Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Anna Bakkebø

Implementation of the Number Theoretic Transform

for Faster Lattice-Based Cryptography

Master's thesis in Natural Schience with Teacher Education
Supervisor: Kristian Gjøsteen

December 2020

NTNU
Norwegian University of Science and Technology

Anna Bakkebø

Implementation of the Number Theoretic Transform

for Faster Lattice-Based Cryptography

Master's thesis in Natural Schience with Teacher Education
Supervisor: Kristian Gjøsteen
December 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Norwegian University of
Science and Technology

Acknowledgements

First of all I want to thank my supervisor Professor Kristian Gjøsteen. I have always had a feeling that you want to help me do the best I can. You have had a good mix between coming with suggestions on what I can do next, but also let me be in control of the thesis. I have also felt that you both care about my thesis, but also about me as a person.

Secondly I want to thank me good friend Elisabeth Enerhaug. You have been a great support for me this year, and to be able to talk about the process that it is to write a master thesis with you have been such a gift to me. You have also been a great source of joy to me this semester, and have made the semester into a semester that I am thankful for.

I would also like to thank my friends Anna Karina Kristianslund, Thomas Schjem and Endre Sørmo Rundsveen for giving me fun breaks from writing the thesis, and for being great friends during the course of my study. I would also like to thank Thale Lund Ness and Marthe Fjellberg for good conversations in the lunch breaks leading up to the due date. There are also many other friends outside of school, that I have not mentioned, that I am also so thankful for in the process of writing this thesis, thank you all. Finally I would also like to thank my family, Mom, Dad, Andreas and Sara Alida, for being supporters during the whole course of my study.

Abstract

This thesis is about implementation of the Number Theoretic Transform, NTT. NTT is an algorithm for multiplying polynomials faster, and through this paper we look at how it is able to do this, and to what extent it computes multiplication faster. Our motivation is to use this for faster lattice-based cryptography. In this paper we have implemented NTT for polynomials in $R_q = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$, and discovered that it does multiply faster, especially when N gets bigger. We have looked at how it would affect the running time of NTRU, which uses multiplication of polynomials in the key generation, encryption and decryption. We have also looked at how it affects the running time of a commitment scheme that uses multiplication of a matrix times a vector where all the inputs are polynomials in R_q. The result showed us that multiplication was much faster, both in NTRU and in the commitment scheme.

Contents

Chapter 1

Introduction

Lattice-based cryptography is gaining more popularity nowadays, as it is believed to be resistant against quantum computers. Many of these lattice-based cryptosystems use multiplications of polynomials as part of encryption and decryption. Being able to compute these multiplications fast would help make these cryptosystems more efficient. In this paper we look at one method for multiplying polynomials faster, namely the Number Theoretic Transform, or NTT for short. We look at how it works, whether it is faster or not, in theory, and have also implemented it in c-code[1] and looked at the difference in runtime with NTT multiplication and normal multiplication.

We have specifically looked at how it is able to make the cryptosystem, NTRU, and a commitment scheme faster. Both NTRU and the commitment scheme use multiplication of polynomials in $R_q = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$ as part of the arithmetic. We have based our work with the Number Theoretic Transform, and NTRU, on the paper "NTTRU: Truly Fast NTRU Using NTT" by Vadim Lyubashecsky and Gregor Seiler (2019). The commitment scheme we have looked at is based on the paper "More Efficient Commitments from Structured Lattice Assumptions" by Baum, Damgård, Lyubashevsky, Oechsner and Peikert (2016).

[1] Source code can be found at https://github.com/annabakkebo/Master_NTT.

Chapter 2

Why NTT

2.1 Preliminaries

2.1.1 Notation

Notation	Description
$f(X)$	Polynomial that is reducible to factors of small degree in $\mathbb{Z}_q[X]$. In this thesis we use $f(X) = X^N + 1$
R_q	The ring $\mathbb{Z}_q[X]/\langle f(X)\rangle$.
q	Prime number q used as modulus in the ring R_q
N	The degree of $f(X)$. In this paper we want N to be on the form $N = 2^k \cdot a$ for some $k, a \in \mathbb{Z}$
β_2	Binomial distribution Generate $a_0, a_1, a_2, a_3 \xleftarrow{\$} \{0, 1\}$ Output $a_0 + a_1 + a_2 + a_3 \mod \pm 3$
β_2^N	Generate polynomial in R_q where the coefficients are generated as in β_2
$\mod \pm q$	Function for modular reduction modulo q mapping onto the space $[-(q-1)/2, (q-1)/2]$

Table 2.1: Notation

Definition 1: A public-key cryptosystem consist of three different algorithms; Key generation (K), encrypt (E) and decrypt (D).

- The key generation (K) does not take anything as input and outputs a secret key sk and a public key pk.

- The encrypt algorithm takes in the message m and the public key pk, and outputs the ciphertext c.
- The decrypt algorithm takes in the ciphertext c and the secret key sk, and outputs the message m.

For a public-key cryptosystem to be valid we need $D(E(m, pk), sk) = m$, for any message m and any pair of keys (sk, pk) output by the key generation.

2.2 NTRU

To give context to why we would want to look at the Number Theoretic Transform, we will look at a cryptosystem called NTRU. NTRU is a lattice-based public-key cryptosystem, which means that it is a public-key cryptosystem that uses lattice-based arithmetic in encryption and decryption. NTRU specifically uses multiplication of polynomials in the ring $R_q = \mathbb{Z}_q[X]/\langle f(X)\rangle$. The key generation produces a secret key, a, and a public key, h. The secret key is produced by:

1. first picking a polynomial who's coefficients are in $\{-1, 0, 1\}$,
2. then multiplying this polynomial with a small prime, p,
3. and adding 1. We add 1 to make it easier to retrieve the message in decryption.

This result is set to the secret key if it is invertible in R_q. The public key, h, is the inverse of the secret key, a, multiplied by a small prime, p, and a polynomial with small coefficients. All of the coefficients in the secret key is at most p, and the coefficients in the public key looks random.

Encryption looks like multiplying the public key with a randomness, and then adding the message. Decryption looks like multiplying the cipher text with the secret key.

2.2.1 Key generation, encryption and decryption

Algorithm 1: Key generation

Output: Secret key a, Public key h

$a' \xleftarrow{\$} \beta_2^N$

$a := pa' + 1$

if a *is not invertible in* R_q **then**

    Restart

$g \xleftarrow{\$} \beta_2^N$

$h := pg/a$

return $(a,\ h)$

Algorithm 2: Encryption

Input: Message m, public key h

Output: Ciphertext c

$r \xleftarrow{\$} \beta_2^N$

$c := hr + m$

return c

Algorithm 3: Decryption

Input: Ciphertext c, Secret key a

Output: Message m

$m := (ca \mod \pm q) \mod \pm p$

return m

Theorem 1: Let key generation, encryption and decryption be as stated in Algorithm 7, 3 and 2. Let $f(X) = X^N + 1$ with $N < (q-1)/4p$. If we encrypt a message m and decrypt the ciphertext produced by the encryption, we end up with m. (i.e $D(E(m,h),a) = m$).

Proof. Let the secret key and public key be as in the description of the key generation algorithm. The key generation algorithm will then return the secret key a and the

public key h. Encrypting the message will return $c = hr + m$. Decryption looks like multiplying the ciphertext with the secret key $a \mod \pm q \mod \pm p$.

$$\begin{aligned} D(c,a) = &\ (c \cdot a \mod \pm q) \mod \pm p \\ = &\ ((hr+m) \cdot a \mod \pm q) \mod \pm p \\ = &\ ((pg/a \cdot r + m) \cdot a \mod \pm q) \mod \pm p \\ = &\ (pgr + ma \mod \pm q) \mod \pm p \\ = &\ ((pgr + m(pa'+1) \mod \pm q) \mod \pm p \\ = &\ ((p(gr+ma') + m) \mod \pm q) \mod \pm p \end{aligned}$$

For us to not have a decryption error we want

$$(p(gr+ma')+m) \mod \pm q = (p(gr+ma')+m) \in R \tag{2.1}$$

For this to be true we need $|(p(gr+ma')+m)| \leq (q-1)/2$. The coefficients in m has absolute value less than or equal to 1. This gives that we need $|p(gr+ma')| < (q-1)/2$. Which leads to $|gr+ma'| < (q-1)/2p$. We know that all coefficients in g, r, a' and m have absolute value at most 1. This leads to the coefficients of gr and ma' are at most N.

$$|gr+ma'| \leq 2N < (q-1)/2p$$

Which confirms that the Equation (2.1) holds.

$$\begin{aligned} D(c,a) = &\ ((p(gr+ma')+m) \mod \pm p \\ = &\ m \mod \pm p \\ = &\ m \end{aligned}$$ □

2.3 Multiplication of polynomials

In the NTRU cryptosystem a crucial step of both encryption and decryption, is multiplication of polynomials. When decrypting we multiply the cipher text polynomial, c, which looks like a random polynomial of degree less than N, with the secret key polynomial, a, of degree less than N. Being able to do multiplications of

polynomials fast over R_q will therefore be advantageous when computing NTRU fast.

Chapter 3

Number theoretic transform

3.1 Chinese remainders theorem

The Number Theoretic Transform is heavily based on the Chinese Remainders Theorem. Let $\{m_1, m_2, ..., m_k\}$ be pairwise coprime integers such that $\prod_{i=1}^{k} m_i = M$. Then the system of equations

$$\begin{aligned} x &= a_1 \mod m_1 \\ x &= a_2 \mod m_2 \\ &\dots \\ x &= a_k \mod m_k \end{aligned}$$

has a unique $x \in \mathbb{Z}_M$ that solves the equations. This version of the Chinese Remainders Theorem is related to coprime integers. We will look more specifically on this theorem in relation to polynomial rings that we will use for our Number Theoretic Transform.

Theorem 2: Let $f(X) \in \mathbb{Z}_q[X]$ be so that $f(X) = \prod_{i=1}^{k} f_i(X)$ where $\langle f_i(X)\rangle + \langle f_j(X)\rangle = \mathbb{Z}_q[X] \quad \forall i \neq j \in \{1, ..., k\}$ (i.e. $f_i(X)$ and $f_j(X)$ are coprime, relatively prime, for all $i \neq j$). Then

$$\mathbb{Z}_q[X]/\langle f(X)\rangle \cong \prod_{i=1}^{k} \mathbb{Z}_q[X]/\langle f_i(X)\rangle$$

Proof. We will give a proof by induction. First we check that the base case holds. If $k = 1$ we have:

$$f(X) = f_1(X)$$

Because $\mathbb{Z}_q[X]/\langle f(X)\rangle = \mathbb{Z}_q[X]/\langle f_1(X)\rangle$ we definitely have that $\mathbb{Z}_q[X]/\langle f(X)\rangle \cong \mathbb{Z}_q[X]/\langle f_1(X)\rangle$. So the Theorem holds for $k = 1$.

We will now assume that it holds for $k = n$ and want to show that it will then hold for $k = n + 1$. Our assumption is:

1. $f(X) = \prod_{i=1}^{n+1} f_i(X)$
2. $\langle f_i(X)\rangle + \langle f_j(X)\rangle = \mathbb{Z}_q[X] \quad \forall\, i \neq j \in \{1, 2, ..., n+1\}$
3. $\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle \cong \prod_{i=1}^{n} \mathbb{Z}_q[X]/\langle f_i(X)\rangle$

What we want to show is that this leads to $\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n+1} f_i(X)\rangle \cong \prod_{i=1}^{n+1} \mathbb{Z}_q[X]/\langle f_i(X)\rangle$. If we can show that

$$\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n+1} f_i(X)\rangle \cong \mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle \tag{3.1}$$

we have by assumption 3 that

$$\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n+1} f_i(X)\rangle \cong \prod_{i=1}^{n+1} \mathbb{Z}_q[X]/\langle f_i(X)\rangle$$

First we start by defining a ring homomorphism:

$$\begin{array}{rcl} h : \mathbb{Z}_q[X]/\langle f(X)\rangle & \to & \mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle \\ p(X) + \langle f(X)\rangle & \mapsto & (p(X) + \langle f_{n+1}(X)\rangle, p(X) + \langle \prod_{i=1}^{n} f_i(X)\rangle) \end{array}$$

In $\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$ we have component wise multiplication and addition. First we check that it is a ring homomorphism. For all p, s in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ we have the following properties:

1. $$\begin{array}{rcl} h(p) + h(s) & = & (p + \langle f_{n+1}(X)\rangle, p + \langle \prod_{i=1}^{n} f_i(X)\rangle) + (s + \langle f_{n+1}(X)\rangle, s + \langle \prod_{i=1}^{n} f_i(X)\rangle) \\ & = & (p + s + \langle f_{n+1}(X)\rangle, p + s + \langle \prod_{i=1}^{n} f_i(X)\rangle) \\ & = & h(p + s) \end{array}$$

   $\Rightarrow h(p + s) = h(p) + h(s)$

2. $h(p)h(s) = (p + \langle f_{n+1}(X)\rangle, p + \langle \prod_{i=1}^{n} f_i(X)\rangle) \cdot (s + \langle f_{n+1}(X)\rangle, s + \langle \prod_{i=1}^{n} f_i(X)\rangle)$
$= (ps + \langle f_{n+1}(X)\rangle, ps + \langle \prod_{i=1}^{n} f_i(X)\rangle)$
$= h(ps)$

$\Rightarrow h(ps) = h(p)h(s)$

3. $h(1) = (1 + \langle f_{n+1}(X)\rangle, 1 + \langle \prod_{i=1}^{n} f_i(X)\rangle)$
$\Rightarrow h(1_{\mathbb{Z}_q[X]/\langle f(X)\rangle}) = 1_{\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle}$

This shows that it is a ring homomorphism. We would now like to prove that this is an isomorphism. The Fundamental Theorem of Isomorphisms (Bhattacharya, Jain, & Nagpaul, 1994, p. 190) states that the image of a homomorphism, ϕ from R, is isomorphic to R modulus the kernel. In our instance the image of h is isomorphic to $\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$ modulus the kernel of h.

$$\operatorname{Im} h \cong (\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle)/\ker h$$

To show that h is an isomorphism we want the kernal of h to be $\langle 0\rangle$ and the image to be $\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$.

First we start by showing that the image of h is $\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$. To do this we want to show

$$\forall (p, s) \in \mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$$
$$\exists t \in \mathbb{Z}_q[X]/\langle f(X)\rangle, \quad \text{such that } h(t) = (p, s)$$

By assumption we have $\langle f_{n+1}(X)\rangle + \langle f_i(X)\rangle = \mathbb{Z}_q[X]$ for all $i \in \{1, ..., n\}$. We therefore have that there exist an $a_i \in f_{n+1}(X)$ and a $b_i \in f_i(X)$ such that $a_i + b_i = 1$. This leads to

$$1 = \prod_{i=1}^{n}(a_i + b_i) \quad \text{where } a_i \in \langle f_{n+1}(X)\rangle \text{ and } b_i \in \langle f_i(X)\rangle \tag{3.2}$$

$$= \underbrace{a_1a_2...a_n + a_1a_2a_3...b_n + ... + a_1...b_i...a_n + ...}_{= a \in \langle f_{n+1}\rangle} + \underbrace{b_1b_2...b_n}_{= b \in \langle \prod_{i=1}^{n} f_i(X)\rangle} \tag{3.3}$$

$$1 = a + b \tag{3.4}$$

Now we can choose $t = bp + as$. We then have

$$\begin{aligned} h(bp+as) &= (bp + as + \langle f_{n+1}(X)\rangle, bp + as + \langle \textstyle\prod_{i=1}^{n} f_i(X)\rangle) \\ &= (bp + \langle f_{n+1}(X)\rangle, as + \langle \textstyle\prod_{i=1}^{n} f_i(X)\rangle) \\ &= ((1-a)p + \langle f_{n+1}(X)\rangle, (1-b)s + \langle \textstyle\prod_{i=1}^{n} f_i(X)\rangle) \\ &= (p + \langle f_{n+1}(X)\rangle, s + \langle \textstyle\prod_{i=1}^{n} f_i(X)\rangle) \end{aligned}$$

This holds for all (p, s) in $\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$, and shows that the image of h is in fact $\mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$.

To finish the proof that it is an isomorphism, we want to show that the kernel to h is $\langle 0\rangle$. The definition of the kernel is the elements in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ that sends to $(0, 0)$. We have that if an element, p, sends to $(0, 0)$ then $p \in \langle f_{n+1}(X)\rangle$ and $p \in \langle \prod_{i=1}^{n} f_i(X)\rangle$. This leads to p being in the intersection, $p \in \langle f_{n+1}(X)\rangle \cap \langle \prod_{i=1}^{n} f_i(X)\rangle$.

By (3.4) we have that $1 = a + b$, where $a \in \langle f_{n+1}(X)\rangle$ and $b \in \langle \prod_{i=1}^{n} f_i(X)\rangle$. Using this we can rewrite p to be $p = pa + pb$.

$$pa \in \langle \prod_{i=1}^{n} (f(X)) \cdot f_{n+1}(X)\rangle = \langle f(X)\rangle$$

$$pb \in \langle f_{n+1}(X) \cdot \prod_{i=1}^{n} (f(X))\rangle = \langle f(X)\rangle$$

Which results in p being an element in $\langle f(X)\rangle$ as it is a sum of two elements in $\langle f(X)\rangle$. Because p is an element in $\langle f(X)\rangle$, we have that $p = 0 + \langle f(X)\rangle$ in $\mathbb{Z}_q[X]/\langle f(X)\rangle$. So the kernel of h is $\langle 0\rangle$.

We have now shown that h is an isomorphism. Showing $\mathbb{Z}_q[X]/\langle \prod_{i=1}^{n+1} f_i(X)\rangle \cong \mathbb{Z}_q[X]/\langle f_{n+1}(X)\rangle \times \mathbb{Z}_q[X]/\langle \prod_{i=1}^{n} f_i(X)\rangle$. By assumption 3 we have that this again is isomorphic to $\prod_{i=1}^{n+1} \mathbb{Z}_q[X]/\langle f_i(X)\rangle$. Proving that if the statement holds for $k = n$, it also holds for $k = n + 1$, thus completing the proof. □

3.1.1 The algorithm

The goal of the Number Theoretic Transform in this paper is to perform multiplication of polynomials fast over R_q. The idea is to split $f(X)$ into coprime factors of small degree, and use the isomorphism that then exists because of Theorem 2. Assume $f(X) = \prod_{i=1}^{k} f_i(X)$ where all the different $f_i(X)$ are relatively prime. The algorithm takes in two polynomials a, b in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ and computes the product. The steps are as follows:

1. Compute $a_i \equiv a \mod f_i(X)$ and $b_i \equiv b \mod f_i(X)$
   Through this step a_i and b_i end up being polynomials of small degree.
2. Compute $c_i \equiv a_i b_i \mod f_i(X)$
   This goes fairly fast as both a_i and b_i are polynomials of small degree.
3. Use the inverse function of the isomorphism to compute $c \mod f(X)$ from all the $(c_1, c_2, ...c_n)$

To see how the Number Theoretic Transform works we will give an example.

Example 1: For the arithmetic to be simple, we will work on the ring $\mathbb{Z}_5[X]/\langle X^2+1\rangle$. In $\mathbb{Z}_5[X]$ we have $X^2+1 = (X-2)(X-3)$ and $\langle(X-2)\rangle+\langle(X-3)\rangle = \mathbb{Z}_5[X]$. In both $\mathbb{Z}_5[X]/\langle X^2+1\rangle$ and in $\mathbb{Z}_5[X]/\langle X-2\rangle$ and $\mathbb{Z}_5[X]/\langle X-3\rangle$ the computations are fairly simple.

When $\mathbb{Z}_5[X]/\langle X^2+1\rangle$ we have that $aX^2 \equiv -a \mod (X^2+1)$. In $\mathbb{Z}_5[X]/\langle X-3\rangle$ we have $aX \equiv 3a \mod (X-3)$, and in $\mathbb{Z}_5[X]/\langle X-2\rangle$ we have $aX \equiv 2a \mod (X-2)$.

First we define the isomorphism and it's inverse:

$$\begin{array}{rcl} h : \mathbb{Z}_5[X]/\langle x^2+1\rangle & \to & \mathbb{Z}_5[X]/\langle X-2\rangle \times \mathbb{Z}_5[X]/\langle X-3\rangle \\ p(X)+\langle f(X)\rangle & \mapsto & (p(X)+\langle X-2\rangle, p(X)+\langle X-3\rangle) \\ a(X)(3-X)+b(X)(X-2) & \leftarrow\!\shortmid & (a(X)+\langle X-2\rangle, b(X)+\langle X-3\rangle) \end{array}$$

In the ring $\mathbb{Z}_5[X]/\langle X^2+1\rangle$ all polynomials are on the form $a_0 + a_1X$ where $a_0, a_1 \in \mathbb{Z}_5$. We will use the steps in the number theoretic transform to compute the multiplication of two polynomials. We will also compute the multiplication directly in $\mathbb{Z}_5[X]/\langle X^2+1\rangle$ to check that NTT indeed computes the multiplication correctly.

First we pick two random polynomials $a(X)$ and $b(X)$ in $\mathbb{Z}_5[X]/\langle X^2+1\rangle$. Which will be on the form $a(X) = a_0 + a_1X$ and $b(X) = b_0 + b_1X$. We then go through the steps of NTT:

1. $a_0 + a_1X \equiv a_0 + 2a_1 \mod (X-2)$

   $a_0 + a_1X \equiv a_0 + 3a_1 \mod (X-3)$

   $b_0 + b_1X \equiv b_0 + 2b_1 \mod (X-2)$

   $b_0 + b_1X \equiv b_0 + 3b_1 \mod (X-3)$

2. $(a_0 + 2a_1)\cdot(b_0 + 2b_1) = a_0b_0 + 2(a_0b_1 + a_1b_0) + 4a_1b_1$

   $(a_0 + 3a_1)\cdot(b_0 + 3b_1) = a_0b_0 + 3(a_0b_1 + a_1b_0) + 9a_1b_1$

   $\equiv a_0b_0 + 3(a_0b_1 + a_1b_0) + 4a_1b_1$

3. $h^{-1}(a_0b_0 + 2(a_0b_1 + a_1b_0) + 4a_1b_1, a_0b_0 + 3(a_0b_1 + a_1b_0) + 4a_1b_1)$

   $=(3-X)(a_0b_0 + 2(a_0b_1 + a_1b_0) + 4a_1b_1)$

   $+ (X-2)(a_0b_0 + 3(a_0b_1 + a_1b_0) + 4a_1b_1)$

   $=3(a_0b_0 + \cancel{2(a_0b_1 + a_1b_0)} + 4a_1b_1) - X(\cancel{a_0b_0} + 2(a_0b_1 + a_1b_0) + \cancel{4a_1b_1})$

   $- 2(a_0b_0 + \cancel{3(a_0b_1 + a_1b_0)} + 4a_1b_1) + X(\cancel{a_0b_0} + 3(a_0b_1 + a_1b_0) + \cancel{4a_1b_1})$

   $=\underline{a_0b_0 + 4a_1b_1 + (a_0b_1 + a_1b_0)X}$

To see that it indeed is $a(X)b(X)$, we will compute it directly over $\mathbb{Z}_5[X]/\langle X^2+1\rangle$ as well:

$$\begin{aligned}(a_0 + a_1X)(b_0 + b_1X) &= a_0b_0 + a_0b_1X + a_1b_0X + a_1b_1X^2\\ &\equiv a_0b_0 - a_1b_1 + a_0b_1X + a_1b_0X\\ &\equiv \underline{a_0b_0 + 4a_1b_1 + (a_0b_1 + a_1b_0)X}\end{aligned}$$

This result is the same as when we multiplied using NTT.

Chapter 4

Running time

We have now seen that the NTT algorithm works. It computes the multiplication of two polynomials correctly. Now we will look at how fast it computes the multiplication, and whether there is reason to believe it is faster or not. First we will look at how fast just normal multiplication is, and then how fast multiplication using the NTT algorithm is.

4.1 Multiplication over $\mathbb{Z}_q[X]/\langle f(X)\rangle$

Let $f(X) = \sum_{k=0}^{N} f_k X^k$. Polynomials in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ are then polynomials of degree less than or equal to $N-1$, where we have the relation

$$X^N = -\sum_{k=0}^{N-1} f_k X^k \tag{4.1}$$

Normal, not NTT, multiplication of two polynomials $a(X)$ and $b(X)$ in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ can be divided into two steps. First multiplying in $\mathbb{Z}_q[X]$, and then using the relation (4.1) so that the result ends up having degree less than N. Multiplication of $a(X)$ and $b(X)$ in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ is computed like this

$$a(X) \cdot b(X) = \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} a_k b_j X^{k+j}$$

The number of multiplications performed in this step is N^2. Then we use the relation (4.1) so that the result have degree less than N. The number of multiplications here is dependent on this relation. In the case where $f(X) = X^N + 1$, which we will use in this paper, this step does not use any additional multiplication, only additional subtractions. For the general case of multiplication, without a specific modulus

polynomial $f(X)$, we can look at multiplication in $\mathbb{Z}_q[X]/\langle f(X)\rangle$ as using $\mathcal{O}(N^2)$ multiplications.

4.2 Multiplication using NTT

Let $f(X) = \prod_{i=0}^{n} f_i(X)$. We want to determine how many multiplications that are performed when multiplying two polynomials a and b in $\mathbb{Z}_q[X]/\langle f(X)\rangle$. Multiplication using NTT is divided into three steps.

1. Compute $a_i \equiv a \mod f_i(X)$ and $b_i \equiv b \mod f_i(X)$
2. Compute $c_i \equiv a_i b_i \mod f_i(X)$
3. Use the inverse function of the isomorphism to compute $c \mod f(X)$ from all the $(c_1, c_2, ...c_n)$

To see the running time of the whole NTT process we can look at the three steps and then add the running time of each step together. When we now describe the running time we compare the polynomials to a vector. Here the vector corresponding to the polynomial, $a(X)$, are a vector of length $\deg a(X)$, where the inputs in the vector are the coefficients in the polynomial.

The first step of NTT is a reduction modulo all the $f_i(X)$. This reduction is linear, and can be compared to using a $\deg f_i(X) \cdot \deg f(X)$ matrix multiplied by the vector corresponding to each of the polynomials a and b. For each of the polynomials we then end up with $\mathcal{O}(\deg f_i(X) \cdot \deg f(X))$ multiplications for each of the $f_i(X)$. This will be performed for all the different $f_i(X)$, so we get $\mathcal{O}(\sum_{i=1}^{n} \deg f(X) \cdot \deg f_i(X))$ multiplications. This step is performed on both polynomials, so all together this step costs $\mathcal{O}(2 \cdot \sum_{i=1}^{n} \deg f(X) \cdot \deg f_i(X))$ multiplications.

Step 2 of the NTT algorithm multiplies each of the n new polynomials. For each of the polynomials the amount of multiplications are $\mathcal{O}((\deg f_i(X))^2)$. All together this step has $\mathcal{O}(\sum_{i=1}^{n}(\deg f_i(X))^2)$ multiplications.

The last step uses the inverse function. This step is very similar to the first step just the other way around. Where the first step can be compared to a $\deg f_i(X) \cdot \deg f(X)$ matrix multiplied with the vector corresponding to the two polynomials, for each of the $i \in \{1, ..., n\}$. This step can be compared to a $\deg f(X) \cdot \deg f_i(X)$ matrix multiplied with each of the c_i. So the amount of multiplications performed in this step will be $\mathcal{O}(\sum_{i=1}^{n} \deg f_i(X) \cdot \deg f(X))$.

The number of multiplications performed all together is then $\mathcal{O}(3 \cdot \sum_{i=1}^{n} \deg f_i(X) \cdot \deg f(X) + \sum_{i=1}^{n}(\deg f_i(X))^2)$. The general case of multiplying, stated in Section 4.1, uses $\mathcal{O}((\deg f(X))^2)$ multiplications. We have that $\sum_{i=1}^{n} \deg f_i(X) = \deg f(X)$. So the number of multiplications performed in the NTT version is

$$\mathcal{O}(3 \cdot \deg f(X) \cdot \deg f(X) + \sum_{i=1}^{n}(\deg f_i(X))^2) = \mathcal{O}((\deg f(X))^2)$$

which is the same as just normal multiplication. All of these estimates are not very specific or accurate, but give us an idea that the general case of NTT is not necessarily much faster than just normal multiplication. As seen in this section, the most costly part of the NTT algorithm is the first and last step, the forward and inverse NTT. So in order for NTT to be advantageous, we need the first and last step to be more efficient. We will therefore now look at some improvements that can help make the NTT algorithm faster.

4.3 Improvements

4.3.1 Deliver polynomials in NTT version

The most costly part of the general case of NTT is the forward and invers NTT, first and last step. Computing the multiplication in NTT version is not as costly. One

improvement that can be done to the NTT algorithm, when implementing it to the NTRU cryptosystem, is to send the polynomials in NTT version. When computing the secret and public key in Algorithm 7, we can send it in NTT version. The same can be done when sending the ciphertext after decrypting the message in Algorithm 2.

4.3.2 When N is a power of two

When N is a power of two we can use a trick in the forward and inverse NTT. We will now look at this. First we need a definition.

Definition 2: Let n be a positive integer. An element ω is a primitive n-th root of unity if $\omega^n = 1$ and $\omega^k \neq 1$ for $k < n$.

Forward NTT

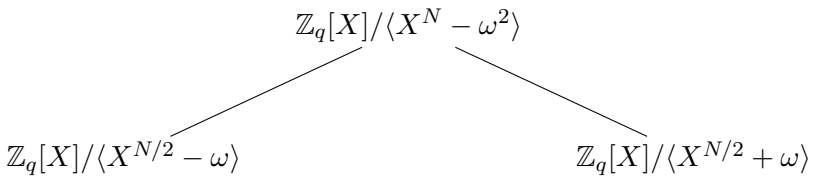


Figure 4.1: NTT first splitting

A modification of the forward NTT is dividing it into different levels and performing a divide and conquer algorithm. Instead of performing forward NTT into polynomials of low degree in one step, we can split it into different levels.

Assume there exist an element $\omega \in \mathbb{Z}_q$ that is a primitive 4-th root of unity, i.e. $\omega^4 = 1$ and $\omega^k \neq 0$ when $k < 4$. From this we know that ω^2 is the primitive 2-nd root of unity, i.e. $\omega^2 = -1$. We can then rewrite our ring using our primitive 4-th root of unity, ω, $\mathbb{Z}_q[X]/\langle X^N + 1 \rangle = \mathbb{Z}_q[X]/\langle X^N - \omega^2 \rangle$. By Theorem 2 we have $\mathbb{Z}_q[X]/\langle X^N - \omega^2 \rangle \cong \mathbb{Z}_q[X]/\langle X^{N/2} - \omega \rangle \times \mathbb{Z}_q[X]/\langle X^{N/2} + \omega \rangle$.

In $\mathbb{Z}_q[X]/\langle X^{N/2} - \omega\rangle$ we have the relation $aX^{N/2} = \omega \cdot a$, and in $\mathbb{Z}_q[X]/\langle X^{N/2} + \omega\rangle$ we have $aX^{N/2} = -\omega \cdot a$. When computing forward NTT of a polynomial, $a(X) = \sum_{i=0}^{N-1} a_i(X)$, this step looks like:

1. multiplying the second half of the coefficients with ω
2. In $\mathbb{Z}_q[X]/\langle X^{N/2} + \omega\rangle$ we subtract these coefficients to the first half of coefficients, in $\mathbb{Z}_q[X]/\langle X^{N/2} - \omega\rangle$ we add these coefficients to the first half of coefficients.

$$\begin{aligned} a(X) \mod (X^{2^{r-1}} - \omega) &= (a_0 + \omega a_{2^{r-1}}) + (a_1 + \omega a_{2^{r-1}+1})X + ... \\ &\quad +(a_{2^{r-1}-1} + \omega a_{2^r-1})X^{2^{r-1}-1} \\ a(X) \mod (X^{2^{r-1}} + \omega) &= (a_0 - \omega a_{2^{r-1}}) + (a_1 - \omega a_{2^{r-1}+1})X + ... \\ &\quad +(a_{2^{r-1}-1} - \omega a_{2^r-1}X)^{2^{r-1}-1} \end{aligned}$$

This step ends up with computing $N/2$ multiplications, $N/2$ subtractions and $N/2$ additions.

Example 2: Let our ring be $\mathbb{Z}_{17}[X]/\langle X^4 + 1\rangle$. In $\mathbb{Z}_{17}$ 4 is a primitive 4-th root of of unity. We can then use this to perform the first splitting, as shown in Figure 4.2.

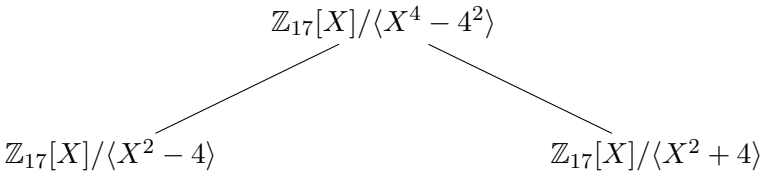


Figure 4.2: NTT using primitive 4-th root of unity

Let $a(X) \in \mathbb{Z}_{17}[X]/\langle X^4 + 1\rangle$. First we multiply the second half of the coefficients with 4.

$$a_2 \cdot 4, \qquad a_3 \cdot 4$$

Then we subtract or add these to the first coefficients resulting in

$$a(X) \mod (X^2 - 4) = (a_0 + 4a_2) + (a_1 + 4a_3)X \tag{4.2a}$$

$$a(X) \mod (X^2 + 4) = (a_0 - 4a_2) + (a_1 - 4a_3)X \tag{4.2b}$$

All together performing $N/2 = 2$ multiplications, additions and subtractions (marked with red).

If we further have primitive 8-th roots of unity, ω_1, we can perform another level of splittings. ω_1^2 would be a primitive 4-th root of unity, which would be used in the first level of splitting. The second level would use different powers of the primitive 8-th root of unity, shown in Figure 4.3. Each of these two new splittings would perform $N/4$ multiplications, additions and subtractions. Since there are two splittings, the number of arithmetic performed in this level is $2 \cdot N/4 = N/2$ multiplications, additions and subtractions.

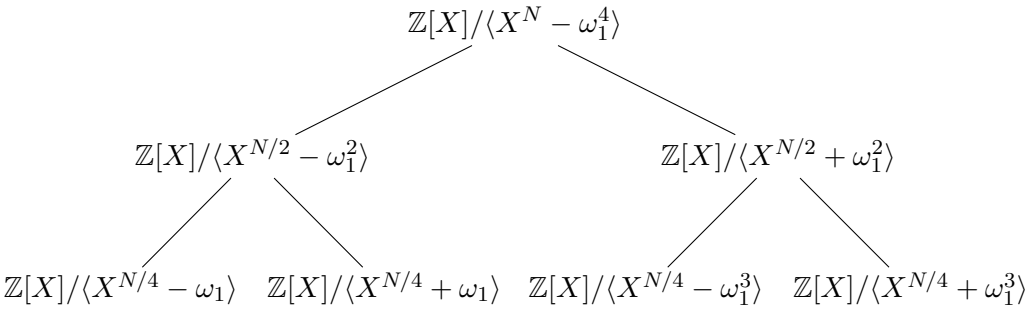


Figure 4.3: NTT with two levels of splittings

Example 3: We can continue on our previous example, Example 2. In $\mathbb{Z}_{17}$ we have primitive 8-th roots of unity. In this example we can use 2, since $2^2 = 4$ which is the primitive 4-th root of unity used in the previous example. The first splitting is already computed, we will now perform the two splitting in the next level.

First we will compute the splitting marked with blue in Figure 4.4. First we will do the multiplication with 2 of the second half of the coefficients in the polynomial.

$$(a_1 + 4a_3) \cdot 2$$

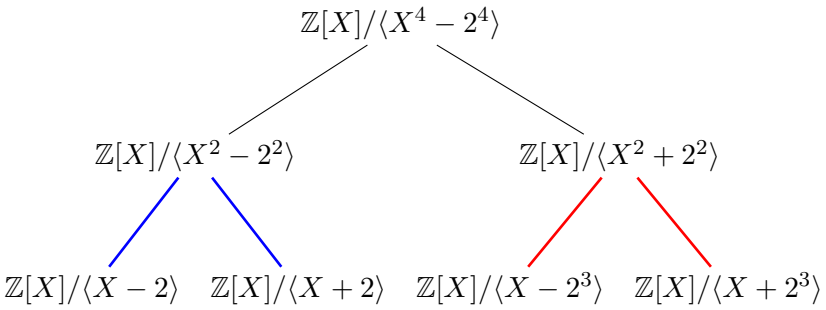


Figure 4.4: NTT with two levels of splittings

Then we subtract or add these to the first half of the coefficients resulting in

$$\begin{aligned} a(X) \mod (X-2) &= (a_0+4a_2)+(a_1+4a_3)\cdot 2 \\ a(X) \mod (X+2) &= (a_0+4a_2)-(a_1+4a_3)\cdot 2 \end{aligned}$$

Then we use the same steps to perform the splitting marked with red in Figure 4.4.

$$(a_1 - 4a_3)\cdot 8$$

$$\begin{aligned} a(X) \mod (X-8) &= (a_0-4a_2)+(a_1-4a_3)\cdot 8 \\ a(X) \mod (X+8) &= (a_0-4a_2)-(a_1-4a_3)\cdot 8 \end{aligned}$$

Both splittings performed $N/4 = 1$ multiplication, addition and subtraction, together resulting in 2 multiplication, addition and subtraction.

If we further would have primitive 16-th roots of unity we could do an additional level of splittings. Each of the 4 new splittings would perform $N/8$ multiplications, additions and subtractions. All together resulting in $N/2$ multiplications, additions and subtractions. If we would have primitive $2N$-th roots of unity, we could perform splittings into linear factors. For every level of splitting using this method, we would perform $N/2$ multiplications, additions and subtractions.

Multiplication in NTT version

The multiplication when the polynomials are in NTT version does not necessarily need to be any faster. We will therefore not write about any improvements of

the multiplication, but will write more specifically how many multiplications and additions or subtractions that will be performed in the multiplication part of the NTT algorithm. Let k be the number of levels we use in the forward NTT. The polynomials in NTT version are all over a ring on the form $\mathbb{Z}_q[X]/\langle X^{N/2^k} - \omega_l \rangle$, where ω_l are different powers of a primitive 2^{k+1}-th root of unity. Multiplication looks like normal multiplication in $\mathbb{Z}_q[X]$, and then using the relation

$$X^{N/2^k} = \omega_l \tag{4.3}$$

Multiplying two polynomials, a and b in $\mathbb{Z}_q[X]/\langle X^{N/2^k} - \omega_l \rangle$ looks like first multiplying all of the coefficients:

$$a \cdot b = \sum_{i=0}^{N/2^k-1} \sum_{j=0}^{N/2^k-1} a_i b_j X^{i+j} \tag{4.4}$$

This step performs $(N/2^k)^2$ multiplications. Then we use the relation (4.3) so that the degree of the result is less than $N/2^k$.

$$a_i b_j X^{i+j} = \omega_l a_i b_j X^{i+j-N/2^k}, \quad \text{when } i+j \geq N/2^k$$

To see how many extra multiplications that are performed during the implementation of this relation, we need to know in how many instances $i + j \geq N/2^k$. When $i = 0$, then $i + j \ngeq N/2^k$. When $i > 0$, then $i + j \geq N/2^k$ in half of the instances. Resulting in $((N/2^k)^2 - N/2^k)/2$ extra multiplications. The number of multiplications performed all together then ends up being $(N/2^k)^2 + ((N/2^k)^2 - N/2^k)/2$.

The number of additions performed in Equation (4.4) is $(N/2^k)^2$. When we use the relation, we do not necessarily add any extra additions or subtractions. When we are implementing this we can perform these two steps simultaneously, and therefore the addition or subtraction would just be performed in front of a different power of X.

For instance assume $a(X) \cdot b(X) = c_1 X + c_2 X^{1+N/2^k}$. Using the relation we would end up with $a(X) \cdot b(X) = (c_1 + \omega_l c_2)X$. The addition is then just moved in front of X instead of $X^{1+N/2^k}$. The number of additions or subtractions performed all together for each of the polynomials is $(N/2^k)^2$.

All of the multiplications and additions will be performed for all the 2^k polynomials. So the number of multiplications performed for all polynomials will be

$$N^2/2^k + N^2/2^{k+1} - N/2$$

The number of additions performed for all the polynomials will be

$$N^2/2^k$$

Inverse NTT

When computing the inverse NTT, we also divide it into the same levels as in forward NTT, shown in Figure 4.1 and 4.3. To give a picture of how this works we need to define how a merging works.

First we define how NTT works when we have just one level of forward and inverse NTT. Let ω be the primitive 4-th root of unity, and the splitting for forward NTT be as shown in Figure 4.1. Let $a_{-\omega} \in \mathbb{Z}_q[X]/\langle X^{N/2} - \omega \rangle$ and $a_{+\omega} \in \mathbb{Z}_q[X]/\langle X^{N/2} + \omega \rangle$. Merging the two polynomials for inverse NTT looks like

1. Add the two polynomials
2. Subtract the two polynomials, $a_{+\omega} - a_{-\omega}$ and multiply by $-\omega^{-1}$, which is ω when ω is primitive 4-th root of unity. In the end we multiply by $X^{N/2}$. (The multiplication by $X^{N/2}$ is just placing the result as the coefficients of the last half of the new polynomial, and does not count as a multiplication when computing the running time.)

All together we end up with $N/2$ additions, subtractions and multiplications. This merging results with a superfluous factor of 2. In the end we will therefore need to multiply by 2^{-1}.

Example 4: We will use the same polynomial as in Example 2, resulting with (4.2a) and (4.2b). That way we can observe that we actually end up with twice polynomial that we started the forward NTT with.

First we add the two polynomials

$$((a_0+4a_2)+(a_0-4a_2)) + ((a_1+4a_3)+(a_1-4a_3))X = 2a_0 + 2a_1X$$

Then we subtract the two polynomials

$$((a_0-4a_2)-(a_0+4a_2)) + ((a_1-4a_3)-(a_1+4a_3))X = -8a_2 - 8a_3X$$

Then multiplying by $-4^{-1} = 4$ and X^2

$$4\cdot(-8)a_2X^2 + 4\cdot(-8)a_3X \cdot X^2 = 2a_2X^2 + 2a_3X^3$$

All together the result of this merging ends up being $2a_0+2a_1X+2a_2X^2+2a_3X^3$, which is twice the polynomial used in the forward NTT. To finish the merging we will then multiply by $2^{-1} = 9$

$$9\cdot 2a_0 + 9\cdot 2a_1X + 9\cdot 2a_2X^2 + 9\cdot 2a_3X^3 = a_0 + a_1X + a_2X^2 + a_3X^3$$

During this merging we have performed $N/2 = 2$ additions, subtractions and multiplications and $N = 4$ multiplication for the finishing part.

If we have performed several levels of splittings in the forward NTT, we will also have several levels of mergings in the inverse NTT. In the instance where we have two levels of forward NTT using a primitive 8-th root of unity, illustrated in Figure 4.3, we will then have two levels of mergings in the inverse NTT. When ω_1 is the primitive 8-th root of unity used in the splitting, we will multiply by $-\omega_1^{-1} = \omega_1^3$ in the merging. The two mergings in the first level of inverse NTT would perform $N/4$ additions, subtractions and multiplications. This would be performed for both mergings, resulting in a total of $N/2$ additions, subtractions and multiplications. For every merging we would end up with a superfluous factor of 2. In the end of inverse NTT we would therefore need to multiply by 2^{-2}.

Example 5: We will now use the same example as in Example 3. We compute the blue merging first. First we add the polynomials

$$((a_0+4a_2) + (a_1+4a_3)\cdot 2)+((a_0+4a_2) - (a_1+4a_3)\cdot 2) = 2\cdot(a_0+4a_2)$$

Then we subtract the polynomials

$$((a_0{+}4a_2) - (a_1{+}4a_3) \cdot 2){-}((a_0{+}4a_2) + (a_1{+}4a_3) \cdot 2) = -4 \cdot (a_0{+}4a_2)$$

Then multiply by $-2^{-1} = 2^3 = 8$ and X

$$8{\cdot}(-4 \cdot (a_0{+}4a_2))X = 2 \cdot (a_0{+}4a_2)X$$

resulting in $2 \cdot (a_0{+}4a_2) + 2 \cdot (a_0{+}4a_2)X$ which is twice (4.2a). This merging performed one addition, one subtraction and one multiplication. The merging marked with red in Figure 4.4 will similarly perform one addition, subtraction and multiplication, and end up being (4.2b) times two. All together this level will perform $N/2 = 2$ additions, subtractions and multiplication.

After the two mergings in the first level, we would perform the merging in the second level, which is shown in Example 4. The only difference is that we now start by merging two times (4.2a) and (4.2b), resulting in four times the polynomial in forward NTT. We will then have to multiply by $2^{-2} = 13$.

$$13{\cdot}4a_0 + 13{\cdot}4a_1X + 13{\cdot}4a_2X^2 + 13{\cdot}4a_3X^3 = a_0 + a_1X + a_2X^2 + a_3X^3$$

We end up with $N/2 \cdot 2 = 4$ additions, subtractions and multiplications, plus an additional $N = 4$ multiplications to finish the inverse NTT.

If we performed three levels of splittings in forward NTT, we would then have three levels of mergings. The first level of merging would then be four mergings that all use $N/8$ additions, subtractions and multiplications, all together performing $N/2$ additions, subtractions and multiplications. The second and third level of merging would also perform $N/2$ additions, subtractions and multiplications. In the end we would get an additional N multiplications to finish the inverse NTT. This would be similar if we would have several mergings. Each level would perform $N/2$ additions, subtractions and multiplications, and in the end we would need an additional N multiplications to finish the inverse NTT.

Summary

The running time of multiplication with the NTT algorithm is dependent on the running time of NTT forward, multiplication in NTT version and inverse NTT. Forward and inverse NTT is dependent on how many levels we perform the splitting and merging. Let k be the number of levels of splitting and merging. The number of multiplications, additions and subtractions performed in the forward NTT algorithm would then be $N/2 \cdot k$, for each polynomial. The multiplication would be $N^2/2^k + N^2/2^{k+1} - N/2$ multiplications and $N^2/2^k$ additions or subtractions. The inverse NTT would perform $N/2 \cdot k$ multiplications, additions and subtraction, plus an additional N multiplications.

Chapter 5

Implementation of NTT

5.1 Choice of ring

When we have implemented the algorithm, we have implemented for several values of q and N. In order to get a good impression of how much faster the NTT algorithm can be, compared to normal multiplication, we want to use a prime q such that $q-1 = 2^r \cdot a$, for some $r \in \mathbb{N}$. This is because in $\mathbb{Z}_q$ we would then have primitive 2^r-th roots of unity. We could then use one of these primitive 2^r-th roots of unity to perform $r-1$ levels of splittings (mergings) in the forward (inverse) NTT. In the results written about in this paper, we have used a q so that $q-1 = 2^{12} \cdot 3$. When using this q, we could see how much faster NTT could be as N got very large.

5.2 Setting of testing

The code[1] was tested on a $2,3$ GHz 8-Core Intel Core i9 processor. We chose to do the multiplication over random polynomials of length N. The code is tested on different sizes of N so that we could see the difference in runtime as N got bigger. For all the different N that were tested, we performed multiplication 40 times over random polynomials, and divided the runtime by 40. In that way we could get an estimate of runtime that were closer to what would be expected. We also implemented code for normal multiplication.

When testing the NTT version, we also multiplied the polynomials with normal multiplication. For every multiplication, we computed the runtime as well as

[1] Source code at https://github.com/annabakkebo/Master_NTT. We ran the main function from main.c from the branch "master".

checking that the result of the multiplication was the same using both NTT and normal multiplication. It was the same random polynomials that were tested in both the normal multiplication and the NTT multiplication.

5.3 Forward NTT to factors of degree 2

Theorem 3: Let $a(X)$ and $b(X)$ be random polynomials in $\mathbb{Z}_q[X]/\langle X^N + 1\rangle$. Let $N = 2^r$. Using the NTT algorithm for multiplication, as described in section 4.3.2, multiplication would be most optimal if the polynomials at the lowest level are polynomials of degree 2.

Proof. The number of multiplications, subtractions and additions performed all together are equal to:

- the number performed in NTT forward of both polynomials
- $+$ the number of arithmetic performed when multiplying at the lowest level
- $+$ the number of arithmetic performed at NTT inverse for the solution polynomial

We then want the sum of these to be as small as possible.

As stated in section 4.3.2 we can compute forward and inverse NTT using 2^{r-1} multiplications and 2^r additions or subtractions at each level.

The number of multiplications at the lowest level is $(\deg(f_i(X)))^2 + (\frac{\deg(f_i(X)}{2})^2$ times the number of polynomials at the lowest level if and only if $\deg(f_i(X)) > 1$. If $\deg(f_i(X)) = 1$, the number of multiplications at the lowest level is N.

Let k be the number of levels performed. We then have the relation $\deg(f_i(X)) = \frac{\deg(f(X)}{2^k}$

Forward NTT ends up being:

$$2 \cdot 2^{r-1} \cdot k$$

Multiplication will be

$$(2^{r-k})^2 + ((2^{r-k})^2 - 2^{r-k})/2) \cdot 2^k = (2^{r-k+1} + 2^{r-k} - 1)\dot{2}^{r-1}$$

Inverse NTT ends up being:

$$2^{r-1} \cdot k + 2^r$$

Multiplications performed all together ends up being:

$$(3k + 2^{r-k+1} + 2^{r-k} + 1) \cdot 2^{r-1}$$

If $k = r$, i.e. we perform NTT into linear factors, we get that the number of multiplications performed all together is

$$(3r + 4) \cdot 2^{r-1}$$

If $k = r - 1$, i.e. the polynomials at the lowest level has degree 2, we get that the number of multiplications performed all together is

$$(3r + 4)2^{r-1}$$

Which is the same number of multiplications as when $k = r$. If $k = r - 2$ we get that the amount of multiplications performed all together is

$$(3r + 7)2^{r-1}$$

If $k = r - 3$ we get that the amount of multiplications performed all together is

$$(3r + 16)2^{r-1}$$

We get the lowest number of multiplications if $k = r$ or $k = r - 1$.

The number of additions or subtractions are:

Forward NTT:

$$2 \cdot 2 \cdot 2^{r-1} \cdot k = 2 \cdot 2^r \cdot k$$

Additions or subtractions at the lowest level:

$$(\frac{2^r}{2^k})^2 \cdot 2^k = 2^{2r-k}$$

Inverse NTT:

$$2 \cdot 2^{r-1} \cdot k = 2^r \cdot k$$

Additions or subtractions performed all together ends up being:

$$3k \cdot 2^r \cdot k + 2^{2r-k} = (3k + 2^{r-k})2^r$$

If $k = r$, i.e. we perform NTT into linear factors, we get that the number of multiplications performed all together is

$$(3r + 2^{r-r})2^r = (3r + 1) \cdot 2^r$$

If $k = r - 1$ the number of additions or subtractions performed all together is

$$(3r - 3 + 2^{r-r+1})2^r = (3r - 1)2^r$$

If $k = r - 2$ the number of additions or subtractions performed all together is

$$(3r - 6 + 2^{r-r+2})2^r = (3r - 2)2^r$$

If $k = r - 3$ the number of additions or subtractions performed all together is

$$(3r - 9 + 2^3)2^r = (3r - 1)2^r$$

Which is the same as when $k = r - 1$. If $k = r - 4$ the number of additions or subtractions performed all together is

$$(3r - 12 + 2^4)2^r = (3r + 4)2^r$$

We perform the lowest number of additions or subtractions if $k = r - 2$. So now we need to see how many multiplications and additions or subtractions are performed when $k = r - 1$ and $k = r - 2$ to see at what level we perform the lowest number of multiplications, additions and subtractions. When $k = r - 1$ the number of multiplications and additions or subtractions are:

$$(3r + 4)2^{r-1} + (3r - 1)2^r = (9r + 2)2^{r-1}$$

When $k = r - 2$ the number of multiplications and additions or subtractions are:

$$(3r + 7)2^{r-1} + (3r - 2)2^r = (9r + 3)2^{r-1}$$

Which gives that the lowest number of multiplications, additions and subtractions is when $k = r - 1$, i.e. the polynomials at the lowest level are 2. □

Theorem 3 gives us that the NTT algorithm is fastest when the degree at the lowest level is 2. When we tested the code, we therefore decided that the polynomials at the lowest level were of degree 2, if possible.

5.4 The code

We have implemented the multiplication in c-code. It is all posted on github and we have devided the method into different files. We have one file called params.h where we can change the variable for the prime modulus, q, and can change the different values for N through the function `void set_N(long power)`. Further we have a file called NTT.h and NTT.c which contains the function for forward and inverse NTT. These functions are also added in Appendix B. They take in pointers to the polynomials that we want to perform forward and inverse NTT on, and changes the polynomials so that they are in NTT version. The reason why we use pointers instead of the actual polynomials, is that in doing so we use less memory for these functions. We use precomputed roots of unity for forward and inverse NTT, which are precomputed through the function `void initiate(long power, long level)`. The `long power` refers to the power of 2 we want N to be, and the `long level` refers to how many levels of NTT forward and inverse we want. In this function we also precompute what $2^{-\text{levels}}$ is.

The functions for multiplications are all stored in the file multiplication.h and multiplication.c. The functions for normal multiplications is given in Appendix A. The function takes in a pointer to the two polynomials that will be multiplied, and a

pointer to the polynomial where the result will be stored. The functions for NTT multiplication, given in Appendix C, takes in:

- pointer to the two polynomials in NTT version
- pointer to the polynomial where the result will be stored
- an array of the roots of unity that will be used in the multiplication
- the size of the polynomials in the lowest level
- how many polynomials there are at the lowest level

The function updates the polynomial where the result will be stored to be the product in NTT version.

5.5 How fast was it

When testing this code we have used the measurement `clock_t`, which is a measurement that counts the number of clock ticks elapsed since the program was launched. We have taken samples of `clock_t` before and after each multiplication, and subtracted them to see how many clock ticks elapsed during the multiplications. The numbers in Table 5.1 and in the plot in Figure 5.1 are both measured in clock ticks.

Figure 5.1 shows the running time, measured in clock ticks, of the NTT algorithm compared to normal multiplication. The plot is given as a logarithmic graph. This is so that we can get a better overview of how fast the two algorithm where. The x-axis shows the value of N and the y-axis shows how long time was used to compute the multiplication. We can see that as N got bigger the runtime grew much more for the normal multiplication than the NTT multiplication.

The graph shows N from $2^3 = 8$ to $2^{16} = 65\,536$. When N is 2^{16} the NTT algorithm computes the multiplication roughly $1\,000$ times faster than the normal

N	Normal multiplication	NTT multiplication
2^3	$\approx 2^{0.38}$	$\approx 2^{0.26}$
2^4	$\approx 2^{1.72}$	$\approx 2^{1.17}$
2^5	$\approx 2^{3.51}$	$\approx 2^{2.30}$
2^6	$\approx 2^{5.45}$	$\approx 2^{3.43}$
2^7	$\approx 2^{7.46}$	$\approx 2^{4.48}$
2^8	$\approx 2^{9.47}$	$\approx 2^{5.58}$
2^9	$\approx 2^{11.51}$	$\approx 2^{6.66}$
2^{10}	$\approx 2^{13.48}$	$\approx 2^{7.78}$
2^{11}	$\approx 2^{15.46}$	$\approx 2^{8.95}$
2^{12}	$\approx 2^{17.48}$	$\approx 2^{10.08}$
2^{13}	$\approx 2^{19.44}$	$\approx 2^{11.09}$
2^{14}	$\approx 2^{21.48}$	$\approx 2^{12.30}$
2^{15}	$\approx 2^{23.47}$	$\approx 2^{13.70}$
2^{16}	$\approx 2^{25.47}$	$\approx 2^{15.22}$

Table 5.1: Running time using normal multiplication and NTT

multiplication. While when N is 2^3 the NTT multiplication is not noticeably faster than normal multiplication. The graph and table just shows how there is a difference in running time as N grows. It is probably not preferred to have an N as high as $2^{16} = 65\,536$. The normal multiplications would then spend roughly 1.5 minutes, while the NTT multiplication would still be under a second. The reason why we have decided to use an N that large in our testing, is that it gives a picture of how much faster the NTT multiplication is as N grows.

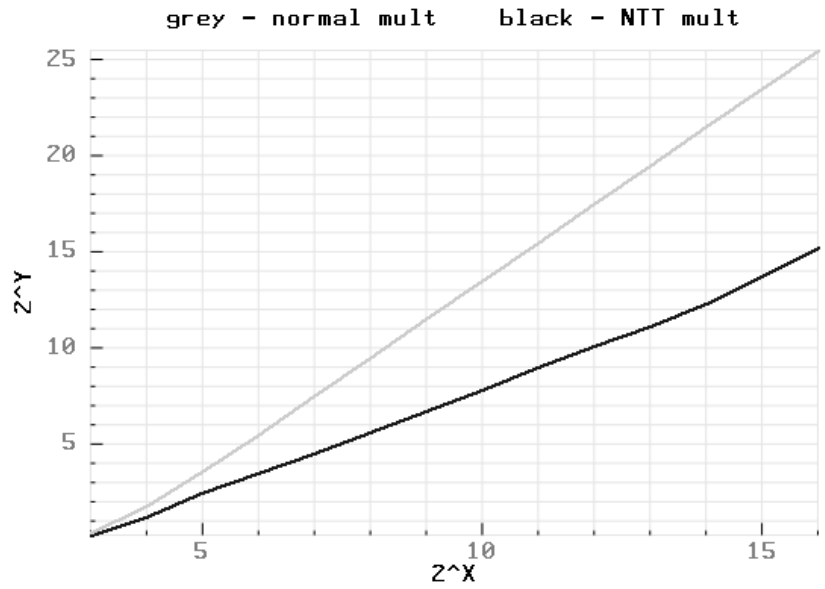


Figure 5.1: Plot for running time given in logarithmic values

Chapter 6

Commitment scheme

6.1 Commitment schemes

We will now look at a commitment scheme that uses multiplication over R_q as part of the arithmetic, but first we will look at what a commitment scheme is.

Definition 3: A commitment scheme consist of three different algorithms: Commitment key generation (CK), commit (C) and open (O)

- The commitment key generation algorithm, CK, takes no input and outputs a commitment key ck.
- The commit algorithm, C, takes as input the commitment key ck, the message m and a randomness r and outputs the commitment c and an opening t.
- The open algorithm, O, takes as input the commitment key ck, the commitment c, the message m, opening t and outputs either 0 or 1

To be a valid commitment scheme we want $O(ck, c, m, t) = 1$.

To give some context as to why we would want a commitment scheme and what purpose it serves: Picture that you have a betting competition with your friend. For instance that you will roll a die, and you are betting at what the number will be. When you are both at the same place you can say, "I bet you will roll a 6 when you roll the die". Your friend can then roll the die and check whether or not you where correct. Now picture that you will do the same, but you are at different places. Now if you say you bet your friend will roll a 6, your friend can trick you and just say that he did not roll a 6. In this situation we can use a commitment scheme. If

you commit 6 through a commitment scheme, your friend can not see what you committed, what you bet on. When he says what he rolled, he does not know what you bet and therefore can not trick you. Then after he says what he rolled, he can open the message and check that you committed what you said you did.

6.2 Hiding and binding

In a commitment scheme we have two properties that we want the scheme to have: Hiding and binding. If a commitment scheme has a hiding property, that means when the receiver gets the commitment he can not determine what the message is. He needs the opening to be able to discern what the message is. If a commitment scheme has a binding property, that means that the commitment can not open to different values. The committer can not change his message after he has committed the message. The commitment can only be opened to the message that was committed.

6.3 The commitment scheme

We will now introduce a commitment scheme written about in the article "More Efficient Commitments from Structured Lattice Assumptions" by Baum, Damgård, Lyubashevsky, Oechsner and Peikert (2016). The reason why this commitment scheme is interesting for us is that most of the arithmetic is used over the ring R_q. We will attempt to use NTT on this scheme and look at whether we can compute the commitments faster with NTT. The parameters used in the commitment scheme is given in the Table 6.1, and are the same as in the paper just mentioned.

Notation	Description
R	The ring $\mathbb{Z}[X]/\langle X^N + 1\rangle$ over which the norms of the vectors are defined
R_q	The ring $\mathbb{Z}_q[X]/\langle X^N + 1\rangle$ over which most of the computations are done
q	Prime q defining the modulus in R_q
k	Width of the commitment matrices, over R_q
l	Dimension of the message space, over R_q
d	Height of the commitment matrix A_1, over R_q
β	Norm bound for honest prover's randomness in ℓ_∞-norm
S_β	Set of all elements in R_q with ℓ_∞-norm at most β
$\mathcal{C}$	Set of all elements in R_q whose ℓ_∞-norm is 1 and ℓ_1 norm is κ
κ	The maximum ℓ_1-norm of the element in $\mathcal{C}$
$\overline{\mathcal{C}}$	The set of differences $\mathcal{C} - \mathcal{C}$ excluding 0
σ	$\sigma = 11 \cdot \kappa \cdot \beta \cdot \sqrt{kN}$, standard deviation

Table 6.1: Notation for the commitment scheme

6.3.1 Key generation, commit and open

Algorithm 4, 4 and 5 defines the key generation, commit and opening of the commitment scheme. The key generation generates two matrices, where part of the matrices are generated randomly from our polynomial ring R_q. The commit algorithm, Algorithm 4, uses the matrices generated by the key generation, the message and a random vector and outputs the ciphertext. The polynomials in the random vector is picked uniformly from S_β, and the message is a vector, of length l, whose elements are polynomials in R_q. If the committer is honest, he outputs the message, the randomness used, and 1 as the opening t.

To check whether the opening is valid, the open algorithm, Algorithm 5, checks

$$f \cdot \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + f \cdot \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix}$$

and that $||r_i||_2 \leq 4\sigma\sqrt{N}$. If the committer would be an honest committer, this f would be 1, and the $\vec{r}$ and $\vec{m}$ would be the ones used in the commitment algorithm. It is also easy to see that this would be a valid opening for the commitment scheme.

Algorithm 4: Key generation

Output: Two matrices $A_1 \in R_q^{d\times k}$ and $A_2 \in R_q^{l\times k}$

1: $A_1' \xleftarrow{\$} R_q^{d\times(k-d)}$

2: $A_1 := \begin{pmatrix} I_d & A_1' \end{pmatrix}$

3: $A_2' \xleftarrow{\$} R_q^{l\times(k-d-l)}$

4: $A_2 := \begin{pmatrix} 0^{l\times d} & I_l & A_2' \end{pmatrix}$

5: **return** $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$

Algorithm 5: Commit

Input: Commitment key $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$, Message $m \in R_q^l$

Output: Commitment $\vec{c} = \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix}$, Opening $t = (\vec{m} \in R_q^l, \vec{r} \in R_q^k, f \in \overline{\mathcal{C}})$

1 $\vec{r} \xleftarrow{\$} S_\beta^k$

2 $\vec{c} := \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix}$

3 $t := (\vec{m}, \vec{r}, 1)$

4 **return** $\vec{c}$, t

Algorithm 6: Open

Input: Opening $t = (\vec{m} \in R_q^l, \vec{r} \in R_q^k, f \in \overline{\mathcal{C}})$, Commitment key $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$

Output: 1 if the opening is valid, 0 if the opening is not valid

if $(f \cdot \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + f \cdot \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix})$ **then**
  if $||r_i||_2 \leq 4\sigma\sqrt{N}$ **then**
    return 1
else
  return 0

6.4 Hiding and binding property

To look at the hiding and binding property we will first give a short introduction to Module-LWE, Module-Learning With Error, and Module-SIS, Module-Short Integer Solution. Both of these problems are known, and the hiding and binding property of the commitment scheme can be compared to these problems. For a more in depth reasoning to this, look at the article mentioned (Baum et al., 2016).

6.4.1 Module-LWE

The problem of Module-LWE can be divided into two different problems, search Module-LWE and decision Module-LWE. The problem that relates to the hiding property of this commitment scheme, is decision Module-LWE. The core of the problem is, given a matrix A, we want to distinguish A times a small vector $\vec{r}$, from a uniformly random vector. The larger $\vec{r}$ can get, i.e. the larger the set from which $\vec{r}$ is picked from, the harder this problem is. This is due to the fact that the function

$$h(\vec{r}) := A \cdot \vec{r}$$

becomes more of an onto function as $\vec{r}$ can get bigger, which results in the product actually being uniformly random.

In relation to our scheme the hiding property is related to whether $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r}$ looks random or not. Whether we can distinguish this result from just a uniformly random vector. If $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r}$ looks random, or is indistinguishable from a uniformly random vector, then $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix}$ also looks random. Which makes it hard to recover $\vec{m}$. This is related to the Decisional Module-LWE. Assume that the $\vec{r}$ is chosen from the same set of vectors as in the commitment scheme. In the article by Baum et al they write that, if an algorithm $\mathcal{A}$ has advantage ϵ in breaking the hiding property of the commitment scheme, then there exists an algorithm $\mathcal{A}'$ that has probability ϵ in solving the Module-LWE(Baum et al., 2016, p. 11-12). So if Module-LWE becomes very hard to solve, the hiding property of the commitment scheme also becomes hard to break

6.4.2 Module-SIS

The binding property of this commitment scheme is related to another known problem, namely the Module-Short Integer Solution problem. The core of this problem is, given a random matrix B, what is the probability that we can find a short vector $\vec{r}$, so that $B \cdot \vec{r} = \vec{0}$. In this problem the solution becomes harder when we want $\vec{r}$ to be shorter. If our random matrix B is on Hermite Normal Form, this becomes the matrix A_1 given in our commitment scheme. The solution to B, when given in Hermite Normal Form, is the same as when B is uniformly random.

The relation to our commitment scheme is that if we are able to open to two different values, then we are able to solve Module-SIS. Assume we have two different valid openings, $t = (\vec{m}, \vec{r}, f)$ and $t' = (\vec{m'}, \vec{r'}, f')$. Assume that f and f' are invertible,

which they will be because of the set these are picked from (Baum et al., 2016, p. 13). As they both are valid openings, we have that the following equations are true.

$$f \cdot \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + f \cdot \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix}$$
$$f' \cdot \begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r'} + f' \cdot \begin{pmatrix} 0^d \\ \vec{m'} \end{pmatrix}$$

Combining these two equations we end up with the relation.

$$f' \cdot (\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r} + f \cdot \begin{pmatrix} 0^d \\ \vec{m} \end{pmatrix}) = f \cdot (\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot \vec{r'} + f' \cdot \begin{pmatrix} 0^d \\ \vec{m'} \end{pmatrix})$$

Which again gives the equation

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \cdot (\vec{r} \cdot f' - \vec{r'} \cdot f) + f \cdot f' \begin{pmatrix} 0^d \\ \vec{m} - \vec{m}' \end{pmatrix} = \begin{pmatrix} 0^d \\ 0^l \end{pmatrix}$$

Now $f \cdot f'(\vec{m} - \vec{m'}) \neq 0$, because f and f' are invertible, and $\vec{m} \neq \vec{m'}$. Because of this we also have that $(\vec{r} \cdot f' - \vec{r'} \cdot f) \neq 0$. Now we have that $A_1 \cdot (\vec{r} \cdot f' - \vec{r'} \cdot f) = 0^d$, and that $(\vec{r} \cdot f' - \vec{r'} \cdot f)$ is short and nonzero. So $(\vec{r} \cdot f' - \vec{r'} \cdot f)$ would be a solution to Module-SIS in Hermite Normal Form.

6.5 Implementation

When we now have implemented this commitment scheme, our main goal was to determine how fast it was with normal multiplication compared to NTT multiplication. All the polynomials where picked randomly from R_q. We did this because, whether we picked the polynomials from the specific sets or randomly over R_q, the running time for multiplication would be roughly the same. As our goal was to

determine whether we could make the commitment scheme more efficient using NTT, we chose to focus on the dimension of the polynomials and matrices, and not from what subset of R_q we picked the polynomials.

6.5.1 Parameters used

When we have tested the code, we have tested it for random values of k, l and d, which determines the sizes of the commitment martices. For our testing we have used the values $k = 8$, $l = 4$ and $d = 3$. All of these values can easily be changed in the file params.h[1].

To test the running time of the commitment scheme, we have chosen to access memory by stack, which affects the parameters we have chosen. In c-code, which we have used for implementation, we have two different types of memory allocation, stack and heap. Stack has a designated size of storage which we can use, where memory allocation runs very fast. The downside, though, is that there is a limited size of storage. In heap we can store larger size of memory, but the allocation is a bit slower. We have therefore chosen to use stack in our testing, so that memory allocation is fast, but will through this be limited by the storage size. In our testing we have therefore chosen to test for N between $2^2 = 4$ and $2^{13} = 8\,192$, which is a smaller N than we used for testing multiplication, written about in Chaper 5. We have also implemented code for accessing memory through heap, which is posted on github[2] but not written about in this paper.

[1] All files are uploaded to https://github.com/annabakkebo/Master_NTT. To test, and change the parameters of the code used for the commitment scheme written about in this thesis go to branch "commitment-scheme".

[2] The code for this implementation by heap is given in the branch "Commitment_scheme_malloc". The branch "commitment-scheme" uses stack for memory allocation.

6.5.2 Code for the commit algorithm

The implementations, for multiplication of the polynomials, are the same as we referred to in Section 5.4. The implementation for the commit-algorithm, Algorithm 4, is given in the file MatrixMultiplication.h and MatrixMuliplication.c[3]. The functions for commit, using normal multiplication, are added to Appendix D. In Section 5.4, where we talked about the functions for multiplication, we mentioned that we used pointers to the polynomials for multiplication. When we implemented commit, which contains multiplication of a matrix times a vector of polynomials, we also used pointers. This for the same reason, to not use more memory by creating a copy of the matrices and vectors.

The implementation of commit using NTT multiplication is given in Appendix E.

6.6 Results of implementation

The result of our testing is very similar to when we multiplied two polynomials (see Section 5.5). We can observe, from Table 6.2 and Figure 6.1, that as N grew bigger the NTT multiplication got much faster than normal multiplication. When N is $2^{13} = 8\,192$, the NTT multiplication is roughly $2^{8.68}$ times faster. When we tested with multiplication of just one polynomial the NTT computed roughly $2^{8.35}$ times faster. So here there is not much difference. This is not very different from what was expected, due to our k, d and l being fairly small. We would then not gain much time from performing forward NTT of the randomness vector, $\vec{r}$, just once.

We can also observe, from Figure 6.1, that when N is very small, there is barely any difference between committing using normal multiplication and NTT multiplication. This is probably due to the fact that when N is small, NTT is not necessarily much faster. In this case the time to go through all the elements of the matrices and the vector takes longer time compared to what time we save through computing using

[3]Both of these files can be found on the github page mentioned.

N	Normal multiplication	NTT multiplication
2^2	$2^{7.62}$	$2^{7.61}$
2^3	$2^{7.53}$	$2^{7.51}$
2^4	$2^{7.73}$	$2^{7.60}$
2^5	$2^{8.18}$	$2^{7.69}$
2^6	$2^{9.22}$	$2^{7.93}$
2^7	$2^{10.85}$	$2^{8.38}$
2^8	$2^{12.73}$	$2^{9.02}$
2^9	$2^{14.71}$	$2^{9.88}$
2^{10}	$2^{16.76}$	$2^{10.88}$
2^{11}	$2^{18.69}$	$2^{11.83}$
2^{12}	$2^{20.67}$	$2^{12.92}$
2^{13}	$2^{22.67}$	$2^{13.99}$

Table 6.2: Running time using normal multiplication and NTT

NTT multiplication. As the plot is given in logarithmic values we will not detect this difference in the running time.

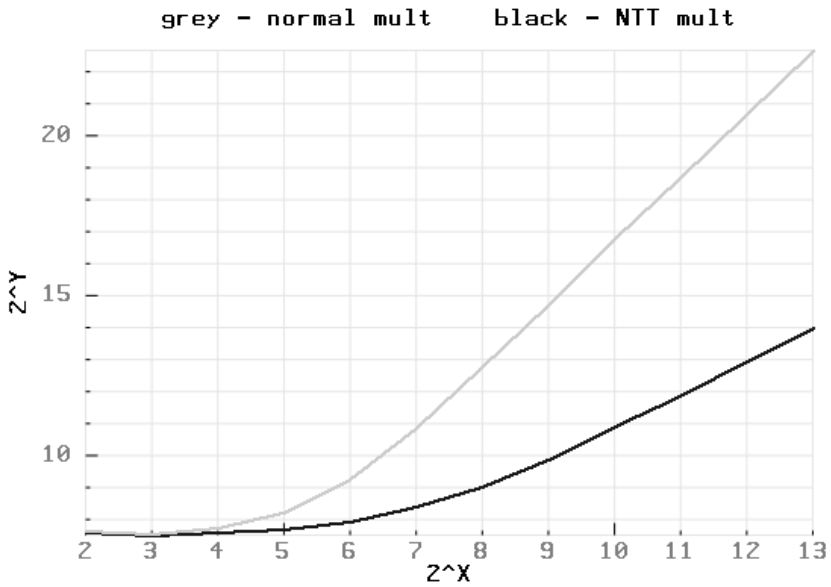


Figure 6.1: Running time using stack in logarithmic values

Chapter 7

Conclusion

In this thesis we have looked at multiplication of polynomials using the Number Theoretic Transform. Our goal with the Number Theoretic Transform was to perform multiplications faster. The motivation for doing this was to be able to multiply polynomials fast, which could help making NTRU and a lattice-based commitment scheme more efficient. We discovered that the general case of NTT was not necessarily faster, but when our ring, $R_q = \mathbb{Z}_q[X]/\langle X^N + 1\rangle$, had a specific structure, we could use a divide and conquer method that would make the NTT multiplication faster. The specific structure of R_q, that we have looked at in this thesis, is:

- q being a prime such that $q - 1 = 2^r \cdot a$
- our modulus polynomial, $f(X) = X^N + 1$, using an N on the form $N = 2^k$

We implemented the NTT multiplication, and normal multiplication for comparison, in c-code. We did this to see if it could improve the efficiency of NTRU and the commitment scheme. In our implementation we tested for different values of N. The results where when N grew bigger, using NTT was much more efficient than normal multiplication.

References

Baum, C., Damgård, I., Lyubashevsky, V., Oechsner, S., & Peikert, C. (2016). *More efficient commitments from structured lattice assumptions.* Cryptology ePrint Archive, Report 2016/997. (https://eprint.iacr.org/2016/997)

Bhattacharya, P. B., Jain, S. K., & Nagpaul, S. R. (1994). *Basic abstract algebra*. Cambridge University Press.

Lyubashevsky, V., & Seiler, G. (2019, May). NTTRU:Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2019*(3), 180-201. Retrieved from https://tches.iacr.org/index.php/TCHES/article/view/8293 doi: 10.13154/tches.v2019.i3.180-201

Appendices

A | Normal multiplication

```
1  /**
2   * Determines what position of the second polynomial that will be
       multiplied by the j-th position of the first
3   * polynomial for the i-th position of the result
4   * @param i integer
5   * @param j integer
6   * @param n Size of the modpol
7   * @return The position of the second polynomial that is multiplied.
8   *
9   */
10 long f(long i, long j, long n) {
11     if (i >= j) {
12         return i - j;
13     } else {
14         return i - j + n;
15     }
16 }
17
18 /**
19  * Normal multiplication modulo x^N+1, the result is stored in result
20  * @param pol1 Pointer to the array of the first polynomial
21  * @param pol2 Pointer to the array of the second polynomial
22  * @param result Pointer to the array where the result is stored
23  * @param n  The degree of the modulo polynomial
24  */
25 void multiplied_normal(long *pol1, long *pol2,long *result, long n){
26     for (long i = 0; i < n; i++) {
27         for (long j = 0; j < n; j++) {
28             long pos2 = f(i, j, n);
29             if (j + pos2 >= n) {
30                 //printf("multipliserer %d og %d og lagrer det i posisjon
       %d (bruker minus)\n",pol1[j],pol2[pos2],i);
31                 result[i] = (result[i] -  pol1[j] * pol2[pos2]) % Q;
32 #if COUNTOPERATIONS==1
33                 Mult_Norm+=1;
34                 AddSub_Norm+=1;
35 #endif
36             } else {
37                 //printf("multipliserer %d og %d og lagrer det i posisjon
       %d\n",pol1[j],pol2[pos2],i);
38                 result[i] = (result[i] + pol1[j] * pol2[pos2]) % Q;
39 #if COUNTOPERATIONS==1
40                 Mult_Norm+=1;
41                 AddSub_Norm+=1;
42 #endif
```

```
43              }
44          }
45      }
46  }
47
48  /**
49   * Normal multiplication modulo x^N+1, the result is stored in result
50   * @param pol1 The first polynomial
51   * @param pol2 The second polynomial
52   * @param result The array where the result is stored
53   * @param n  The degree of the modulo polynomial
54   */
55  void multiplied_normal2(struct pol *pol1, struct pol *pol2,struct pol *
        result, long n){
56      multiplied_normal(pol1->coeffs,pol2->coeffs,result->coeffs,n);
57  }
```

B | NTT forward and inverse

```
1
2 long NTT_forward[NUM_POLYNOMIALS*2]; // Array of the roots of unity that
      will be used for forward NTT
3 long NTT_roots[NUM_POLYNOMIALS]; // Array of roots of unity that is used
      at the lowest level
4
5 /**
6  * Stores the roots of unity that is used for the forward splitting
7  * @param roots List of roots of unity
8  * @param NTT_forward  List where the forward NTT will be stored
9  * @param n  The previous root of unity that is used x^N- w^n
10 * @param move How many positions forward the next step should be stored
11 * @param direction Whether you're splitting to the right or left. Left:
       true, right: false, start with false.
12 * @param start The level where it is started
13 * @param stop  How many levels the splitting should be
14 */
15 void initiate_NTT_forward(long *roots, long *NTT_forward, long n,
16                           long move, bool direction , long start,
17                           long  stop){
18     if (start==stop){
19         return;
20     }
21     else{
22         NTT_forward[0] = roots[n/2];
23         long current = start +1;
24         long next_move= move*2;
25         if(direction){
26             initiate_NTT_forward(roots,
27                                  NTT_forward + next_move,
28                                  n / 2,
29                                  next_move,
30                                  true,
31                                  current,
32                                  stop);
33             initiate_NTT_forward(roots,
34                                  NTT_forward + next_move + 1,
35                                  n / 2 + PRIMITIVE_N / 2,
36                                  next_move + 1,
37                                  false,
38                                  current,
39                                  stop);
40         } else{
41             initiate_NTT_forward(roots,
42                                  NTT_forward + next_move - 1,
```

```
                                    n / 2,
                                    next_move - 1,
                                    true,
                                    current,
                                    stop);
            initiate_NTT_forward(roots,
                                    NTT_forward + next_move,
                                    n / 2 + PRIMITIVE_N / 2,
                                    next_move,
                                    false,
                                    current,
                                    stop);
        }
    }
}

/**
 * Creates list of roots that the multiplication will do modulo when doing
     multiplication after NTTforward
 * @param NTT_forward The NTT_forward list
 * @param level How many levels that will be computed
 * @param NTT_roots The list where the roots will be stored
 */
void initiate_NTT_roots(long *NTT_forward, long level, long *NTT_roots){
    long polynomials=1;
    for (long i=0;i<level-1; i++){
        polynomials=polynomials*2;
    }
    for (long i =0; i<polynomials;i++){
        NTT_roots[i*2]=-NTT_forward[i+polynomials-1];
        NTT_roots[i*2+1]=NTT_forward[i+polynomials-1];
    }
}

/**
 * Updates the array of polynomial to contain the two splitted polynomials
     .
 * The first half of the array will be modulos x^n-w, the second half
     modulos x^n+w
 * @param pol polynomial that will be splitted
 * @param n the degree of the two new polynomials, ie half the degree of
     the input polynomial
 * @param w from x^n+w or x^n-w, ie the w in the two new modulos
     polynomials
```

```
 */
void splitting(long *pol, long n, long w){
    long a;
    for(long i=0; i<n; i++){
        a = pol[i+n]*w;
        pol[n+i]= (pol[i]-a)%Q;
        pol[i]=(pol[i]+a)%Q;
    }
}

/**
 * Computes forward NTT
 * @param pol  array of the coeefficients of the polynomial that the
    forward NTT will be performed on
 * @param NTT_forward list of the different roots of unity that it will be
     splitted modulo
 * @param move initiated at 0, this showes how long to move forward on the
     list of the forward NTT
 * @param start how many levels performed, start at 0
 * @param levels  how many levels the NTT will be performed
 * @param n  The length of the polynomial that will be splitted
 */
void forward_NTT(long *pol, long *NTT_forward, long move, long start,
                 long levels, long n){
    if(start==levels){
        return;
    }
    start++;
    if(move==0){
        splitting(pol,n/2,NTT_forward[0]);
        forward_NTT(pol, NTT_forward + 1, 1, start, levels, n / 2);

    }
    else{
        move=move*2;
        for(long i=0; i<move;i++){
            splitting(pol+i*n,n/2,NTT_forward[i]);
        }
        forward_NTT(pol, NTT_forward + move, move, start, levels, n / 2);
    }

}

/**
```

```
123  * Computes forward NTT
124  * @param pol  the polynomial that the forward NTT will be performed on
125  * @param NTT_forward list of the different roots of unity that it will be
         splitted modulo
126  * @param move initiated at 0, this showes how long to move forward on the
         list of the forward NTT
127  * @param start how many levels performed, start at 0
128  * @param levels  how many levels the NTT will be performed
129  * @param n  The length of the polynomial that will be splitted
130  */
131 void forward_NTT2(struct pol *polynomial, long *NTT_forward, long move,
132                   long start, long levels, long n){
133     forward_NTT(polynomial->coeffs,NTT_forward,move,start,levels,n);
134 }
135
136 /**
137  *Updates the array to contain the merging of the two previous polynomials
138  * @param pol array containing the two polynomials that will be merged
139  * @param n degree of the two polynomials
140  * @param w some power of the root of unity used for merging
141  */
142 void merging(long * pol, long n, long w){
143     long a;
144     for (long i =0; i < n; i++){
145         a = pol[i]+pol[n+i];
146         pol[n+i]= ((pol[n+i]-pol[i])*w)%Q;
147         pol[i]=a%Q;
148     }
149 }
150
151 /**
152  * Computes inverse NTT
153  * @param pol Array of the coefficients of the polynomial that the inverse
         NTT will be performed on
154  * @param NTT_forward The list for the roots of unity for the forward NTT
       + the integer so that the start point is the list for the last level
155  * @param move The length of the roots for the first level
156  * @param start initiated at 0
157  * @param levels how many levels that will be performed
158  * @param n Twice the length of each of the small polynomial
159  */
160 void inverse_NTT(long *pol, long *NTT_forward, long move, long start,
161                  long levels, long n){
162     if(start==levels){
```

```
163         return;
164     }
165     long i=move-1;
166     for(long j=0;j<move;j++){
167         merging(pol+j*n,n/2,NTT_forward[i]);
168         i=i-1;
169     }
170     move=move/2;
171     start++;
172     inverse_NTT(pol, NTT_forward - move, move, start, levels, n * 2);
173 }
174 
175 /**
176  * Computes inverse NTT
177  * @param pol The polynomial that the inverse NTT will be performed on
178  * @param NTT_forward The list for the roots of unity for the forward NTT
        + the integer so that the start point is the list for the last level
179  * @param move The length of the roots for the first level
180  * @param start initiated at 0
181  * @param levels how many levels that will be performed
182  * @param n Twice the length of each of the small polynomial
183  */
184 void inverse_NTT2(struct pol *polynomial, long *NTT_forward, long move,
185                   long start, long levels, long n){
186     inverse_NTT(polynomial->coeffs, NTT_forward, move, start,  levels, n);
187 }
188 
189 /**
190  * Multiplying by the inverse of 2^LEVEL
191  * @param pol Array of the coefficients of the polynomial right after the
        inverse_NTT to finnish the inverse algorithm
192  * @param inverse The inverse of the power of two
193  */
194 void inverse_finnish(long *pol, int inverse){
195     for(long i=0; i<get_N(); i++){
196         pol[i]=(pol[i]*inverse)%Q;
197     }
198 }
199 /**
200  * Multiplying by the inverse of 2^LEVEL
201  * @param pol The polynomial right after the inverse_NTT to finnish the
        inverse algorithm
202  * @param inverse The inverse of the power of two
203  */
```

```
204 void inverse_finnish2(struct pol *polynomial, int inverse){
205     inverse_finnish(polynomial->coeffs,inverse);
206 }
```

C | NTT multiplication

```
1  /**
2   * Determines what position of the second polynomial that will be
       multiplied by the j-th position of the first
3   * polynomial for the i-th position of the result
4   * @param i integer
5   * @param j integer
6   * @param n Size of the modpol
7   * @return The position of the second polynomial that is multiplied.
8   */
9  long f(long i, long j, long n) {
10     if (i >= j) {
11         return i - j;
12     } else {
13         return i - j + n;
14     }
15 }
16
17 /**
18  * Updates result to be the multiplication of pol1 and pol2
19  * @param pol1 The first polynomial
20  * @param pol2  The second polynomial
21  * @param result Polynomial/ array where the result is stored
22  * @param w  The w when the modpol is X^N+w
23  * @param n The degree of the modpol
24  */
25 void step_multiplied_NTT(long *pol1, long *pol2, long *result,
26                          long w, long n){
27     for (long i = 0; i < n; i++) {
28         for (long j = 0; j < n; j++) {
29             long pos2 = f(i, j, n);
30             if (j + pos2 >= n) {
31                 result[i] = (result[i] - w * pol1[j] * pol2[pos2]) % Q;
32             } else {
33                 result[i] = (result[i] + pol1[j] * pol2[pos2]) % Q;
34             }
35         }
36     }
37 }
38
39 /**
40  * Updates result to be the multiplication of pol1 and pol2
41  * @param pol1 Pointer to the array of the first polynomial
42  * @param pol2  Pointer to the array of the second polynomial
43  * @param result Pointer to the array of the olynomial where the result is
       stored
```

```
44  * @param roots  list of the roost, w, used for the different polynomials
       X^N+w
45  * @param sizeofpol The degree of the modpols
46  * @param amountofpol Number of polynomials that will be multiplied
47  */
48 void multiplied_NTT(long *pol1, long *pol2, long *result,
49                     long* roots, long sizeofpol, long amountofpol){
50     long j=0;
51     for(long i=0;i<amountofpol;i++){
52         step_multiplied_NTT(pol1+j,pol2+j,result+j,roots[i],sizeofpol);
53         j+=sizeofpol;
54     }
55 }
56
57 /**
58  * Updates result to be the multiplication of pol1 and pol2
59  * @param pol1 The first polynomials
60  * @param pol2  The second polynomials
61  * @param result Polynomial/ array where the result is stored
62  * @param roots  list of the roost, w, used for the different polynomials
       X^N+w
63  * @param sizeofpol The degree of the modpols
64  * @param amountofpol Number of polynomials that will be multiplied
65  */
66 void multiplied_NTT2(struct pol *ppol1, struct pol *ppol2,
67                      struct pol *presult, long* roots,
68                      long sizeofpol, long amountofpol){
69     multiplied_NTT(ppol1->coeffs,ppol2->coeffs,presult->coeffs,
70                    roots,sizeofpol,amountofpol);
71 }
```

D | Commit using normal multiplication

```
/**
 * Adding all coefficients of two polynomials
 * @param pol1 The first polynomial that is to be added
 * @param pol2 The second polynomial that is to be added
 * @param size The degree of the two polynomials
 * @return The sum of the two polynomials
 */
struct pol addPolynomials(struct pol pol1, struct pol pol2, long size) {
    struct pol result;
    for (int i = 0; i < size; i++) {
        result.coeffs[i] = (pol1.coeffs[i] + pol2.coeffs[i]) % Q;
    }
    return result;
}

/**
 * Multiplies a row by a vector and returns the result polynomial
 * @param row The row that is to be multiplied
 * @param vector The vector that is to be multiplied
 * @param size The size of the row and vector
 * @return  The polynomial that is the row multiplied by the vector
 */
struct pol multiplyRowByVectorNormal(struct pol *row,
                                     struct pol *vector,
                                     int size) {
    struct pol result;
    struct pol zeropol;
    for (int i = 0; i < get_N(); i++) {
        zeropol.coeffs[i] = 0;
    }
    result = zeropol;
    for (int i = 0; i < size; i++) {
        struct pol step_result = zeropol;
        multiplied_normal2(&row[i], &vector[i], &step_result, get_N());
        result = addPolynomials(result, step_result, get_N());
    }
    return result;
}

/**
 * Multiplies the matrix A_1 by the randomnessvector r
 * where A_1 = [I_D  A_1_marked]
 * @param A_1_marked Pointer to the last part of the A_1 vector with
    random polynomials as input
 * @param randomness Pointer to the randomness vector
```

```
45  * @param commit Pointer to the commitvector where the result is stored
46  */
47 void pmatrixTimesVectorNormalA_1(struct A_1_marked *A_1_marked,
48                                  struct randomness_vector_K *randomness,
49                                  struct comitment_vector_DL *commit) {
50     for (int i = 0; i < D; i++) {
51         struct pol step_result;
52         step_result = multiplyRowByVectorNormal(A_1_marked->pol[i],
53                                                 randomness->pol + D,
54                                                 K - D);
55         step_result = addPolynomials(step_result,
56                                      randomness->pol[i],
57                                      get_N());
58         commit->pol[i] = step_result;
59     }
60 }
61
62 /**
63  * Multiplies the matrix A_2 by the randomnessvector r
64  * where A_2 = [0^(LxD)   I_L  A_2_marked]
65  * @param A_2_marked Pointer to the last part of the A_2 vector with
       random polynomials as input
66  * @param randomness Pointer to the randomness vector
67  * @param commit Pointer to the commitvector where the result is stored
68  */
69 void pmatrixTimesVectorNormalA_2(struct A_2_marked *A_2_marked,
70                                  struct randomness_vector_K *randomness,
71                                  struct comitment_vector_DL *commit) {
72     for (int i = 0; i < L; i++) {
73         struct pol step_result;
74         step_result = multiplyRowByVectorNormal(A_2_marked->pol[i],
75                                                 randomness->pol + (D + L),
76                                                 K - D - L);
77         step_result = addPolynomials(step_result,
78                                      randomness->pol[i + D],
79                                      get_N());
80         commit->pol[i + D] = step_result;
81     }
82 }
83
84 /**Commits the message m by computing
85  * A_1  * r    +  0^d
86  * A_2             m
87  * Using normal multiplication
```

```
 * @param A_1_marked pointer to a D times K-D vector used as the last part
     of the A_1 vector
 * @param A_2_marked pointer to a L times (K-D-L) vector used as the last
    part of the A_1 vector
 * @param randomness pointer to the randomnessvector r of length K
 * @param message pointer to the message m of length L
 * @param commit pointer to the commitment vector of length D+L
 */
void pcommitNormal(struct A_1_marked *A_1_marked,
                   struct A_2_marked *A_2_marked,
                   struct randomness_vector_K *randomness,
                   struct message_vector_L *message,
                   struct comitment_vector_DL *commit) {
    pmatrixTimesVectorNormalA_1(A_1_marked, randomness, commit);
    pmatrixTimesVectorNormalA_2(A_2_marked, randomness, commit);
    for (int i = 0; i < D; i++) {
        commit->pol[i + L] = addPolynomials(commit->pol[i],
                                            message->pol[i],
                                            get_N());
    }
}
```

E | Commit using NTT multiplication

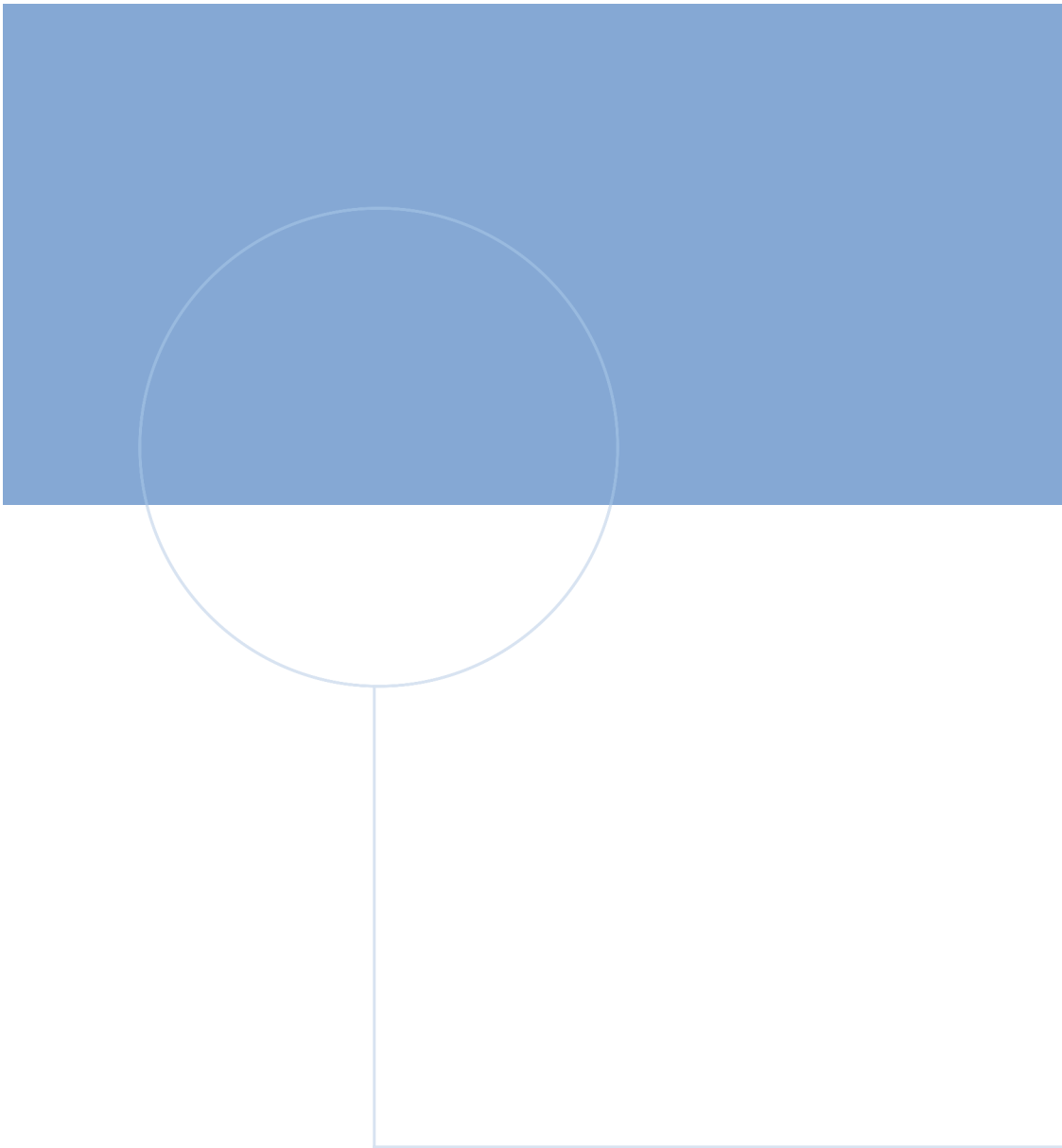
```
/**
 * Multiplies a row by a vector and returns the result polynomial
 * @param row The row that is to be multiplied in NTT version
 * @param vector The vector that is to be multiplied in NTT version
 * @param size The size of the row and vector
 * @return  The polynomial that is the row multiplied by the vector in NTT
     version
 */
struct pol multiplyRowByVectorNTT(struct pol *row,
                                  struct pol *vector,
                                  int size) {
    struct pol result;
    struct pol zeropol;
    for (int i = 0; i < get_N(); i++) {
        zeropol.coeffs[i] = 0;
    }
    result = zeropol;
    for (int i = 0; i < size; i++) {
        struct pol step_result = zeropol;
        multiplied_NTT2(&row[i], &vector[i], &step_result, NTT_roots,
                        get_sizeofpol(), get_num_polynomials());
        result = addPolynomials(result, step_result, get_N());
    }
    return result;
}

/**
 * Computes NTT forward of all the polynomials in the matrices and vector
 * @param A_1_marked D times (K-D) matrix
 * @param A_2_marked L times (K-D-L) matrix
 * @param randomness vector of length K
 */
void forwardNTT_matrices_vector(struct A_1_marked *A_1_marked,
                                struct A_2_marked *A_2_marked,
                                struct randomness_vector_K *randomness) {
    for (int i = 0; i < D; i++) {
        for (int j = 0; j < K - D; j++) {
            forward_NTT2(&A_1_marked->pol[i][j], NTT_forward, 0, 0,
                         get_Level(), get_N());
        }
    } // forward NTT of the A_1_marked matrix
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < K - D - L; j++) {
            forward_NTT2(&A_2_marked->pol[i][j], NTT_forward, 0, 0,
                         get_Level(), get_N());
```

```
45          }
46      } // forward NTT of the A_2_marked matrix
47      for (int i = 0; i < K; i++) {
48          forward_NTT2(&randomness->pol[i], NTT_forward, 0, 0,
49                       get_Level(), get_N());
50      } //forward NTT of the randomness vector
51  }
52
53  /**
54   * Inverse NTT of a vector with length D+L
55   * @param vector Vector of length D+L in NTT version
56   */
57  void inverseNTT_commitmentvectorDL(struct comitment_vector_DL *vector) {
58      for (int i = 0; i < D + L; i++) {
59          inverse_NTT2(&vector->pol[i],
60                       NTT_forward + get_move() - 1,
61                       get_move(),
62                       0,
63                       get_Level(),
64                       get_sizeofpol() * 2);
65          inverse_finnish2(&vector->pol[i],
66                           inverses_power_of_two[get_Level()]);
67      }
68  }
69
70  /**
71   * Multiplies the matrix A_1 by the randomnessvector r
72   * where A_1 = [I_D  A_1_marked]
73   * @param A_1_marked Pointer to the last part of the A_1 vector with
       random polynomials as input
74   * @param randomness Pointer to the randomness vector
75   * @param commit Pointer to the commitvector where the result is stored
76   */
77  void pmatrixTimesVectorNTTA_1(struct A_1_marked *A_1_marked,
78                                struct randomness_vector_K *randomness,
79                                struct comitment_vector_DL *commit) {
80      for (int i = 0; i < D; i++) {
81          struct pol step_result;
82          step_result = multiplyRowByVectorNTT(A_1_marked->pol[i],
83                                               randomness->pol + D,
84                                               K - D);
85          step_result = addPolynomials(step_result,
86                                       randomness->pol[i],
87                                       get_N());
```

```
88          commit->pol[i] = step_result;
89      }
90  }
91
92  /**
93   * Multiplies the matrix A_2 by the randomnessvector r
94   * where A_2 = [0^(LxD)   I_L  A_2_marked]
95   * @param A_2_marked Pointer to the last part of the A_2 vector with
         random polynomials as input
96   * @param randomness Pointer to the randomness vector
97   * @param commit Pointer to the commitvector where the result is stored
98   */
99  void pmatrixTimesVectorNTTA_2(struct A_2_marked *A_2_marked,
100                                struct randomness_vector_K *randomness,
101                                struct comitment_vector_DL *commit) {
102     for (int i = 0; i < L; i++) {
103         struct pol step_result;
104         step_result = multiplyRowByVectorNTT(A_2_marked->pol[i],
105                                              randomness->pol + (D + L),
106                                              K - D - L);
107         step_result = addPolynomials(step_result,
108                                      randomness->pol[i + D],
109                                      get_N());
110         commit->pol[i + D] = step_result;
111     }
112 }
113
114 /**
115  * Commits the message m by computing
116  * A_1    * r     + 0^d
117  * A_2              m
118  * Using NTT multiplication
119  * @param A_1_marked pointer to a D times K-D vector used as the last part
       of the A_1 vector
120  * @param A_2_marked pointer to a L times (K-D-L) vector used as the last
       part of the A_1 vector
121  * @param randomness pointer to a randomnessvector r of length K
122  * @param message pointer to the message m of length L
123  * @param commit pointer to the commitment vector of length D+L
124  */
125 void pcommitNTT(struct A_1_marked *A_1_marked,
126                 struct A_2_marked *A_2_marked,
127                 struct randomness_vector_K *randomness,
128                 struct message_vector_L *message,
```

```
129                 struct comitment_vector_DL *commit) {
130     forwardNTT_matrices_vector(A_1_marked, A_2_marked, randomness);
131     pmatrixTimesVectorNTTA_1(A_1_marked, randomness, commit);
132     pmatrixTimesVectorNTTA_2(A_2_marked, randomness, commit);
133     inverseNTT_commitmentvectorDL(commit);
134     for (int i = 0; i < D; i++) {
135         commit->pol[i + L] = addPolynomials(commit->pol[i],
136                                             message->pol[i],
137                                             get_N());
138     }
139 }
```



NTNU

Norwegian University of
Science and Technology