COMPACT AND FLEXIBLE NTRU IMPLEMENTATION ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SINAN EMIR KORKMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CYBER SECURITY

AUGUST 2022

Approval of the thesis:

COMPACT AND FLEXIBLE NTRU IMPLEMENTATION ON FPGA

submitted by **SINAN EMIR KORKMAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

Prof. Dr. [illegible]
Dean, Graduate School of **Graduate School of Informatics** _____

Assoc. Prof. Dr. Cihangir TEZCAN
Head of Department, **Cyber Security** _____

Assoc. Prof. Dr. Cihangir TEZCAN
Supervisor, **Cyber Security Department, METU** _____

Examining Committee Members:

Prof. Dr. Ali Aydın SELÇUK
Computer Engineering Dept., TOBB ETÜ _____

Assoc. Prof. Dr. Cihangir TEZCAN
Cyber Security Dept., METU _____

Prof. Dr. Banu Günel KILIÇ
Information Systems Dept., METU _____

Date: 23.08.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Sinan Emir KORKMAZ

Signature :

ABSTRACT

COMPACT AND FLEXIBLE NTRU IMPLEMENTATION ON FPGA

KORKMAZ, Sinan Emir
M.S., Department of Cyber Security
Supervisor: Assoc. Prof. Dr. Cihangir TEZCAN

August 2022, 78 pages

At the dawn of quantum computing, our most trusted cryptosystems are at significant risk. All vastly used and standardized public-key algorithms such as RSA and ECC were designed to withstand the attacks from classical computers by using integer factorization and discrete logarithm problems. However, quantum computers can generate the whole solution space for these problems that contains all the possible keys and reduce it to the correct key in polynomial time. Therefore, we need to start using a new public key encryption algorithm before the first full-scale quantum computer starts to work. To select this new algorithm, NIST organized a competition in 2016. They received 59 submissions in the field of encryption. With the passing rounds, algorithms are heavily investigated according to security and performance metrics by researchers all around the globe.

In this thesis work, we focused on the NTRU, one of the third-round candidate algorithms. This lattice-based algorithm uses Shortest Vector Problem as the encryption function and has the ability to provide secrecy against quantum computers. We worked on the hardware implementation of the NTRU. By implementing an algorithm on FPGA, we can benefit from gate-level parallelism and design algorithm-specific logical blocks. We implemented encryption, decryption, and data processing modules with our proposed improvements. The key generation module is not implemented because generated keys can be loaded manually and have a long life of usage. Our design focuses on resource optimization and flexibility. These properties enabled us to provide a suitable solution for low-power embedded network devices.

Keywords: NTRU, FPGA, Post Quantum Cryptography, Public Key Cryptography, Cryptography

ÖZ

NTRU ALGORİTMASININ FPGA ÜZERİNDE GERÇEKLENMESİ

KORKMAZ, Sinan Emir
Yüksek Lisans, Siber Güvenlik Bölümü
Tez Yöneticisi: Doç. Dr. Cihangir TEZCAN

Ağustos 2022 , 78 sayfa

Kuantum bilgisayarların artan bir ivme ile geliştiği günümüzde, güvenli iletişimimizi sağlayan açık anatarlı şifreleme algoritmaları tehdit altındadır. Şu an kullandığımız, güvenirliğini ispat ederek standartlaşmış RSA ve ECC gibi algoritmaların klasik bilgisayar yapıları tarafından yapılan saldırılara karşı dayanımı yüksektir. Bu yeteneklerini çarpanlara ayırma ve ayrık logaritma problemleri sayesinde kazanmaktadırlar. Fakat, sahip oldukları eşsiz paralel işlem yeteneği ile kuantum bilgisayarlar bu algoritmalara ait bütün çözüm olasılıklarını aynı anda oluşturup doğru olana polinom zamanda indirgeyebilirler. Bu sebeple yeni bir açık anahtarlı şifreleme algoritmasına geçiş yapmamız gerekmektedir. Bu geçişi standardize edebilmek için American Standartlar ve Teknoloji Enstitüsü (NIST) bir yarışma düzenlemektedir. 2016 yılında başlayan ve 59 algoritmanın şifreleme alanında katıldığı bu yarışmanın her turunda katılımcı algoritmalarin güvenlik ve performans yetenekleri dünyanın her yerindeki bağımsız araştırmacılar tarafından incelenmektedir.

Bu tez çalışmasında 3. tur katılımcısı NTRU algoritması incelenmiş olup, alanda programlanabilir kapı dizisi (FPGA) üzerinde gerçeklenmiştir. NTRU algoritması çok boyutlu kafes yapısına sahiptir ve en kısa vektör sorusunu şifreleme fonksiyonu olarak kullanır. Bu sayede kuantum bilgisayarlara karşı güvenlik sağlayabilmektedir. Donanım üzerinde yapılan tasarımlar kapı seviyesinde paralelleştirme ve özel tasarlanmış mantıksal yapılardan faydalanabilirler. Şifreleme, şifrenin çözülmesi ve bilgilerin paketlenmesi işlemleri önerilen iyileştirmelerle gerçeklenmiştir. Şifre üretimi, üretilen şifrelerin uzun ömürleri ve elle yüklenebilecek olmaları sebebiyle tasarıma dahil edilmemiştir. Tasarım düşük kaynak tüketimi ve esneklik kriterlerine göre yapılmıştır. Bu kriterlere uyum sağlayarak ağ üzerinde çalışan düşük güçlü gömülü cihazlar için uygun bir çözüm oluşturulmuştur.

Anahtar Kelimeler: NTRU, FPGA, Açık Anahtarlı Şifreleme, Kuantum Sonrası Kriptografi, Şifreleme

to my family

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my thesis advisor Assist. Prof. Dr. Cihangir TEZCAN for the precious guidance and generous help that makes this work possible. Also, I want to especially thank him for introducing me to the world of Cryptography.

I want to thank my family and friends that support me unconditionally in every situation and give me the strength to continue.

I would like to thank TUALCOM for supporting this thesis. Working here is an excellent adventure as I am gaining new skills and gathering more knowledge.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
ALM	Adaptive Logic Modules
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machines
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
CVP	Closest Vector Problem
DAC	Digital to Analog Converter
DFF	Data Flip Flop
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
EEA	Extended Euclidean Algorithm
EEPROM	Electronically Erasable Programmable Read-Only Memory
EMI	Electromagnetic Interference
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GCD	Greatest Common Divisor
GF	Galois Field
GPU	Graphical Processing Unit

HDL	Hardware Description Language
HSTL	High Speed Transceiver Logic
HSUL	High-Speed Unterminated Logic
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IO	Input Output
JTAG	Joint Test Action Group
KEM	Key Exchange Mechanism
LSB	Least Significant Bit
LUT	Look-Up Table
LVDS	Low Voltage Differential Signaling
MSB	Most Significant Bit
MUX	Multiplexer
NIST	National Institute of Standards and Technology
NTRU	Nth degree-Truncated Polynomial Ring Units
PCI	Peripheral Component Interconnect
PLL	Phase Locked Loop
qbit	Quantum bit
QFT	Quantum Fourier Transform
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSA	Rivest–Shamir–Adleman
RTOS	Real Time Operating System
SATA	Serial AT Attachment
SD	Secure Digital
SIMD	Single Instruction Multiple Data

SISD	Single Instruction Single Data
SM	Switch Matrix
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SSTL	Stub Series Terminated Logic
SVP	Shortest Vector Problem
UART	Universal Asynchronous Receiver-Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CHAPTER 1

INTRODUCTION

At the dawn of quantum computing, our most trusted cryptosystems are at significant risk. The level of secrecy reduction has to be addressed separately for different types of algorithms. Because while the effect of quantum computers on private key encryption algorithms may be ignored or easily overcome, it is catastrophic for the public key encryption algorithms.

Firstly, private key encryption algorithms such as Triple-DES and AES will lose their security level by half. An AES256 key attacked by a quantum computer can provide the same secrecy as an AES128 key attacked by a conventional computer (Bonnetain et al., 2019). By just increasing the key size, one can safely continue to use the current standard algorithms like AES. However, these algorithms need secret keys that have to be determined on a secure channel before communicating freely. In many real-life scenarios, having access to a medium that prevents eavesdropping or tampering is impossible.

Using a secret algorithm might come up to mind. Because even if the adversaries know the key, they will have no idea about how to use it. From a layman's perspective, we can ensure security by using a confidential algorithm. However, this security by obscurity approach is not accepted in many fields. According to the security assessments, the secret key has to be the only element that ensures secrecy in an open and well-known system. Using the security by obscurity approach keeps the attacker in the dark, but it will also keep the defenders in similar conditions. Without an open-source algorithm and open discussion platform like academia, the defenders cannot determine their algorithm's strengths and weaknesses. The current general practice for security is to use well-tested and standardized algorithms according to the secrecy needs. This practice is formalized by NIST by stating, "System security should not depend on the secrecy of the implementation or its components." (Scarfone et al., 2008)

Public key encryption algorithms are designed to communicate in an open world without any shared secret. Public key encryption algorithms use two different keys. One is shared with the world, and it can be used by anyone who wants to send a message to the key's owner. If this message is received by an adversary who knows the public key, there will not be a security breach because there is no practically possible way to extract the information from the encrypted message. This process can only be done by someone who knows the private key by using the trap function of the algorithm.

Increasing the key size to increase the security approach is impractical in the public

key cryptography algorithms while working against quantum computational devices. Because, in public key algorithms, we can change the time complexity of searching for the correct key by using quantum computers. However, in private key algorithms, the usage of quantum computers only enables us to make a reduction.

The underlying mechanism of quantum computers is storing many possibilities in their smallest units, which are named qbits, unlike the classical computers that can hold two possibilities in their bits. Quantum computers do not need to try the possible keys one by one while attacking an algorithm because they have the capability of working with multiple possibilities at the same time. By using Shor's Algorithm, quantum computers can generate a solution space containing all the possible keys and reduce it to the correct key in polynomial time for the problems of integer factorization and discrete logarithm. The polynomial time solution to our most used public key algorithms is the dead of the secrecy in today's communication practices. Therefore, we need to start using a new public key encryption algorithm before the first full-scale quantum computer is deployed.

1.1 NIST Post Quantum Cryptography Challenge

To select this new algorithm for standardization, NIST is organizing a competition. NIST Post Quantum Cryptography Competition was formally called for the proposals on January 03, 2017, and the deadline for round 1 submissions ended on November 30, 2017. You can see the complete timetable of the challenge in Table 1.1. The table was adapted from
https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline

Table 1.1: NIST PQC Timeline

April 2-3, 2015	NIST held a Cybersecurity in a Post-Quantum World workshop
Feb 24-26, 2016	NIST calls for submissions in PQCrypto 2016
April 28, 2016	NISTIR 8105 report was released
Dec 20, 2016	Formal call for Round 1 submissions
Nov 30, 2017	Deadline for Round 1 submissions
Dec 4, 2017	The Ship Has Sailed: The NIST Post-Quantum Crypto "Competition" presentation at AsiaCrypt 2017
Dec 21, 2017	Round 1 contestants are announced
Apr 11, 2018	Let's Get Ready to Rumble - The NIST PQC "Competition" presentation at PQCrypto 2018
April 11-13, 2018	1. PQC Standardization Conference
January 30, 2019	Round 2 contestants are announced
March 15, 2019	2. Round's submission packages deadline
May 8-10, 2019	Round 2 of the NIST PQC "Competition" - What was NIST Thinking? presentation at PQCrypto 2019

Table 1.1 (continued).

August 22-24, 2019	2. PQC Standardization Conference
July 22, 2020	Round 3 contestants are announced
October 1, 2020	3. Round's submission packages deadline
June 7-9, 2021	3. PQC Standardization Conference
July 5, 2022	Announcing which PQC candidates to be standardized and the fourth round candidates
2022/2024	Draft Standards Available

The competition was categorized into three security levels according to an exhaustive key search method and scaled to the AES algorithm with key sizes of 128, 192, and 256 bits. A total of 59 encryption/KEM (Key Exchange Mechanism) and 23 digital signature algorithms are submitted (Moody, 2017). You can see submitted encryption algorithms in Table 1.2.

Table 1.2: NIST PQC Round 1 Submissions

Name	Type	Result	Source
BIG QUAKE	Code-based	Eliminated	Bardet et al., 2017
BIKE	Code-based	Passed	Aragon, Barreto, et al., 2017
CFPKM	Multivariate	Eliminated	Chakraborty et al., 2017
Classic McEliece	Code-based	Passed	Bernstein, Chou, et al., 2017
Compact-LWE	Lattice	Eliminated	D. Liu et al., 2017
CRYSTALS-KYBER	Lattice	Passed	Bos et al., 2018
DAGS	Code-based	Eliminated	Banegas et al., 2018
Ding Key Exchange	Lattice	Eliminated	Ding et al., 2017
DME	Multivariate	Eliminated	Luengo et al., 2017
Edon-K	Code-based	Withdrawn	Gligoroski and Gjøsteen, 2017
EMBLEM & R.EMBLEM	Lattice	Eliminated	Seo et al., 2017
FrodoKEM	Lattice	Passed	Naehrig et al., 2017
Giophantus	Multivariate	Eliminated	Akiyama et al., 2017
Guess Again	Other	Eliminated	Shpilrain et al., 2017
HILA5	Lattice	Merged to Round5	Saarinen, 2017
HK17	Other	Withdrawn	Hetch and Kamlofsky, 2017

Table 1.2 (continued).

HQC	Code-based	Passed	Melchor, Aragon, Bettaieb, Bidoux, Blazy, Deneuville, Gaborit, Persichetti, et al., 2017
KCL	Lattice	Eliminated	Zhao et al., 2017
KINDI	Lattice	Eliminated	El Bansarkhani, 2017
LAC	Lattice	Passed	Lu et al., 2018
LAKE	Code-based	Merged to ROLLO	Aragon, Blazy, et al., 2017b
LEDAkem	Code-based	Merged to LEDAcrypt	Baldi et al., 2018
LEDApkc	Code-based	Merged to LEDAcrypt	Baldi et al., 2017
Lepton	Code-based	Eliminated	Y. Yu and Zhang, 2017
LIMA	Lattice	Eliminated	Albrecht, Lindell, et al., 2017
Lizard	Lattice	Eliminated	Cheon et al., 2018
LOCKER	Code-based	Merged to ROLLO	Aragon, Blazy, et al., 2017a
LOTUS	Lattice	Eliminated	Le Trieu Phong et al., 2017
McNie	Code-based	Eliminated	J.-L. Kim et al., 2018
Mersenne-756839	Other	Eliminated	Aggarwal et al., 2017
NewHope	Lattice	Passed	Alkim et al., 2016
NTRUEncrypt	Lattice	Merged to NTRU	Hoffstein et al., 2017
NTRU-HRSS-KEM	Lattice	Merged to NTRU	Hülsing et al., 2017
NTRU Prime	Lattice	Passed	Bernstein et al., 2016
NTS-KEM	Code-based	Passed	Albrecht, Cid, et al., 2017
Odd Manhattan	Lattice	Eliminated	Plantard, 2017
Ouroboros-R	Code-based	Merged to ROLLO	Melchor, Aragon, Bettaieb, Bidoux, Blazy, Deneuville, Gaborit, Hauteville, et al., 2017
PQ RSA-Encryption	RSA	Eliminated	Bernstein, Heninger, et al., 2017
QC-MDPC KEM	Code-based	Eliminated	Yamada et al., 2017
Ramstake	Code-based	Eliminated	Szepieniec, 2017
RLCE-KEM	Code-based	Eliminated	Wang, 2017
Round2	Lattice	Merged to Round5	Baan et al., 2017

Table 1.2 (continued).

RQC	Code-based	Passed	Melchor, Aragon, Bettaieb, Bidoux, Blazy, Deneuville, Gaborit, and Zémor, 2017
RVB	Other	Withdrawn	Brands and Roellgen, 2015
SABER	Lattice	Passed	D'Anvers et al., 2018
SIKE	Elliptic Curve	Passed	Azarderakhsh et al., 2017
SRTPI	Multivariate	Eliminated	Peretz and Granot, 2017
Three Bears	Lattice	Passed	Hamburg, 2017
Titanium	Lattice	Eliminated	Steinfeld et al., 2017

Submissions can be categorized into the following algorithms according to their cryptographic primitives:

- Lattice-Based: One of the most popular algorithms for the NIST challenge's first round and have complete dominance in the third round. In these algorithms, the lattices are used as the encryption primitive or to prove the security of the underlining algorithm.
- Code Based: In these types of algorithms, we are using the error correcting codes (ECC) as cryptographic elements. Error Correcting Codes are designed to maintain a communication channel in a highly noisy environment. They carry out their missions by generating redundant information and adding it to the original messages. The most known examples are LDPC (Low-Density Parity-Check) and RS (Reed Solomon) codes. Because of their massive key sizes, they were not very popular in the field of cryptography. However, their immunity against quantum computers is making them popular now.
- Multivariate: These algorithms are based on the polynomials constructed by more than one variable. For example : $P(x, y, z) = x^3 * z + 3 * x^2 * y^3 * z - x * z^2 + 1$. Introduced by (Matsumoto & Imai, 1988). They show presence in the first stage of the challenge. However, all three multivariate-based algorithms are eliminated in this stage.
- RSA: Original integer factorization-based RSA algorithm participates with increased key sizes. It provides security by using a key that is big enough to resist a search attack carried out in polynomial time. Because of the massive keys, this method is impractical and eliminated in the first stage.
- Elliptic Curve: SIKE (Supersingular Isogeny Diffie–Hellman Key Exchange) algorithm's used elliptic curves as their primitives. Security against quantum computers comes with generating isogeny from the elliptic curves.

Second-stage submissions and their results are listed in Table 1.3.

Table 1.3: NIST PQC Round 2 Submissions

Name	Type	Result
BIKE	Code-based	Alternate Candidate
Classic McEliece	Code-based	Passed
CRYSTALS-KYBER	Lattice	Passed
FrodoKEM	Lattice	Alternate Candidate
HQC	Code-based	Alternate Candidate
LAC	Lattice	Eleminated
LEDAcrypt	Code-based	Eleminated
NewHope	Lattice	Eleminated
NTRU	Lattice	Passed
NTRU Prime	Lattice	Alternate Candidate
NTS-KEM	Code-based	Passed
ROLLO	Code-based	Eleminated
Round5	Lattice	Eleminated
RQC	Code-based	Eleminated
SABER	Lattice	Passed
SIKE	Elliptic Curve	Alternate Candidat
Three Bears	Lattice	Eleminated

While this thesis was in progress, the NIST announced CRYSTALS-KYBER as the first algorithm to be standardized at the end of round 3. You can see the submitted algorithms and their result in Table 1.4. Also, for digital signature purposes, CRYSTALS-DILITHIUM, FALCON, and SPHINCS+ algorithms will be standardized. While NIST decided on the first algorithm to standardize, the competition still continues, and it is in its fourth round with the reaming algorithms. These algorithms can be seen in Table 1.4

Table 1.4: NIST PQC Round 3 Submissions

Name	Type	Result
BIKE	Code-based	Passed
Classic McEliece	Code-based	Passed
CRYSTALS-KYBER	Lattice	Standardized
FrodoKEM	Lattice	Eleminated
HQC	Code-based	Passed
NTRU	Lattice	Eleminated
NTRU Prime	Lattice	Eleminated

Table 1.4 (continued).

NTS-KEM	Code-based	Eleminated
SABER	Lattice	Eleminated
SIKE	Elliptic Curve	Passed

Most of the participant algorithms are mathematically secure from the beginning of the first stages of the competition. However, their performance metrics are open to improvements. Also, the hardware implementation of these algorithms is highly important because many network-capable devices have to adapt to the new algorithm. With the increasing number of user devices, using specific logic blocks makes more sense than the CPU for server-like devices. Also, many embedded devices have to adopt the new standard algorithm to connect to a network. As this thesis work started long before the round 4 announcement, we chose the NTRU algorithm for development.

1.2 NTRU

NTRU was designed as a property algorithm by Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. In 2013, it was released under GNU License. The algorithm uses latices and has the shortest vector problem as the underlying security primitive. The algorithm was a third-round participant in the NIST PQC Competition.

This algorithm has only three parameters named N, q, p. The parameter N defines the length of the polynomials. q, p parameters are used to determine the Galois Fields. Therefore, modulo operations are calculated according to these parameters.

1.3 Effects of Quantum Computers on Cryptography

As mentioned, quantum computers are the new era of computing devices. Up to them, we do not have a new method of computing. The improvements have come from increasing the clock speed, utilizing parallelism, or optimizing the algorithms. However, quantum computers let us use the superposition of physical systems. While classical computation devices can hold two possibilities in their smallest building blocks named bits, a quantum computer can hold all the possible states of their qbits. This will let us calculate all the possible results at the same time. The problem of handling everything at the same time is eliminating the wrong solutions from this superposition state. For this purpose, we have special algorithms and quantum circuits designed for these specific algorithms.

For cryptographic purposes, we are interested in two quantum algorithms: Grover's Search algorithm and Shor's algorithm. Grover's algorithm is an unconstructed search algorithm. This algorithm is important for us. Because by using that, a quantum computer can halve the security of any private key encryption algorithms. By oversimplifying the working mechanisms, Grover's algorithm does not try to generate the

solution. It tries to check every possible solution's correctness in a superposition. While generating a correct solution for a problem is hard, checking a solution's correctness might be easy. For example, solving a maze might require many trials and returns to the previous states. However, checking a list of orders and deciding if they are a solution to this maze is easy.

We use Shor's algorithm while factorizing integers. The integer factorization problem is the underlying security mechanism of the RSA. To briefly explain Shor's algorithm, we start with a random guess and check if it is a factor of the number N. When we fail, we do not select a new test number; the following test number is derived from the previous one with a rule. A quantum computer can generate all the numbers that will be generated by following this rule in a superposition. We can calculate the result of every generated number under modulo by N. After that, we collapse the superposition to a single result. However, the result contains periodic signals that generate the same output. If we feed this state to a Quantum Fourier Transform (QFT) module, we can find this period. The period and the selected random number then be used to factor the big integer (N).

Grover's and Shor's algorithms are resource hungry in terms of qbits and gates, and we will not see a quantum computer that can risk our current public key encryption algorithms in the near future. However, the general practice for states is to keep the secrets for 25 or 50 years. Even if we do not have the technology today, we might gather the encrypted data at the moment and break them when quantum computers are ready to make the necessary calculations. With that limitations, we are required to mitigate a new quantum-safe algorithm as soon as possible. You can see this scenario in Figure 1.1. In this figure; x represents the time passed before mitigation, y is the required time before the information becomes public, and z is the time passed before the first full-scale quantum computer started to work.

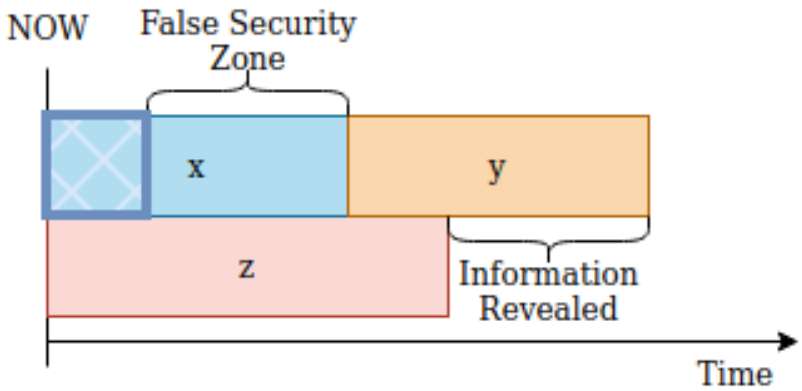


Figure 1.1: Unwanted information reveal

1.4 Reasons to work on FPGA

We selected our computational units as FPGAs (Field-Programmable Gate Array). Because they are the irreplaceable component of the military, aerospace, and communication areas, even if they are sometimes backed up by processors for the time-independent computational workloads, many designs from these fields do not include an onboard processor because of the restrictions and security concerns. FPGAs have more resilience against temperature and radiation. They might generate countless high-speed communication ports as long as their IO (Input Output) port numbers let them. The FPGAs might have hundreds of IO ports. They can support high-speed parallel synchronized IO operations. This process is essential for reading the ADCs and feeding DACs used in communication and digital signal processing. Their incredible power in parallel architecture makes them a perfect fit for working with high computational loads in real-time systems. While normal processors have interrupt controllers to respond to the changing situation and support RTOSes (Real-Time Operating System) for time-bounded calculations, we can control everything in the FPGA within a one-clock beat precision. This response time beats everything that a processor can offer.

1.5 Other NTRU Implementations

(Bailey et al., 2001) is probably the first hardware implementation of the NTRU algorithm. The purpose of that paper is to investigate four aspects. Firstly, they compared Karatsuba-Ofman polynomial multiplication (Karatsuba & Ofman, 1962) and their first algorithm. According to their results, algorithm 1 outperforms the Karatsuba-Ofman algorithm. Secondly, the paper provides results for three different microprocessors, which are MC68EX328 Dragonball, Intel 80386, and 37MHz ARM7 performance comparisons. Thirdly, they provided closed source FPGA implantations that build on a fixed parameter and can only perform encryption. They used the parameters of (N, p, q) = (251, X + 2, 128). In that implementation, they defined the p as a polynomial. Because $x + 2$ and 128 are co-prime, this selection is valid.

(O'Rourke, 2002) only focuses on the polynomial multiplication process of the NTRU. Starting with software designs and comparison, they went forward to hardware and provided a scalable polynomial multiplier core design on FPGA. Their design uses the number of gates between 1483 and 19270 according to the selected parameters.

(Kaps, 2006) in that doctoral thesis, they focused on the different algorithms' power efficiencies and analyzed Rabin's Scheme (Rabin, 1979), NtruEncrypt, and ECC. The NTRU algorithm is implemented only for encryption, while the decryption part is discarded. The implementation for encryption uses 523 combinational elements and 2327 storage elements in their base design. But it might be increased up to 7000 and 9200, respectively, if the parallelization degree is increased. However, the parameters used are insufficient to provide reliable secrecy with a maximum of 61 bits of security against exhaustive search.

(Atici et al., 2008) is designed to be used in RFIDs and in the nodes of the sensor

networks. This purpose forced the implementation to be highly power efficient. The implementation is for ASICs and is focused on power consumption. But the evolution of the NTRU algorithm makes this design's parameters obsolete. The used parameters are N=167, q=128, and p=3. The design spends 1651 combinational and 8848 non-combinational elements for encryption and decryption.

(Kamal & Youssef, 2009) proposed a new approach by utilizing the statistical properties of the distance between the non-zero elements in the polynomials. Also, they provided speed improvements for encryption and decryption processes. They managed to increase the calculation speed by approximately 160%; furthermore, if the parallelization is increased, it can reach up to 216%. They used the parameters up to N= 251, q=128, p=3, which are insecure as general purpose cases. Their implementation spends 4838 FFs and 21654 LUTs.

(B. Liu & Wu, 2015) implemented a hardware multiplier for truncated polynomial rings using LFSRs (Linear feedback shift registers) and claimed the design has the best area-delay product. Their implementation uses 7294 logic elements, 3768 combinational elements, and 3526 registers.

(Farahmand et al., 2019) worked on a hardware and software co-design and compared the different approaches (RTL vs. HLs) to the full software implementation. The main goal of this design was speed. This design's encryption parameters have to be decided before the synthesis process. They used a minimum of 44257 LUTs and 29655 FFs, and for increased security, 76972 LUTs and 49674 FFs were spent.

(Wera, 2020) focuses on IoT devices. Therefore one of the design criteria is compactness. That design supports big polynomials for security, but changing the parameters in runtime is not possible.

(Keersmaekers, 2021) replaces costly polynomial multiplications with NTT (Number Theoretic Transform). This improves the time complexity. However, it doubles the polynomial sizes. Because NTRU was not designed according to this transformation.

(H. Yu et al., 2021) focuses on the performance comparison between FPGA and GPU architectures. As expected, FPGAs can perform better with increasing parallelism. To get the best performance, they increased the resource utilization and spent half of the resources of a high-end FPGA (Stratix V).

(Dang et al., 2021) compares CRYSTALS-Kyber, NTRU, and Saber according to their performances. To do that, they implemented these algorithms on hardware.

(Hoffstein et al., 2017) investigates the security promises of the algorithm with different parameters and messages. You can see the simplified results in Table 1.5

1.6 Our Contribution

In this thesis work, we targeted FPGA-based embedded network devices. As these devices' resources are utilized for their own missions, our implementation needs to be minimal. To succeed, we stored data in RAMs. Contrary to many implementations,

Table 1.5: NTRUEncrypt security estimation according to Hoffstein in his 2017 article

N	q	Original Security Estimate	search cost
401	2048	112	145
439	2048	128	147
593	2048	192	193
743	2048	256	256

we make the polynomial multiplication in serial. With that method, we were able to decrease the needed data at every cycle and did not have to store the coefficients in FFs. This helps us to develop a flexible architecture to support different parameters. One word of memory can hold our four coefficients. As we can reach four coefficients at the same time, we are able to speed up the serial calculation with a small shifter and adder module. Also, by compressing the data, we stored different keys in RAM without spending additional resources. That technique decreases the costly flash operations before communicating with other devices that have different capabilities or are using different parameters. As we are targeting network devices, we have to be ready for fragmented packets. In this scenario, we can partially process the data, contrary to waiting for the remaining part. Lastly, we implemented a serializer and deserializer module for packing the NTRU's word size information to bytes to increase the efficiency of the communication channel.

CHAPTER 2

QUANTUM COMPUTING

The first computation devices were analog. They made calculations by generally rotating disks, roads, shafts, and gears. One of the well-known early analog computers is the "Antikythera mechanism". By using this hand-powered solar system model, the Ancient Greeks were able to calculate astronomical events years in advance (Efstathiou & Efstathiou, 2018). With the evolution of technology, the paradigm shifted to electronic devices. In these devices, the decisions are given by the voltage levels, not by the gear positions. From their emergence, they have been improved drastically, starting from vacuum tubes; now we have a few atoms wide transistors. The computation power that we have today might be enough to sustain a modern world's needs. However, tomorrow's world requires new techniques. Quantum computing is the new horizon.

Quantum computers use quantum superposition and quantum entanglement to work. These properties let a quantum computer hold every possible state of the qbits and make calculations on every possibility at the same time. While quantum computers will bring new horizons, they will not replace classical computational devices. The classical computers were designed as self-sufficient systems. However, a quantum computer needs many other complex systems to operate; even one of the helper systems is a classical computer. You can see an example diagram of the quantum computer in Figure 2.1. To control the qbits, we are using microwave signals. The signals are generated and read by analog components with the help of FPGAs. FPGAs in the systems are connected to a classical computer for the calculations. Also, the classical computer works as an interface module. (Sanders, 2017)

Many research centers and companies around the world are currently trying to increase the qbit numbers of quantum computers. While we have many quantum computers with small qbits, making a quantum computer with many more qbits is still a serious challenge. We cannot combine qbits like we have combined the classical processor cores in a CPU. Combining quantum computers like combining processors in a server or cloud does not work either. All of the qbits in the system have to be a part of a single circuit.

We stated that quantum computers could store multiple possibilities on their qbits. Storing the data is meaningless on its own. Because when we need to receive the results from the quantum computers, it will give out one of the random states. In an unprocessed quantum system, all the probabilities of the states are equal. To get meaningful results from quantum computers, we need to manipulate the possibilities. This manipulation process requires additional qbits and gates. In (Orman, 2021), they

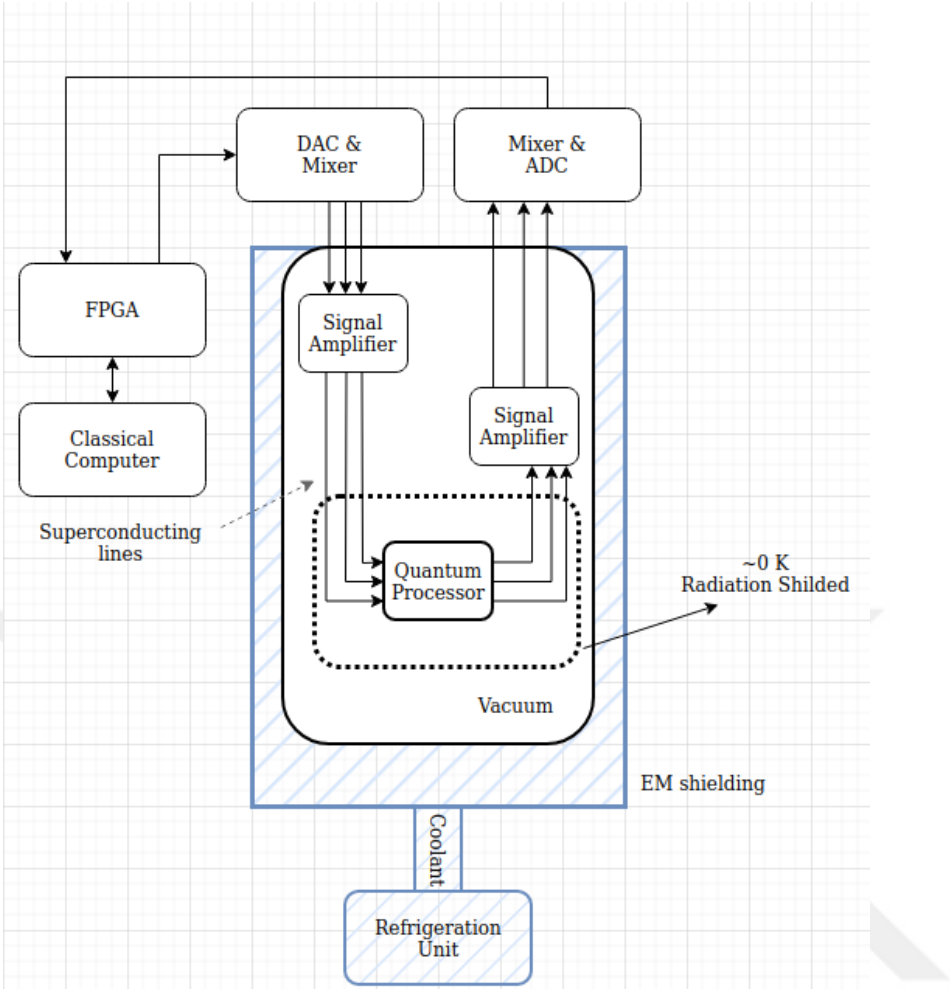


Figure 2.1: Basic diagram of a quantum Computer

investigated the qbit requirements to break the standardized public key encryption algorithms and provide time comparisons between classical computers and quantum computers. The classical computer selected for that study was Intel's Xeon Gold 6130. This processor has 16 physical cores that operate at 2100 MHz. The result of that study can be summarized in Table 2.1 for RSA. Also, they provided calculations for ECC. You can see the results in Table 2.2

Maintaining the entangled quantum states is named quantum coherence. Also, we can define quantum coherence as the evaluation of how much the system obeys the pure coherent states, not the probability distributions of the states. As quantum coherence is highly difficult to maintain, the researchers developed different methods like using ion traps, optical traps, superconductive circuits, or using semiconductors as carriers to maintain the coherence. However, quantum decoherence can still occur with interference from outside. Quantum computers generally work in isolation with near 0-kelvin temperatures to avoid the spontaneous self-decoherence effect. While it can happen destructively, we also need quantum decoherence to get the result of the calculations. After the possibilities are arranged to collapse the superposition to the

Table 2.1: Quantum computer vs. classical computer comparison for breaking RSA and required resources according to Orman

	bit size	qbits logical	qbits physical	Toff gates	Quantum device time (h)	Classical device time (h)
RSA 240	797	2406	8M	150M	1	$8.766 * 10^6$
RSA 1024	1024	3092	10M	400M	1.5	$7.305 * 10^8$
RSA 2048	2048	6189	20M	2700M	7	$5,264 * 10^{18}$
RSA 3072	3072	9287	40M	9900M	13	$1,930 * 10^{25}$
RSA 4096	4096	12386	50M	23000M	20	$4,387 * 10^{30}$

Table 2.2: Required time for a quantum computer to break ECDLP and required resources according to Orman

bit size	qbits logical	qbits physical	Toff gates	Quantum device time (h)
224	2042	6M	84.3 B	288
256	2330	7M	126 B	432
384	3484	10M	452 B	1440
521	4719	14M	1140 B	3840

correct solution, we need to break the entanglement of the system. Before making another calculation, we need to establish the quantum coherence again. The time spent while doing that is also a critical criterion while evaluating quantum computers.

You can see the maximum number of qbits in one system according to years in Figure 2.2. The exponential increasing trend might indicate they will be ready for full-scale computations in the not far future. IBM is expecting to active Osprey with 433 qbits in 2022, Condor with 1,221+ qbits in 2023, and Kookaburra with 4,158+ qbits in 2025, according to their roadmap (Gambetta, 2022). While many research centers and private companies are trying to increase the qubit numbers in the system, there are active researches to optimize the quantum algorithms to decrease the qbit requirements. (Roetteler et al., 2017) in that article, they optimized Shor's algorithm for working on the discrete logarithm. The results of that article are used in (Orman, 2021).

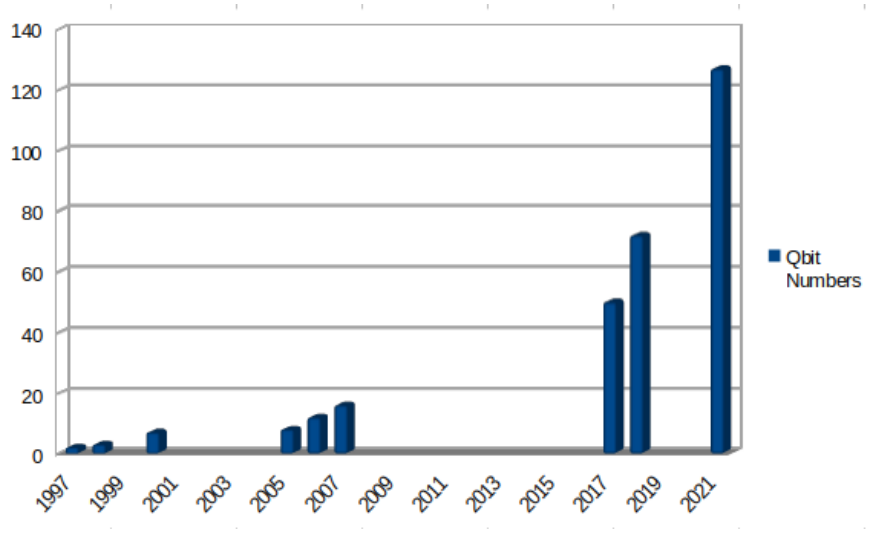


Figure 2.2: Maximum utilized qbits in single system vs. years

Quantum computers can be epoch-making in the following areas:

Cryptography: With the help of Shor's and Grover's algorithms, quantum computers have significant effects on all the well-known encryption algorithms. The result can be ignored in the private key encryption algorithms because doubling the key size can restore the previous security level. However, currently, standardized public key encryption algorithms will not be able to provide security. Because from classical computers' perspective, their underlying problem is non-polynomial. On the contrary, a functional quantum computer can solve these problems in polynomial time.

Search Problems: There is no way to optimize unstructured search problems while using classical computers. The time complexity of this process is $O(N)$. But, using Grover's algorithm on a quantum computer, the time complexity will be dropped to $O(\sqrt{N})$.

Quantum Systems Simulation: As the behavior of the atomic and molecular particles are tough to predict. Classical computers use massive resources to do the calculations; however, a quantum computer might be able to achieve this task effectively.

Computer-aided Chemistry and Biology: Processes like genome sequencing and designing job-specific proteins are taking too much time. Because these systems are not constructed with basic rules and resemblance in the construction of the blocks does not guarantee similarity of the functionality. In the future, quantum computers might replace supercomputers in this area.

2.1 Grover's Search Algorithm

Grover's algorithm works in unstructured search spaces. For classical computers, this job has a time complexity of $O(N)$. Because there is no way to optimize this process, and the only way to guarantee success is to try every possibility. However, quantum computers can do this job with $O(\sqrt{N})$ time complexity by using Grover's Search Algorithm. This algorithm was developed by Lov Grover in 1996 (Grover, 1996). Because Grover's Algorithm does not use any internal structure of the dataset, it can be used in most search problems and can improve the performance quadratically. In this section, we will explain Grover's Search Algorithm without getting deep into the quantum circuits.

In this algorithm, we have a black box module named Oracle. We show the possibilities as quantum states like $|0\rangle$, $|1\rangle, |2\rangle, |3\rangle, ..., |7\rangle$. For this example, we have eight different states. The oracle adds a negative phase to the correct solution state. We can represent our oracle in a matrix form.

$$\text{oracle} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

At this point, we generated a correct solution pattern. We still do not know the solution. But, now we can check whether a solution is correct or not. After we feed our oracle with the solutions, the correct one still has the same possibility of the wrong states. But, the correct solution has a phase difference. The generated solution space is fed into the amplification unit. The amplification unit will increase the possibility of the correct state while decreasing the possibilities of the wrong states. We need to repeat this process several times to increase the possibility of the correct state before collapsing the superposition. The Grover's algorithm's visual representation is provided in Figure 2.3.

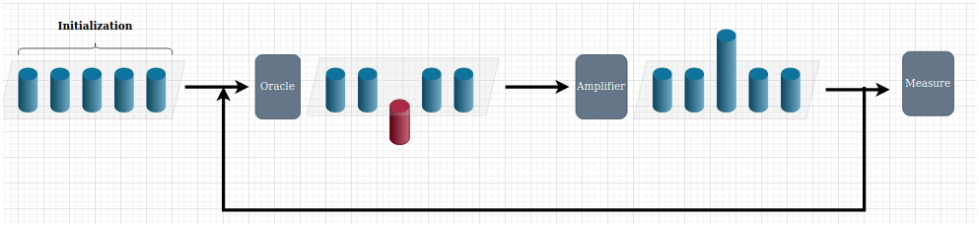


Figure 2.3: Visual Representation of the Grover's Search Algorithm

2.2 Shor's Algorithm

Peter Shor developed this algorithm in 1994 (Shor, 1994). For classical computing devices, our best way to factorize a number is General Number Field Sieve with $O(e^{1.9(\log N)^{1/3}(\log\log N)^{2/3}})$ time complexity (Pomerance, 1996). This complexity is big enough to accept that RSA is secure for practical key sizes like 3072 bits. While running on a quantum computer, Shor's algorithm has a time complexity of $O(\log N)$. This reduction makes today's trustworthy RSA encryption completely insecure for a future with quantum computers.

In Shor's algorithm, we are transforming the problem of integer factorization to frequency determining. Assumed, we are trying to factorize the number N. Firstly, we need to pick a random number g; it should be less than N. If we are not extremely lucky, $gcd(N, g)$ will be equal to 1. After that, we can take our unsuccessful guess and convert it to a better one by calculating the new guess by $g_{new} = g^{p/2} + 1$. If we try all the p values, we will find a common factor with a 37.5% chance. For classical computers, following this algorithm does not make any performance improvements. In contrast, quantum computers can make miracles with the following steps.

A quantum computer can calculate every exponent of a number at the same time with a gate. However, the resultant information is not functional yet. Because all these values are in a superposition, and if we want a result, the superposition will collapse to a random state. You can see this process in Equation 2.1. After that, we will feed this superposition to another gate that can calculate the difference between each state and $m * N$. You can see the result of this gate in Equation 2.2. There will be periodically repeating numbers in the results.

$$|1\rangle + |2\rangle + |3\rangle + |4\rangle + ... \Rightarrow |1, g\rangle + |2, g^2\rangle + |3, g^3\rangle + |4, g^4\rangle + ... \tag{2.1}$$

$$|1, g\rangle + |2, g^2\rangle + |3, g^3\rangle + |4, g^4\rangle + ... \Rightarrow |1, a\rangle + |2, b\rangle + |3, c\rangle + ...+ |8, a\rangle + |15, a\rangle + ... + |22, a\rangle + ... \tag{2.2}$$

After that, we collapse the result to a random state and feed it to the QFT module Equation 2.3. This module will find the periodicity of the input. The result will be equal to $1/p$. If the p is even we can calculate the $g^{p/2} + 1$. If the calculated value is not a multiple of N, it will share a factor with it, and we can calculate the GCD with EEA.

$$|1, g\rangle + |8, a\rangle + |15, a\rangle + ... + |22, a\rangle + ... \Rightarrow |1/7\rangle \tag{2.3}$$

In the above example, we showed the quantum states of Shor's algorithm while factorizing a number that satisfies $gcd(N, 7) \neq 1$.

CHAPTER 3

NTRU

NTRU (Nth degree-Truncated Polynomial Ring Units) algorithm was developed in 1996 by three mathematicians, Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman (Hoffstein et al., 1998). Firstly, it was a proprietary algorithm owned by Security Innovation and distributed under paid licenses. In 2013, NTRU was released under GNU Public License (GPL) and became open source. Daniel Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal released a changed version of NTRU which named NTRUPrime. They changed the algebraic structure of the original NTRU algorithm with worries about security vulnerabilities in 2006. However, through the NIST PQC challenge, no attack was found to exploit these worries. In the third round, the NTRU was a finalist, and NTRUPrime was an alternate candidate. However, they both could not see round four. An NTRU-based digital signature algorithm, Falcon, will be standardized.

NTRU algorithm has three parameters, and they are (N, p, q). The parameter N defines the length of polynomials used in the encryption. p and q values are used in the modulo calculations. Even though these parameters can be selected freely, some parameters are optimized for efficiency and investigated more heavily for their security promises. For example, establishing q as a power of 2 converts the modulo process to binary "and" gates. If we take p as a small prime number, we can quickly write a hardware accelerator for the modulo. Some of the most used ones are listed in Table 3.1. The usage of the parameters is explained in the following sections with numeric examples.

NTRU algorithm uses Shortest Vector Problem (SVP) for cryptographic functions. SVP is lattice-based, unlike RSA's integer factorization or ECC's discrete logarithm.

Table 3.1: Some of the recommended parameters

Security Margin	N	p	q
128 bit	509	3	2048
192 bit	677	3	2048
256 bit	821	3	4096
256 bit	701	3	8192

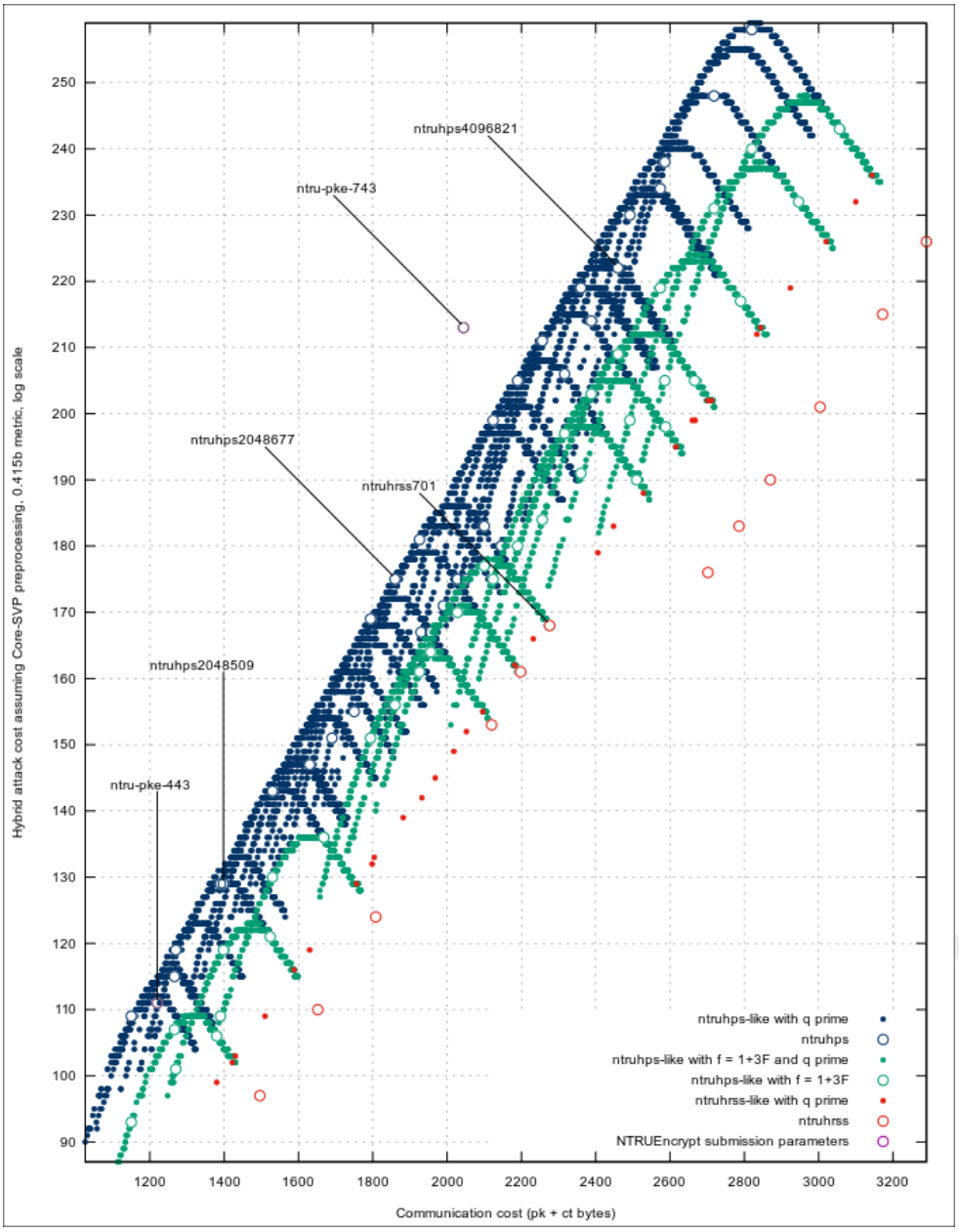
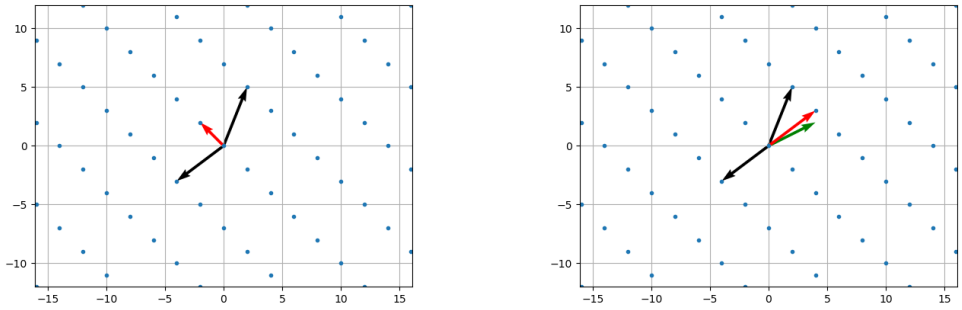


Figure 3.1: NTRU options security comparison

3.1 Lattice

Lattice is the space of possible vectors that can be generated from the given basis vectors using only integer coefficients. The more mathematical definition can be provided with $L \subset \mathbb{R}^n$, $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, $L = \left\{ \sum a_i \mathbf{b}_i \ : \ a_i \in \mathbb{Z} \right\}$. For example, for (7,4) and (3,5) basis vectors, our lattice includes (17,13), (4,-1), but (3,7) cannot be a part of our lattice.



(a) SVP

(b) CVP

Figure 3.2: Lattice Problems

3.2 Shortest Vector Problem

With a provided lattice basis vectors, the Shortest Vector Problem is trying to find the non-zero $\vec{u}$ that minimizes the $|\vec{u}|$. SVP is a sub-problem of the Closest Vector Problem (CVP). In CVP, a random $\vec{w}$ is also provided with basis vectors and trying to find the $\vec{u}$ that minimizes the $|\vec{u} - \vec{w}|$. So, CVP can be transformed to SVP by selecting $\vec{w} = \vec{0}$. You can see the visual representations of SVP and CVP in Figure 3.2.

For example, if basis vectors are given as $\vec{b_1} = (66, 216, 131)$, $\vec{b_2} = (73, 207, 113)$, and $\vec{b_3} = (28, -48, -84)$ the shortest vector in this lattice should be $\vec{u} = (-2, 0, 1)$ with $|\vec{u}| = 2.08$. To get $\vec{u}$, we should select $a_1 = 1219$, $a_2 = -1208$, $a_3 = 276$. With that coefficients we can satisfy the equation $\vec{u} = \vec{b_1} * a_1 + \vec{b_2} * a_2 + \vec{b_3} * a_3$.

3.3 Galois Field

The name was given in honor of French mathematician Évariste Galois. The Galois Fields have finite number of elements in them. The elements in this kind of field should satisfy the following field axioms.

1. Associativity of "+" and "*":
   $\forall\ a, b, c\ \exists\ F$
   $(a + b) + c = a + (b + c)$
   $(a * b) * c = a * (b * c)$

2. Commutativity of "+" and "*":
   $\forall\ a, b\ \exists\ F$
   $a + b = b + a$

$a * b = b * a$

3. Identity of "+" and "*":
   $\forall\, a \,\exists\, F$
   $a + 0 = a$
   $a * 1 = a$

4. Inverse of "+" and "*":
   $\forall\, a \,\exists\, F$
   $a + (-a) = 0$
   $a * (a^{-1}) = 1$ if $a \neq 0$

5. Distributivity of "*" over "+" and "+" over "*":
   $\forall\, a, b, c \,\exists\, F$
   $a * (b + c) = a * b + a * c$
   $(a + b) * c = a * c + b * c$

In Figure 3.1, you can see the effect of the different parameters. This graph is taken from the supplementary materials of the NTRU's NIST PQC round 3 submission package.

The following sections will provide numeric examples of how the NTRU algorithm work. The parameters are selected as $N = 17, p = 3, q = 64$. These parameters are too weak for real-life data security purposes; they are selected to be used in demo calculations. The calculations are provided to help the users to familiarize themself with the NTRU and provide a better understanding by showing the numeric examples. Before starting the key generation process, we will explain the Extended Euclidean Algorithm because it will be used while generating the keys.

3.4 Extended Euclidean Algorithm

The Euclidean Algorithm calculates the greatest common divisor (GCD) of two numbers with an iterative approach. With the Extended Euclidean Algorithm, we can calculate the GCD with x and y coefficients that satisfies the equation $a * x + b * y = GCD(a, b)$. EEA is an iterative algorithm. We start by dividing the bigger integer by the smaller integer. In iterative cycles, we divide the divisor by the remainder until we find a constant value as a quotient. The algorithm is expressed in Algorithm 1:

Algorithm 1: Extended Euclidean Algorithm

Initialization:

$a = num1$
$b = num2$
$s2 = 1$
$t2 = 0$
$s1 = 0$
$t1 = 1$

while $r \neq 0$ **do**
  $q = int(a/b)$
  $r = a - b * q$
  $s = s2 - q * s1$
  $t = t2 - q * t1$
  $a, b = b, r$
  $s2, t2, s1, t1 = s1, t1, s, t$
end
$gcd(num1, num2) = a$

In a numeric example, we will take the inverse of polynomial $x^4 + x^3 + x^2 + 1$ with respect to $x^3 + 1$ in the $GF(7)$. To solve this problem with EEA, we need to convert the equation to $a * (x^4 + x^3 + x^2 + 1) + b * (x^3 + 1) = 1$ mod 7. Some polynomials are underlined to flag them for the next steps. They will be regarded as variables in the backtracking algorithms.

$$\underline{(x^4 + x^3 + x^2 + 1)} = \underline{(x^3 + 1)} * (x + 1) + \underline{(x^2 - x)} \tag{3.1}$$

$$\underline{(x^3 + 1)} = \underline{(x^2 - x)} * (x + 1) + \underline{(x + 1)} \tag{3.2}$$

$$\underline{(x^2 - x)} = \underline{(x + 1)} * (x - 2) + 2 \tag{3.3}$$

We calculated the $GCD(x^4 + x^3 + x^2 + 1, x^3 + 1) = 2$. To find the inverses, GCD should be equal to 1. Therefore, we need to multiply both sides of the equation by 4 before finalizing the calculations. To get $4 * 2 = 1$ mod 7 . Before proceeding we need to rewrite these equations (3.1) to (3.4), (3.2) to (3.5) and (3.3) to (3.6).

$$\underline{(x^4 + x^3 + x^2 + 1)} - \underline{(x^3 + 1)} * (x + 1) = \underline{(x^2 - x)} \tag{3.4}$$

$$\underline{(x^3 + 1)} - \underline{(x^2 - x)} * (x + 1) = \underline{(x + 1)} \tag{3.5}$$

$$\underline{(x^2 - x)} - \underline{(x + 1)} * (x - 2) = 2 \tag{3.6}$$

We substitute the $x + 1$ in the (3.6) with the (3.5).

$$\underline{(x^2 - x)} - (\underline{(x^3 + 1)} - \underline{(x^2 - x)} * (x + 1)) * (x - 2) = 2 \tag{3.7}$$

$$\underline{(x^2 - x)} * (x^2 - x - 1) - \underline{(x^3 + 1)} * (x - 2) = 2 \tag{3.8}$$

Now, we need to write the equation (3.4) into (3.10).

$$(\underline{(x^4 + x^3 + x^2 + 1)} - \underline{(x^3 + 1)} * (x + 1)) * (x^2 - x - 1) - \underline{(x^3 + 1)} * (x - 2) = 2 \tag{3.9}$$

$$\underline{(x^4 + x^3 + x^2 + 1)} * (x^2 - x - 1) + \underline{(x^3 + 1)} * (-x^3 + x + 3) = 2 \tag{3.10}$$

The last step is to make it compatible with the defined Galois Field.

$$\underline{(x^4+x^3+x^2+1)} * (x^2-x-1) + \underline{(x^3+1)} * (-x^3+x+3) = 2 \bmod 7 \quad (3.11)$$

We will multiply the two sides of the equation by 4.

$$\underline{(x^4+x^3+x^2+1)} * (4*x^2-4*x-4) + \underline{(x^3+1)} * (-4*x^3+4*x+12) = 8 \bmod 7 \quad (3.12)$$

With taking the modulo.

$$\underline{(x^4+x^3+x^2+1)} * (4*x^2+3*x+3) + \underline{(x^3+1)} * (3*x^3+4*x+5) = 1 \bmod 7 \quad (3.13)$$

The inverse of the $(x^4+x^3+x^2+1)$ is equal to $(4*x^2+3*x+3)$ under $(3*x^3+4*x+5)$ on $GF(7)$.

3.5 Key Generation

Like all public key algorithms, there are two sets of keys, public and private. To generate these keys, two parties have to agree on the parameters. These parameters are named (N,p,q) and can be shared on an unsecured channel or used as fixed numbers for the implementations. After parameters are selected, we need to generate two polynomials called (f,g) as our key generators. The key generation steps are listed below.

- We need to randomly select f polynomial. The degree of the polynomial should be equal to or smaller than N-1, and its coefficients should be one of these integers (-1,0,1). Also, the selected f polynomial should have inverses under both p and q. We will name the inverse polynomials as f_p, f_q. The inverses can be formulated as

  $$f_p * f = 1 (mod\ p)$$
  $$f_q * f = 1 (mod\ q)$$

  If the inverses do not exist, we need to generate a new f polynomial and calculate the inverses again.

  As mentioned above, we will be using the $NTRU(N = 17, p = 3, q = 64)$ parameters to generate our public and private keys. We started by randomly selecting the f and g functions.

  $$f(x) = -x^{16} - x^{15} + x^{12} + x^{11} + x^9 + x^8 - x^5 - x^3 + x^2$$

  $$g(x) = x^{15}+x^{14}-x^{13}-x^{12}-x^{11}+x^{10}-x^9+x^8-x^7-x^6+x^5+x^4-x^3-x^2+x+1$$

  Before we continue with inverse function calculations, we might check if the function's coefficients are balanced. $f(x)$ has 5 ones and 4 negative ones, $g(x)$ has 8 ones and 8 negative ones. To calculate the inverse functions, we can use the Extended Euclidean Algorithm. The numeric calculations can be carried out in subsection Extended Euclidean Algorithm.

$$f'_p(x) = -x^{15} - x^{14} - x^{13} - x^{12} - x^{11} - x^{10} - x^8 - x^7 - x^5 + x$$

$$\begin{aligned} f'_q(x) = &-28 * x^{16} - 23 * x^{15} + 22 * x^{14} - 14 * x^{13} + 16 * x^{12} \\ &- 23 * x^{11} + 8 * x^{10} + 32 * x^9 - 26 * x^8 - 27 * x^7 - 29 * x^6 - 22 * x^5 \\ &+ 5 * x^4 + 11 * x^3 - 18 * x^2 - 25 * x + 14 \end{aligned}$$

We can calculate the preliminary $h(x)$ with multiplying $p, f'_q(x), g(x)$

$$\begin{aligned} h_{pre}(x) = &-20 * x^{31} - 25 * x^{30} + 17 * x^{29} - 15 * x^{28} + 29 * x^{27} \\ &+ 4 * x^{26} + 26 * x^{25} - 22 * x^{24} - 14 * x^{23} - 29 * x^{22} - 2 * x^{21} + 2 * x^{20} \\ &+ 17 * x^{19} + 10 * x^{18} - 12 * x^{17} - x^{16} - 31 * x^{15} + 8 * x^{14} + 20 * x^{13} \\ &+ 6 * x^{12} - 18 * x^{11} + 13 * x^{10} - 25 * x^9 + 21 * x^8 + 23 * x^7 + 12 * x^6 \\ &+ x^5 + 27 * x^4 + 12 * x^3 + 21 * x^2 + 31 * x - 22 \end{aligned}$$

The last step to finish the key generation is taking the modulo of $h_{pre}(x)$ under $x^{17} - 1$.

$$\begin{aligned} h(x) = &-x^{16} - 31 * x^{15} - 12 * x^{14} - 5 * x^{13} + 23 * x^{12} + 31 * x^{11} - 22 * x^{10} - 21 * x^9 \\ &- 17 * x^8 + x^7 - 2 * x^6 - 28 * x^5 + 25 * x^4 + 14 * x^3 - 26 * x^2 - 23 * x + 30 \end{aligned}$$

- After f_q is calculated, we can generate our public key and finish the key generation process. $h = p * f_q * g$
- Private keys : f, f_p, g
  Public key: h

3.6 Encryption

NTRU algorithm works on the polynomials. Therefore, our secret message has to be converted into a polynomial, and our coefficient rule above is still effective. With these constraints, it is convenient to take the binary value of the message as the polynomial. Also, on the encryption side, we need to generate the random polynomial r. This random polynomial prevents attackers from finding the secret message. We have to change the random polynomial at every encryption.

$$e = (((r * h + m)\ mod (x^N - 1))\ (mod\ q))$$

Anybody, who knows the r, can calculate the secret message with $m = e - r * h$. Therefore, the r polynomial should be generated randomly, and after usage, it should be destroyed.

In this example, we will encrypt the data "SE". Firstly, the ASCII message should be converted to binary. $"SE" = 16'b0101001101000101$. After the data is digitalized, we need to generate the message polynomial. Also, we need to use the

$R = x^N - 1 = x^{17} - 1$ and a random generated polynomial.

$$m = x^{15} + x^{13} + x^9 + x^7 + x^6 + x^3 + x$$

$$r = x^{15} + x^{14} - x^{13} - x^{12} - x^{11} + x^{10} - x^9 + x^8 - x^7 - x^6 + x^5 + x^4 - x^3 - x^2 + x + 1$$

We will use the $h(x)$ polynomial generated in subsection Key Generation.

$$h(x) = -x^{16} - 31 * x^{15} - 12 * x^{14} - 5 * x^{13} + 23 * x^{12} + 31 * x^{11} - 22 * x^{10} - 21 * x^9 - 17 * x^8 + x^7 - 2 * x^6 - 28 * x^5 + 25 * x^4 + 14 * x^3 - 26 * x^2 - 23 * x + 30$$

$$r(x) * h(x) = -x^{31} + 32 * x^{30} + 22 * x^{29} + 15 * x^{28} - 2 * x^{27} - 27 * x^{26} - 27 * x^{25} - 10 * x^{24} - 29 * x^{23} - 20 * x^{22} - 24 * x^{21} + 26 * x^{20} - 13 * x^{19} + 11 * x^{18} - 23 * x^{17} + 2 * x^{16} + 19 * x^{15} + 18 * x^{14} + 16 * x^{13} + 25 * x^{12} - 28 * x^{11} - 5 * x^{10} - 17 * x^9 + 4 * x^8 - 17 * x^7 - 20 * x^6 + 16 * x^5 - 10 * x^4 - 19 * x^3 - 15 * x^2 + 7 * x + 30$$

We can add the message polynomial.

$$r(x) * h(x) + m(x) = -x^{31} + 32 * x^{30} + 22 * x^{29} + 15 * x^{28} - 2 * x^{27} - 27 * x^{26} - 27 * x^{25} - 10 * x^{24} - 29 * x^{23} - 20 * x^{22} - 24 * x^{21} + 26 * x^{20} - 13 * x^{19} + 11 * x^{18} - 23 * x^{17} + 2 * x^{16} + 20 * x^{15} + 18 * x^{14} + 17 * x^{13} + 25 * x^{12} - 28 * x^{11} - 5 * x^{10} - 16 * x^9 + 4 * x^8 - 16 * x^7 - 19 * x^6 + 16 * x^5 - 10 * x^4 - 18 * x^3 - 15 * x^2 + 8 * x + 30$$

We need to take the mod under $R(x)$.

$$(r(x) * h(x) + m(x)) \bmod R(x) = e = 2 * x^{16} + 20 * x^{15} + 17 * x^{14} - 15 * x^{13} - 17 * x^{12} - 13 * x^{11} - 7 * x^{10} + 21 * x^9 - 23 * x^8 - 26 * x^7 + 16 * x^6 - 4 * x^5 + 30 * x^4 + 8 * x^3 - 28 * x^2 + 19 * x + 7$$

We can convert the encrypted polynomial to an array.

$$e = [7, 19, -28, 8, 30, -4, 16, -26, -23, 21, -7, -13, -17, -15, 17, 20, 2]$$

Finally, we can pack the array's elements into binary format. Every element can be represented with 6 bits. Because $q = 64 = 2^6$.

$$e = 102'b000111\_010011\_100100\_001000\_011110\_$$
$$111100\_010000\_100110\_101001\_010101\_111001\_$$
$$110011\_101111\_110001\_010001\_010100\_000010$$
$$e = 102'h74E421EF109A9579CEFC51502$$

To demonstrate the dangers of using the same random polynomial twice, we will provide an example. If we wanted to encrypt the message "SM". Converting ASCII to binary we get $m2 = "SM" = 16'b0101001101001101$. After encrypting this data,

we found the e polynomial coefficients as

$$e2 = [7, 19, -28, 8, 30, -4, 16, -26, -23, 21, -7, -13, -16, -15, 17, 20, 2]$$

$$e2 = 102'h74E421EF109A9579CF0C51502$$

To analyze the results, we take $e \oplus e2$.

$$102'h74E421EF109A9579CEFC51502 \oplus 102'h74E421EF109A9579CF0C51502$$

$$= 1F000000$$

If we convert the binary information to the polynomials, we can see only one coefficient is changed. Every coefficient is represented with 6 bits in the encrypted message. The change starts at the 25th bit and keeps 5 bits. From this, an adversary can deduce only the fourth bit is changed in the message.

3.7 Decryption

To decrypt the encrypted message, we need to spend more computational power compared to encrypting the secret message.

$$a = (((e * f)\ mod(x^N - 1))\ (mod\ q))$$

$$b = a(mod\ p)$$

$$m = (((b * f_p)\ mod(x^N - 1))\ (mod\ p))$$

We will decrypt the message that encrypted the previous part. Also the $f(x)$ and $f_p^{'}$ polynomials will be used.

$$e = 2 * x^{16} + 20 * x^{15} + 17 * x^{14} - 15 * x^{13} - 17 * x^{12} - 13 * x^{11} - 7 * x^{10} + 21 * x^9 - 23 * x^8 - 26 * x^7 + 16 * x^6 - 4 * x^5 + 30 * x^4 + 8 * x^3 - 28 * x^2 + 19 * x + 7$$

$$f = -x^{16} - x^{15} + x^{12} + x^{11} + x^9 + x^8 - x^5 - x^3 + x^2$$

$$f_p^{'} = -x^{15} - x^{14} - x^{13} - x^{12} - x^{11} - x^{10} - x^8 - x^7 - x^5 + x$$

The process starts with multiplying e and f polynomials.

$$e(x) * f(x) = -2 * x^{32} - 22 * x^{31} - 37 * x^{30} - 2 * x^{29} + 34 * x^{28} + 52 * x^{27} + 57 * x^{26} - 10 * x^{25} - 8 * x^{24} + 56 * x^{23} - 8 * x^{22} - 32 * x^{21} - 78 * x^{20} - 126 * x^{19} + 21 * x^{18} + 39 * x^{17} - 4 * x^{16} + 30 * x^{15} - 33 * x^{14} + 34 * x^{13} + 62 * x^{12} + 15 * x^{11} - 2 * x^{10} - 46 * x^9 + 19 * x^8 - 6 * x^7 + 3 * x^6 + 29 * x^5 - 47 * x^4 + 12 * x^3 + 7 * x^2$$

$$a_{pre} = e(x) * f(x) \bmod R(x) = -4 * x^{16} + 28 * x^{15} - 55 * x^{14} - 3 * x^{13} + 60 * x^{12} + 49 * x^{11} + 50 * x^{10} + 11 * x^9 + 9 * x^8 - 14 * x^7 + 59 * x^6 + 21 * x^5 - 79 * x^4 - 66 * x^3 - 119 * x^2 + 21 * x + 39$$

$$a = a_p re \bmod q = -4 * x^{16} + 28 * x^{15} + 9 * x^{14} - 3 * x^{13} - 4 * x^{12} - 15 * x^{11} - 14 * x^{10} + 11 * x^9 + 9 * x^8 - 14 * x^7 - 5 * x^6 + 21 * x^5 - 15 * x^4 - 2 * x^3 + 9 * x^2 + 21 * x - 25$$

$$b = a \bmod p = -x^{16} + x^{15} - x^{12} + x^{10} - x^9 + x^7 + x^6 + x^3 - 1$$

$$b * f_p^{'} = x^{31} + x^{27} + x^{26} - x^{25} + 2 * x^{24} + x^{23} - x^{22} - x^{21} - 2 * x^{20} - 4 * x^{18} - 3 * x^{17} - 2 * x^{15} - x^{14} - 2 * x^{13} - x^{10} + x^8 + 2 * x^7 + x^5 + x^4 - x$$

$$m_{pre} = -2 * x^{15} - 2 * x^{13} + x^9 + 4 * x^7 + x^6 - 2 * x^3 - 5^x - 3$$

$$m = m_{pre} \bmod p = x^{15} + x^{13} + x^9 + x^7 + x^6 + x^3 + x$$

Decryption computations are completed. After that, we need to convert this data to binary form.

$$m = 01010011\_01000101$$

Converting binary data to ASCII, we can recover the secret text.

$$m = "SE"$$

CHAPTER 4

FIELD-PROGRAMMABLE GATE ARRAY

FPGA is a reconfigurable integrated circuit. The user can design the inner structure of the IC using FPGAs. To do that, the user has to use one of the Hardware Description Language (HDL) like Verilog or VHDL. FPGAs behave like Application-Specific Integrated Circuits (ASIC). However, it can be updated or completely redesigned, unlike ASICs. An FPGA contains look-up tables (LUT), flip-flops (FF), block memory (BRAM), and interconnections. With these primitive blocks, FPGAs can mimic logic gates like "Not", "And", "Or" and create complex logic functions. Also, some FPGAs can include hard coded multicore ARM processors, Digital Signal Processing (DSP) blocks, and specific high-speed digital communication lines like JESD or artificial intelligence cores. You can see an FPGA model in Figure 4.1.

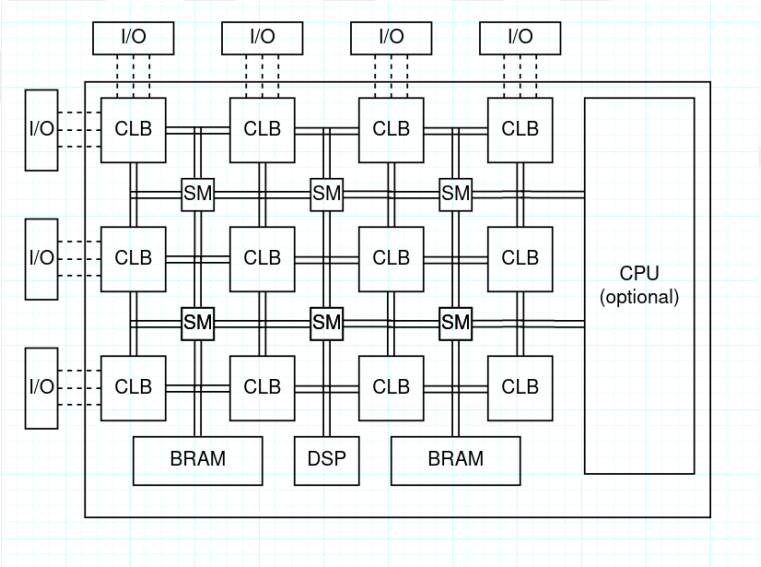


Figure 4.1: FPGA model representation

The CLBs are the core component of the desired logical functions. They are the decision units. They can change their outputs according to the inputs. Also, they have the ability to store the previously generated output results by using the internal flip-flop (FF). A basic model of CLB is provided in Figure 4.2.

Our model CLB has a LUT4 primitive; with that block, we can hold 16 scenarios and generate two possible outputs. With the FF block, we can store this result under the

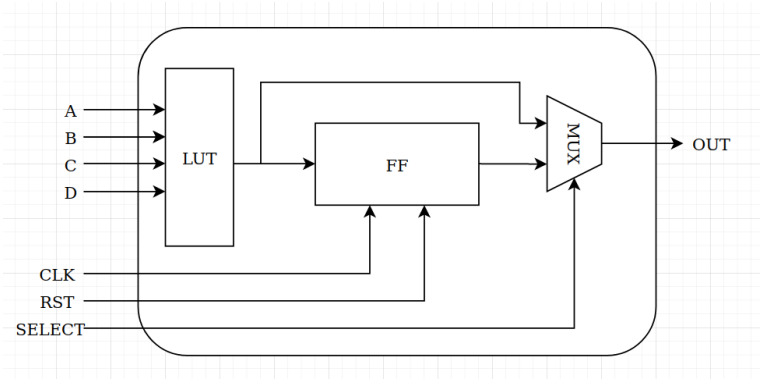


Figure 4.2: Simple model of CLB

control of the other logical blocks. It can save a new value whenever it requires to do that. The FF block is made from SR-latch. By connecting the reset pin to the set pin through a "NOT" gate, we can make a Data Flip Flop (DFF). In DFFs, the signal applied to the D pin can be saved when Enable signal is provided. We can model it with a 5 NAND gate like in Figure 4.3. Lastly, the CLB block has a multiplexer (MUX). The multiplexer in the circuit can be used to select which data we want to see in the output. It can output the stored data or directly bypass the logical function's result.

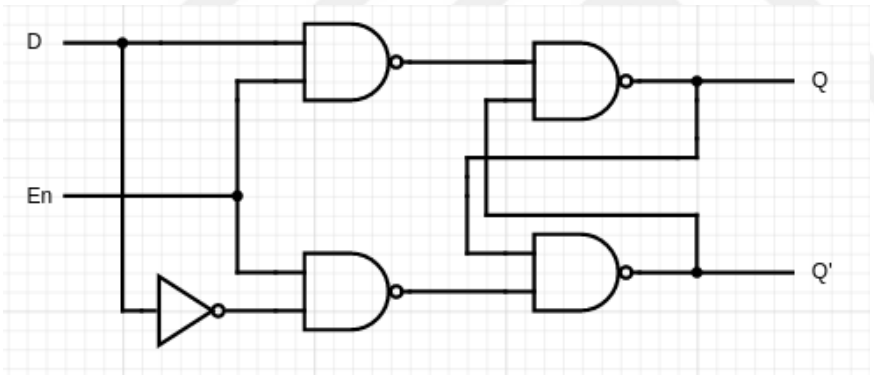


Figure 4.3: Inside of the Data Flip Flop

To generate complex and meaningful logical functions, we need to connect many CLBs to each other. To do that, we need to carry the output signals in the FPGA fabric. This job can be accomplished by the Switch Matrix elements. They can generate physical lines between CLBs. The length of these lines is generally the limiting factor of the maximum clock speed that the design can operate.

The IO ports of the FPGA devices are more complicated than the microprocessors'. They support single-ended or differential signals, can change the drive strength and slew rate, can activate termination resistors, and are able to support different standards like LVDS, HSTL, SSTL, and HSUL. You can see a basic representation of an FPGA IO port in Figure 4.4 without the differential signaling support. The pull-up and

pull-down controller allows us to change the input characteristics of the pin. This characteristic determines the value of the input while a driving signal does not exist. The enable signal lets us make the buffer's output floating (high impedance) while the pin is in input mode. Also, from that buffer, we can adjust the output parameters like driving strength and slew rate.

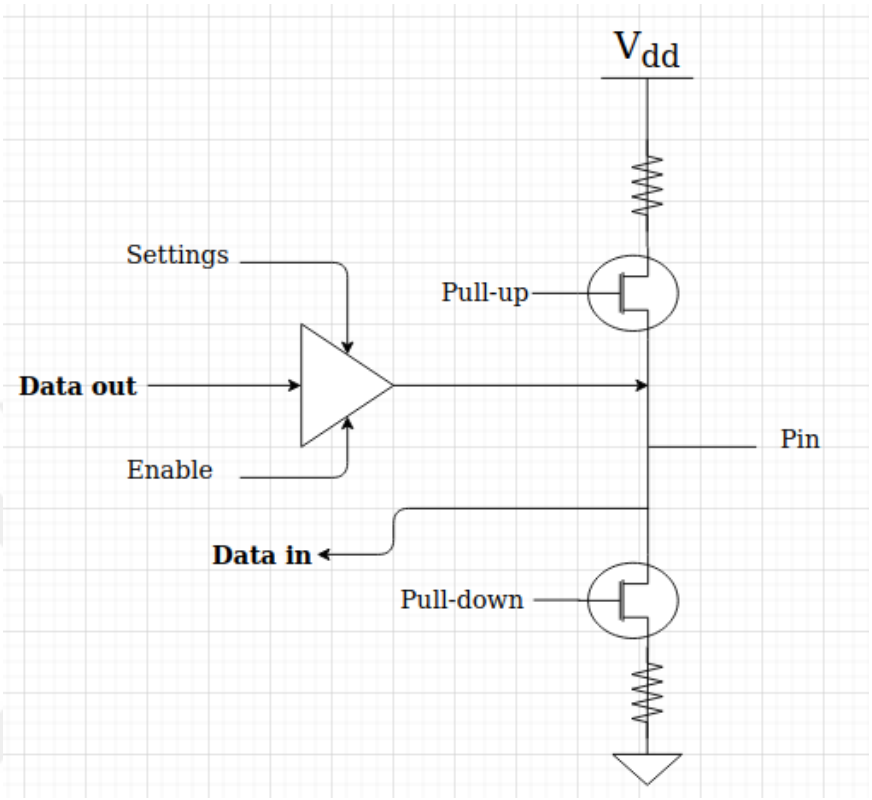


Figure 4.4: Basic electronic design of the FPGA IO port

Combining the CLBs, switch matrices, and IO ports, we can create every logical function we need. However, to improve the functionality, we need to add BRAM (Block RAM), DSP, and PLL (Phase Locked Loop) blocks. Also, the new trend is to include internal RISC-based CPUs. The BRAM blocks increase the data storage capabilities of our design without needing to keep the necessary data in the FFs. DSP blocks contain specialized functions like multiplication and accumulation. These blocks are hard coded to the FPGA fabric to increase the performance. The last mention-worthy block is the PLLs. By using PLL blocks, we can change the frequencies of the clock signals.

As mentioned before, FPGA architecture is defined by the user and does not dictate number formats like processors. In the provided examples, "N" represents the bit number of the system, and "n" represents the order of the specific bit starting from 0. The most used data formats are listed below:

- Unsigned: Without any special rule, every nth bit represents the 2^n component of the number.

- Signed Magnitude: Despite the unsigned numbers, the most significant bit represents the sign of the number. For example, $4'b0111$ is equal to 7 and $4'b1111$ is equal to -7.

- Two's Compliment: In this mode, the most significant bit has a value of -2^{N-1}, and other bits have their normal value of 2^n. If we use the previous examples, $4'b0111$ will be equal to 7. However, $4'b1111$ will be equal to -1. In 2's complement system we can represent the numbers $-(2^{N-1}), -(2^{N-1} - 1), ..., -1, 0, 1, ...,$
  $(2^{N-1}-2), (2^{N-1}-1)$. For a four-bit number system, we can represent numbers from -8 to 7.

- One's Compliment: Like the two's complement system, the most significant bit has a special value; in 1s complement, this value is $2^{N-1} - 1$. This will generate the representable numbers $-(2^{N-1}-1), -(2^{N-1}-2), ..., -1, -0, 0, 1, ..., (2^{N-1}-2), (2^{N-1} - 1)$.

  The most important feature of this format is the negative zero. In 2's complement mode, we have uneven rounding errors around 0. Because every positive number that is smaller than 1 will be rounded to 0, every negative number that is bigger than -1 will be rounded to -1. However, by using 1's complement method, we can round numbers bigger than -1 to -0. This will create symmetry in rounding with negative and positive numbers.

The steps to deploy a code on an FPGA can be different from vendor to vendor. However, we can inspect the steps from one vendor, for example, Xilinx, and develop a general understanding of the process. Xilinx generates FPGA binary files in the following four steps.

- Synthesis: In this step, the HDL code is converted to the logical functions, and the netlist code is generated. Even if the structural block is changing between different FPGA classes, this process is independent for all of them.

- opt_design: This is the first FPGA-dependent step. The netlist is optimized to the characteristics of the FPGA. From the netlist, the logical functions are converted to the FPGA's structural elements (LUT2, LUT3, LUT4, LUT5, and LUT6 primitives).

- place_design: In this step, the decided primitives are selected from the fabric of the chip. The more successful results in place design step can make the routing easier.

- route_design: Because of the physical limitations of the three-dimensional real world, we cannot connect all the logic blocks in the perfect way with zero distances to the other blocks. Therefore, we need to find an optimum solution for the connection paths. In this step, the implementation tool decides which block connection can handle how much time delay and try to generate a data path with switch matrices that satisfies these requirements. This step generally takes most of the time in the implementation step.

- write_bitstream: In this step, the design is converted to the FPGA programming files. These files can be written to the SD Memory Card or EEPROM flash, or they can be directly used with a JTAG connection.

There are three different technologies for FPGA logical structures. The most commonly used one is SRAM. In SRAM-based FPGA, bit data is stored with transistors. These kinds of FPGAs are easy to produce and can gain advantages from the developments in the CPU and GPU design. Because they use the same architecture. However, they are more vulnerable to outside disturbers like power fluctuations or radiation. Therefore, even if they are perfect for general purpose use cases, it needs extra protective precaution for a space mission. Their most important advantage is this class of FPGA can be programmed infinitely many times.

The second type of FPGAs is Antifuse FPGAs. A normal fuse is a circuit protector. Suppose we draw a high current from a fuse. It will break, and the circuit becomes open. However, the process is reversed in anti-fuse designs. If we apply high voltage to an anti-fuse, the nonconducting part of the circuits are welded together, and the circuit becomes closed. Activating the anti-fuses is done at one time while programming the FPGA. The problem is we only program this class of FPGAs one time. However, their resistance to power problems and radiation makes them more suitable for aerospace industries. Also, their routing performance is generally better because of the fixed circuits. They have lower power requirements. Thus, they produce less heat. An example of an anti-fuse is provided in Figure 4.5.

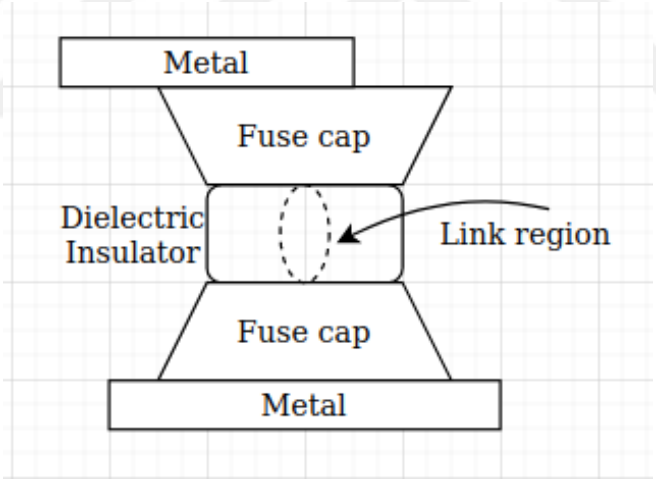


Figure 4.5: Anti fuse example

Every switch is open at the beginning of programming because of the dielectric insulator. With applying a high voltage while programming, the insulator deforms, and a silicon-based link forms between the fuse caps. Because forming conductive links takes time and is done one by one, programming an anti-fuse FPGA is a slow process compared to others.

The last type of FPGAs is Flash-based ones. These type of FPGAs can be programmed several times, similar to the SRAM-based ones, and has fixed circuit prop-

erties like anti-fuse-based ones. They are the new players on the field.

The parallelism capability and highly configurable architecture come with developmental difficulties. In microprocessors, there is only a couple of registers and some basic interrupts. Their architecture is generally RISC-based and does not include highly complex runtime optimization tools. The computation blocks are not specialized. Because of this, we need to combine the basic operations to get the required calculations. They are mostly used in embedded devices. The microprocessors are generally required to communicate with other chips, read data from measurement tools, or drive actuators. To do these jobs, they have many communication ports like UART, SPI, or I2C. Generally, microprocessors carry their own RAM and ROM blocks on them to increase their usability.

The CPU is the workhorse of computational devices and can be seen as an art piece. It is so flexible that it can process every computational work of the user. Generally, they read a single instruction and execute them on single data. The architecture is named SISD. They lack parallelism to the rest of the list. However, they generally have faster clocks because of the optimization efforts put into them. To get the best performance, processors have powerful runtime optimization tools like adaptive branch predictors, out-of-order executors, and virtual cores. The conventional design principle of the CPU is taking steps to increase the computational power in the expanse of area usage. Therefore they have many specialized computational blocks for complex calculations. Their main architecture is CISC-based. CPUs do not have standalone applications and have to communicate with the other parts of the computers; this requires high-speed communication. To enable that, it implements PCI and SATA protocols.

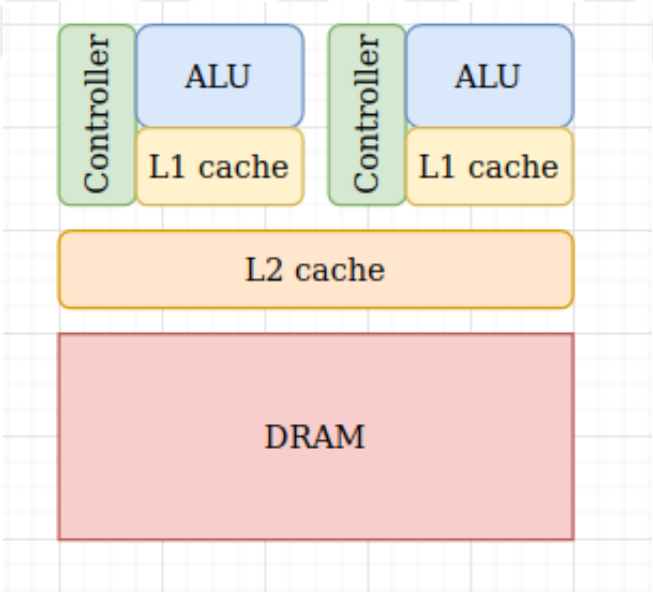


Figure 4.6: Sample CPU architecture

You can see a simplified multicore CPU design in Figure 4.6. We have a single dram block and combined L2 cache for most used or recently accessed data. A simplified core contains a controller that reads the instructions and arranges the control signals,

an L1 cache that exclusively belongs to the controller, and ALU that makes the calculations.

GPUs are developed to manage the visual data to put on the screen. Because of the timing requirements to generate video signals and the requirements of manipulating many pixels at the same time, this is not a CPU-friendly process. To handle a problem like that, we need many processing units. These units can be small and have to calculate only basic operations for that time. Generally, we need to make the same calculations on many pixels. This problem can be solved by architectures similar to the SIMD. However, in their evolution throughout time, they became much more capable; they are specialized to handle computationally intense parallel works like image manipulation, video processing, gaming, and simulations. With the current trend of popularity in Artificial Intelligence and crypto-currency mining, they gained new computational capabilities for their core units and increased numbers of these units. A simple GPU with three cores is provided in Figure 4.7. In reality, the core numbers are much higher. The main difference between GPU and CPU is the number of ALUs connecting to one controller. In GPU architecture, a single controller is connected to multiple ALUs to increase the parallelism.

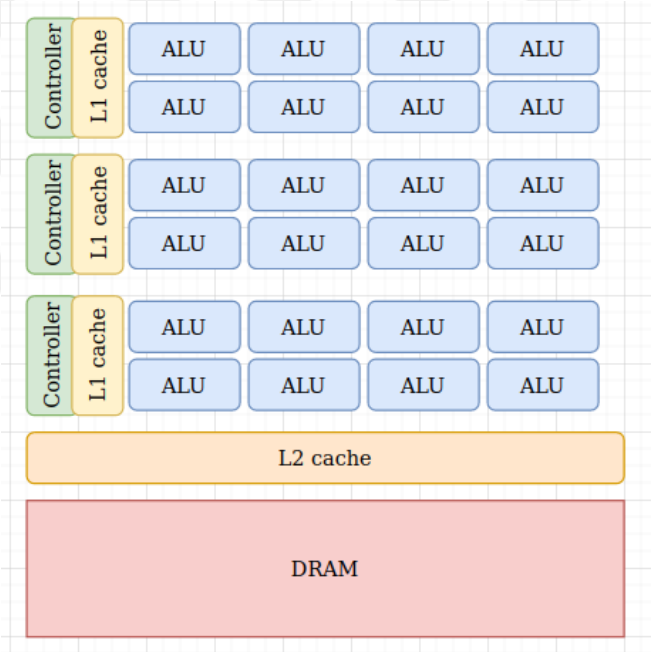


Figure 4.7: Sample GPU architecture

FPGA is like the Swiss knife of computation devices. It can simultaneously read practically infinite peripheral devices, can run huge neural networks, demodulate radio signals, encrypt data streams and write to disks, and drive other units like GPUs at the same time. Contrary to the CPUs' approach of allocating time slots for the required computations, every functionality is implemented in different parts of the fabric. The limiting factor is the number of logical elements in the FPGA. It is an empty canvas for digital circuits. Every need of the user can be implemented on this canvas.

ASICs are miniaturized circuit blocks that can include both analog and digital functionalities. These circuits are hard coded on the silicone. From the digital design perspective, it came after FPGA prototyping. FPGAs have multiple layers of fundamental blocks. However, the control of the layers' placement is in the hands of the manufacturer. While designing the chip, one can include analog components, improve the layer placements, and optimize critical parts. Most importantly, after the initial cost of development, the ASICs are highly cost-efficient for mass production.

In Figure 4.8, we provided a general rule of parallelism versus development speed. There can be outliers, like writing a program with a high-level programming language for CPU might be easier than developing a micro-processor program with assembly language.

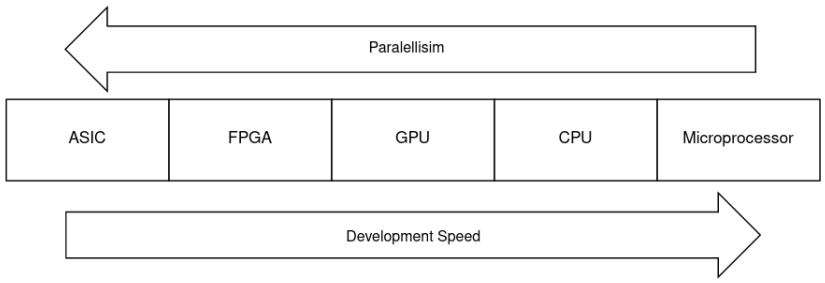


Figure 4.8: Parallelism vs. development Speed

The explained architecture styles require different development strategies and optimization techniques. We have to consider the hardware accelerators deployed with our target device, memory response time, cache size, core count, clock frequency, and inner behaviors of the system. Generally, the slowest part of a computational device is the memory because we can increase the computational power by increasing the frequency or using more computational cores. However, using a brute-force-like technique does not solve the memory response time. We need to adapt our programming approach to go around this bottleneck. In (Tezcan, 2021), they prevented the bank conflicts in shared memory accesses. In this way, they broke the speed record of AES operations on GPU, with 878.6 Gbps throughput using RTX 2070 Super. That result can beat the AES hardware core in the processors or FPGA implementations like COPACOBANA and RIVYERA.

As stated before, FPGA and ASIC-based devices can work efficiently on the calculation that requires bit manipulation. However, CPUs and GPUs have fixed word sizes that their ALUs can process. Therefore, using an algorithm designed for embedded devices might be inefficient for more complex computational devices. However, in (Tezcan, 2022), they changed the bitwise operations in the DES, 3DES, and Present algorithms to table-based operations. That way, they provided an optimization method that can be used to search 3.87 billion keys per second for DES and 1.89 billion keys for Present on an RTX 3070. These results show that the 80-bit key size is too short for an acceptable secrecy level.

Because of their user-designed hardware architectures, FPGA is immune to hardware-

based attacks like Meltdown, Spectra (Kocher et al., 2019), or Row hammer (Y. Kim et al., 2014). Also, because everything is restrained by the application-specific hardware, there is no way to use a bug to exploit the whole system. They have specific vulnerabilities like securing the code. This problem is mitigated by writing encrypted bitstream to the flash memory. This problem does not occur in the anti-fuse FPGAs because the code is converted into the fixed circuit while programming.

4.1 Side-Channel Attack

The side-channel attacks are carried out from outside the cryptographic device without gaining access to the internal mechanisms. In these attack methods, adversaries observe the cryptographic device while working and try to gather useful information about how the encryption or decryption processes keep going. The information gathered might be the time needed to encrypt the data, the power spent to make the necessary calculation, and the thermal images of the processing units. These types of attacks generally require specific measurement tools and controlled access to the interested devices. The inner mechanism of side-channel attacks and some methods to prevent them is investigated in (Standaert, 2010).

Timing Attack: In these types of attacks, the adversaries can measure the time spent to encrypt the data. By analyzing the timings, the number of calculations can be guessed. The elapsed time can be determined from another application that measures the processor usage or spying on the IO activities like ethernet port or serial terminals. After that, the adversaries can deduce the distribution of the bits. The implementation should not take conditional computational shortcuts and spend the same amount of time for every encryption or decryption process to mitigate the timing attacks. Generally, the civil systems are optimized for speed to keep up with countless requests with fewer resources. Therefore, many systems we use daily can be open to this type of attack. For example, in (Brumley & Boneh, 2005), they showed the OpenSSL is vulnerable, and they extracted private keys from a web server.

Power-Analysis Attack: Doing complex calculations requires more energy than keeping the device idle. If the adversaries have access to the power line of the cryptographic device, they can use a sensitive oscilloscope to profile the time versus energy usage. This information can be used to determine the device's state and periodic processes. With the developments in the artificial intelligence field, the data captured from the power lines can be interpreted more accurately (Lerman et al., 2014).

In Figure 4.9, you can see the power consumption while doing RSA encryption steps. The figure is adapted from https://commons.wikimedia.org/wiki/File:Power_attack.png.

Environment Monitoring: We gather the highly specific side channels and try to find ways to exploit them in this category. For a hypothetical cryptographic device, these attacks might be observing the Electromagnetic Interfaces (EMI) with an antenna and spectrum analyzer, capturing the coil whining noise from

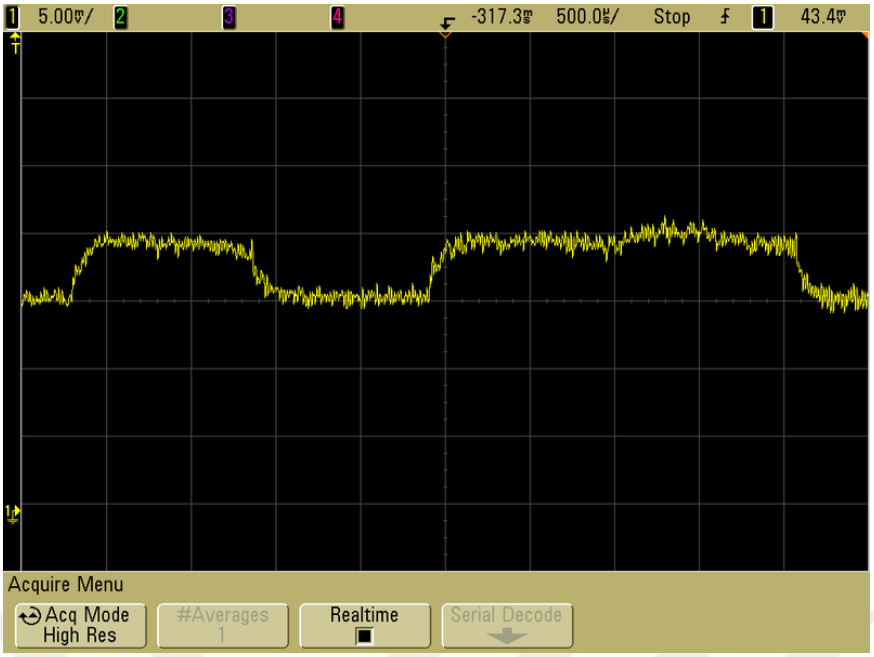


Figure 4.9: Current drawn while processing

the power supply with a microphone, and measuring the voltage surges in the whole power system. The adversaries can be highly creative for specific devices under specific environments. An out-of-topic interesting example might be the transcribing of a conversation by only using a high-speed camera that records a foil bag of potato chips in the room.
https://news.mit.edu/2014/algorithm-recovers-speech-from-vibrations-0804

CHAPTER 5

IMPLEMENTATION

We choose to implement the algorithm on FPGA because many communication devices that use Software Radio Techniques do not include a processor, specifically in military or aerospace areas. Because FPGAs are irreplaceable for SDR-based projects for the required IO bandwidth and can do every calculation a CPU does. Because of this reason, while a device is equipped with an FPGA, the designers might not add additional processors. Also, traditionally FPGAs are more resistant to harsh environments like temperature changes and radiation.

The main idea behind our implementation is to provide a secure communication channel for remote embedded devices. More specifically, we targeted small ad-hoc and mesh network capable embedded devices. From a node's perspective, the connected devices and the communication routes to the unseen devices can be changed in seconds in this type of network. A mesh network representation is provided in Figure 5.1. Also, we might need to generate clearance levels in the network. Furthermore, some of the devices can be captured by adversaries after the network is established. The symmetric key encryption algorithms are inadequate to deal with these kinds of scenarios. However, because of their efficiency in both speed and hardware requirements, they are still indispensable. We need to use public key encryption algorithms to overcome the shortness of the symmetric key encryption algorithms. We have been using the RSA and ECC algorithms to communicate on insecure channels. But due to the recent developments in quantum computers, today's standard PKE algorithms are risky to use in devices that will be supported for a long time.

After we have defined the use case of the implementation, we can determine the restrictions. First of all, the targeted devices have different main roles other than encrypting and decrypting PKE messages. These different roles might need to be performed with various kinds of hardware. In some of the used hardware and main role combinations, most of the resources might be utilized. Therefore our implementation has to be highly area efficient. Most of the other implementations make the multiplications in parallel and store the data in registers between operations. Unfortunately, this approach requires so many resources that an embedded device might be unable to spare. Therefore, we will shift the workload to the BRAM modules from logical gates at the expense of calculation time. To provide freedom in parameter selection, we do not restrict the bit sizes of the coefficients. N and q parameters of the NTRU can be changed on the fly with the help of the ram-based design.

As we are targeting to optimize the area usage of the device, we are choosing to make the calculations serially. However, parallelization and pipelining optimizations can be

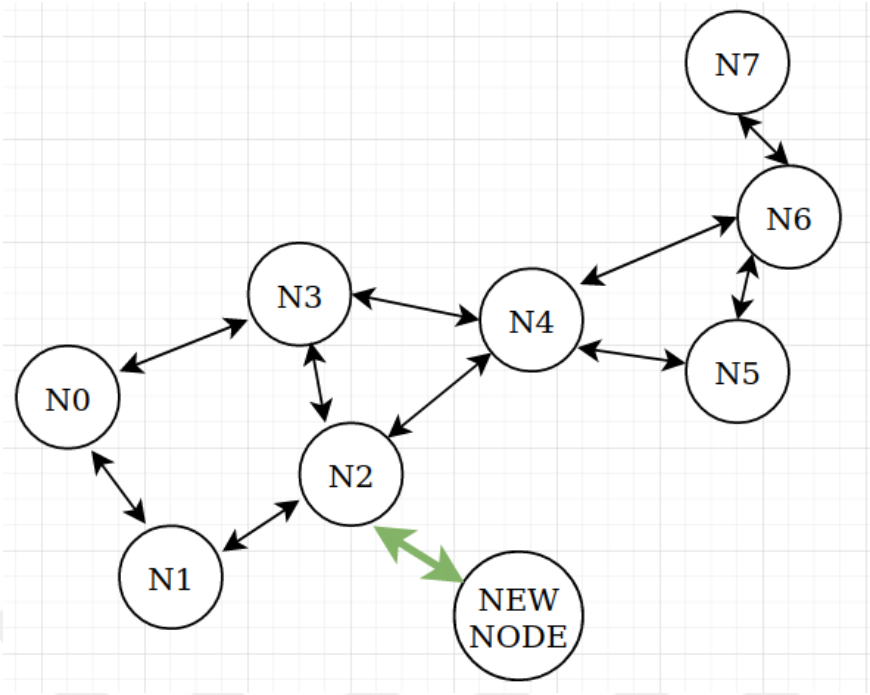


Figure 5.1: Representation of Mesh Network

implemented with little to no hardware cost. For example, the read requests from the RAMs are executed in parallel, and the new requests are issued before the previous one is even completed. Without parallelism and pipelining, when we issue a read command, we need to wait to get the response and have to gather all the necessary data before calculations.

In this design, we wanted to generate an optimum solution for embedded devices with low processing power on a network. Our optimization proposals are generated for these devices.

In our algorithm, we used the same processing architecture for both encryption and decryption. Because in the NTRU algorithm, the encryption calculations can be done with the first part of the decryption module. The implementation utilizes three block ram modules to store all the necessary information to make the cryptographic calculations. In most of the FPGA architectures, to reach data stored in the Bram takes two cycles without output registering. You can see the timetable of issued commands and spend time executing them in Figure 5.2 with a serial non-pipelined architecture. For every step of the polynomial multiplication and summation, the device spends eight cycles.

We took two optimization steps for the data storing part. Firstly, we parallelized the commands. As we are working on the FPGA, Bram interfaces are independent of each other. So, we can issue all the commands at the same time. Also, even if the BRAMs have a latency value of 1, their throughput is one too. This means we can issue new commands before the previous one is finished and retrieve the results in the issued order. You can see the parallel and pipelined read and write architecture in

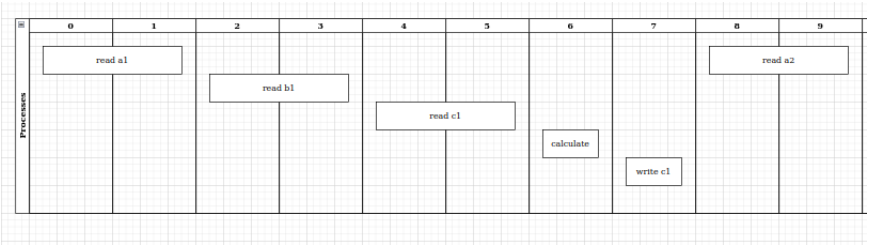


Figure 5.2: Serial and blocking operations

Figure 5.3.

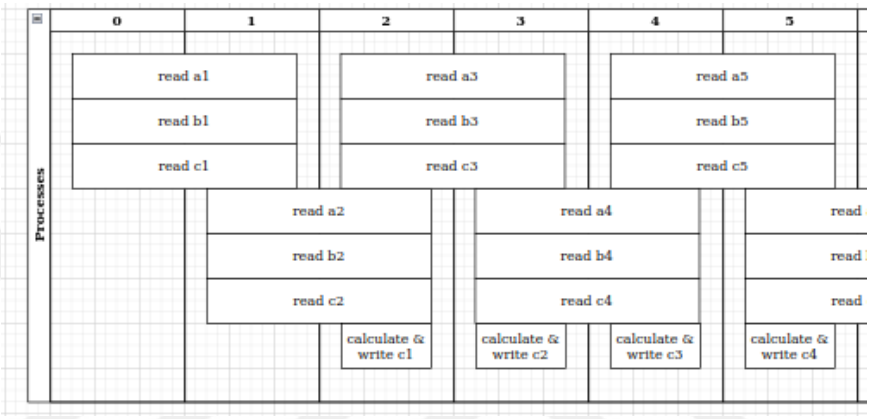


Figure 5.3: Parallel and pipelined operations

Most popular FPGA manufacturers design their BRAM modules dual ported. The first port does not have writing ability; it can only read the memory. On the contrary, the second port can execute write operations without reading what is stored in the memory. In Xilinx FPGAs, these modules are selected with "Dual port RAM". Xilinx also offers a "true dual port ram" that can write and read from both of the ports. However, this option will double the resource usage. As we do not require it, we selected to continue with "dual port ram". This feature enables us to simultaneously execute read and write operations. However, as shown in Figure 5.3, we also combined the calculation and write steps. This is done by only using combinational logic without registering the data before writing it to the RAM. While polynomial multiplexing requires two values, we do not need to read two of them from memory at every cycle. One of the variables can be stored in a register and used repetitively.

5.1 Memory Scheme

We designed our memory scheme to maximize memory utilization without decreasing the flexibility of our design. We wanted to change the NTRU parameters on the fly. Therefore, the implementation should handle the changing numbers of coefficients

and numbers of bits in them.

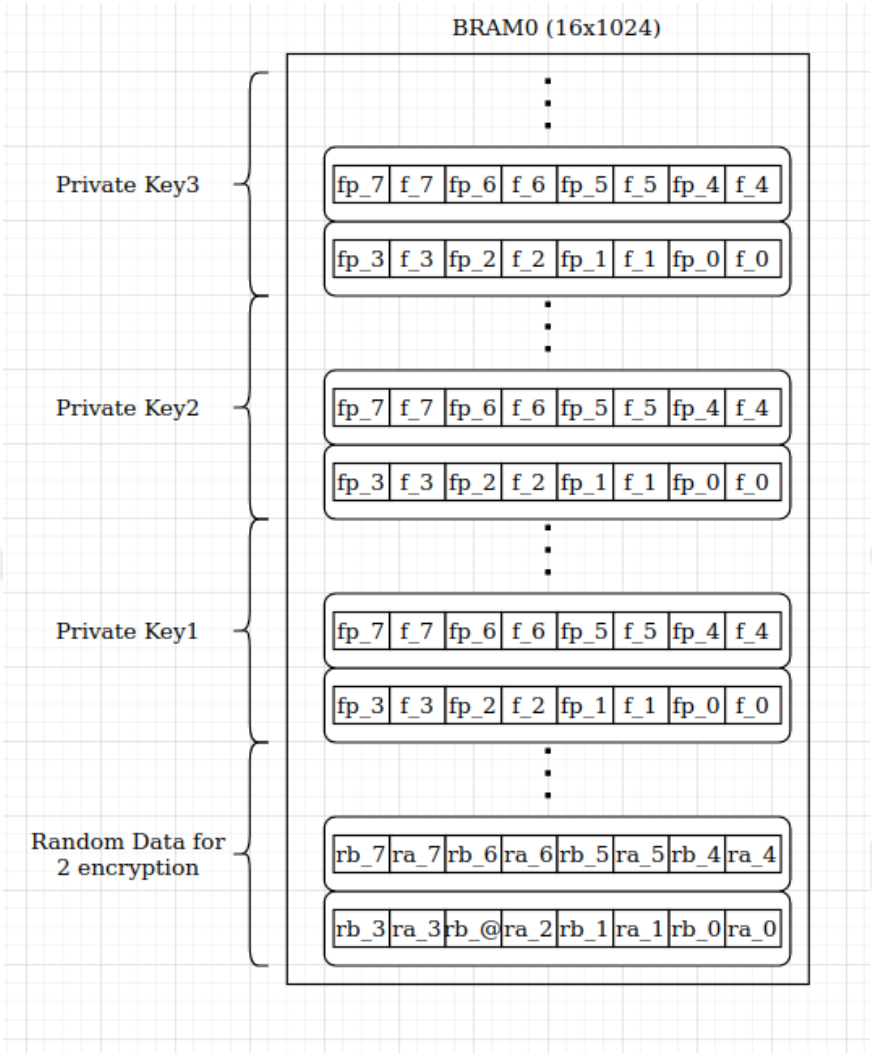


Figure 5.4: Data plan in the first RAM

In both encryption and decryption, we need three memories to store the required data, inter-products, and the results. The first RAM is designed to be less volatile than the other ones. It holds the random numbers for the encryption and the private keys for the decryption. Because we predetermined the q value as 3, everything we store in this memory block will be 2 bits long. In Xilinx FPGAs, the main memory has 18 bits width and 1024 address depth. Therefore we can combine multiple data into a single ram word. We utilized 16 bits of the memory width. The proposed method to store data can be seen in Figure 5.4.

The lower quarter of the ram is allocated for the random data. For deterministic systems, random number generation is a challenging process because of the required entropy. However, our target devices have radio communication units and can constantly read noise from their ADCs. The noise readings can be fed into hashing modules or feedback systems to ensure randomness. Even if we work on a real-time

system, true random generating can still be slow. Therefore, having a complete set of random data makes us ready to start the encryption immediately. Also, using random data that was generated an unknown time ago can increase the resistance against side-channel attacks.

Our 2-bit random variables are stored in 4 bits of words to use the same logic with the decryption. Therefore, we can store one additional random data set in the same memory region. If the random number generation is slow, we can do consecutive encryptions with that method. However, we need to select which data set we want to use when we read the word. To do that, we can deploy a basic 2-bit multiplexer.

The other three quarters are reserved for the private keys. We have two polynomials as our private keys. We can combine the f and f_p coefficient into one 4-bit word as we can store four combined words in a single memory location. As we can reach 4 of them, we will use the 4-step multiplier and adder in Figure 5.7. We will select the f or f_p polynomial by using a multiplexer according to the steps of decryption.

In a centralized network, we do not need to use different keys to communicate with different nodes. However, we assume our target devices are on a mesh network. In these types of systems, we might not be in charge of all the nodes and cannot control which device supports which parameters. Therefore, we are storing multiple private keys in the memory.

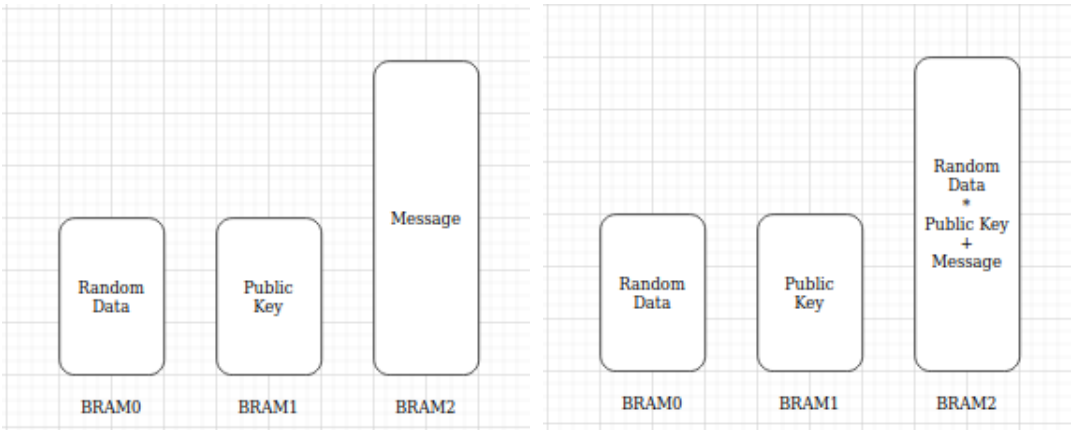
The second memory stores the public key in encryption calculations. We assumed we would not send multiple messages to the same node one after another. Therefore, we will not store the public keys of other nodes in the memory. It might be provided at the beginning of the communication or can be stored in flash memory. While decrypting, this memory starts with the encrypted message.

While encrypting, the third memory starts with the secret message and directly forms the encrypted message. In decryption, we do not need to load any data to this memory before starting.

The last 2 bits of the first RAMs address are used to select which region we want to reach. We assumed the maximum of the N parameter would be 1024 in this implementation, considering the suggested parameters. However, it can be easily increased in the future.

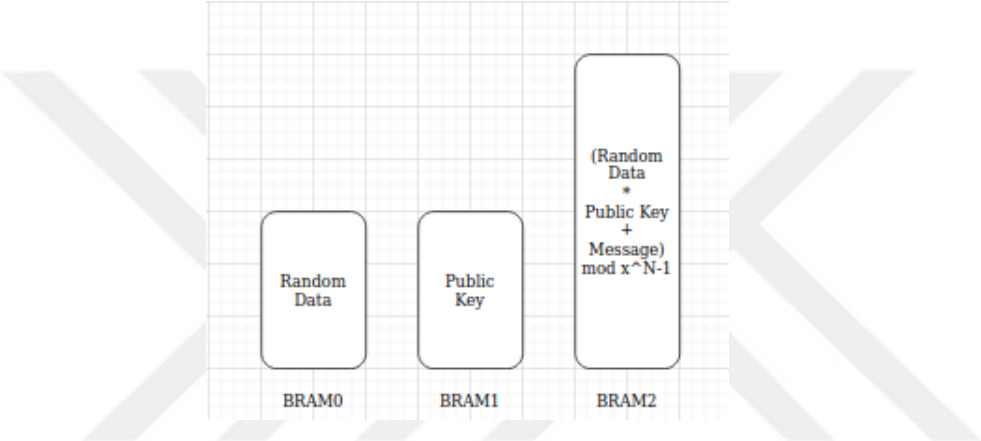
5.2 Standalone Encryption

Firstly, we implemented the encryption and decryption parts independently. Also, this version does not compress the data and stores all the coefficients in different memory locations. We provided the rams' contents through the encryption in Figure 5.5.



(a) Start of the encryption

(b) After the polynomial multiplication



(c) Finalization

Figure 5.5: Contents of the memory while encrypting

To start the encryption, we assumed that we had enough random data, received the public key, and generated the secret message. We start with setting the memory addresses for all three of them. The first one and the second one will be set to N. However, the third memory will be set to $2 * N$ as it is the sum of two addresses because the polynomial multiplication will increase the order of the polynomials.

When we set the address registers, the BRAM module starts the read operation. In FPGA architecture, this read operation takes two cycles. However, we can issue new read orders before the first one ends. Therefore we will decrease the read address of the second and third memories.

Two clocks after the start, the data can be read in the output port of the memories. We can start the polynomial multiplication by using these coefficients. In our architecture, we do not need to read the first polynomial's coefficients at every turn. So we can cache this value and use it directly. We will multiply the outputs of the first and second RAMs and add them to the output of the third RAM. We will write the

calculated value to the third RAM. The write address should be read address minus 2 by considering the latency of the memory modules.

We are not changing the read address of the first memory. However, the other addresses will be decreased by one at every clock until we reach the 0 in the second and third read addresses. After that, we will continue calculations for two more cycles because of the latency.

Now, we will decrease the first memory's read address by one. Set the second one to N and third one to $address1 + address2$. By repeating the previous and this step up until the first memory read address extends 0.

After we calculated the polynomial multiplexing, we can continue with calculating $mod(x^N - 1)$. To do that, we need to read the content of the $x + N$ and x addresses in the third memory. After that, we will calculate the $bram3[x] + bram3[x + N]$ and write back to the third memory's x location. In this process, x starts from $N - 1$ and goes to 0.

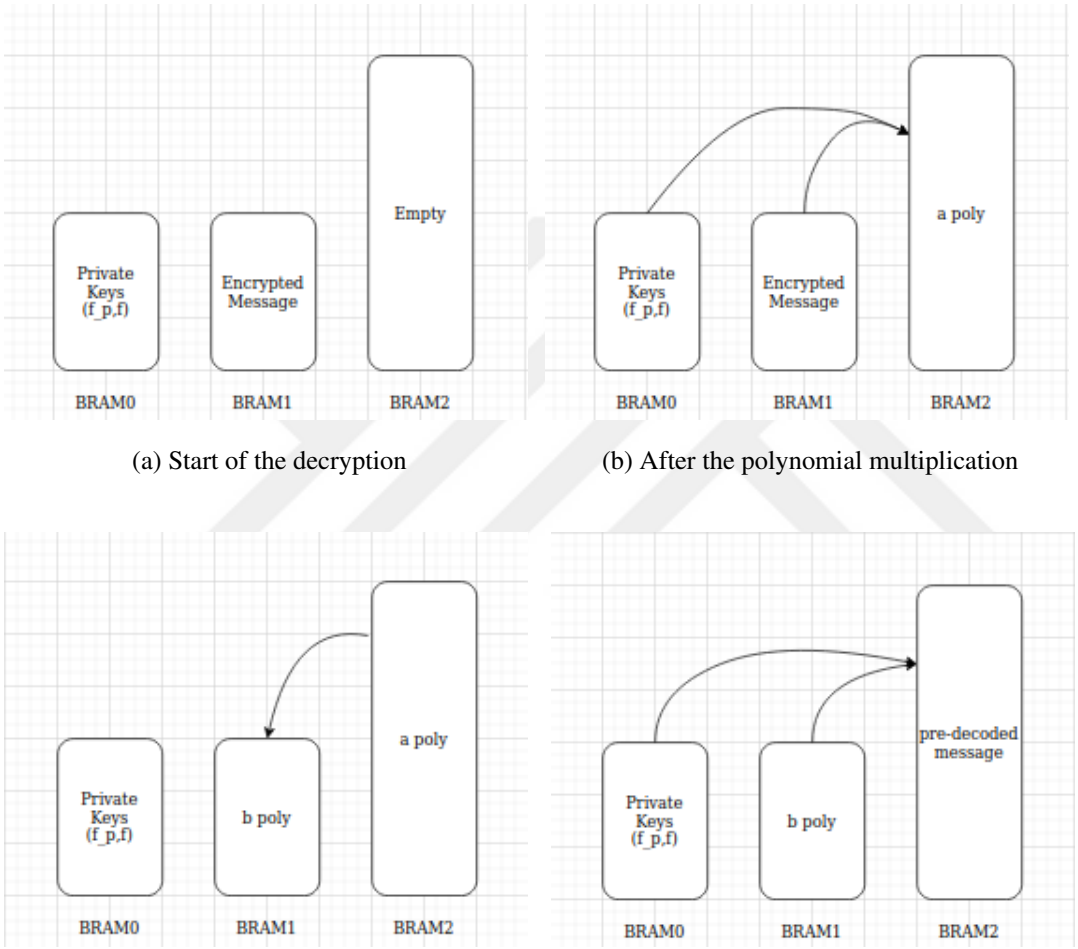
The last step should be taking $mod\ q$ of the results. This step is done while reading the data from memory with the help of the "mode_by_q" module. The decryption method is provided in the Algorithm 2. This module spends 344 LUT, 115 FF, and 4 BRAM (18K).

Algorithm 2: NTRU encryption algorithm on FPGA

Initialization:
$i = N; j = N$ cache $bram0[i]$ into var_a
while $i >= 0$ **do**
  while $j >= 0$ **do**
    read bram1[j] ;read bram2[i+j]
    write $var_a * bram1\_output + bram2\_output$ into $bram2[i + j - 2]$
    $j = j - 1$
  end
  $i = i - 1; j = N$
  cache $bram0[i]$ into var_a
end
$j = N - 1$
while $j >= 0$ **do**
  temp= bram2[j+N]
  read bram2[j]
  load $bram2\_output + temp$ into $bram2[j - 2]$
  $j = j - 1$
end
We will get the results through the mod_by_q module

5.3 Standalone Decryption

The beginning of the decryption is the same as the encryption processes. We used the same polynomial multiplication algorithm. Therefore, the algorithm will not be repeated in this subsection. We assumed the multiplication of the f polynomial and the secret message is calculated in the same way in Section 5.2. You can see the information stored in the memories in Figure 5.6a.



(a) Start of the decryption

(b) After the polynomial multiplication

(c) After the modulo calculation

(d) Final polynomial multiplication

Figure 5.6: Contents of the memory while decrypting

After the polynomial multiplication, we need to calculate the $mod(x^N - 1)$. The difference between the encryption and decryption in this step is the target memory of the process. While encrypting, we are writing the calculated values to BRAM2. In contrast, we store the result in BRAM1 when we decrypt a message. We will read the values in address $a + N$, a in the BRAM2 and write the result to BRAM1's a address.

However, we need to use an intermediate step before writing. As stated before, we stored the information in $GF(2^{16})$ for unifying the calculations. The next step in the calculations requires changing the field from $GF(q)$ to $GF(p)$. To get ready for this process, we fed the results to the "mode_by_q" module, and the output will be written to the BRAM1. The a value should start with N and goes to 0. With that, we destroyed the encrypted message by writing the inter-product on it. You can see values stored in the RAMs in Figure 5.6b. In this figure, the polynomial b is equal to $(amod(x^N - 1))modq)$.

The next calculation is the polynomial multiplication of the f_p and b. However, b is still in the $GF(q)$. We connected the "mode_by_3" module's input to the output of the BRAM1. And use the output of this module. As we stored f_p in $GF(p)$ and b polynomial is converted to $GF(p)$ while reading the value. We can calculate the multiplication. The process is the same as before. We calculated the results with two while loop serially and wrote the results to BRAM2.

The last step is to calculate the result of $(f_p * b)mod(x^N - 1)$ and write it to BRAM2 with the same method explained before. The components should be read from BRAM2, and the result will be written to the same memory. Because our registers are designed to work in $GF(2^{16})$, the calculations made in $GF(p)$ can be easily made and stored in the same logic. Now, we have the pre-decoded message ready in the BRAM2. We will use the same "mode_by_3" module to convert the results while reading the pre-decoded message; the output of the module gives us the decoded message. We added multiplexers to the input of the "mode_by_3" module to decrease resource utilization by using the same module for different parts of the calculations.

The algorithm is provided in Algorithm 3. This implementation uses 623 LUTs, 152 FFs, and 4(18K) BRAMs.

Algorithm 3: NTRU decryption algorithm on FPGA

Initialization:
$i = N; j = N$ cache $bram0[i]$'s first 2 bits into var_a
while $i >= 0$ **do**
  while $j >= 0$ **do**
    read bram1[j] ;read bram2[i+j]
    write $var_a * bram1\_output + bram2\_output$ into $bram2[i + j - 2]$
    $j = j - 1$
  end
  $i = i - 1; j = N$
  cache $bram0[i]$'s first 2 bits into into var_a
end
$j = N - 1$
while $j >= 0$ **do**
  temp= bram2[j+N]
  read bram2[j]
  load $bram2\_output + temp$ into $bram1[j - 2]$ through the mod_by_q module
  $j = j - 1$
end
$i = N; j = N$ cache $bram0[i]$ into var_a **while** $i >= 0$ **do**
  while $j >= 0$ **do**
    read bram0[j] ;read bram1[i+j] through the mod_by_3 module
    write $var_a * mod\_by\_3\_output + bram2\_output$ into $bram2[i + j - 2]$
    $j = j - 1$
  end
  $i = i - 1; j = N$
  cache $bram0[i]$'s second 2 bits into into var_a
end
$j = N - 1$
while $j >= 0$ **do**
  temp= bram2[j+N]
  read bram2[j]
  load $bram2\_output + temp$ into $bram2[j - 2]$
  $j = j - 1$
end
We will get the results through the mod_by_3 module

5.4 Combined Encryption and Decryption

In this module, we combined the architectures of encryption and decryption because we can use the decryption module's first polynomial multiplication part for encryption. Also, in this module, we realized our improvements. The first improvement is in memory management. We actualized the method mentioned in Section 5.1. With that method, we increased the memory efficiency and were able to store multiple keys at

the same time. Without using this method, we would have needed to read flash memory more frequently. Flash memory operations are generally avoided because they are very slow compared to BRAMs. For example, a general-purpose flash memory Infineon S25fl256's read speed is 6.25 MBps in normal mode.

By implementing the proposed memory improvements, we can reach eight polynomial coefficients in one clock. However, in our design, only four of them will be useful for each step of the calculations. The desired coefficients are selected through four 2-bit multiplexers. The outputs of these multiplexers are routed to the multi-stage processing unit. You can see this process in Figure 5.7. The calculations are carried out with four multipliers and four adders connected to four 16-bit registers.

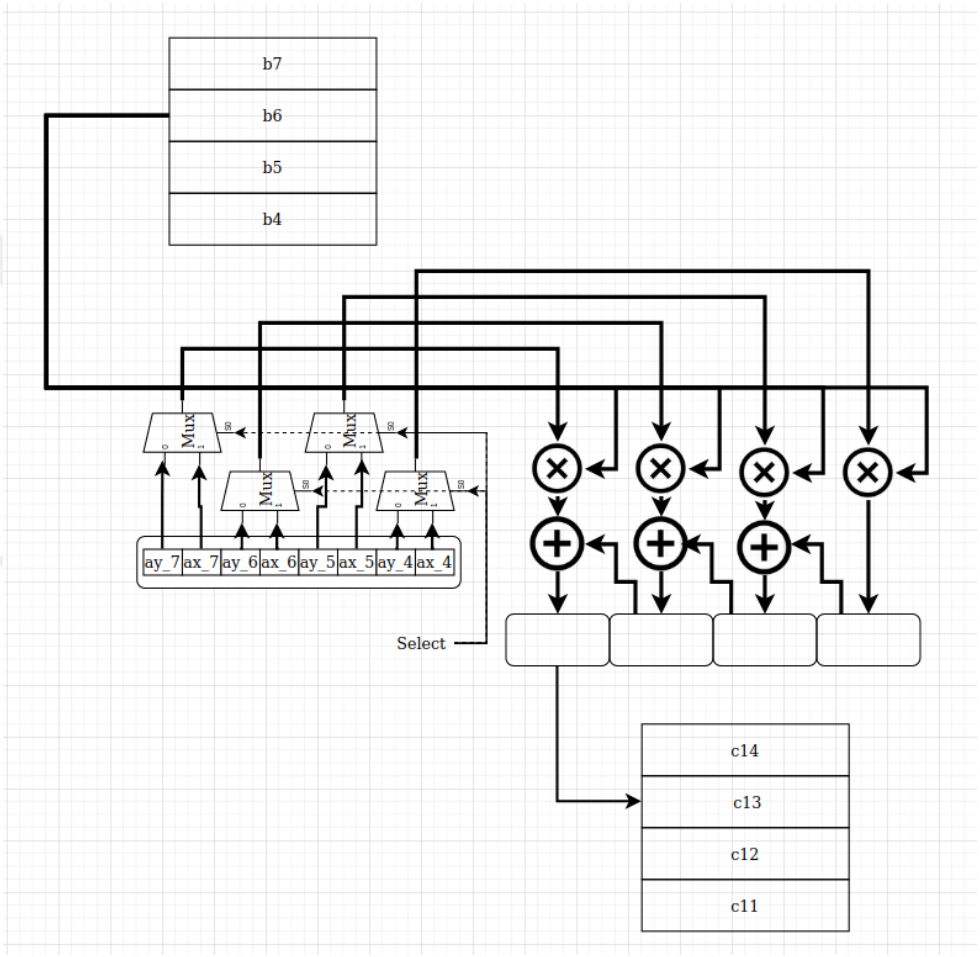


Figure 5.7: Polynomial calculation architecture

In the design provided in Figure 5.7, we assumed we were using the least significant bits of the coefficient words. To do that we select, ax_4, ax_5, ax_6, ax_7 with using the multiplexers. The selected coefficients will be multiplied with the selected coefficient coming from the BRAM1 in parallel. The calculated result should be accumulated before being written to the memory. The right-most register does not have an accumulator and directly takes the multiplication result. For others, we will load the summation of the right neighbor's result and the result of the multiplication.

The left-most register will feed results to the memory module. A symbolic example is provided in Figure 5.8. This example shows the contents of the registers for three clocks.

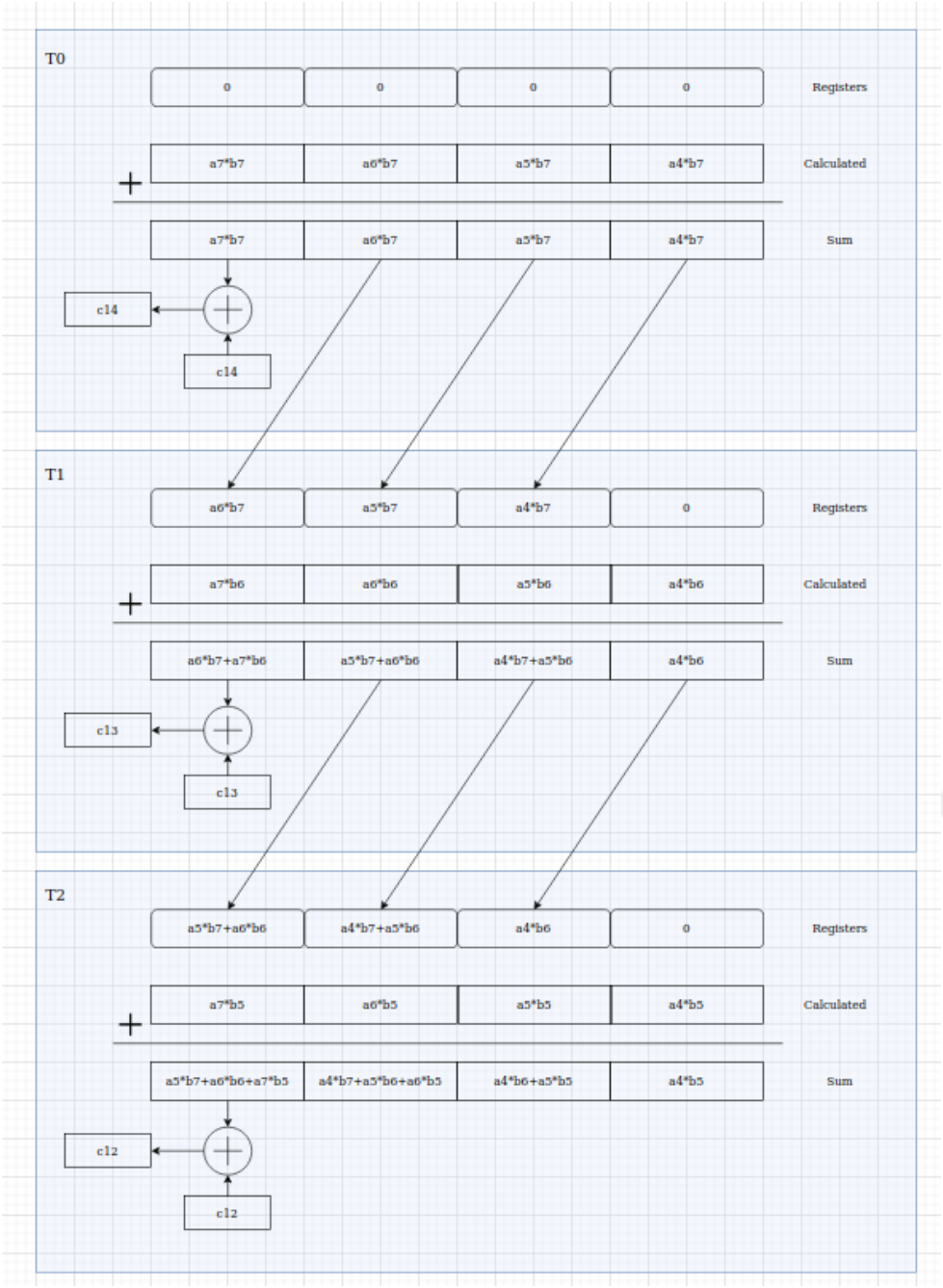


Figure 5.8: Shifter and adder module's contents with respect to time

In this symbolic example, the registers should be cleared when the address of the first RAM changes. Therefore we started with 0 in the registers. The multiplication results will be added to the contents of the registers. For the next clock, we will shift the results to the left while rewriting the register.

We spend some resources on multiplying four coefficients at the same time. To increase the calculation speed four times, we spent 64 FFs to store the inter-products. While we are reaching four coefficients from the first polynomial, we only can read one coefficient from the second one without increasing the BRAM usage. Because the coefficients in the BRAM1 are bigger.

Another optimization was done by changing the $mod(x^N - 1)$ operation. In normal calculations, this operation takes $2 * N$ time. Because we need to issue two read operations from the same memory. $mod(x^N - 1)$ operation matches the a and $a + N$ addresses and adds the value inside the $a + N$ address to the value in address a. To optimize this operation, we can calculate $mod(x^N - 1)$ while doing polynomial multiplications. As we know the N value before starting the calculations, we do not need to write the results to addresses bigger than N. If the result should be written to the $a + N$ address, we can directly add the result to the content of the a address. This method lets us reduce memory utilization significantly. In the split encryption and decryption architecture, we need to use two 18K RAM modules for BRAM2. However, by matching the addresses in the multiplication process, we can decrease the memory utilization to one 18K RAM. In total, our consumption is dropped from four to three BRAMs.

In a network, we cannot control the incoming packets and their lengths. Therefore, in encryption, the public key might come in multiple packets at different times. The standard architecture requires all the data to be ready before starting the polynomial multiplication. However, by adding two variables and keeping track of the valid data parts, we were able to work on partial data. The content of the BRAM0 is not volatile. We kept track of the last available data address in the BRAM1. Normally, our implementation reads every data in the BRAM1 and then decreases one address in BRAM0. If we do not have the complete data, we will save the last address that contains valid data. We multiplied the polynomial coefficients up to that memory address. After that, we decreased the BRAM0's address and repeated the process until we reached 0 in the address of the BRAM0. At his point, we partially multiplied two polynomials. For the rest of the data, we can wait for the next packet if we did not receive it yet. After the packet is received, we reset the address of the BRAM0 and continue to the polynomial multiplication from that point. If a packet is received while we are multiplying the polynomials, the incoming data can still be written to the RAM because we use two ports RAMs in our implementation.

5.5 Modulo by 3 Module

In (Atici et al., 2008), they used a state machine-based mod-3 calculation module. It is a highly area-efficient method. However, it will spend the clock period times the bit width of the variables. For the suggested parameters, this algorithm needs to spend 14 clocks just to take one number mod by 3. At that point, by spending 21 LUT, we can generate the results in 1 clock cycle. Also, our method can deal with negative numbers.

In binary representation, every bit has fixed values. When we take modulo by 3, the least significant bit of the number is one. $2^0 = 1 = 1 mod 3$. Every bit after the first

one, we can easily calculate the modulo by multiplying the result with 2. Therefore, the modulo result of the bits in the number will be $..., 1, 2, 1, 2, 1$. By using these, we can sum the multiplication of the precalculated modulo and bits' actual value. However, for a 14-bit number, this value can reach up to 21. To ease up the decision process, we added another calculation step (Step 2 in Figure 5.9). With that step, the summation of the bits can be up to 7. This result is small enough to be used directly. However, we also consider the negative numbers. If the number is negative, we add 2 to the number and put it directly into the decision tree. You can see the calculation process in Figure 5.9. In the figure, the straight lines represent multiplication by 1, and dashed lines represent multiplication by -1.

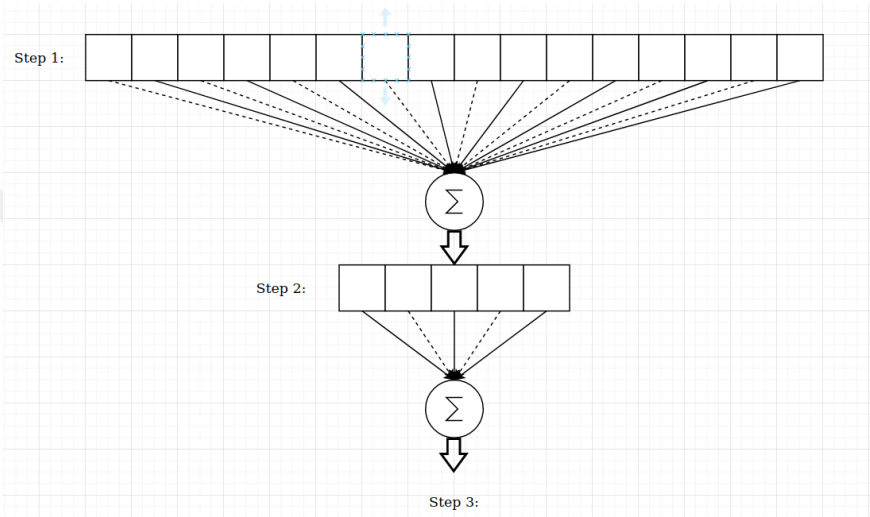


Figure 5.9: Mod3 module architecture

5.6 Modulo by 5 Module

This module is implemented to give a user a chance to change the p parameter of the NTRU. We used a similar algorithm to Section 5.5. However, in there, we have the ability to represent every bit with a 1 or -1. In this scenario, our numbers have to be in the range of 1 to 4. Zero is omitted because any power of 2 cannot be a multiple of 3. Because the multipliers are bigger, we need to increase the bit size of step 2. It might reach up to 40. After that step, we added another block to sum the bits and decrease the value up to 13 in step 3. The value might be big for going to the decision process. However, adding another layer will decrease the maximum value to 10, and implementing this step might be wasteful according to resource management. The module spends 30 LUTs. The increase is 42% compared to the "modulo by 3" module.

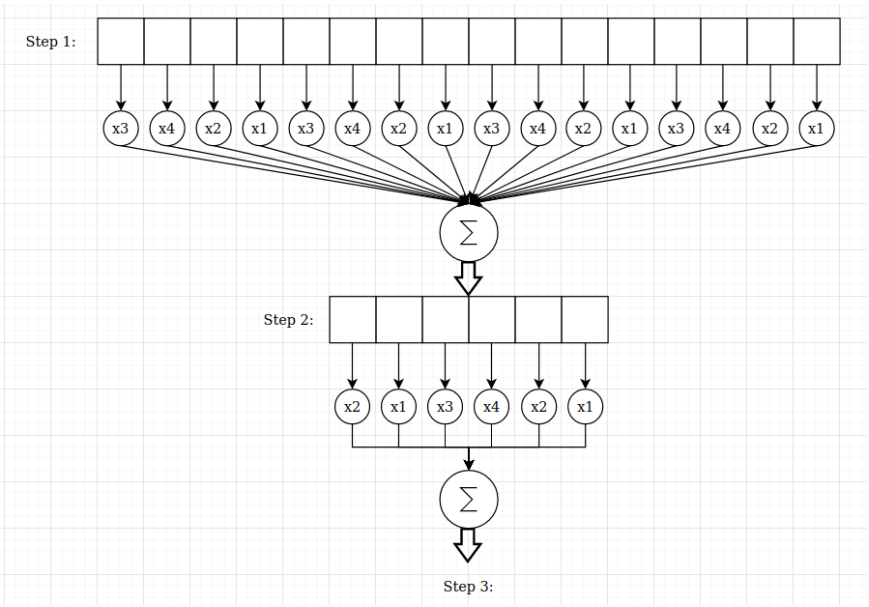


Figure 5.10: Mod5 module architecture

5.7 Modulo by q

Because we restrained the q value as a power of 2. Taking modulo does not require any division or value-specific binary coding. We can execute the modulo by a simple mask operation of desired bit size. If we need signed numbers, we can make the most significant bit of the signed value and extend it to the right. This architecture is provided in Figure 5.11.

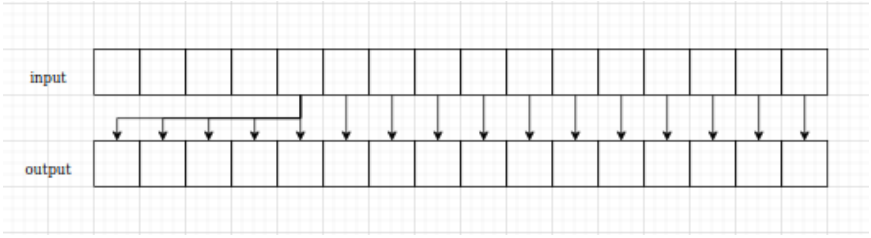


Figure 5.11: Mod q architecture

5.8 Serialization and Deserialization Module

As we designed our implementation to work on changing parameters of the algorithm, we cannot optimize the word length of the calculations. Also, as we assumed our devices are on a network, encryption word length probably will not be equal to communication word length; we can choose to use zero-padding the data to fit them into communication channels. However, this method will decrease the efficiency of

the communication channel. To avoid this problem, we implemented a serializer module.

This module reshapes the incoming stream of changing size inputs to the byte-sized output stream. The module truncates the input data and reshapes it according to the new data length. By using this module, we can increase the efficiency up to 33% if the minimum suggested q is used. Also, the system can support increased q for future security needs. The architecture can be seen in Figure 5.12.

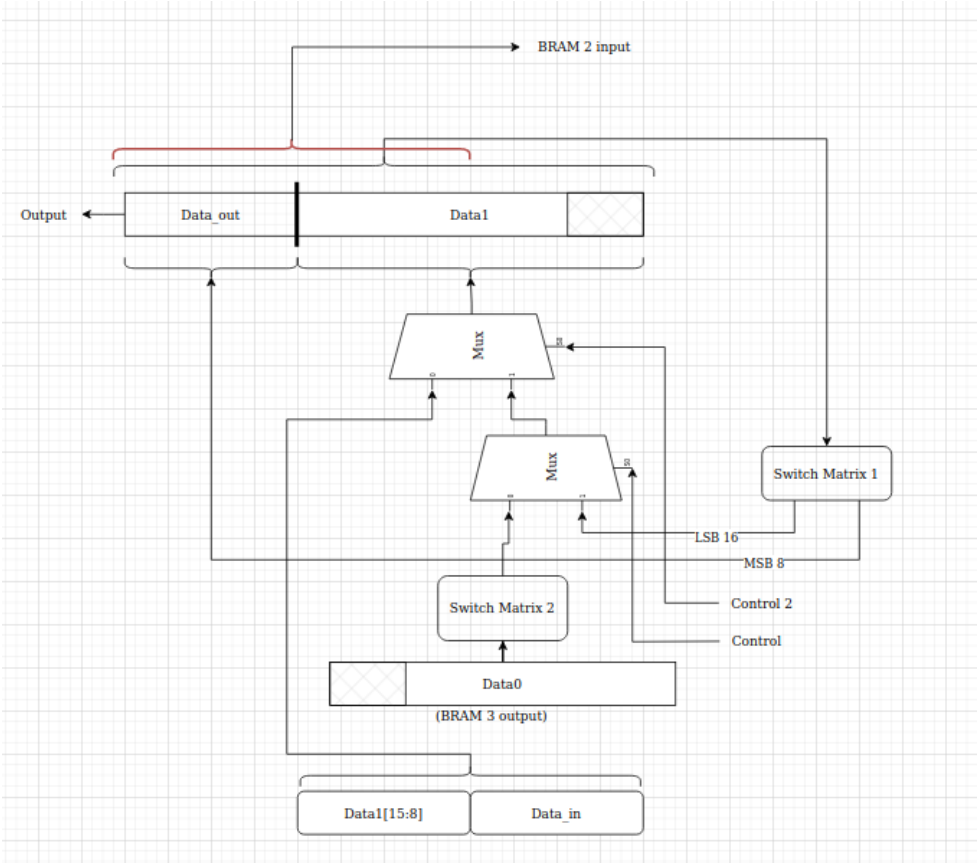


Figure 5.12: Combined Serialization and Deserialization Architecture

5.8.1 Switch Matrix

This module is used in Section 5.8. The switch matrix is used to make a combinational shift register that can change the shifting amount. The interconnections between the source and target bits make the shifting operation possible. You can see a switch matrix representation in Figure 5.13. In this method, only one of the switches in both rows and columns should be closed at the same time. If we assume $wxyz$ bits are input and $abcd$ bits are output with the same order. If we close the "xa", "yb", and "zc" switches, the matrix will perform the one-bit shift left operation.

Also, we can perform signed shift operations on this same module. In this mode, we

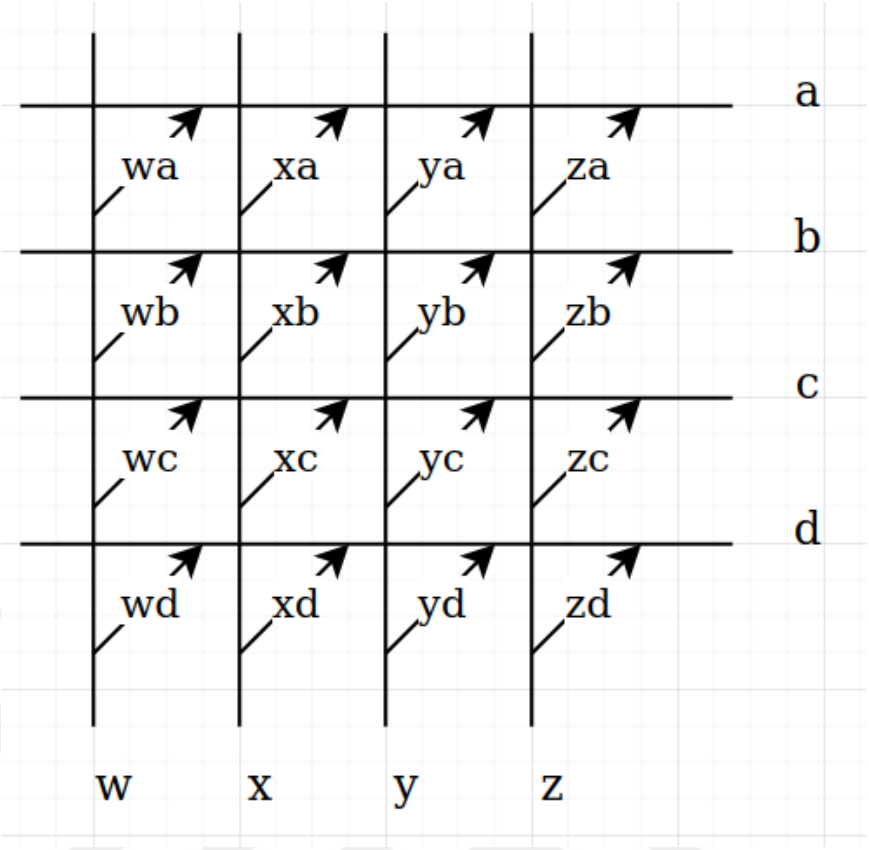


Figure 5.13: Switch matrix design to make adjustable shifters

need to make a sign extension. The sign extension is an exception to the rule stated in the first paragraph. Multiple switches in one column should be closed at the same time. For example, if we closed the "wa", "wb", "xc", and "yd" switches, we can divide a signed 2's complement number by 2.

5.8.2 Serialization

The minimum suggestion for the q parameter is 2048. That will generate coefficients with a length of 10 bits and can be reached to 12 bits for today's security needs and can be increased in the future. The nonstandard bit size will cause inefficiencies while communicating. To solve that issues, we need to combine the coefficients and parse them into bytes. In this process, the data's bit locations in the memory and the stream have to change. To solve that issue, we are using shift registers. We also need to use shift registers in the deserialization module, and they will be the most resource-hungry part of this design; we will combine the serialization and deserialization modules. You can see the combined architecture in Figure 5.12.

While encrypting, the encrypted message will be generated in the last memory module. Also, in decryption, the decrypted message will be formed in the same memory. Therefore, we connected the BRAM3's output to this module. While serializing, we

execute the following steps.

- In normal operations, our data is aligned to the right. Because with that method, we can operate the computational modules without changing anything despite the changing bit sizes. However, to prepare the data, we need to make it aligned to the left side of the register. This means the most significant bit affected by the $modq$ operation (ex. if q=2048, the 10. bit from the LSB) will be the most significant bit in the buffer register named "Data1" in the reference design. To succeed, we need to use a switch matrix module (Switch Matrix 2) that works as an adjustable shifter. The shifting amount will be fixed for every q value, and we only need to change it if we change the NTRU parameters.
- After the data is aligned, we need to transfer it to the data buffer register "Data1" by selecting the multiplexer on the way. This is our new data coming directly from memory, and it is ready to be written on the buffer. However, before that, the buffer needs to not contain any useful information in it. Therefore, information inside the buffer has to be transferred into the Data_out register. Before forming the Data_out and sending it, we have two possibilities.
- The first one is that we have enough more than eight meaningful bits in the buffer (16 bits). In this scenario, we can transfer one byte directly to the Data_out by using the Switch Matrix 1. This module might need to change the shifting amount of every usage according to the remaining bits in the buffer. To compile with the other parts, we need to split the output of this module's output and feed the Data1 and Data_out from different paths.
- The second possibility is that we have less than 8 bits in the buffer. In that scenario, firstly, we need to read the remaining meaningful x bits from the buffer to Data_send registers. To do that, we will use the Switch Matrix 1 to shift x bits of data and load it to the Data_out, Data1 combination. After that, we can load the available data that comes from the memory into the Data1 buffer. Finally, we need to use the Switch Matrix 1 to one more time for shifting $8 - x$ bits. At this point, Data_out is ready to send outside.
- Whenever we have meaningful 8 bits in the Data_out register, we can make the valid flag 1. After that, we can continue with the previous steps.

You can see the algorithm in Algorithm 4.

Algorithm 4: Serialization algorithm

Initialization:

$rd\_address = N$

if $Q == 1024$ **then**

$bits\_total = 10$

end

else if $Q == 2048$ **then**

$bits\_total = 11$

end

else if $Q == 4096$ **then**

$bits\_total = 12$

end

else if $Q == 8192$ **then**

$bits\_total = 13$

end

while $read\_address >= -1$ **do**

  if $bits\_in\_out < 8$ & $bits\_in\_buffer >= (8 - bits\_in\_out)$ **then**

    set SM1 shift amount to $8 - bits\_in\_out$

    $bits\_in\_buffer = bits\_in\_buffer + bits\_in\_out - 8$

    $bits\_in\_out <= 8$

  end

  else if $bits\_in\_out < 8$ & $bits\_in\_buffer < (8 - bits\_in\_out)$ **then**

    set SM1 shift amount to $bits\_in\_buffer$

    $bits\_in\_buffer = 0$

    $bits\_in\_out = bits\_in\_out + bits\_in\_buffer$

  end

  else

    set SM1 shift amount to 0

  end

  load SM1's output to Data_out,Data1

  if $bits\_in\_out == 8$ **then**

    $data\_out\_valid = 1$

    $bits\_in\_out <= 0$

  end

  $data\_out\_valid = 0$

  if $bits\_buffer == 0$ **then**

    load SM2's output to Data1

    $read\_address = read\_address - 1$

    $bits\_in\_buffer = bits\_total$

  end

end

An example of the serialization process is provided in Table 5.1. In this example, the RAM outputs a fixed value for convenience. The output value is 10 bits and equal to $10'h30f$. The SM1 module aligns this data to the left by converting it to $10'hc3c0$. The red colored bits represent the output of the memory module. The blue-colored bits are the output of the module.

Command	Buffer	Valid flag	Output value
Initialize	00000000_00000000_00000000	0	
load	00000000_11000011_11000000	0	
shift 8	11000011_11000000_00000000	1	c3
shift 2	00001111_00000000_00000000	0	
load	00001111_11000011_11000000	0	
shift 6	11110000_11110000_00000000	1	f0
shift 4	00001111_00000000_00000000	0	
load	00001111_11000011_11000000	0	
shift 4	11111100_00111100_00000000	1	fc
shift 6	00001111_00000000_00000000	0	
load	00001111_11110000_11000000	0	
shift 2	00111111_11000011_00000000	1	3f

Table 5.1: Serialization example

5.8.3 Deserialization

Assuming we have matched with our module or an efficient one like ours in the network, the data will come as serialized into bytes. Therefore, we need to unpack the data into NTRU polynomials' word size. As the serialization part contains the required shifters, we can combine two architectures to decrease resource utilization. With the following steps, we can convert the incoming data stream to the coefficients of the NTRU polynomials. We will use the same architecture in Figure 5.12. In this mode, data will be provided from outside of the module, and deserialized information will be formed in the Data_out and the most significant 8 bits of the Data1.

- Firstly the incoming byte will be loaded to the least significant 8 bits of the buffer register Data1. To comply with the serialization architecture, we will load the Data1 register in a single step. To do that, we will cascade the Data1's bits upper half and Data_in and load it to Data1. With this method, we can easily store the remaining information in the buffer because the most significant 8-bit is not changed.
- After the new data is loaded, we should shift it to the most significant 8 bits of the buffer. This task will be completed by the Switch Matrix 1. As the least significant 8-bit is empty, we can load new data with the same method in the first step.
- Now, we have 16 bits in our buffer and can generate one coefficient of the polynomial. To do that, we need to shift the data by $x - 8$ bits by using the

Switch Matrix 1. We assumed x is the bit size of the polynomials' coefficients. We have the coefficient in the Data_out and Data1's most significant 8-bit. We will load that to BRAM1. Because in our architecture, the encrypted message is stored in the second RAM module.

- We have $16 - x$ bits in the Data1 register' lower byte as aligned to the left. We need to carry this into the upper byte with Switch Matrix 1 to load a new value into the lower byte. This step will align the remaining data to the right side of the upper byte in the Data1.
- We can repeat these steps up until all the information is unpacked into the coefficients.

  The numerical example for deserialization is provided in Table 5.2. In this example, the red values come from the input stream. The blue ones are ready to be written to the RAM.

Incoming byte	Command	Buffer	Valid flag	Output
c3	Initialize	00000000_00000000_00000000	0	-
c3	load	00000000_00000000_11000011	0	-
f0	shift 8	00000000_11000011_00000000	0	-
f0	load	00000000_11000011_11110000	0	-
fc	shift 2	00000011_00001111_11000000	1	30f
fc	shift 6	11000011_11110000_00000000	0	-
fc	load	11000011_11110000_11111100	0	-
3f	shift 4	00111111_00001111_11000000	1	30f
3f	shift 4	11110000_11111100_00000000	0	-
3f	load	11110000_11111100_00111111	0	-
3f	shift 6	00111111_00001111_11000000	1	30f

Table 5.2: Deserialization example

The deserialization algorithm is provided in Algorithm 5.

Algorithm 5: Deserialization algorithm

Initialization:

$wr\_address = N$

$bits\_total <= (q == 1024)?10 :$
$(q == 2048)?11 :$
$(q == 4096)?12 :$
$(q == 8192)?13 :$
14

if *data_in_valid* **then**
  $data\_temp[15:0] = data\_in\_temp[15:8], data\_in$
  $bits\_in\_buffer = 8$
  $bits\_in\_out = 0$
end
while $wr\_address >= 0$ **do**
  if $bits\_in\_out < (bits\_total - 8)$ **then**
    set SM1 shift amount to 8
    $data\_in\_temp = SM1output$
    $bits\_in\_out <= 8;$
  end
  else
    set SM1 shift amount to $bits\_total - bits\_in\_out$
    load SM1's output to Data_out,Data1
    write to RAM
    $wr\_address = wr\_address - 1$
    set SM1 shift amount to $bits\_in\_buffer - bits\_total + bits\_in\_out$
    load SM1's output to Data_out,Data1
    $bits\_in\_buffer = 0$
    $bits\_in\_out = bits\_in\_buffer - bits\_total + bits\_in\_out$
  end
  if $data\_in\_valid$ **then**
    $data\_in\_temp[15:0] = data\_in\_temp[15:8], data\_in$
    $bits\_in\_buffer = 8$
    $bits\_in\_out = 0$
  end
end

CHAPTER 6

RESULTS

In this thesis work, we implemented different architectures for NTRU. We have encryption, decryption, and combined implementation. Even if the encryption and decryption modules are designed to be lightweight, they are capable of working with changing parameters. However, they lack some of the improvements mentioned in the combined architecture. In our design, we fixed the p parameter and used a restriction on q as it should be a power of 2. N can be anything the user wants. We provided the flexibility of changing parameters in between calculations.

6.1 Limitations

Some of the general use low-cost FPGAs from different vendors are given in Table 6.1 and Table 6.2 to provide a perspective for the reader. For Xilinx FPGAs, every slice has four LUTs with six inputs, and in Intel Cyclone5 FPGAs, the ALM blocks contain two LUTS with four inputs. These tables show that our implementation is suitable for the general-purpose low-power FPGAs.

Table 6.1: Possible target FPGAs from Xilinx and their resources

	Logic Cells	Slices	CLB FF	BRAM (Kb)	DSP
XC7A15T	16,640	2,600	20,800	900	45
XC7A35T	33,280	5,200	41,600	1,800	90
XC7A50T	52,160	8,150	65,200	2,700	120
XC7A100T	101,440	15,850	126,800	4,860	240

Many FPGA models have more resources than those given in the tables. These models are more expensive, power-hungry, and require more complex circuit designs. They are not very popular with the embedded systems because of the power requirements and the costs.

Table 6.2: Possible target FPGAs from Intel and their resources

	Logic Elements	ALM	ALM Registers	RAM	DSP
5CEA2	25,000	9,434	37,736	1.956 Mb	25
5CEA4	49,000	18,480	73,920	3.383 Gb	66
5CEA5	77,000	29,080	116,320	4.884 Gb	150
5CEA7	149,500	56,480	225,920	7.696 Mb	156

6.2 Implementation Results

We started with standalone encryption and decryption modules with a more standardized approach. After that, we combined the encryption and decryption modules into a single combined module to decrease resource utilization. Also, a combined module is more suitable for a network without a controller node. If one node can be a master in one communication and a slave in the other, it must deploy both encryption and decryption features.

We designed serialization and deserialization modules with a combined shift register-based architecture. However, for use with standalone operations, only serialization and deserialization modules can be generated by deactivating the other part. This module is designed to effectively use the communication line by arranging the NTRU's word size into bytes. To keep resource utilization low, we make the operations sequentially. Therefore, this module cannot generate new data at every clock cycle. While making serialization, the module requires three cycles in the worst case to generate a new output value. While deserializing the data, required clock cycles at the worst case increase to 4.

Table 6.3: Resource utilizations of our modules

	LUT	FF	BRAM
our encryption	344	115	4 (18K)
our decryption	623	152	4 (18K)
our combined and improved architecture	984	385	3 (18K)
our serialization architecture	90	51	0
our deserialization architecture	105	54	0
our ser-des architecture	178	73	0

The characteristics of the implementation are defined by the use cases. We focused on the embedded devices on a mesh network. The features we implemented are listed below:

- Low resource usage: To decrease resource utilization, we stored data in the RAMs. Also, the combined architecture lets us use shared logic between different operations.
- RAM optimization: With packing the data and rearranging the intermediate products, we successfully stored several keys in the memory and kept the required RAM module number low.
- On-the-fly parameter change: This feature lets the user communicate with different devices with different capabilities or change the security level to balance the needs of secrecy and speed. We can support N numbers up to 1024, q can be 1024, 2048, 4096, 8192, 16384, 32768, 65536. The p value should be 3.
- One clock Galois Field operations: We can easily calculate the $mod\ q$ and $mod\ p$ using hardware accelerators in a single clock.
- Serialization and deserialization of different size words: With changing parameters, our bit sizes in the calculations are changing too. This situation requires a solution to efficiently pack and unpack the data into bytes. This module uses additional 178 LUTs and 73 FFs.

You can see the resource utilization of our modules and compare it to others in Table 6.4. While making a comparison, one should remember all these designs has different goals and features.

Table 6.4: Resource comparison between different implementations

Implementation	LUT	FF	BRAM	Features
(Farahmand et al., 2019)	76972	49674	1 (36K)	• Supports NTRU(743,3,2048) and NTRU(443,3,2048) • Uses a co-processor and only makes polynomial multiplication in hardware
(B. Liu & Wu, 2015)	3768	3526	x	•Results are for NTRU(251,3,128) •For different parameters the design has to change •That design is LFSR-based
(Kamal & Youssef, 2009)	4838	21654	x	•Results are for (251, 3, 128) •Has options with different level of parallelism.
(Atici et al., 2008)	1651	8848	x	•Fixed parameters NTRU(167,3,128) •Designed for power efficiency

Table 6.4 (continued).

(Kaps, 2006)	523	2327	x	•Supports NTRU(167, 3, 128) •That design is used to compared different algorithms' hardware efficiency
(Wera, 2020)	2354	2068	4(36K)	•Uses additional 103 DSP units •Supports ntru-hps4096821 •Also, their SW/HW interface uses additional 1499 slices
(Keersmaekers, 2021)	1764	1553	6(36K)	•Uses NTT method •Supports different parameter sets •Interface module uses additional 3980 LUTs and 4750 FFs
Our architecture	984	385	3 (18K)	Our design features are listed above.

6.3 Timing Results

For both encryption and decryption processes, the runtime does not change according to the changing q. This result can be seen in Figure 6.1; it is only dependent on the N parameter. The time complexity of the algorithm is dependent on the N^2 because of the polynomial multiplication. You can see the results in Figure 6.2. The tests are made with a 100 MHz clock.

Our combined encryption and decryption module can work approximately four times faster than standalone modules. These improvements mainly come from the shifter and adder architecture. Also, doing the last modulo calculations while multiplying the polynomials decreases the time we spend by $2 * N$ clock periods. You can see the result in Figure 6.3.

We assumed we were directly converting the plaintext's bits to polynomial coefficients. Therefore, if we select N as 821, we can encrypt 821 bits of data at every round of encryption. N bit input will generate $N * log_2 q$ bit output while encrypting. The decryption process reverses this operation. The throughput of plaintext input before encrypting and recovered message output after decrypting is provided in Figure 6.4. While getting these results, we used a clock speed of 100 MHz.

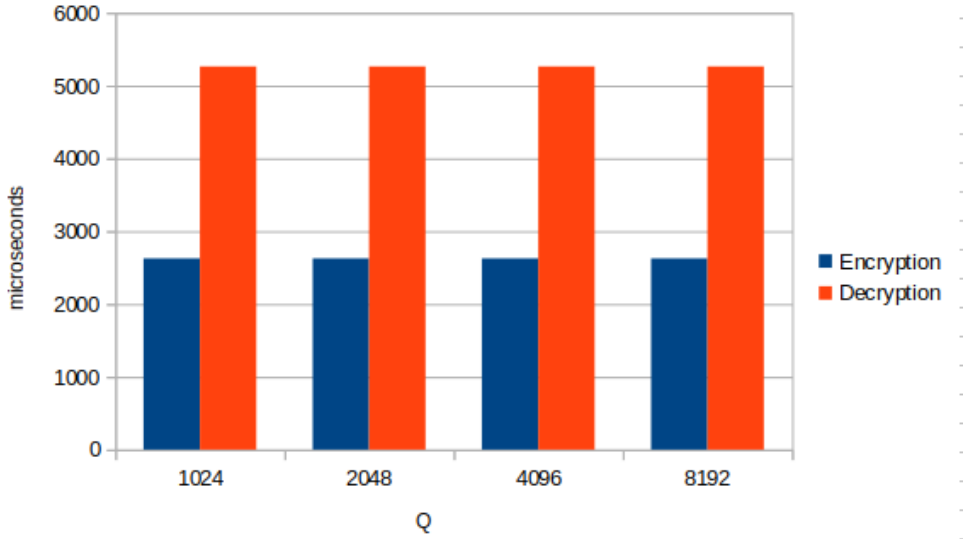


Figure 6.1: Visualization of time spent while encrypting and decrypting messages with respect to Q with a 100 MHz clock

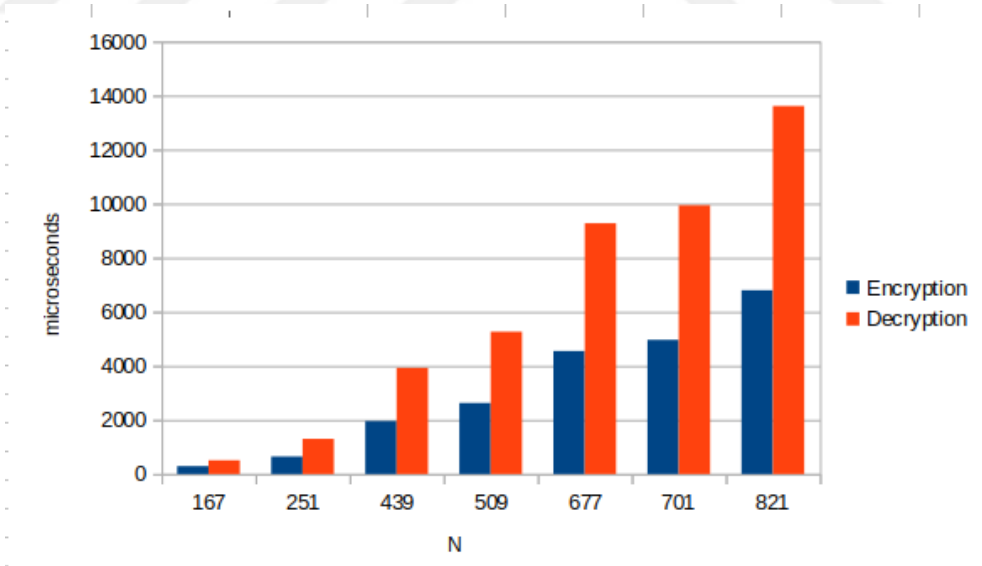


Figure 6.2: Visualization of time spent while encrypting and decrypting messages with respect to N with a 100 MHz clock

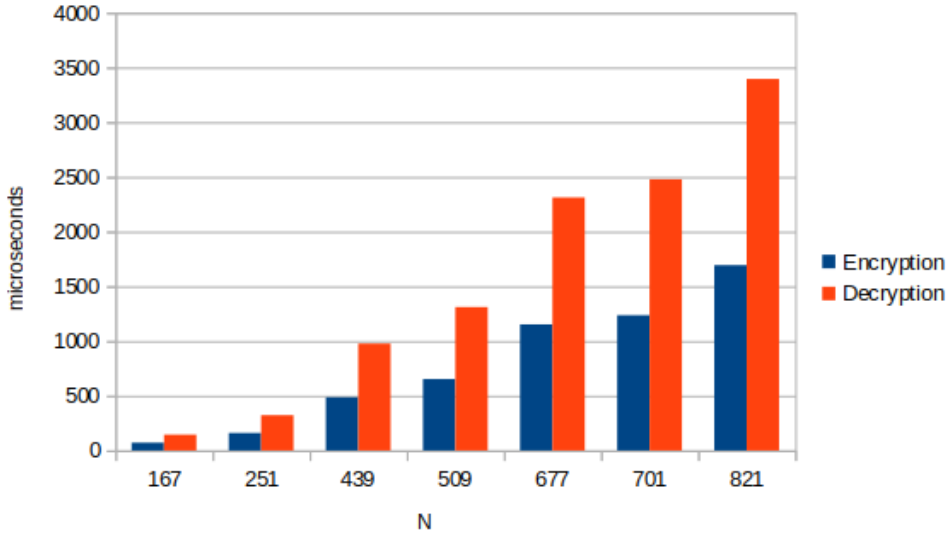


Figure 6.3: Visualization of time spent while encrypting and decrypting on combined architecture with respect to N with a 100 MHz clock

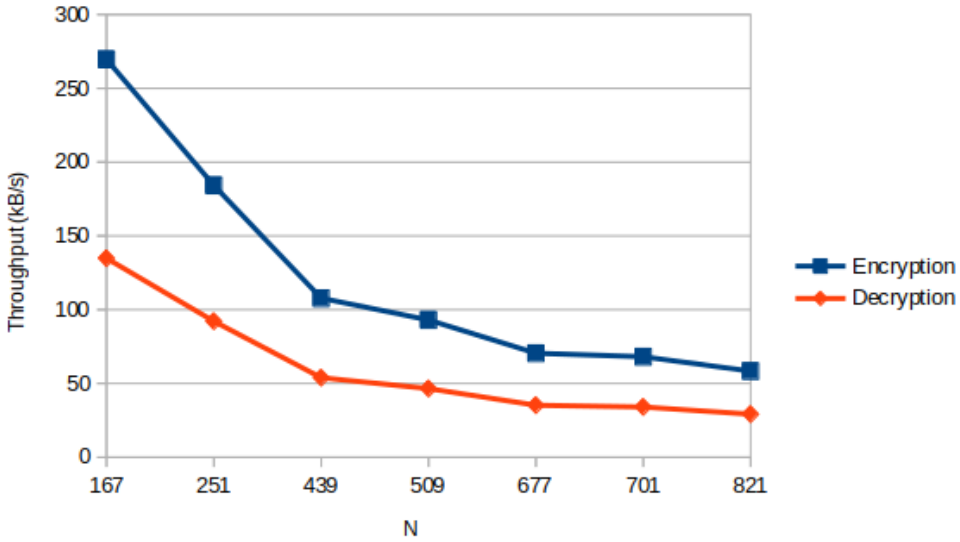


Figure 6.4: Throughput of encryption and decryption with respect to N with a 100 MHz clock

CHAPTER 7

CONCLUSION

With the development in the quantum computing field, some of our most trusted algorithms will not be functioning in the near future. To prepare for this scenario, NIST organized the Post-Quantum Cryptography Challenge. While the work of this thesis work completed, NIST decided that NTRU would not be standardized.

In this thesis, we implemented one of the contestant algorithms on FPGA. The implementation specifically targeted embedded devices with low computational powers. One of the general use cases for this type of device might be ad-hoc mesh network nodes. This was assumed because of the public key cryptographic needs. As we are not planning to work on servers that require high-speed solutions, we traded the speed with low area in our design. Also, we proposed and actualized some methods to increase the efficiency of our design.

In this implementation, we tried to use the FPGA's potential by pipelining and parallelism. The user can change the encryption and decryption parameters on the fly. We shifted the data storage from registers to BRAMs. A data storing scheme was developed to let the user store multiple decryption keys and/or multiple random numbers for encryption. We do not store the encryption keys long-term. Because it is assumed that we will not use the same key repeatedly after the private key is established or the command is sent. However, the random data is saved for performing encryption immediately and eliminate the waiting period. The intermediate products of the calculations are written on the no longer needed data. We designed a selective data filter to overcome the overhead of memory cleaning operations both before starting the process and while cleaning the ram between intermediate cycles. As we are trying to maximize efficiency, we pack multiple data into a single ram slot. This method lets us reach multiple data in a single clock. A shift register-based algorithm was implemented to decrease the time required for encryption or decryption. Finally, we proposed a control algorithm that can work on incomplete data. On a network between distant devices, this will enable us to calculate preliminary results with every coming packet. With that method, the encryption can be finalized shortly after the last packet has arrived.

Using the mentioned methods, we provided a compact and flexible public key cryptographic system for FPGA-based embedded network devices. This design is lightweight enough to be used by already deployed systems in the field without needing to change the hardware for more resources.

7.1 Future Work

To decrease resource utilization, we defined some restrictions on the parameters. The restrictions can be loosened up to increase flexibility. However, this process will increase resource utilization. If a full autonomous design is required, the key generation part can be implemented. Also, the security characteristics of the implementation can be investigated for side-channel attacks.

REFERENCES

Aggarwal, D., Joux, A., Prakash, A., & Santha, M. (2017). Mersenne-756839. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Akiyama, K., Goto, Y., Okumura, S., Takagi, T., Nuida, K., Hanaoka, G., Shimizu, H., & Ikematsu, Y. (2017). A public-key encryption scheme based on non-linear indeterminate equations (giophantus). *IACR Cryptol. ePrint Arch.*, *2017*, 1241.

Albrecht, M., Cid, C., Paterson, K. G., Tjhai, C. J., & Tomlinson, M. (2017). Ntskem. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Albrecht, M., Lindell, Y., Orsini, E., Osheter, V., Paterson, K. G., Peer, G., & Smart, N. P. (2017). *Lima: A pqc encryption scheme* (tech. rep.). https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Round-1-Submissions

Alkim, E., Ducas, L., Pöppelmann, T., & Schwabe, P. (2016). Post-quantum key exchange—a new hope. *25Th {USENIX} security symposium ({USENIX} security 16)*, 327–343.

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

Aragon, N., Blazy, O., Deneuville, J.-C., Gaborit, P., Hauteville, A., Ruatta, O., Tillich, J.-P., & Zémor, G. (2017a). Locker-low rank parity check codes encryption.

Aragon, N., Blazy, O., Deneuville, J.-C., Gaborit, P., Hauteville, A., Ruatta, O., Tillich, J.-P., & Zémor, G. (2017b). LAKE - Low rAnk parity check codes Key Exchange [Submission to the NIST post quantum standardization process. 2017]. https://hal.archives-ouvertes.fr/hal-01946967

Atici, A. C., Batina, L., Fan, J., Verbauwhede, I., & Yalcin, S. B. O. (2008). Low-cost implementations of ntru for pervasive security. *2008 International Conference on Application-Specific Systems, Architectures and Processors*, 79–84.

Azarderakhsh, R., Campagna, M., Costello, C., Feo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., et al. (2017). Supersingular isogeny key encapsulation. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Baan, H., Bhattacharya, S., Garcia-Morchon, O., Rietman, R., Tolhuizen, L., Torre-Arce, J.-L., & Zhang, Z. (2017). Round2: Kem and pke based on glwr. *Cryptology ePrint Archive*.

Bailey, D. V., Coffin, D., Elbirt, A., Silverman, J. H., & Woodbury, A. D. (2001). Ntru in constrained devices. *International Workshop on Cryptographic Hardware and Embedded Systems*, 262–272.

Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., & Santini, P. (2017). Ledapkc: Low-density parity-check code-based public-key cryptosystem. *NIST PQC Round 1 Submission Package*. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Round-1-Submissions

Baldi, M., Barenghi, A., Chiaraluce, F., Pelosi, G., & Santini, P. (2018). Ledakem: A post-quantum key encapsulation mechanism based on qc-ldpc codes. *International Conference on Post-Quantum Cryptography*, 3–24.

Banegas, G., Barreto, P. S., Boidje, B. O., Cayrel, P.-L., Dione, G. N., Gaj, K., Gueye, C. T., Haeussler, R., Klamti, J. B., N'diaye, O., et al. (2018). Dags: Key encap-

sulation using dyadic gs codes. *Journal of Mathematical Cryptology*, *12*(4), 221–239.

Bardet, M., Barelli, E., Blazy, O., Canto Torres, R., Couvreur, A., Gaborit, P., Otmani, A., Sendrier, N., & Tillich, J.-P. (2017). BIG QUAKE BInary Goppa QUAsi–cyclic Key Encapsulation [submission to the NIST post quantum cryptography standardization process]. https://hal.archives-ouvertes.fr/hal-01671866

Bernstein, D. J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., et al. (2017). Classic mceliece: Conservative code-based cryptography. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Bernstein, D. J., Chuengsatiansup, C., Lange, T., & Van Vredendaal, C. (2016). Ntru prime. *IACR Cryptol. ePrint Arch.*, *2016*, 461.

Bernstein, D. J., Heninger, N., Lou, P., & Valenta, L. (2017). Post-quantum rsa. *International Workshop on Post-Quantum Cryptography*, 311–329.

Bonnetain, X., Naya-Plasencia, M., & Schrottenloher, A. (2019). Quantum security analysis of aes. *IACR Transactions on Symmetric Cryptology*, *2019*(2), 55–93.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., & Stehlé, D. (2018). Crystals-kyber: A cca-secure module-lattice-based kem. *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 353–367.

Brands, G., & Roellgen, C. (2015). QRKE: quantum-resistant public key exchange [Withdrawn.]. *CoRR*, *abs/1510.07456*. http://arxiv.org/abs/1510.07456

Brumley, D., & Boneh, D. (2005). Remote timing attacks are practical. *Computer Networks*, *48*(5), 701–716.

Chakraborty, O., Faugère, J.-C., & Perret, L. (2017). *CFPKM : A Key Encapsulation Mechanism based on Solving System of non-linear multivariate Polynomials*

(Research Report). UPMC - Paris 6 Sorbonne Universités ; INRIA Paris ; CNRS. https://hal.inria.fr/hal-01662175

Cheon, J. H., Kim, D., Lee, J., & Song, Y. (2018). Lizard: Cut off the tail! a practical post-quantum public-key encryption from lwe and lwr. *International Conference on Security and Cryptography for Networks*, 160–177.

Dang, V., Mohajerani, K., & Gaj, K. (2021). High-speed hardware architectures and fair fpga benchmarking of crystals-kyber ntru and saber. *NIST 3rd PQC Standardization Conf.*

D'Anvers, J.-P., Karmakar, A., Roy, S. S., & Vercauteren, F. (2018). Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. *International Conference on Cryptology in Africa*, 282–305.

Ding, J., Takagi, T., Gao, X., & Wang, Y. (2017). *Ding key exchange* (tech. rep.).

Efstathiou, K., & Efstathiou, M. (2018). Celestial gearbox. *Mechanical Engineering*, *140*(09), 31–35.

El Bansarkhani, R. (2017). Key encapsulation and encryption based on lattices. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Farahmand, F., Nguyen, D. T., Dang, V. B., Ferozpuri, A., & Gaj, K. (2019). Software/hardware codesign of the post quantum cryptography algorithm ntruencrypt using high-level synthesis and register-transfer level design methodologies. *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 225–231.

Gambetta, J. (2022). Ibm quantum roadmap to build quantum-centric supercomputers. https://research.ibm.com/blog/ibm-quantum-roadmap-2025

Gligoroski, D., & Gjøsteen, K. (2017). Post-quantum key encapsulation mechanism edon-k. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 212–219.

Hamburg, M. (2017). *Module-lwe key exchange and encryption: The three bears* (tech. rep.).

Hetch, P., & Kamlofsky, J. (2017). Hk17: Post quantum key exchange protocol based on hypercomplex numbers. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Hoffstein, J., Pipher, J., Schanck, J. M., Silverman, J. H., Whyte, W., & Zhang, Z. (2017). Choosing parameters for ntruencrypt. *Cryptographers' Track at the RSA Conference*, 3–18.

Hoffstein, J., Pipher, J., & Silverman, J. H. (1998). Ntru: A ring-based public key cryptosystem. *International Algorithmic Number Theory Symposium*, 267–288.

Hülsing, A., Rijneveld, J., Schanck, J. M., & Schwabe, P. (2017). Ntru-hrss-kem. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Kamal, A. A., & Youssef, A. M. (2009). An fpga implementation of the ntruencrypt cryptosystem. *2009 International Conference on Microelectronics-ICM*, 209–212.

Kaps, J.-P. E. (2006). *Cryptography for ultra-low power devices* (Doctoral dissertation). Worcester Polytechnic Institute.

Karatsuba, A. A., & Ofman, Y. P. (1962). Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk*, *145*(2), 293–294.

Keersmaekers, K. (2021). A compact and flexible hardware accelerator for ntru polynomial multiplication using ntt. *Master's thesis, KU Leuven*.

Kim, J.-L., Kim, Y.-S., Galvez, L., Kim, M. J., & Lee, N. (2018). Mcnie: A code-based public-key cryptosystem. *arXiv preprint arXiv:1812.05008*.

Kim, Y., Daly, R., Kim, J. S., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., & Mutlu, O. (2014). Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 1–19. https://doi.org/10.1109/SP.2019.00002

Le Trieu Phong, T. H., Aono, Y., & Moriai, S. (2017). *Lotus* (tech. rep.).

Lerman, L., Bontempi, G., Markowitch, O., et al. (2014). Power analysis attack: An approach based on machine learning. *Int. J. Appl. Cryptogr.*, *3*(2), 97–115.

Liu, B., & Wu, H. (2015). Efficient architecture and implementation for ntruencrypt system. *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 1–4.

Liu, D., Li, N., Kim, J., & Nepal, S. (2017). Compact-lwe: Enabling practically lightweight public key encryption for leveled iot device authentication. *IACR Cryptol. ePrint Arch.*, *2017*, 685.

Lu, X., Liu, Y., Zhang, Z., Jia, D., Xue, H., He, J., & Li, B. (2018). LAC: practical ring-lwe based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, 1009. https://eprint.iacr.org/2018/1009

Luengo, I., Avendaño, M., & Marco, M. (2017). Dme: A public key, signature and kem system based on double exponentiation with matrix exponents. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Matsumoto, T., & Imai, H. (1988). Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. *Workshop on the Theory and Application of of Cryptographic Techniques*, 419–453.

Melchor, C. A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., Persichetti, E., Zémor, G., & Bourges, I. (2017). Hamming quasi-cyclic (hqc). *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Melchor, C. A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., Hauteville, A., Zémor, G., & Bourges, I.-C. (2017). Ouroboros-r. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Melchor, C. A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., & Zémor, G. (2017). Rank quasi-cyclic (rqc). *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Moody, D. (2017). The ship has sailed: The nist post-quantum cryptography "competition"(invited talk). *Advances in Cryptology-ASIACRYPT*.

Naehrig, M., Alkim, E., Bos, J., Ducas, L., Easterbrook, K., LaMacchia, B., Longa, P., Mironov, I., Nikolaenko, V., Peikert, C., et al. (2017). Frodokem. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Orman, H. (2021). Internet security and quantum computing. *IACR Cryptol. ePrint Arch.*, 1637. https://eprint.iacr.org/2021/1637

O'Rourke, C. M. (2002). Efficient ntru implementations. *Master's thesis, Worcester Polytechnic Institute*.

Peretz, J., & Granot, N. (2017). Tpsig, nist submission. *Internet: https://csrc. nist. gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions*.

Plantard, T. (2017). *Odd manhattan* (tech. rep.).

Pomerance, C. (1996). A tale of two sieves. *Notices Amer. Math. Soc*.

Rabin, M. O. (1979). *Digitalized signatures and public-key functions as intractable as factorization* (tech. rep.). Massachusetts Inst of Tech Cambridge Lab for Computer Science.

Roetteler, M., Naehrig, M., Svore, K. M., & Lauter, K. (2017). Quantum resource estimates for computing elliptic curve discrete logarithms. https://doi.org/10.48550/ARXIV.1706.06752

Saarinen, M.-J. O. (2017). Hila5: On reliability, reconciliation, and error correction for ring-lwe encryption. *International conference on selected areas in cryptography*, 192–212.

Sanders, B. C. (2017). *How to build a quantum computer*. IOP Publishing Bristol.

Scarfone, K., Jansen, W., Tracy, M., et al. (2008). Guide to general server security. *NIST Special Publication*, *800*(123).

Seo, M., Kim, S., Lee, D. H., & Park, J. (2017). Emblem and r. emblem.

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. *Proceedings 35th annual symposium on foundations of computer science*, 124–134.

Shpilrain, V., Bessonov, M., & Grigoriev, D. (2017). *Guess again* (tech. rep.).

Standaert, F.-X. (2010). Introduction to side-channel attacks. In *Secure integrated circuits and systems* (pp. 27–42). Springer.

Steinfeld, R., Sakzad, A., & Zhao, R. K. (2017). Titanium: Proposal for a nist post-quantum public-key encryption and kem standard. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Szepieniec, A. (2017). Kem proposal for nist poc project. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Tezcan, C. (2021). Optimization of advanced encryption standard on graphics processing units. *IEEE Access*, *9*, 67315–67326. https://doi.org/10.1109/ACCESS.2021.3077551

Tezcan, C. (2022). Key lengths revisited: Gpu-based brute force cryptanalysis of des, 3des, and PRESENT. *J. Syst. Archit.*, *124*, 102402. https://doi.org/10.1016/j.sysarc.2022.102402

Wang, Y. (2017). Rlcekeyencapsulation mechanism (rlce-kem) specifcation. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Wera, M. (2020). A compact hw-sw codesign of ntru kem. *Master's thesis, KU Leuven*.

Yamada, A., Eaton, E., Kalach, K., Lafrance, P., & Parent, A. (2017). Qc-mdpc kem. *NIST PQC Round 1 Submission Package*. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Round-1-Submissions

Yu, H., Zhang, C., & Jiang, H. (2021). A fpga-based heterogeneous implementation of ntruencrypt. In *Advances in parallel & distributed processing, and applications* (pp. 461–475). Springer.

Yu, Y., & Zhang, J. (2017). Lepton: Key encapsulation mechanisms from a variant of learning parity with noise. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.

Zhao, Y., Jin, Z., Gong, B., & Sui, G. (2017). A modular and systematic approach to key establishment and public-key encryption based on lwe and its variants. *A Submission to the NIST Post-Quantum Cryptography Standardization Process*.