HIGH PERFORMANCE NUMBER THEORETIC TRANSFORMS IN CRYPTOGRAPHY

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

BY

METIN EVRIM ULU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

JANUARY 2020

Approval of the thesis:

HIGH PERFORMANCE NUMBER THEORETIC TRANSFORMS IN CRYPTOGRAPHY

submitted by **METIN EVRIM ULU** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Ömür Uğur
Director, Graduate School of **Applied Mathematics** _____

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** _____

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Ali Doğanaksoy
Cryptography, METU _____

Assoc. Prof. Dr. Murat Cenk
Cryptography, METU _____

Assoc. Prof. Dr. Ali Ulaş Özgür Kişisel
Mathematics, METU _____

Assoc. Prof. Dr. Fatih Sulak
Mathematics, Atılım University _____

Assist. Prof. Dr. Nurdan Saran
Computer Engineering Department, Çankaya University _____

Date: **15.01.2020**

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

Name, Last Name: METIN EVRIM ULU

Signature :

ABSTRACT

HIGH PERFORMANCE NUMBER THEORETIC TRANSFORMS IN CRYPTOGRAPHY

Ulu, Metin Evrim
Ph.D., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk

January 2020, 69 pages

Theoretical advances in physics opened up a new window into quantum computation. This window rendered a number of mathematically hard problems unusable for cryptographic applications. For instance, Shor showed that it is possible factor integers by a quantum algorithm efficiently thus rendering the standard public-key encryption scheme RSA insecure. In February 2016, NIST launched a standardization process for post-quantum cryptography algorithms to study the effect of quantum computing on the current generation of cryptographic algorithms and to build the next generation cryptosystems that are resistant to quantum attacks.

One type of quantum safe cryptographic systems is based on lattices. In order to improve the performance in lattice based systems, Number Theoretic Transforms (NTT) are used. In this thesis, the performance of NTT in cryptography is studied. First, Peikert's Scheme and its realization BCNS Algorithm and NewHope key encapsulation method is discussed. Next, SWIFFTX hash function that uses NTT as a building block is presented. Finally, an efficient GPU implementation of SWIFFTX hash function is provided. Experimental results indicate almost 10x improvement in speed and 5 Watts decrease in power consumption per 2^{16} hashes.

Keywords: NTT, PQC, NewHope, SWIFFT, SWIFFTX

ÖZ

KRİPTOGRAFİ'DE YÜKSEK PERFORMANSLI SAYI KURAMSAL DÖNÜŞÜMLER

Ulu, Metin Evrim
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk

Ocak 2020, 69 sayfa

Son yıllarda meydana gelen fizik'teki teorik gelişmeler, kuantum bilgisayarları üretmenin ve üzerinde hesaplama yapmanın olanaklı olduğunu gösterdi. Bu durum, varolan ve kullanılan kriptografik algoritmaları değişik seviyelerde güvensiz hale getirdi. Bunun sebebi ise, algoritmalarda kullanılan matematiksel çözümü zor olan problemlerin kuantum bilgisayarlarda verimli olarak çözülebilmesi. Örnegin Shor, tam sayıları asal çarpanlarına ayıran bir algoritmayı henüz kuantum bilgisayarlar var olmadan tasarlamıştı. Bu sayede, kuantum bilgisayarlar üzerinde, açık anahtar RSA kriptosistemi kırılabiliyor. Şubat 2016'da, NIST mevcut durumu incelemek, kuantum hesaplamanın varolan kriptosistemler üzerindeki etkisini çalışmak ve yeni nesil kuantum saldırılara dayanıklı sistemler geliştirebilmek için bir çalışma grubu kurdu.

Kuantum ataklara karşı dayanıklı olan kriptografik sistemlerin bir çeşidi latis tabanlı sistemlerdir. Bu tür sistemlerde, performansı arttırmak için Sayı Kuramsal Dönüşümler kullanılmaktadır. Bu tezde, bu dönüşümlerin kriptografik uygulamalardaki pereformansları çalışılmıştır. Öncelikle, latis tabanlı Peikert şeması, O'nun gerçekleştirimi olan BCNS algoritması ve NewHope algoritması incelenmiştir. Ardından, bahsi geçen dönüşüm'leri kullanan SWIFFTX özet fonksiyonu sunulmuştur. Son olarak, bu fonksiyona ait verimli bir GPU uygulaması verilmiştir. Yapılan testlerde görülmüştür ki, bu yeni uygulama fonksiyona 10 kat hız kazandırmış, aynı zaman da 2^{16} girdi için 5 Watt güç tasarrufu sağlamıştır.

Anahtar Kelimeler: NTT, PQC, NewHope, SWIFFT, SWIFFTX

ACKNOWLEDGMENTS

I would like to thank to my family for their patience and support during my studies. It would be impossible to accomplish without them. Also, I would like to thank to my supervisor Assoc. Prof. Dr. Murat Cenk for his guidance, encouragement and profound comments. I wish him and his family a long and prosperous life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
CCA	Choosen Ciphertext Attack
CPA	Choosen Plaintext Attack
CUDA	Compute Unified Device Architecture
DFT	Discrete Fourier Transform
DIT	Decimation in Time
DIF	Decimation in Frequency
FFT	Fast Fourier Transform
GNU	GNU is not Unix
GPU	Graphical Processing Unit
ITS	Independent Thread Scheduling
NTT	Number Theoretic Transform
PQC	Post-Quantum Cryptography
PTX	Parallel Thread Execution
REDC	Montgomery Reduction Algorithm
SIMD	Single-instruction Multiple-data
SM	Streaming Multiprocessor

CHAPTER 1

INTRODUCTION

Cryptography is a branch of mathematics that studies the security of information in the presence of adversaries. This information can be the data on a secure channel or just the files on your computers hard drive. Therefore the level of security required might change variably.

A famous example is the German device called Enigma that has been used during World War II. Although being a mechanical device, operated by rotors etc., the keyspace of this device was $26^4 \approx 2^{18}$ (i.e. more than 18-bits). This number is much lower than the size of the keyspace of DES, a symmetric cipher that has been used in the late nineties internationally. DES had a keyspace of 2^{56}. Enigma has been cracked in 1932 by cryptanalysis techniques and this allowed Allies to listen to the communication among German troops.

Many ciphers have been designed since and most of them are shown to broken by emerging cryptanalysis techniques. Some of these techniques are purely mathematical deeply rooted in the concrete branches of mathematics such as topology, algebraic geometry and analysis, some of them are rooted in the advances in computer science and engineering. Giant supercomputers made the computations feasible and faster and this made the impossible possible. Unfortunately, these developments posed certain challenges over the cryptographers who design these ciphers which are, theoretically, secure under certain assumptions.

Although the field is dominated by mathematicians and computer scientists, physicists have also showed interest in computation, computability and cryptography. A

prominent example might be by the famous Richard P. Feynman. In 1982, Feynman [12] showed that a quantum computer can outperform a regular one by employing subatomic particles. However, this computer was not a binary but a probabilistic one. Developments such as these made awareness in the scientific community to look deeper into this subject and the field of quantum computing has born.

Industry is still in its infancy yet promising results emerge every day. In 2017, Google announced its 20-qubit computer. Similarly, IBM, Intel, Rigetti followed them. In 2018, Google announced their 72-qubit quantum computer. In the following year, IBM announced a 53-qubit one. These numbers coincide with the famous Moore's Law which states the number of transistors in a dense integrated circuit doubles about every two years. If an analog rule is true for quantum-computers then in 10+ years, there will be a quantum computer of 2048-qubits.

Although realizations of quantum computers are still new, theoretical models have been developed to estimate or better prove what a quantum computer can do. For instance, Shor [32] showed that a problem in the NP class called Prime Factorization can be solved in the existence of such a quantum computer. This proof was only the beginning. In 1996, Grover [15] showed that a search on the keyspace can be done faster requiring only square-root of the keyspace computations in the worst case.

These facts led to deeper studies and cryptography community now develops new ciphers and algorithms based on the fact that a competitive quantum computer might be available to an adversary. In April 2016, NIST published a report [3] on post-quantum cryptography. In this report, it is noted that commonly deployed RSA algorithm will be insecure by 2030. Therefore, post-quantum resistant algorithms are required and the Post-Quantum-Cryptography Project was born.

In terms of post-quantum security, symmetric ciphers has an advantage that doubling the keysize restores their original security. This is not true for public-key and signature schemes. This is why the NIST Project is focused on them. There are five classes defined by NIST. The lattice based schemes is one of them. We will focus on this type and delve into the details.

In this thesis, a number of the post-quantum cryptography algorithms that employ

Number Theoretic Transforms are discussed. As mentioned before, these algorithms are based on problems that are proved to be quantum resistant. Our focus is on the efficient implementations of these algorithms on today's computers.

We start with defining building blocks of these algorithms, namely, the Fourier Transforms and efficient methods of modular arithmetic in the following two chapters. Then, several post-quantum algorithms are presented. Finally, an efficient implementation of a hash algorithm on GPU is discussed in detail.

CHAPTER 2

NUMBER THEORETIC TRANSFORMS

2.1 Introduction

In mathematics, Discrete Fourier Transform (DFT) is a function that converts a string of equally-spaced samples of a function to a same length string of equally-spaced samples of discrete-time Fourier transform that is a complex-valued function of frequency. DFT can be calculated over any ring.

The material in this chapter follows [10] and [9].

Definition 1. *Let $\mathbb{C}$ be the algebraically closed complex field and $n \in \mathbb{N}$. An $\omega \in \mathbb{C}$ is called n-th root of unity if it satisfies $\omega^n = 1$.*

Definition 2. *An $\omega \in \mathbb{C}$ is called principal n-th root of unity if it is a n-th root unity and satisfies $\omega^m \neq 1$ with $n > m \in \mathbb{N}$.*

From now on, assume $\omega_n \in \mathbb{C}$ is a principal n-th root of unity.

Lemma 1. $\omega_{dn}^{dk} = \omega_n^k$.

Proof. $\omega_{dn}^{dk} = (e^{\frac{2\pi i}{dn}})^{dk} = (e^{\frac{2\pi i}{n}})^k = \omega_n^k \quad \square$

Corollary 1. *Let $n \in N$ be an even integer, then $\omega_n^{n/2} = \omega_2 = -1$.*

Proof. The first equality is a direct consequence of the previous lemma. Now suppose $\omega_2 = 1$ then, $\omega_n^{n/2} = 1$ which contradicts to the fact that $\omega_n \in \mathbb{C}$ is a principal n-th root of unity. $\square$

Lemma 2. *Let $n, k \in \mathbb{N}$ such that $n \nmid k$, then*

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Proof. If $n \mid k$ such that $k = nk'$ then $\sum_{j=0}^{n-1} (\omega_n^{nk'})^j = \sum_{j=0}^{n-1} (1)^{k'j} = n$ trivially. If $n \nmid k$ then,

$$\omega_n^k \sum_{j=0}^{n-1} (\omega_n^k)^j = \sum_{j=0}^{n-1} (\omega_n^k)^{j+1} = \sum_{j=0}^{n-1} (\omega_n^k)^j$$

$$(\omega_n^k - 1) \sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Finally, $\omega_n^k \neq 1$ implies the desired result. □

2.2 Fourier Transform

Definition 3. *Let $\mathcal{R}$ be a ring, $n \in \mathbb{N}$ and $\omega_n \in \mathcal{R}$ be a principal n-th root of unity such that $\omega_n^n = 1$ and $\omega_n^m \neq 1$ for any $n > m \in \mathbb{N}$. The discrete Fourier transform is defined as follows:*

$$DFT : \mathcal{R}^n \to \mathcal{R}^n$$

$$A = (a_0, \ldots, a_{n-1}) \to F = (f_0, \ldots, f_{n-1})$$

where $f_i = \sum_{j=0}^{n-1} a_j \omega_n^{ij}$.

The vector A can also be viewed as an element of $\mathcal{R}[x]$ such that $A = \sum_{i=0}^{n-1} a_i x^i$ where $degree\ A = n$. Then the equation, $DFT(A) = F$ is simply calculated by evaluating the polynomial A at $\omega_n^j, \forall j \in \mathbb{Z},\ 0 \leq j \leq n-1$ such that $f_j = A(\omega_n^j)$.

Definition 4. *Number theoretic transforms (NTT) are specialized discrete Fourier transforms where the ring $\mathcal{R}$ is a finite field.*

Theorem 1. *Horner's Rule: The polynomial $A(x)$ can be evaluated in n multiply-add operations.*

Proof. Consider the following:

$$A(x) = (((((a_{n-1}x + a_{n-2})x + a_{n-3})x + \ldots$$

Each parenthesis can be calculated in a single multiply-add operation. By considering the polynomial evaluation as left-associative fold, $A(x)$ can be evaluated in n multiply-add operations. □

Theorem 2. *The naive implementation of DFT has arithmetic complexity of* $\mathcal{O}(n^2)$

Proof. Calculation of each $f_j = A(\omega_n^j)$ has complexity of $\mathcal{O}(n)$ by the Horner's Rule and the result follows. □

An algorithm invented by Cooley-Tukey, called Fast Fourier Transform, allows the calculation of DFT and has arithmetic complexity of $\mathcal{O}(n\ logn)$ when n is a power of two. Basically, the algorithm employs divide-and-conquer strategy with the following outline:

1. Divide the problem into two sub-problems of half-size,
2. Solve each sub-problem,
3. Obtain the final solution for the original problem by combining two solutions.

There are in fact, two variants of Fast Fourier transform. The first one is the Cooley-Tukey algorithm that defines two sub-problems by decimating the input in time.

Definition 5. *Radix-2 Decimation-In-Time (DIT) Fast Fourier Transform (FFT).*

Let $N \in \mathbb{N}$ *be a power of two and* $X = FFT_N(x)$ *be the fast Fourier transform such that*

$$X_r = \sum_{k=0}^{N-1} x_k \omega_n^{rk}, \quad \forall r \in \mathbb{Z},\ 0 \leq r < N, \tag{2.1}$$

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} \omega_N^{r(2k)} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} \omega_N^{r(2k)}, \quad \forall r \in \mathbb{Z},\ 0 \leq r < N, \tag{2.2}$$

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} \omega_{\frac{N}{2}}^{r(k)} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} \omega_{\frac{N}{2}}^{r(k)}, \quad \forall r \in \mathbb{Z},\ 0 \leq r < N, \tag{2.3}$$

since $\omega_N^2 = \omega_{\frac{N}{2}}$. *Now define the two sub-problems.*

$$Y_r = \sum_{k=0}^{\frac{N}{2}-1} y_k \omega_{\frac{N}{2}}^{rk}, \quad \forall r \in \mathbb{Z},\ 0 \leq r \leq \frac{N}{2} - 1, \tag{2.4}$$

$$Z_r = \sum_{k=0}^{\frac{N}{2}-1} z_k \omega_{\frac{N}{2}}^{rk}, \quad \forall r \in \mathbb{Z},\ 0 \leq r \leq \frac{N}{2} - 1, \tag{2.5}$$

where $y_k = x_{2k}$ *and* $z_k = x_{2k+1},\ \forall k \in \mathbb{Z}, 0 \leq k < \frac{N}{2} - 1$. *It is now possible to combine the two results by the following:*

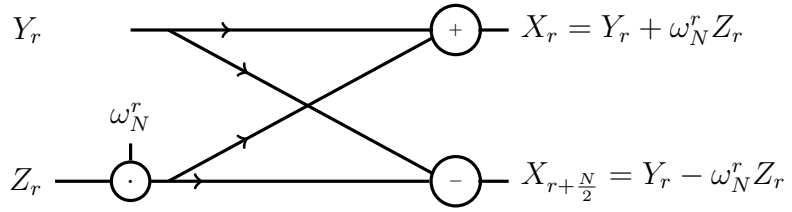
$$X_r = Y_r + \omega_N^r Z_r, \quad \forall r \in \mathbb{Z},\ 0 \leq r \leq \frac{N}{2} - 1 \tag{2.6}$$

$$X_{r+\frac{N}{2}} = Y_r - \omega_N^r Z_r, \quad \forall r \in \mathbb{Z},\ 0 \leq r \leq \frac{N}{2} - 1 \tag{2.7}$$

using the fact $\omega_N^{\frac{N}{2}+r} = -\omega_N^r,\ \forall r \in \mathbb{Z}, 0 \leq r \leq \frac{N}{2} - 1$.

In the following figure, Cooley-Tukey butterfly is depicted in which two sub-problems are combined to get the desired final result. Note that, Z_r is multiplied by the twiddle factor before entering the network.

Figure 2.1: Cooley-Tukey Butterfly



The second type of FFT is Decimation-In-Frequency Fast Fourier Transform.

Definition 6. *Radix-2 Decimation-In-Frequency (DIF) Fast Fourier Transform (FFT)*

Let $N \in \mathbb{N}$ *be a power of* 2 *and* $X = FFT_N(x)$ *be the fast Fourier transform such*

that

$$X_r = \sum_{l=0}^{N-1} x_l \omega_N^{rl}, \quad \forall r \in \mathbb{Z}, 0 \leq r < N, \tag{2.8}$$

$$X_r = \sum_{l=0}^{\frac{N}{2}-1} x_l \omega_N^{rl} + \sum_{l=\frac{N}{2}}^{N-1} x_l \omega_N^{rl}, \quad \forall r \in \mathbb{Z}, 0 \leq r < N, \tag{2.9}$$

$$X_r = \sum_{l=0}^{\frac{N}{2}-1} x_l \omega_N^{rl} + \sum_{l=0}^{\frac{N}{2}-1} x_{l+\frac{N}{2}}\, \omega_N^{r(l+\frac{N}{2})}, \quad \forall r \in \mathbb{Z}, 0 \leq r < N, \tag{2.10}$$

$$X_r = \sum_{l=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}}\, \omega_N^{r\frac{N}{2}})\, \omega_N^{rl}, \quad \forall r \in \mathbb{Z}, 0 \leq r < N, \tag{2.11}$$

For even $r = 2k,\ k \in \mathbb{Z}$*,*

$$X_{2k} = \sum_{l=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}}\, \omega_N^{kN})\, \omega_N^{2kl}, \quad \forall k \in \mathbb{Z}, 0 \leq k \leq \frac{N}{2} - 1, \tag{2.12}$$

$$= \sum_{l=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}})\, \omega_{\frac{N}{2}}^{kl}, \quad \forall k \in \mathbb{Z}, 0 \leq k \leq \frac{N}{2} - 1, \tag{2.13}$$

$$\tag{2.14}$$

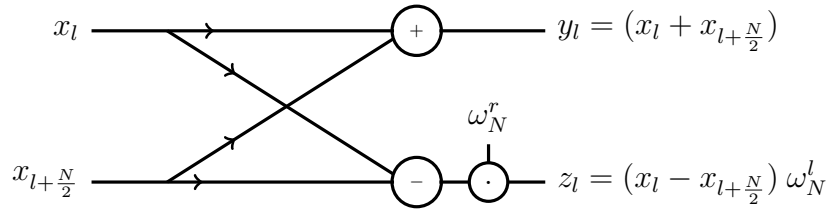
For odd $r = 2k + 1,\ k \in \mathbb{Z}$*,*

$$X_{2k+1} = \sum_{l=0}^{\frac{N}{2}-1} (x_l + x_{l+\frac{N}{2}}\, \omega_N^{(2k+1)\frac{N}{2}})\, \omega_N^{(2k+1)l}, \quad \forall k \in \mathbb{Z}, 0 \leq k \leq \frac{N}{2} - 1, \tag{2.15}$$

$$= \sum_{l=0}^{\frac{N}{2}-1} ((x_l - x_{l+\frac{N}{2}})\, \omega_N^{l})\, \omega_{\frac{N}{2}}^{kl}, \quad \forall k \in \mathbb{Z}, 0 \leq k \leq \frac{N}{2} - 1. \tag{2.16}$$

Now define $Y_k = X_{2k}$*,* $Z_k = X_{2k+1}$*,* $y_l = x_l + x_{l+\frac{N}{2}}$ *and* $z_l = (x_l - x_{l+\frac{N}{2}})\, \omega_N^l$*.*

Figure 2.2: Gentleman-Sande Butterfly



Theorem 3. *The arithmetic complexity of DIT FFT algorithm is* $\mathcal{O}(N\ logN)$*.*

Proof. There are two sub-problems of size $N/2$. Total cost of N X_r is N additions and $N/2$ multiplications.

$$T(N) = \begin{cases} 2T(N/2) + NAdditions + \frac{N}{2}Multiplications, & ifN = 2^n \geq 2, \\ 0, & ifN = 1 \end{cases}$$

Observe the recurrence relation, for instance for $N = 2^3$,

$$\begin{aligned} T(8) &= 2T(4) + 2^3 Add. + 2^2 Mult. \\ &= 2(2T(2) + 2^2 Add. + 2Mult.) + 2^3 Add. + 2^2 Mult. \\ &= 4T(2) + 2^4 Add. + 2^3 Mult. \\ &= 4(2T(1) + 2Add. + 1Mult.) + 2^4 Add. + 2^3 Mult. \\ &= 8T(1) + (2^3 + 2^4)Add. + (2^2 + 2^3)Mult. \\ &= NlogNAdditions + \frac{N}{2}logNMultiplications \end{aligned}$$

This shows that the complexity is bounded by $\mathcal{O}(NlogN)$. □

Theorem 4. *The arithmetic complexity of DIF FFT algorithm is* $\mathcal{O}(N\ logN)$.

Proof. The proof is almost the same as the DIT case. The only difference in the algorithm is that additions and subtractions are executed before the multiplication by the twiddle factors. □

2.3 Fast Polynomial Multiplication

There are various applications of Fast Fourier transform. One of them is Fast Polynomial Multiplication. The idea is basically evaluating operands at roots of unity, multiplying results point-wise then, interpolating back to the resulting polynomial in order to get the desired result.

Consider two polynomials of degree N defined by

$$A_N(z) = \sum_{l=0}^{N} a_l z^l = a_N z^N + \ldots + a_1 z + a_1 \tag{2.17}$$

and

$$B_N(z) = \sum_{l=0}^{N} b_l z^l = b_N z^N + \ldots + b_1 z + b_0 \tag{2.18}$$

where $z \in \mathbb{C}$ and $a_l, b_l \in \mathbb{C},\ l \in \mathbb{Z},\ 0 \leq l \leq N$. The product of polynomials $A_N(z)$ and $B_N(z)$ is a polynomial of degree $2N$ defined by

$$C_{2N}(z) = A_N(z)B_N(z) = \sum_{l=0}^{2N} c_l z^l = c_{2N} z^{2N} + \ldots + c_1 z + c_0 \tag{2.19}$$

where $a_k = b_k = 0$ for $N < k \leq 2N$ and

$$c_l = \sum_{k=0}^{l} a_k b_{l-k}, \quad l \in \mathbb{Z}, 0 \leq l \leq 2N. \tag{2.20}$$

Theorem 5. *Naive polynomial multiplication of two polynomials of degree N has arithmetic complexity of $\mathcal{O}(N^2)$.*

Proof. Each c_l requires N multiplications and $N-1$ additions at maximum and there are $2N+1$ coefficients and the result follows. □

Definition 7. *Polynomial Multiplication using DFT*

Assuming two operands have degree N, the algorithm is composed of the following steps:

1. *Pre-processing: Add N leading zero coefficients to the operands $A_N(z)$ and $B_N(z)$ to obtain $\tilde{A}_{2N}(z)$ and $\tilde{B}_{2N}(z)$ for which $\tilde{a}_l, \tilde{b}_l = 0, l \in \mathbb{Z}, N < l \leq 2N$.*
2. *Fast polynomial evaluation: Evaluate $\tilde{A}_{2N}(z)$ and $\tilde{B}_{2N}(z)$ at powers of $2N+1$-th root of unity, that is*

$$\tilde{A}_r = \tilde{A}_N(\omega_{2N}^r) = \sum_{l=0}^{2N} a_l \omega_{2N}^{rl}, \quad r \in \mathbb{Z}, 0 \leq r \leq 2N \tag{2.21}$$

and

$$\tilde{B}_r = \tilde{B}_N(\omega_{2N}^r) = \sum_{l=0}^{2N} b_l \omega_{2N}^{rl}, \quad r \in \mathbb{Z}, 0 \leq r \leq 2N \tag{2.22}$$

3. *Point-wise multiplication:*

$$C_r = \tilde{A}_r \tilde{B}_r,\ r \in \mathbb{Z}, 0 \leq r \leq 2N \tag{2.23}$$

4. *Fast polynomial interpolation: Now, we have* $C_r = \sum_{l=0}^{2N} c_l \omega_{2N+1}^{rl}$ *which is basically the DFT of the polynomial* $C_{2N}(z)$*. Therefore it is possible to obtain* $2N+1$ *coefficients using inverse DFT such that:*

$$c_l = \frac{1}{2N+1} \sum_{r=0}^{2N} C_r \omega_{2N}^{-rl}, \quad l \in \mathbb{Z}, 0 \leq l \leq 2N. \tag{2.24}$$

This defines a method to multiply two polynomials of degree N *by using DFT and obtain* $C_{2N}(z)$*, a polynomial of degree* $2N$*.*

Theorem 6. *The polynomial multiplication algorithm is called Fast Polynomial Multiplication if it employs FFT for steps 2 and 4 and it has arithmetic complexity of* $\mathcal{O}(NlogN)$*.*

Proof. Steps 2 and 4 have complexities of $\mathcal{O}(2N\ log2N)$ using the either butterfly defined previously. Point-wise multiplication has complexity of $\mathcal{O}(2N)$ so the algorithm has a total complexity of $\mathcal{O}(2N\ log2N + 2N) \approx \mathcal{O}(NlogN)$ where N is a power of 2. □

CHAPTER 3

OPTIMIZATIONS IN MODULAR ARITHMETIC

3.1 Introduction

The multiplication of coefficients of polynomials are performed in prime finite fields and it requires modular reduction having high cost. Therefore, we need optimizations in order get high performance. In this chapter, two methods of optimizing modular arithmetic in modulus $n \in \mathbb{Z}$ are discussed. These are Short Barrett Reduction and Montgomery Forms.

3.2 Short Barrett Reduction

Barrett reduction is introduced in 1986 by P.D. Barrett. Basically, the algorithm calculates $c = a \; mod \; n$ for $n \in \mathbb{Z}$.

Definition 8. *Short Barrett Reduction*
Let $\frac{1}{n} = s \in \mathbb{Q}$ *be the multiplicative inverse of* n *in* $\mathbb{Q}$. *Then,*

$$a \; mod \; n \equiv a - \lfloor as \rfloor \; mod \; n \tag{3.1}$$

The reduction can be implemented naively as follows.

```
1   unsigned int reduce(unsigned int n, unsigned int a) {
2     unsigned int q = a / n;
3     return a - q * n;
4   }
```

3.3 Montgomery Forms

Next, we discuss discoveries of Peter L. Montgomery.

Definition 9. *Montgomery Form of an element in* $\mathbb{Z}/n\mathbb{Z}$

Let $n \in \mathbb{N}$*. Then, Montgomery form of an element* $a \in \mathbb{Z}/n\mathbb{Z}$ *is*

$$aR \bmod n \tag{3.2}$$

where $R \in \mathbb{Z}$ *is called the auxiliary modulus with* $(n, R) = 1$.

The auxiliary modulus R defined above is usually a power of the base that the number a is represented in. The reason for this is basically, if the division operation in a processing unit by this modulus is faster than the normal division then, the following optimizations can lead to a better performance.

Theorem 7. *Montgomery Modular Addition*
The addition of two forms $aR, bR \in \mathbb{Z}/n\mathbb{Z}$ *is*

$$aR + bR \bmod n \equiv (a + b)R \bmod n \tag{3.3}$$

Theorem 8. *Montgomery Modular Multiplication*
The multiplication of two forms $aR, bR \in \mathbb{Z}/n\mathbb{Z}$ *is*

$$(aR)(bR)R^{-1} \bmod n \equiv (ab)R \bmod n \tag{3.4}$$

The proofs of the above theorems are trivial so omitted. The multiplicative inverse of R in $\mathbb{Z}/n\mathbb{Z}$ can be pre-calculated so the only overhead is the extra multiplication by the inverse.

Theorem 9. *Montgomery Modular Reduction - The REDC Algorithm*
Suppose $a' \in \mathbb{Z}/n\mathbb{Z}$ *is the Montgomery form of* $a \in \mathbb{Z}/n\mathbb{Z}$*. Then* a *can be calculated from* a' *by applying the following steps.*

Let $n' \in \mathbb{Z}/n\mathbb{Z}$ *be a number such that* $RR^{-1} - nn' = 1 \in \mathbb{Z}$*.*
Let $m \equiv (((a' \bmod R)n') \bmod R)$.

Let $t = \frac{a'+mn}{R}$. *Then,*

$$a = \begin{cases} t - n, & if\ t \geq n, \\ t, & otherwise. \end{cases} \tag{3.5}$$

Proof. To prove the correctness of the algorithm, observe the following equation.

$$a' + mn \equiv (((a'\ mod\ R)n')\ mod\ R)n \equiv a' + a'n'n \equiv a' - a' \equiv 0\ mod\ R \tag{3.6}$$

This shows that $t \in \mathbb{Z}$. Next,

$$t - n \equiv t\ mod\ n \equiv (a' + mn)R^{-1} \equiv a'R^{-1}\ mod\ n \tag{3.7}$$

This shows that a' is the Montgomery form of a in $\mathbb{Z}/n\mathbb{Z}$. Finally, to prove the last step, observe that $m \in [0, R-1]$, therefore

$$a' + mn \in [0, U] \quad where\ U = (Rn - 1) + (R - 1)n < 2Rn \tag{3.8}$$

Hence, $t < 2N$ and the result follows. □

Naturally, in the case of a single multiplication the overhead is heavy. However, if a number of multiplications are done consecutively, this procedure has a significant gain since division by R is faster than usual division. In most of the cases, R is taken to be a power 2 and division is implemented by a shift right operation.

CHAPTER 4

THE USE OF NUMBER THEORETIC TRANSFORMS IN CRYPTOGRAPHY

In this chapter, I discuss some of the post-quantum cryptography algorithms and explore the application of number theoretic transform in these algorithms. First, definitions of some computational problems on lattices are given. Then, two algorithms, namely, BCNS and NewHope algorithms and the use of NTT in them are discussed. Finally, hash algorithms SWIFFT and SWIFFTX are discussed.

4.1 Computational Problems on Lattices

Definition 10. *Let $n \in \mathbb{Z}$. A* lattice L *in $\mathbb{R}^n$ is a subgroup of the additive group $\mathbb{R}^n$ which is isomorphic to the additive group $\mathbb{Z}^n$, and which spans the real vector space $\mathbb{R}^n$. Given a set of basis vectors $\{b_1, \ldots, b_n\}$,* L *is defined as:*

$$\mathrm{L} = \{\sum_{i=1}^{n} a_i \cdot b_i : a_i \in \mathbb{Z}\} = \{B \cdot a : a \in \mathbb{Z}^n\} \tag{4.1}$$

B is called basis matrix.

Following [33], a lattice is a discrete version of a vector subspace. This discreteness results in the existence of a well-defined smallest element, excluding the zero vector. Now define the non-zero smallest element by

$$\lambda_1(L) \coloneqq min\{\|\mathbf{x}\| : \mathbf{x} \in L, \mathbf{x} \neq 0\} \tag{4.2}$$

The successive minimal $\lambda_i(L)$ can be also be defined as the smallest radius $r \in \mathbb{R}$

s.t. the n-dimensional ball of radius r centered at the origin contains $i \in \mathbb{Z}$ linearly independent lattice points.

The basis vectors can be left-multiplied by a uni-modular integer matrix to get another basis for the same lattice. This means

$$B' = B \cdot U \tag{4.3}$$

where the elements of the matrix U are integers with $det(U) = 1$. Moreover, the determinant of a basis matrix is an invariant. Given a basis matrix B for a lattice L,

$$\Delta(L) = \left|det(B^t \cdot B)\right|^{1/2} \tag{4.4}$$

is called the discriminant of the lattice. If L has full rank then,

$$\Delta(L) = |det(B)| \tag{4.5}$$

holds. The volume of the fundamental parallel-piped of the lattice bases are denoted by the above expression, $\Delta(L)$, and this volume is constant for a lattice L.

There exists a constant γ_n which is dependent on n, s.t.

$$\lambda_1(L) \leq \sqrt{\gamma_n} \cdot \Delta(L)^{1/n}. \tag{4.6}$$

This is proved by Hermite. γ_n is only known for $1 \leq n \leq 8$. Nevertheless, for arbitrary lattices and for $n = 1$, Hermite showed that the following holds.

$$\lambda_1(L) \approx \sqrt{\frac{n}{2\pi e}} \cdot \Delta(L)^{1/n} \tag{4.7}$$

The following theorems are included for the sake completeness.

Theorem 10. *(Minkowski). Let L be a rank-n lattice and $\mathbb{C} \subset L$ be a convex symmetric body about the origin with $Vol(\mathcal{C}) > 2^n \cdot \Delta(L)$. Then $\mathcal{C}$ contains a non-zero vector $\boldsymbol{x} \in L$.*

Corollary 2. *(Minkowski's Theorem). Let L be a n-dimensional lattice. Then the following holds.*

$$\lambda_1(L) \leq \sqrt{n} \cdot \Delta(L)^{1/n} \tag{4.8}$$

The *dual* lattice L^* is the set of all vectors $\mathbf{y} \in \mathbb{R}^n$ such that $\mathbf{y} \cdot \mathbf{x}^T \in \mathbb{Z}$ for all $\mathbf{x} \in L$. Given a basis matrix B, it is possible to compute the basis matrix B^* of L^* via $B^* = (B^{-1})^T$. Therefore $\Delta(L^*) = 1/\Delta(L)$ holds. The first minimum of L and the n-th minimum of L^* are linked by the transference theorem by Banaszczyk. It shows that for all n-dimensional lattices, the following holds.

$$1 \leq \lambda_1(L) \cdot \lambda_n(L^*) \leq n. \tag{4.9}$$

Thus a lower bound on $\lambda_1(L)$ can be translated into an upper bound on $\lambda_n(L^*)$ and vice versa.

Definition 11. *The (Approximate) Shortest Vector Problem (SVP).*

Let $\boldsymbol{B}$ be a lattice basis. Then, the SVP problem asks to find a shortest nonzero lattice vector $\boldsymbol{v} \in \mathcal{L}(\boldsymbol{B})$ with $\|\boldsymbol{v}\| = \lambda_1(\mathcal{L}(\boldsymbol{B}))$. In the γ-approximate γ-SVP, for $\gamma \geq 1$, it asks to find a shortest nonzero lattice vector $\boldsymbol{v} \in \mathcal{L}(\boldsymbol{B}) \setminus \{0\}$ of norm at most $\|\boldsymbol{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L}(\boldsymbol{B}))$.

Definition 12. *The (Approximate) Closest Vector Problem (CVP).*

Let again $\boldsymbol{B}$ be a lattice basis and let t be the target vector. The CVP problem asks to find the lattice vector $\boldsymbol{v} \in \mathcal{L}(\boldsymbol{B})$ such that the distance to the target $\|\boldsymbol{v} - \boldsymbol{t}\|$ is minimized. In the γ-approximate γ-CVP, for $\gamma \geq 1$, the problem asks to find a lattice vector $\boldsymbol{v} \in \mathcal{L}(\boldsymbol{B})$ such that $\|\boldsymbol{v} - \boldsymbol{t}\| \leq dist(\boldsymbol{t}, \mathcal{L}(\boldsymbol{B}))$ where $dist(\boldsymbol{t}, \Lambda) = inf\{\|\boldsymbol{v} - \boldsymbol{t}\| : \boldsymbol{v} \in \Lambda\}$ is the distance of $\boldsymbol{t}$ to Λ.

In the SVP problem, there can still be more than one shortest nonzero vectors. The γ-SVP gets difficult as γ gets closer to 1. The Lenstra, Lenstra, Lovász (LLL) algorithm [19] solves the SVP problem in polynomial time and with exponential approximation factor $2^{\mathcal{O}(n)}$. Algorithms that can produce an exact solution or approximate solution of SVP problem with $poly(n)$ factors either run in $2^{\mathcal{O}(n)}$ and require exponential space or in $2^{\mathcal{O}(n\ logn)}$ and require only polynomial space.

The NP-hardness of SVP was shown in [35] for the l_∞ norm. In [1], it is proved that SVP is NP-hard for the l_2 norm using randomized reductions and that the corresponding decision problem is NP-complete. The NP-hardness of CVP is proved in [35]. Nevertheless, practical cryptosystems employ only sub-classes of CVP or SVP and they are not supposed to be NP-hard (see [17], Remark 6.24).

Definition 13. *LWE Search Problem*

Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be an integer lattice. Let $q \in \mathbb{Z}$ be a prime and let $n = poly(q)$. Let $U_{s,\phi}$ be a uniform distribution on $\mathbb{Z}_q^n \times \mathbb{T}$ obtained by choosing a vector $\boldsymbol{a}_i \in \mathbb{Z}_q^n,\ \forall i \in \mathbb{Z}, 0 \leq i < n$ uniformly at random, choosing numbers $\boldsymbol{e}_i,\ \forall i \in \mathbb{Z}, 0 \leq i < n$ according to a probability distribution ϕ on $\mathbb{T}$.

A LWE Search problem asks the questions of finding $s \in \mathbb{Z}_q^n$ given the set $(\boldsymbol{a}_i, \boldsymbol{b}_i),\ i \in \mathbb{Z}, 0 \leq i < n$ where

$$\boldsymbol{b}_i = \frac{< \boldsymbol{a}_i, \boldsymbol{s} >}{q} + \boldsymbol{e}_i \tag{4.10}$$

and $\boldsymbol{s} \in \mathbb{Z}_q^n$ is a fixed sample from $U_{s,\phi}$.

Note that, $n = poly(q)$ denotes that $n \in \mathbb{N}$ is bounded by a polynomial such that $|n| < |g(n')|$ for some $n' \in \mathbb{N}$ where $g(n') = q^{n'}$.

Definition 14. *LWE Decision Problem*

Let $poly(q) = n$ be defined in LWE Search Problem. A LWE Decision Problem asks the question of given the set $S = \{(\boldsymbol{a}_i, \boldsymbol{c}_i)\}$ where $\boldsymbol{c}_i \in \mathbb{Z}_q^n,\ i \in \mathbb{Z},\ 0 \leq i < n$, decide whether $\boldsymbol{c}_i$'s are randomly sampled from $U_{s,\phi}$ or the set is an output defined above in a LWE Search Problem.

In [31], it has been showed that the above two problems are equivalent under the assumption of $n = poly(q)$. This also bounds the number of queries to the search problem to polynomial time.

Theorem 11. *A LWE Search and Decision problems are polynomially equivalent if $n = poly(q)$.*

Proof. ($\Longrightarrow$) Assume the existence of a LWE Search Oracle. Solve the problem to get a candidate $\mathbf{s} \in \mathbb{Z}_q^n$. Then calculate the following set.

$$\{\frac{< \mathbf{a}_i, \mathbf{s} >}{q} - \mathbf{b}_i\}, \quad i \in \mathbb{Z}, 0 \leq i < n \tag{4.11}$$

Now, if the set has the same distribution with the errors $\{\mathbf{e}_i\}$ then output YES otherwise output NO.

($\Longleftarrow$) Assume the existence of a LWE Decision Oracle. Then, construct the following set.

$$\{(\mathbf{a}_i + (r, 0, \ldots, 0), \mathbf{b}_i + \frac{rk}{q})\} \tag{4.12}$$

where $k \in \mathbb{Z}_q$ is the guessed value of the first coordinate of $\mathbf{s} \in \mathbb{Z}_q^n$ and $r \in \mathbb{Z}_q$ is sampled uniformly. Now, if the guess $k \in \mathbb{Z}_q$ was correct, the transformation takes the distribution $U_{s,\phi}$ to itself and to a uniform distribution otherwise. This question is answered by the LWE oracle. Since there are q values, q queries are enough to decide a coordinate of $\mathbf{s}$. The other coordinates are decided similarly and the result follows. □

The above proof shows that the number of queries to the decision oracle are bounded in polynomial time if $n = poly(q)$.

Definition 15. *RLWE Search Problem*

Let $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ be a ring and $n \in \mathbb{N}$ is a power of two. Let $a_i(x) \in R_q$ be a set of random polynomials, $e_i(x) \in R_q$ be a set of random polynomials with small coefficients, $s(x) \in R_q$ be a random polynomial again with small coefficients and

$$b_i(x) = a_i(x)s(x) + e_i(x) \tag{4.13}$$

$\forall i \in \mathbb{Z}$ such that $0 \leq i < n$.

A RLWE Search Problem asks the question of finding the unknown polynomial $s(x)$ given the set of $(a_i(x), b_i(x))$.

A random polynomial in R_q means that the coefficients are sampled from the set $\mathbb{Z}_q$ with uniform distribution. A random polynomial with small coefficients in this definition denotes a polynomial again with coefficients drawn from a bounded subset of $\mathbb{Z}_q$. In particular cases, this set is set to be the $\{\lfloor -q/4 \rfloor, \ldots, \lfloor q/4 \rfloor - 1\}$ with uniform distribution. Finally, errors $e_i(x)$ have the coefficients drawn from the same set but this time with a specific distribution, for instance a Gaussian.

The other version of this problem is as follows.

Definition 16. *RLWE Decision Problem*

Decide given $(a_i(x), b_i(x)),\ \forall i \in \mathbb{Z}, 0 \leq i < n$, *whether the polynomials* $b_i(x)$ *were constructed as* $b_i(x) = a_i(x)s(x) + e_i(x),\ \forall i \in \mathbb{Z}, 0 \leq i < n$ *or randomly generated from the ring* R_q.

4.2 BCNS Key Exchange Algorithm

The algorithm is based on the RLWE decision problem. To be able to define the algorithm in terms of Peikert ([28]), the following definitions are required.

Definition 17. *Let* $q \in \mathbb{N}$. *Define the modular rounding function*

$$\lfloor \cdot \rceil_{q,2} : \mathbb{Z}_q \to \mathbb{Z}_2 \tag{4.14}$$

$$x \mapsto \lfloor x \rceil_{q,2} = \lfloor \frac{2}{q}\, x \rfloor mod\ 2 \tag{4.15}$$

and the cross-rounding function

$$\langle \cdot \rangle : \mathbb{Z}_q \to \mathbb{Z}_2 \tag{4.16}$$

$$x \mapsto \langle x \rangle_{q,2} = \lfloor \frac{4}{q}\, x \rfloor mod\ 2 \tag{4.17}$$

Both functions are extended to elements of R_q *coefficient-wise. For* $f = f_{n-1}X^{n-1} + \ldots + f_1 X + f_0 \in R_q$, *define*

$$\lfloor f \rceil_{q,2} = (\lfloor f_{n-1} \rceil_{q,2}, \ldots, \lfloor f_0 \rceil_{q,2}) \tag{4.18}$$

$$\langle f \rangle_{q,2} = (\langle f_{n-1} \rangle_{q,2}, \ldots, \langle f_0 \rangle_{q,2}) \tag{4.19}$$

Definition 18. *The doubling function is defined as follows.*

$$dbl : \mathbb{Z}_q \to \mathbb{Z}_{2q} \tag{4.20}$$

$$x \mapsto dbl(x) = 2x - e \tag{4.21}$$

where e *is sampled from the set* $\{-1, 0, 1\}$ *with probabilities* $p_{-1} = p_1 = \frac{1}{4}$ *and* $p_0 = \frac{1}{2}$.

Lemma 3. *For odd* q, *if* $v \in \mathbb{Z}$ *is uniformly random and* $\bar{v} \xleftarrow{\$} dbl(v) \in \mathbb{Z}_{2q}$, *then* $\lfloor \bar{v} \rceil_{2q,2}$ *is uniformly random given that* $\langle \bar{v} \rangle_{2q,2}$ *is uniformly random.*

Definition 19. *The reconciliation function is defined as follows.*

$$rec(w,b) = \begin{cases} 0, & if w \in I_b + E \mod 2q, \\ 1, & otherwise, \end{cases} \tag{4.22}$$

where $b \in \{0,1\}$, $I_0 = \{0,1,\ldots \lfloor \frac{q}{2} \rceil - 1\}$, $I_1 = \{-\lfloor \frac{q}{2} \rfloor, \ldots, -1\}$ *and* $E = [-\frac{q}{4}, \frac{q}{4}) \subset \mathbb{R}$.

Lemma 4. *For odd* q, *let* $v = w + e \in \mathbb{Z}_q$ *for* $w, e \in \mathbb{Z}_q$ *such that* $(2e \pm 1 \mod q) \in E$. *Let* $\hat{v} = dbl(v)$. *Then* $rec(2w, \langle \hat{v} \rangle_{2q,2}) = \lfloor \hat{v} \rceil_{2q,2}$.

Definition 20. *BCNS Key Exchange Algorithm*

The algorithm defined in ([28]) is a realization of Peikert's Scheme. Rounding and cross-rounding functions are defined above and this algorithm allows parties to exchange keys of length 1024-bits.

Figure 4.1: BCNS Key Exchange Algorithm

Decision RLWE Parameters: $q = 2^{32} - 1, n = 1024$, χ_σ with $\sigma = 8/\sqrt{2\pi}$, $a \xleftarrow{\$} U(R_q)$		
Alice		**Bob**
$a, e \xleftarrow{\$} \chi$		$s', e' \xleftarrow{\$} \chi$
$b \leftarrow as + e \in R_q$	$\xrightarrow{b}$	$b' \leftarrow as' + e' \in R_q$
		$e'' \leftarrow \chi$
		$v \leftarrow bs' + e'' \in R_q$
		$\bar{v} \xleftarrow{\$} dbl(v) \in R_{2q}$
	$\xleftarrow{b',c}$	$c \leftarrow \langle \bar{v}_{2q,2} \rangle \in \{0,1\}^n$
$k_A \leftarrow rec(2b's, c) \in \{0,1\}^n$		$k_B \leftarrow \lfloor \bar{v}_{2q,2} \rceil \in \{0,1\}^n$

For this algorithm, parameters are chosen to be $n = 1024$, $q = 2^{32} - 1 = 4,294,967,295$, $\sigma = 8/\sqrt{2\pi} \approx 3.192$. Polynomial arithmetic is done by computing the discrete Fourier transform via Fast Fourier transform (FFT) algorithms.

4.3 New Hope

NewHope[5, 3] is one of the NIST PQC submission that is based on lattices. It has high performance due to the use NTT. It is currently a second round candidate. In this section, we present the basics of the NewHope and provide the the use of NTT in it.

The NewHope is a bundle of key encapsulation mechanisms (KEM) denoted as NewHope-CPA-KEM and NewHope-CCA-KEM. The security of the algorithms are based on the conjectured quantum hardness of the Ring Learning with Errors (RLWE) problems. These schemes are derivatives of the NewHope-SIMPLE [4]. NewHope-SIMPLE is semantically secure public-key encryption (PKE) scheme with respect to adaptive chosen plaintext attacks (CPA) denoted as NewHope-CPA-PKE. On the other hand, this public-key encryption is only used inside the key encapsulation mechanisms of NewHope-CPA-KEM and NewHope-CCA-KEM and not to be used as an independent CPA-secure PKE scheme. The reason for that is, the PKE scheme does not accept arbitrary length messages. The description of the PKE scheme is given in the following algorithms 1, 2 and 3.

Algorithm 1: NewHope-CPA-PKE Key Generation

1 **function** `NewHope-CPA-PKE.Gen()`
2 $\textit{seed} \xleftarrow{\$} \{0,\ldots,255\}^{32}$
3 $z \leftarrow \mathrm{SHAKE256}(64, \textit{seed})$
4 $\textit{publicseed} \leftarrow z[0:31]$
5 $\textit{noiseseed} \leftarrow z[32:63]$
6 $\hat{\mathbf{a}} \leftarrow \mathrm{GenA}(\textit{publicseed})$
7 $\mathbf{s} \leftarrow \mathrm{PolyBitRev}(\mathrm{Sample}(\textit{noiseseed}, 0))$
8 $\hat{\mathbf{s}} \leftarrow \mathrm{NTT}(\mathbf{s})$
9 $\mathbf{e} \leftarrow \mathrm{PolyBitRev}(\mathrm{Sample}(noiseseed, 1))$
10 $\hat{\mathbf{e}} \leftarrow \mathrm{NTT}(\mathbf{e})$
11 $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
12 **return** $(pk = \mathrm{EncodePK}(\hat{\boldsymbol{b}}, publicseed), sk = \mathrm{EncodePolynomial}(\hat{\boldsymbol{s}}))$

The main data structures of the algorithm are the elements of R_q and vectors. Other than those, byte arrays are employed. For instance, in the key generation algorithm $\textit{seed} \xleftarrow{\$} \{0,\ldots,255\}^{32}$ is used to sample a byte array of 32 uniform integers in the range 0 to 255.

$\mathrm{SHAKE256}(l, d)$ hash function [13] is employed for compression. Here, l is the number of output bytes and the input data is a byte array d. The amount of data to be absorbed is the length of d. For instance, $v \leftarrow \mathrm{SHAKE256}(64, seed)$ is used in the

Algorithm 2: NewHope-CPA-PKE Encryption

function NewHope-CPA-PKE.ENCRYPT(
  $pk \in \{0, \ldots, 255\}^{7 \cdot 4/n+32}$,
  $\mu \in \{0, \ldots, 255\}^{32}$,
  $coin \in \{0, \ldots, 255\}^{32}$
)
$(\hat{\mathbf{b}}, \textit{publicseed}) \leftarrow \text{DecodePk}(pk)$
$\hat{\mathbf{a}} \leftarrow \text{GenA}(\textit{publicseed})$
$\mathbf{s}' \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 0))$
$\mathbf{e}' \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 1))$
$\mathbf{e}'' \leftarrow \text{Sample}(coin, 2)$
$\hat{\mathbf{t}} \leftarrow \text{NTT}(\mathbf{s}')$
$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \text{NTT}(\mathbf{e}')$
$\mathbf{v} \leftarrow \text{Encode}(\mu)$
$\mathbf{v}' \leftarrow \text{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mathbf{v}$
$h \leftarrow \text{Compress}(\mathbf{v}')$
return $(c = \text{EncodeC}(\hat{\boldsymbol{u}}, \text{h}))$

Algorithm 3: NewHope-CPA-PKE Decryption

function NewHope-CPA-PKE.DECRYPT(
  $c \in \{0, \ldots, 255\}^{7 \cdot n/4+3 \cdot n/8}$,
  $sk \in \{0, \ldots, 255\}^{7 \cdot n/4}$
)
$(\hat{\mathbf{u}}, h) \leftarrow \text{DecodeC}(c)$
$\hat{\mathbf{s}} \leftarrow \text{DecodePolynomial}(sk)$
$\mathbf{v}' \leftarrow \text{Decompress}(h)$
$\mu \leftarrow \text{Decode}(\mathbf{v}' - \text{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}}))$
return (μ)

algorithm where a 32 byte random seed is hashed. The result is a byte array consisting 64 elements in the range $\{0, \ldots, 255\}$.

A common notatiton is used throughout the algorithm. Bracket notation is used to access element of a byte array (i.e. $v[i]$). Ranges are accessed via colons (i.e. $x \leftarrow v[i:j]$). In the algorithm, $r \leftarrow \{0, \ldots, 255\}^x$ means a byte array of length x. Similarly, $r \leftarrow R_q$ means that r is polynomial R_q with zero coefficients. Bit operators $\ll, \gg, |, \&$ are defined and used in the C programming language context. Implicit reductions occur when a left shift operation is executed. The modulus depends on the size of operands. The $a|b$ denotes a bit-wise 'or'. Besides, the $a\&b$ denotes a bit-wise 'and'. A byte $a[i]$ is converted to a positive integer z via $z = b2i(a[i])$. For hexadecimal representation, the prefix $0x$ is employed. Finally, $HW(b)$ is used to denote the Hamming weight of b.

In the algorithm, NewHope-CPA-PKE.ENCRYPT does not have direct access to a random number generator. The user supplies a seed $coin \in \{0, \ldots, 255\}^{32}$ and all randomness is generated from that. This seed should be generated by a true random value generator. Decryption is deterministic and does not require random values. A centered binomial distribution ψ_k of parameter $k = 8$ is used for the distribution of the RLWE secret and error. It is possible to sample from ψ_k for integer $k > 0$ by computing the following,

$$\sum_{i=0}^{k-1} b_i - b'_i \tag{4.23}$$

where $b_i, b'_i \in \{0, 1\}$ are uniform independent bits. The distribution ψ_k is centered (its mean is 0), has variance $k/2$. This gives a standard deviation of $\zeta = \sqrt{8/2}$. Sampling algorithm from ψ_8 is given in Algorithm 4.

The Sample function in the algorithm 4 takes care of the deterministic sampling. There are two input. First one is a 32-byte seed and the second one is an integer parameter $0 \leq \mathit{nonce} \leq 2^8$ for domain separation. This is because, it allows multiple polynomials to be sampled from the same seed. The output of Sample is $\mathbf{r} \in R_q$ where all n coefficients are independently distributed according to ψ_8.

In algorithm 5, GenA algorithm is presented. This function has single input called *seed*. The output is $\hat{\mathbf{a}} \in R_q$ and it is assumed to be in the NTT domain. This is

Algorithm 4: Deterministic sampling of polynomials in R_q from ψ_8^n

function `Sample(`$seed \leftarrow\in \{0,\ldots,255\}$, ***positive integer*** $nonce$`)`

r $\leftarrow R_q$

extseed $\leftarrow \{0,\ldots,255\}^{34}$

extseed$[0:31] \leftarrow$ *seed*$[0:31]$

extseed$[32] \leftarrow$ *nonce*

for $i = 0$ *to* $(n/64) - 1$ **do**

  $extseed[33] \leftarrow i$

  $buf \leftarrow \text{SHAKE256}(128, extseed)$

  for $j = 0$ *to* 63 **do**

    $a \leftarrow buf[2 * j]$

    $b \leftarrow buf[2 * j + 1]$

    $r_{64*i+j} = HW(a) + q - HW(b)\ mod\ q$

  endfor

endfor

return `(`$r \in R_q$`)`

due to the sampling of coefficients of the polynomial from a uniform distribution, NTT maps polynomials with uniform coefficients to polynomials with uniform coefficients. In the function GenA, $state \leftarrow SHAKE128Absorb(d)$ is employed to expand the seed. Inputs to this function is a byte array d. The output is a byte array of length 200. This byte array includes the internal state after absorbind d. Pseudo-randomness is obtained by $buf, state \leftarrow SHAKE128Squeeze(j, state)$. This function takes a positive integer j which is used to determine the size of the output blocks of SHAKE128.

Algorithm 5: Deterministic generation of $\hat{\mathbf{a}}$ by expansion of a seed

```
1  function GenA(seed ∈ {0,...,255}^32)
2  â ← R_q
3  extseed ← {0,...,255}^33
4  extseed ← seed[0 : 31]
5  for i from 0 to (n/64) − 1 do
6      ctr ← 0
7      extseed[32] ← i
8      state ← SHAKE128Absorb(extseed)
9      while ctr < 64
10         buf, state ← SHAKE128Squeeze(1, state)
11         j ← 0
12         for j < 168 and ctr < 64 do
13             val ← b2i(buf[j]) | (b2i(buf[j + 1]) ≪ 8)
14             if val < 5 · q then
15                 â_{i∗64+ctr} ← val
16                 ctr ← ctr + 1
17             end
18             j ← j + 2
19         end
20 end
21 return (â ∈ R_q)
```

It is possible to convert NewHope-CPA-PKE to an IND-CPA-secure KEM by using

Table 4.1: Parameter sets for NewHope Algorithm

Parameter Set	NewHope512	NewHope1024
Dimension n	512	1024
Modulus q	12289	12289
Noise Parameter k	8	8
NTT parameter $\zeta = \sqrt{\omega}$	10968	7
NTT parameter ω	3	49
NTT parameter ω^{-1}	8193	1254
NTT parameter ζ^{-1}	3656	8778
NTT parameter $n^{-1}\ mod\ q$	12265	12277
Decryption error probability	2^{-213}	2^{-216}
Claimed post-quantum bit-security	101	233
NIST Security Strength Category	1	5

Table 4.2: Parameter sets for NewHope Algorithm

Parameter Set	$\|pk\|$	$\|sk\|$	$\|ciphertext\|$
NewHope512-CPA-KEM	928	869	1088
NewHope1024-CPA-KEM	1824	1792	2176
NewHope512-CCA-KEM	928	1888	1120
NewHope1024-CCA-KEM	1824	3680	2208

the PKE scheme to convey a secret, K. In this scheme, coins and the secret K are derived by hashing random coins. These input random coins are not directly used since it might be extract information and analyse the state of random number generator. The final shared secret is derived from the secret K by hashing. The resulting NewHope-CPA-KEM algorithms are given in [4].

In [4], two parameter sets are specified for the NewHope algorithm. These are given in Table 4.1. Note that, the algorithm defined above corresponds to NewHope1024 with $n = 1024$. Similarly, 4.2 depicts possible public key, secret key and ciphertext sizes.

After a brief description of the algorithm, it is now possible delve into the details of the use of NTT in this algorithm. There is an interesting point, however, that should be noted. In [5], a version of this algorithm is given that employs reconciliation functions. These functions basically removes the errors from the exchanged key where

two parties have only approximates of. On the contrary, in this thesis, the version without those reconciliations [4] is discussed. Nevertheless, the optimizations given in [5] are also valuable so they are discussed first.

In [5], two implementations are mentioned (Portable C and AVX). In these implementations, different optimization pathways are taken for different kinds of operations. In this thesis, I will focus on the portable C implementation. This one aims for general purpose use, not specific to a particular architecture. There are several areas demanding optimization. These are NTT specific ones, arithmetic related ones and random sampling optimizations.

I start with NTT. Since $log\ q = 14$, ring elements are kept in $uint16\_t$, namely, *unsigned short*. Since $n = 1024 = 2^{10}$ and Fast Fourier Transform has a complexity of $\mathcal{O}(n\ logn)$, there are 10 butterfly stages where each stage has 512 butterflies. Authors mention that they have picked *Gentleman-Sande* type FFT (DIF) algorithm. In this algorithm, input should be bit-reversed for the output to be in correct places after the evaluation. However, it is possible to omit some of the operations as follows. First, NTT is not applied to $\hat{\mathbf{a}}$ at all since it is derived from a random seed, and the FFT of a random seed is also random, $\hat{\mathbf{a}}$ can directly be used as its FFT. Second, in the application of NTT to $\mathbf{s}$ and $\mathbf{e}$, bit-reversal operation can be omitted since they are also derived from a discrete binomial distribution. Nevertheless, the NTT cannot be omitted since $\hat{\mathbf{s}}$ and $\hat{\mathbf{e}}$ are chosen to be *small* (i.e. coefficients are small in q). Similarly, $\hat{\mathbf{t}}$ is computed in a similar manner and bit-reversal can be omitted. On the other hand, NTT^{-1} operations do still require bit-reversals so this optimization is not valid.

Now, it is possible to optimize each butterfly stage independently. There are 5120 butterflies in total each consisting an addition, a subtraction and a multiplication by the twiddle factor. Authors mention that Montgomery Forms are used with $R = 2^{18}$. This aligns with the fast 18-bit shift right operation implemented on a general purpose processing unit. Therefore, all twiddle factors $\omega^i,\ i \in \mathbb{Z}, 0 \leq i < 512$ are kept in memory as $2^{18}\omega^i$. Also, these factors should be kept in 32-bit unsigned integers since $18 + 14 = 32$. After each butterfly, a 32-bit integer value is reduced using Montgomery reduction. using the algorithm in Figure 4.2. In this algorithm, $R = 2^{18}$,

$m = a' * 12287$ since

$$R^{-1}R - n'n = 576 \times 262144 - 12287 \times 12289 = 1 \tag{4.24}$$

Therefore, it is an application of the REDC algorithm mentioned in the previous section. Note that only least significant 18-bits of m is taken into account since when multiplied by q the most significant bits will overflow 32-bit integer u anyway.

Figure 4.2: Montgomery reduction ($R = 2^{18}$)

```
1  uint16_t mred(uint32_t a) {
2    uint32_t u;
3    u = (a * 12287);
4    u &= ((1 << 18) - 1);
5    a += u * 12289;
6    return a >> 18;
7  }
```

There is subtle point to note however. This algorithm will fail for integers above $2^{32} - q(R-1) = 1073491969$. This can be observed by considering the maximum value of u which is $R - 1$ and the following.

$$a = a + u = 2^{32} - q(R-1) + (R-1)q = 2^{32} \tag{4.25}$$

Next, it is necessary to reduce after modular additions. For this task, algorithm in Figure 4.3 is employed. This routine reduces any 16-bit unsigned integer to an integer of at most 14-bits which is congruent modulo q.

Figure 4.3: Short Barrett Reduction

```
1  uint16_t bred(uint16_t a) {
2    uin32_t u;
3    u = ((uint32_t) a * 5) >> 16;
4    a -= u * 12289;
5  }
```

There is an interesting observation done by the authors. They realized that addition of two 2^{14} bits results in at maximum 2^{16} bits. Therefore, since short Barrett reduction can handle up to 16-bits, Barrett reductions are done only in every second level. The algorithm for this is given in Figure 4.4. In line 4 of this algorithm, $3 * q$ is added to

the operand of mred. This is to avoid unsigned underflow in the subtraction operation. Coefficients never grow more than 15-bits and $3q = 36867 > 2^{15}$.

Figure 4.4: The Gentleman-Sande butterfly inside odd levels of NTT computation. All $a[j]$ and W are of type *uint16_t*.

```
1    W = omega[jTwiddle++];
2    t = a[j];
3    a[j] = bred(t + a[j+d]);
4    a[j+d] = mred(W * ((uint32_t)t + 3*12289 - a[j+d]);
```

If we set $t = 2^{15} - 1$, $a[j + d] = 0$ and $W = q - 1$, we obtain $12288 * (2^{15} + 3q) = 855662592 < 2^{30}$ which is safe for REDC algorithm.

Next, I proceed into the details of the use of NTT in [4].

First, BitRev and PolyBitRev are defined as follows.

$$BitRev(v) = \sum_{i=0}^{log_2(n)-1} (((v \gg i)\ \&\ 1) \ll (log_2(n) - 1 - i)) \tag{4.26}$$

$$PolyBitRev(\mathbf{s}) = \sum_{i=0}^{n-1} s_i X^{BitRev(i)} \tag{4.27}$$

Similar to the previous paper, it is said that bit-reversals are omitted for the forward transformations as all inputs are only random noise. Forward NTT is defined as follows.

$$NTT(\mathbf{g}) = \hat{\mathbf{g}} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \tag{4.28}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \gamma^j g_j \omega^{ij}\ mod\ q \tag{4.29}$$

where $\gamma = \sqrt{\omega}$. The function NTT^{-1} is the inverse number theoretic transform. The computation is essentially the same except that it uses $\omega^{-1}\ mod\ q$.

$$NTT^{-1}(\hat{\mathbf{g}}) = \mathbf{g} = \sum_{i=0}^{n-1} g_i X^i, \tag{4.30}$$

$$g_i = (n^{-1}\gamma^{-i} \sum_{j=0}^{n-1} \hat{g}_j \omega^{-ij})\ mod\ q. \tag{4.31}$$

This concludes the discussion of NTT in NewHope scheme.

4.4 SWIFFT and SWIFFTX

SWIFFT is a collection of compression functions [21, 22, 29]. The security of it is based on the computationally hard lattice problems that provides this function with the property of being provably collision resistant. Therefore, it may be used in digital signatures and authentication protocols. However, the SWIFFT compression function has some undesirable properties such as linearity and lack of pseudo-randomness. In order to remedy this situation and remove these undesirable properties, a new compression function called SWIFFTX, one of the candidates of SHA-3 competition, was proposed in [6].

SWIFFTX has 256-byte input blocks and 65-byte output blocks. In the default configuration, input byte string is shaped as a 32 column matrix where each column comprises 8 bytes. The initial round first executes a Number Theoretic Transform (NTT) on each column and the result is a 64 by 32 matrix. This matrix is then multiplied by three different constant matrices $A_i, \forall i \in \mathbb{Z}, 0 \leq i < 3$ separately. Next, the diagonals of these three matrices are extracted to form three vectors of dimension 64. These vectors are then translated to byte strings by a translation algorithm and results are concatenated to form a single byte string. To provide non-linearity, this byte string is passed through a SBox before fed into the second round. The second round is similar to the first one yet only a single matrix multiplication is done where constants are provided by A_0. Only 25 columns of A_0 are used in matrix multiplication. Finally, there is a carry propagation operation at the end of the round that assembles the final byte of the output. In SWIFFTX, arithmetic is carried out in the finite field of characteristic $p = 2^8 + 1 = 257$. The total number of constants in the matrices $A_i, \forall i \in \mathbb{Z}, 0 \leq i < 3$ is $3NM$ where $N = 64$ is the number of rows and $M = 32$ is the number of columns. These constants are designed to be random and derived from the expansion of the transcendental number π via a certain algorithm (see [22]) in order values to fit into the given field.

In this section, I provide a description for the SWIFFT and the SWIFFTX. Let $p =$

$2^8 + 1 = 257$, $N = 64$ and $M = 32$ with $2N \mid p - 1$. These are the concrete parameters given in [22].

Definition 21. *Let $n \in \mathbb{Z}^+, \mathbf{X} \in \mathbb{F}_p^{n \times n}, \mathbf{Y} \in \mathbb{F}_p^n$. Define the column operator $C_j : \mathbb{F}_p^{n \times n} \to \mathbb{F}_p^n$ as $C_j(\mathbf{X}) = \mathbf{Y}$ where $y_i = x_{ij}$, $\forall i \in \mathbb{Z}$, $0 \leq i < n$.*

Definition 22. *Let $n \in \mathbb{Z}^+, \mathbf{X} \in \mathbb{F}_p^{n \times n}, \mathbf{Y} \in \mathbb{F}_p^n$ be a square matrix. Define the main diagonal operator $D : \mathbb{F}_p^{n \times n} \to \mathbb{F}_p^n$ as $D(\mathbf{X}) = \mathbf{Y}$ where $y_i = x_{ii}$, $\forall i \in \mathbb{Z}$, $0 \leq i < n$.*

Definition 23. *The Number Theoretic Transform employed in SWIFFT is defined as $NTT_N : \mathbb{F}_2^N \to \mathbb{F}_p^N$ where $NTT_N(u_0, \ldots, u_{n-1}) = (v_0, \ldots, v_{n-1})$, $v_j = \sum_{i=0}^{N-1} u_i \omega^{2ij}$, $\forall j \in \mathbb{Z}$, $0 \leq j < N$, and $\omega \in \mathbb{F}_p$ is the 2N-th root of unity such that $\omega^{2N} = 1$.*

Definition 24. *Define the unary operator $E_M : \mathbb{F}_2^{N \times M} \to \mathbb{F}_p^{N \times M}$ as $E(\mathbf{X}) = \mathbf{YX}$, where $\mathbf{Y} \in \mathbb{F}_p^{N \times N}$ with $y_{ij} = \omega^{2j} \in \mathbb{F}_p$, $\forall i, j \in \mathbb{Z}$, $0 \leq i, j < N$ and ω is the 2N-th root of unity such that $\omega^{2N} = 1$.*

It is now possible to define the first part of the SWIFFT compression function in terms of the primitives above.

Definition 25. *Let $\mathbf{U}, \mathbf{A} \in \mathbb{F}_p^{N \times M}$. Define $SWIFFT'$ as follows:*

$$SWIFFT'_M : \mathbb{F}_2^{N \times M} \times \mathbb{F}_p^{N \times M} \to \mathbb{F}_p^N$$
$$\mathbf{U} \times \mathbf{A} \mapsto D(\mathbf{V}\mathbf{A}^\mathbf{T})$$

where $C(\mathbf{V})_j = NTT_N \circ C_j \circ E_M(\mathbf{U})$, $\forall j \in \mathbb{Z}$, $0 \leq j < M$.

The above definition shows how to calculate j-th column of the matrix $\mathbf{V}$ denoted by $C(\mathbf{V})_j$. Finally, $\mathbf{V}$ is multiplied by $\mathbf{A}^\mathbf{T}$ and D is applied to obtain the result.

$SWIFFT'$ basically captures the crucial part of the $SWIFFT$. The rest deals with the translation of vectors with elements in $\mathbb{F}_p$ to vectors with elements in $\mathbb{F}_2$.

Definition 26. *Let $\mathbf{X} \in \mathbb{F}_p^N$ be a matrix and let $N' = N/8$. Define the map G as:*

$$G : \mathbb{F}_p^N \to \mathbb{F}_p^{N' \times N'}$$
$$\mathbf{X} \mapsto \mathbf{Y}$$

where $y_{ij} = x_{i+8*j}$, $\forall i, j \in \mathbb{Z}$, $0 \leq i, j < 8$.

Definition 27. *Let $N' = N/8$. Define the translation map T as:*

$$T : \mathbb{Z}_p^{N'} \to \mathbb{Z}_{256}^{N'} \times \mathbb{Z}_{256}$$
$$\mathbf{a} = \sum_{i=0}^{N'-1} a_i p^i \mapsto \mathbf{b} = \sum_{i=0}^{N'-1} b_i 256^i \times ((\mathbf{a} - (\mathbf{a}\ mod\ 256^{N'})) \gg N).$$

The above function basically translates vectors from base 256 to base 257 with the rightmost component being the carry. It is now possible to define SWIFFT.

Definition 28. *Let $\mathbf{U} \in \mathbb{F}_p^{N \times M}$ be an input matrix and $\mathbf{A} \in \mathbb{F}_p^{N \times M}$ be a constant matrix. Then, $SWIFFT_M$ is defined as:*

$$SWIFFT_M : \mathbb{F}_2^{N \times M} \times \mathbb{F}_p^{N \times M} \to \mathbb{Z}_{256}^{N} \times \mathbb{Z}_{256}$$
$$\mathbf{U} \times \mathbf{A} \mapsto \sum_{i=0}^{N'-1} \pi_1(a_i) 256^{iN'} \times \bigvee_{i=0}^{N'-1} \pi_2(a_i) 2^i$$

where $N' = N/8$, $\mathbf{U}' = SWIFFT'_M(\mathbf{U}, \mathbf{A})$, $a_i = T \circ C_i \circ G(\mathbf{U}')$ and π_j is the projection operator onto the j-th component.

SWIFFTX employs SWIFFT as a building block. However, there are two variations. The first round employs $SWIFFT_M$ with parameter $M = 32$ and the second round sets $M = 25$ denoted by M' in the following definition.

Definition 29. *Let $\mathbf{X} \in \mathbb{F}_2^{N \times M}$ be an input matrix and $\mathbf{A_i} \in \mathbb{F}_p^{N \times M}$, $\forall i \in \mathbb{Z}$, $0 \leq i < 3$ be three constant matrices. Then, $SWIFFTX_M$ is defined as follows:*

$$SWIFFTX_M : \mathbb{F}_2^{N \times M} \to \mathbb{Z}_{256}^{N+1}$$
$$\mathbf{X} \mapsto \pi_1(\mathbf{Z}) \,||\, \pi_2(\mathbf{Z})$$

where $\mathbf{Y_i} = SWIFFT_M(\mathbf{X}, \mathbf{A_i})$, $\forall i \in \mathbb{Z}$, $0 \leq i < 3$, $\mathbf{U} = \pi_1(\mathbf{Y_1}) \,||\, \pi_1(\mathbf{Y_2}) \,||\, \pi_1(\mathbf{Y_3})$, $\mathbf{V} = \pi_2(\mathbf{Y_1}) \,||\, \pi_2(\mathbf{Y_2}) \,||\, \pi_2(\mathbf{Y_3})$, $\mathbf{P} = 0 \in \mathbb{Z}_{256}^5$ is a five byte padding, $\mathbf{Z} = SWIFFT_{M'}(SBox(\mathbf{U} \,||\, \mathbf{V} \,||\, \mathbf{P}), \mathbf{A_0})$, $SBox$ is a lookup operation and $||$ is the concatenation operator.

In the following chapter, I present an efficient GPU implementation of this hash function.

CHAPTER 5

NUMBER THEORETIC TRANSFORMS IN SWIFFTX

In this chapter, number theoretic transforms in SWIFFTX are discussed. The composition of NTT_N and the multiplication function E_M is core to our discussion. However, it is sufficient to deal only with a column to understand the transform. For this purpose, the following definition is provided.

$$F : \{0,1\}^{64} \mapsto \mathbb{Z}_{257}^{64} \tag{5.1}$$

$$(x_0 \ldots, x_{63}) \to (y_0, \ldots, y_{63}) \tag{5.2}$$

where $y_i = \sum_{k=0}^{63}(x_k \cdot \omega^k) \cdot (\omega^2)^{ik} = \sum_{k=0}^{63} x_k \cdot \omega^{(2i+1)k} \in \mathbb{Z}_{257}$. In terms of the previous definitions, the equality $F = C_l \circ NTT_N \circ E_M$ holds for a column $l \in \mathbb{Z},\ 0 \leq l < M$.

Now if we write the index i for the input as $i = i_0 + 8i_1,\ i \in \mathbb{Z}, 0 \leq i < 63$ and k as $k = k_0 + 8k_1, k \in \mathbb{Z}, 0 \leq k < 63$ we then have the following:

$$y_{i_0+8i_1} = \sum_{k=0}^{63} x_k\, \omega^{(2i_0+16i_1+1)k}, \tag{5.3}$$

$$= \sum_{k_0=0}^{7}\sum_{k_1=0}^{7} x_{k_0+8k_1}\, \omega^{(2i_0+16i_1+1)(k_0+8k_1)}, \tag{5.4}$$

$$= \sum_{k_0=0}^{7}(\omega^{16})^{i_1 k_0}\left(\omega^{(2i_0+1)k_0} \cdot \sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} \cdot x_{k_0+8k_1}\right). \tag{5.5}$$

where $\omega^{16i_1\ 8k_1} = 1$ since $\omega^{128} = 1$. Now set $m_{k_0,i_0} = \omega^{(2i_0+1)k_0}$ and $t_{k_0,i_0} =$

$\sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} x_{k_0+8k_1}$ and the equation becomes

$$y_{i_0+8i_1} = \sum_{k_0=0}^{7} \omega^{16i_1k_0}(m_{k_0,i_0} \cdot t_{k_0,i_0}). \tag{5.6}$$

For each $k_0 \in \mathbb{Z}, 0 \leq k_0 < 8$, consider the vector $t_{k_0} = (t_{k_0,0}, \ldots, t_{k_0,7})$. This vector can take 256 different values and the result depends only on the input bits $x_{k_0}, x_{k_0+8\cdot 1}, \ldots, x_{k_0+8\cdot 7}$. This is the first observation done by the authors in [6]. Therefore in the implementation, the input is decimated as follows. Let input be $X_0, \ldots, X_7$ where $X_{k_0} = (x_{k_0}, x_{k_0+8\cdot 1}, \ldots, x_{k_0+8\cdot 7}) \in \mathbb{F}_2^8$ where $k_0 \in \mathbb{Z},\ 0 \leq k_0 < 8$. Now t_{k_0} can be found by a single table lookup operation such that $\mathbf{t}_{k_0} = T(X_{k_0})$. This table has 256 entries with entries in $\mathbb{Z}_{257}$. The multipliers $\mathbf{m}_{k_0} = (m_{k_0,0}, \ldots, m_{k_0,7})$ can also be pre-computed similarly.

The second observation done by the authors in [6] is the following. For a fixed $i_1 \in \mathbb{Z}, 0 \leq i_1 < 8$, the output $\mathbf{y} = F(\mathbf{x})$ can be partitioned into 8 vectors of dimension 8.

$$\mathbf{y}_{i_1} = (y_{8i_1}, y_{8i_1+1}, \ldots, y_{8i_1+7}) \in \mathbb{Z}_{257}^8 \tag{5.7}$$

Now for any $i_0 = 0, \ldots, 7$, the i_0-th component of $\mathbf{y}_{i_1}$ depends only on the i_0-th component of $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$. Besides, the same operation is executed on every coordinate. This allows parallelization of the computation of the output vectors $y_0, \ldots, y_7$ using SIMD instructions. Therefore, each $\mathbf{y}_{i_1}$ can be calculated as follows:

$$\mathbf{y}_{i_1} = \sum_{k_0=0}^{7} \omega^{16i_1k_0}(\mathbf{m}_{k_0} \cdot \mathbf{t}_{k_0}) \tag{5.8}$$

where vectors $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$ are component-wise multiplied. Finally, the resulting equation is a 8 dimensional Discrete Fourier transform using ω^{16} as an 8-th root of unity in $\mathbb{Z}_{257}$. It is shown previouly that it is possible to implement this in $\mathcal{O}(N\ logN)$ with $N = 8$ using FFT.

Now in the algorithm, ω is chosen to be $42 \in \mathbb{Z}_{257}$. This leads to $\omega^{16} \equiv 2^2 \equiv 4 \ mod \ 257$. Therefore, multiplication by $\omega^{16}, \omega^{32}, \omega^{48}$ can be implemented by left shifting by $2, 4, 6$ positions respectively.

Reductions in $\mathbb{Z}_{257}$ can be implemented as $x \equiv (x \ \& \ 255) - (x \gg 8) \ mod \ 257$. The modular reduction is still necessary since there is risk of underflow (i.e. the result being negative).

This concludes our discussion of NTT in SWIFFTX. In the next chapter, an efficient GPU implementation of SWIFFTX is given.

CHAPTER 6

AN EFFICIENT GPU IMPLEMENTATION OF SWIFFT AND SWIFFTX

6.1 Introduction

In this text, I present an efficient parallel implementation of SWIFFTX on GPU. In order to obtain high performance, we have optimized memory access according to memory transaction coalescing rules and optimized arithmetic operations using intrinsics. These are essential for realization of a fast implementation. Furthermore, shared memory is used to hold all intermediate values. Representing elements of $\mathbb{F}_{257}$ in *signed char* posed a certain challenge however, this is resolved by a map and another additional small routine. Moreover, the serial base 257 to base 256 translation algorithm is parallelized by using a binomial matrix. Experimental results (Section 6.6) show that our implementation is approximately 1000 times faster than the single-threaded x86 reference implementation and 10 times faster than the ported reference implementation. In terms of power consumption, our implementation performs 5 Watts better per 2^{16} hashes and 13 Watts better per 2^{19} hashes.

The rest of the chapter is organized as follows: In Section 6.2, GPU programming is discussed. In Section 6.3, the reference x86 implementation of SWIFFTX is presented. This implementation is ported to CUDA without applying a particular optimization. Its characteristic is evaluated to determine performance bottlenecks. In Section 6.4, a parallel version of the SWIFFTX algorithm is presented. In order to achieve our goal, we investigate further possible optimizations specific to the given hardware and propose solutions to discords between hardware and software. In Sec-

tion 6.5, further improvements such as improving cache hits rates and fixing memory bank conflicts are discussed. In Section 6.6, a methodology to evaluate performance of implementations is develop and results are obtained. These results are mostly GPU specific.

6.2 Compute Unified Device Architecture - CUDA

Compute Unified Device Architecture (CUDA) is very different from general purpose architectures such as x86 and AMD64. It has a great number of threads. A group of 32 threads is called a warp. This is the minimal number of threads that can be spawned simultaneously. Warps can be arranged to form a block. Therefore, the number of threads in a block is a multiple of 32. Blocks can also be arranged to form larger blocks called grids. Blocks and grids can be 1, 2 or 3 dimensional to fit into the requirements of the implementation. In this text, GP104 (GP104-400-A1) chip manufactured by NVIDIAis targeted. This chip is a member of the sixth generation NVIDIA Pascalmicro-architecture. In this particular chip, the number of threads in a block is limited to $2^{10} = 1024$ threads. A Streaming Multiprocessor (SM) in this chip can run two 1024-thread blocks simultaneously and there are 20 of them.

In terms of cache, GP104 has 48 KiB Unified Cache and 2 MiB L2 Cache. In CUDA, Unified Cache can be used for local/global loads/stores. L2 is a little bit larger and can be employed for caching global loads/stores. There is also a Texture Cache in Unified Cache which is used for loading constants. Among others, GP104 has another 96 KiB local fast memory per SM. This memory is called Shared Memory and it is particularly useful in terms of optimizing an implementation. Basically, the name *shared* comes from the fact that this memory area can be divided into smaller chunks and moreover, can be shared among a block. Although this region is declared to be fast as registers (2 clocks), it has a limited size and most of the time, determines the maximum number of warps that can be spawned simultaneously along with other factors such as number of registers per block, number of threads per block and number of threads per multiprocessor.

6.3 The Reference Implementation

Next we continue with the x86 reference implementation included in CryptoStreams [14]. The outline of this implementation is given in Figure 6.1. We have kept the variable names unaltered so that the reader can trace them back to the source code. Apart from the definition of SWIFFTX in the previous section, this implementation re-uses the common NTT output for the sake of performance. Assuming a word is 16-bits, elements of $\mathbb{F}_p$ are kept in words. Powers of 64-th root of unity are centered toward zero in the initialization stage. Moreover, NTT is performed via a lookup table. Similarly, SBox lookup is done on byte basis. Translation to base 256 from $\mathbb{F}_p$ is done in 6 iterations in a very efficient manner. Although this implementation is very efficient on x86, it still runs on a single-thread. We ported this implementation to CUDA without applying any further optimizations other than migrating constants to the device memory.

Figure 6.1: SWIFFTX Algorithm

```
1  ALGORITHM: SWIFFTX
2  INPUT:  uint8_t input[256]; // Input
3          int16_t A_0[N*M], A_1[N*M], A_2[N*M]; // Constants
4  OUTPUT: uint8_t output[65]; // Output
5
6  int32_t fftOut[N*M]; // NTT Output
7  int32_t sum[3*N]; // Three vectors of dimension N
8  uint8_t intermediate[3*N+8]; // Output of the first round
9
10 doNTT_32(input, fftOut); // 32 Column NTT
11 // Multipl. and Diagonal
12 doMultiply_and_Diag_3(fftOut, A_0, A_1, A_2, sum);
13 doTranslate_3(sum, intermediate); // Translate to base 256
14 doSBox(intermediate); // Apply SBox
15 doNTT_25(intermediate, fftOut); // 25 Column NTT
16 doMultiply_and_Diag_1(fftOut, A_0, sum); // Multipl. and Diagonal
17 doTranslate(sum, output); // Translate to base 256
```

Unlike x86, CUDA architecture provides a high number of registers. The maximum number of registers per block a CUDA kernel can employ is 255. In cases where more registers are required, spills occur and load/stores are served by Unified Cache. The

reference implementation is register rich. With a block size of a warp (32 threads), the compiler decides to use 228 registers (Figure 6.2) for the default optimization level 3 (-O3) although we haven't forced any loop to unroll. Since the implementation is for x86, it does not employ any shared memory.

According to GP104 specification [23], each SM has a 256 KiB register file. Assuming all 32 bits, a SM can hold up to 65536 registers simultaneously. This kernel has a block size of 32 (a single warp) and employs 228 registers. A calculation shows, each block requires 7296 registers. Therefore, each SM can run $65536/7296 \approx 8.9$ warps. NVIDIAVisual Profiler ($nvvp$, [26]) tells that the actual value is 8 warps. Moreover, each SM can run 2048 threads or 64 warps simultaneously, so the utilization is only 12.5%. Contrary to its high register usage, the kernel still requires an additional 8656 bytes stack frame which further slows the execution down.

Figure 6.2: Reference Implementation Compiler Stats

```
 ptxas info : 16786 bytes gmem
 ptxas info : Compiling entry function '_Z14swifftx_kernelPhS_i'
for 'sm_61'
 ptxas info : Function properties for _Z14swifftx_kernelPhS_i
8656 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
 ptxas info    : Used 228 registers, 340 bytes cmem[0]
```

6.4 A Parallel Implementation

In this section, we present a parallel CUDA implementation of the algorithm. The outline of our implementation is given in Figure 6.3.

Figure 6.3: Our Proposed Parallel SWIFFTX Algorithm

```
ALGORITHM: SWIFFTX
INPUT:  uint8_t input[256]; // Input
        int8_t A_0[N*M], A_1[N*M], A_2[N*M]; // Constants
OUTPUT: uint8_t output[65]; // Output
NUMBER OF THREADS PER BLOCK: 64

__shared__ int16_t S_fftOut[N*(M+2)]; // Adjusted NTT Output
__shared__ int16_t S_sum[3*N+12]; // Adjusted sum output
__shared__ uint8_t S_intermediate[4*N]; // Enlarged intermediate
uint32_t *_input = S_sum;
// Copy input to shared (4-bytes per thread)
doParallel_Copy(input, _input); __syncThreads();
// 32 Column NTT
doParallel_NTT_32((int8_t *)_input, S_fftOut); __syncThreads();
// Multiply and take the diagonal
doParallel_Multiply_and_Diag_3(S_fftOut, A_0, A_1, A_2, S_sum);
__syncThreads();
// Fix leaps
doParallel_Adjust(S_fftOut, S_sum); __syncThreads();
// Translate to base 256
doParallel_Translate_3(S_sum, S_intermediate); __syncThreads();
// Apply SBox
doParallel_SBox(S_intermediate); __syncThreads();
// 25 Column NTT
doParallel_NTT_25(S_intermediate, S_fftOut); __syncThreads();
// Multipl. and Diagonal
doParallel_Multiply_and_Diag_1(S_fftOut, A_0, S_sum);
__syncThreads();
// Translate to base 256
doParallel_Translate(S_sum, S_intermediate); __syncThreads();
// Copy results back to device memory
doParallel_Copy(intermediate, output);
```

We select 64 threads, 2 warps per block. This number matches the number of rows of constant matrices A_i, $\forall i \in \mathbb{Z}$, $0 \leq i < 3$. The details of the proposed parallel

implementation are as follows.

First, we reserve enough shared memory per block for fast access to intermediate values. These intermediate values are NM words for the NTT output (*fftOut[]*), $3N$ words for the diagonals (*sum[]*) and $3N + 8$ bytes for the output of the first round (*intermediate[]*). In total, $(NM)2 + (3N)2 + (3N + 8) = 4680$ bytes. To enable fast access to 256 bytes hash input, at the beginning of the kernel, we copy the input to a shared memory location, specifically, to the space reserved for the diagonal output. This space will not be used until NTT is completed therefore we can use it temporarily. Copy is implemented by pointer dereferencing. A 64-thread block can copy the input in a single step $(256\ bytes = 64 \times 4\ bytes)$. For the hash output, only 17 threads are running, the others are idle $(17 \times 4\ bytes = 68\ bytes > 65\ bytes)$. The input is fetched from memory via 128-byte transactions, obeying the memory coalescing rules. Similarly, the hash output is written to the device memory via 64-byte transactions almost all the time except the last 8 bytes.

Now, NTT for a column can be done in 8 steps or strides. Therefore, we divide the NTT into 8 strides where each stride i is responsible for output row $8k + i, \forall k \in \mathbb{Z},\ 0 \leq k < 8$. A 64-thread block can process 8 columns at once and 32 columns in 4 steps. Furthermore, the multiplication at the beginning of the NTT is transformed into a lookup. The size of the lookup table is $256 \times 8 \times 8$ words with values in $\mathbb{F}_{257}$. These values are constant and served by the Texture Cache. At the end of the NTT, we transpose the output to help the inner product operation in the next stage. Reduction of the field elements is accomplished by the following macro:

$$\#define \quad Q\_REDUCE(a) \quad (((a)\ \&\ 0xFF) - ((a) \gg 8))$$

Basically, this macro subtracts the $8 - 15$th bits from the $0 - 7$th bits. This reduction is an instrinsic property of the nega-cyclic field.

We have the NTT output in shared memory. We need to calculate three diagonals for products $A_i V^T, \forall i \in \mathbb{Z},\ 0 \leq i < 3$. Although this seems to be a straightforward calculation, Pascal tuning guide [25] informs that the multiplication is a multi-clock operation in GP104 and the compiler can compile a single multiplication up to 20 instructions. To remedy this situation, we employ an intrinsic called *__dp_2a*, a two-way dot product. The definition of this operator is given in Figure 6.4 (PTX Manual

[24], Section 9).

Figure 6.4: dp2a two-way dot product-accumulate operator

```
Syntax:
 dp2a.mode.atype.btype d, a, b, c;
 .atype = .btype = { .u32, .s32 };
 .mode = { .lo, .hi };

Description:

Two-way 16-bit to 8-bit dot product which is accumulated
in 32-bit result. Operand a and b are 32-bit inputs. Operand
a holds two 16-bits inputs in packed form and operand b holds
4 byte inputs in packed form for dot product. Depending on the
.mode specified, either lower half or upper half of operand b
will be used for dot product. Operand c has type .u32 if both
.atype and .btype are .u32 else operand c has type .s32 .

Semantics:
 d = c;
 // Extract two 16-bit values from a 32-bit input and sign or
 // zero extend based on input type.
 Va = extractAndSignOrZeroExt_2(a, .atype);
 // Extract four 8-bit values from a 32-bit input and sign or
 // zero extend based on input type.
 Vb = extractAndSignOrZeroExt_4(b, .btype);
 b_select = (.mode == .lo) ? 0 : 2;
 for (i = 0; i < 2; ++i) {
     d += Va[i] * Vb[b_select + i];
 }
```

Now we face the problem that the entries in matrices A_i do not fit into *int8_t*'s (*signed char*). According to C++11 standard, *signed char* can hold values from -128 to 127 if the compiler employs *Two's complement* representation. Fortunately, the CUDA compiler *nvcc* employs *Two's complement* representation; therefore, we can map our field according to the following function.

$$f : \mathbb{F}_{257} \rightarrow Int8$$

$$f(a) = \begin{cases} a & if\ 0 \leq a \leq 127, \\ 127 & if\ a = 128, \\ a - 257 & otherwise. \end{cases}$$

This representation is different than the *diminished-one number system* employed in [16]. In [16], $\mathbb{F}_{257}$ is considered to be the integers in the range 0 to 256 inclusive and the field is mapped to 8-bits by subtracting 1 from each element while excluding the zero. The zero case is detected by an additional signal and handled exclusively. However, in GPU, we have no way of knowing whether a value is zero or not unless a predicate is executed. Unfortunately, predicates are sources of divergence therefore very expensive especially in loop bodies, hence we propose a slightly altered approach by defining the above function f. The codomain of f is selected to be 8-bit $signed\ char$ just to make it compatible with the $\_\_dp\_2a$ operator. We have determined 23 $+128$'s in $A_i, \forall i \in \mathbb{Z},\ 0 \leq i < 3$. There are nine in A_0, seven in A_1, and seven in A_2. We first replace those values with $+127$'s and do the multiplication. After computing the diagonal, we add missing values by a small routine called *doParallel_Adjust* which only employs 23 of the 64 threads in a block. This calculation is done as follows:

$$\begin{aligned} sum_j &= \sum_i a_{ji} \times fftOut_{ij}, \\ sum_j &= \sum_i b_{ji} \times fftOut_{ij} + \sum_i 128 \times fftOut_{ij}, && b_{ij} \neq 128, \\ sum_j &= \sum_i b_{ji} \times fftOut_{ij} + \sum_i 127 \times fftOut_{ij} + \sum_i fftOut_{ij}, \\ sum_j &= \sum_i c_{ji} \times fftOut_{ij} + \sum_i fftOut_{ij}, && c_{ij} \neq 128. \end{aligned}$$

with $\forall j \in \mathbb{Z},\ 0 \leq j < N$ and $\forall i \in \mathbb{Z},\ 0 \leq i < M$. Note that, the rightmost summation on the last line is the residue that needs to be added to its respective row j. To keep $doParallel\_Adjust$ procedure simple, we represent a particular adjustment in a dword. The first byte is the matrix the entry is in, the second is the row, the third is the column and the fourth is always zero. The last byte is kept for the sake of memory alignment. Totally, it consumes $23 \times 4 = 92$ bytes.

Since $\_\_dp\_2a$ is a two-way dot product, 32 columns can be processed totally in 8 calls to variants $\_\_dp\_2a\_lo$ and $\_\_dp\_2a\_hi$. Specifically, this means fetching 2 dwords from *fftOut*, a dword from A_i and computing the dot product twice using each variant once. This is a prominent improvement over the original iteration count of 32 and now, the loop can be unrolled without overloading the instruction fetch queue. Moreover, this approach also halves the memory transaction size required for fetching the entries in the matrices A_i.

Next step is to translate the diagonal entries in $\mathbb{F}_{257}$ to $\mathbb{F}_{256}$. In the reference implementation this is done efficiently in 6 iterations. However, each column has to be processed by a single thread. To make it parallel, we employ the following binomial matrix:

```
typedef int8_t swift_int8_t;
swift_int8_t binom1[8*8] =
  1, 1, 1, 1,   1,  1,  1,  1,
  0, 1, 2, 3,   4,  5,  6,  7,
  0, 0, 1, 3,   6, 10, 15, 21,
  0, 0, 0, 1,   4, 10, 20, 35,
  0, 0, 0, 0,   1,  5, 15, 35,
  0, 0, 0, 0,   0,  1,  6, 21,
  0, 0, 0, 0,   0,  0,  1,  7,
  0, 0, 0, 0,   0,  0,  0,  1,
;
```

This matrix is based on the fact that the equation $257^n = (256+1)^n = \sum_i \binom{n}{i} 256^i$ holds. Therefore, the elements b_{ij} of binom1 are defined as follows:

$$b_{ij} = \begin{cases} \binom{j}{i} & if\ i \leq j, \\ 0 & otherwise, \end{cases} \qquad \forall i, j \in \mathbb{Z},\ 0 \leq i, j \leq 7.$$

We compute the product of this $binom1$ matrix and the vector sum again using $\_\_dp\_2a$ operator. Then for 24 columns, we serially propagate carry bits using only 24 threads. Finally using three threads we propagate three final carry bytes and write them to the 25-th column.

Next, SBox lookup is done. In SWIFFTX, an 8 by 8 bits SBox is employed to provide non-linearity and this table is accessed byte by byte. Inputs and outputs are $3N + 8 = 204\ bytes$ long. In GP104, shared memory banks are 4-bytes wide. Processing the input byte by byte therefore creates 4 times more shared memory write transactions than necessary. Instead, we lookup 4 values, combine them using logical shifts and write them at once to comply with the physical shared memory structure. This concludes the first stage of the algorithm.

Second stage starts with a NTT executed on 25 columns. This does not fit well into our 64-thread per block implementation. We execute three and a half NTT iterations to process 25 columns. Adjustment is done only on eight values since the ninth $+128$ is in column 30. Translation to $\mathbb{F}_{256}$ is applied using the same technique but this time the output is only $8 + 1 = 9$ columns. Finally, the 65-byte output in the shared memory is written to device memory dword by dword to comply with the memory transaction coalesing rules. This finalizes major optimizations done on the algorithm.

6.5 Further Improvements and Occupancy Analysis

In Pascal micro-architecture, shared memory is divided into 16 banks where each bank is 4 bytes. This physical constraint leads to conflicts while writing the NTT output. To overcome this situation, we add two unused rows to the transposed NTT output and adjust pointer arithmetic accordingly. Now, the NTT output is $(M+2)N = 34 \times 64$ words. In pointer arithmetic, multiplication by 34 is implemented as shift by 5 plus shift by 1. However, this introduces latency when compared to a single shift. Similarly, we add four rows to *sum* area, it is $3N + 12$ words now. Finally, to prevent any other alignment issues, we set the size of the *intermediate* area to $4N$ bytes. Since we copy the input to a shared memory location at begining of the kernel, we do not face any global memory access inefficiencies related to the input. Similarly, the output is written directly from shared memory to the global memory at the end of the kernel therefore it is efficient. However, the size of the output is 65 bytes. If several thousands of hashes are calculated in bulk, this leads to an alignment issue for the output of the consecutive hashes. Therefore, we modify the output size and set it to 72 bytes to prevent any issues of this kind.

Now, NTT lookup table is of size $256 \times 8 \times 8$ words. Each lookup returns a word. $nvvp$ shows an inefficiency in global load L2 transaction, specifically, the ideal is 2 but the current is 4 transactions per access. This can be remedied only if the lookup returns a dword. However, the table size in that case becomes too large to fit into L2 Cache therefore left unoptimized. Similarly, $nvvp$ shows an inefficiency of 7.5 to 1 L2 transactions per access while looking up the table SBox. We have tried to implement the same routine by an 16×16 bits SBox yet the speed is reduced, therefore we left it as is.

Finally, we calculate the occupancy. In one hand, *nvcc* compiler tells that our kernel employs 48 registers. GP104, register file is 256 KiB, assuming all 4 bytes, there are 65536 registers in total per SM. A thread employs 48 and a block employs $48 \times 64 = 3072$ registers. Dividing 65536 by this number leads to ≈ 21.3 block limit. On the other hand, we have the following definitions:

```
__shared__ int16_t S_fftOut[(M+2)*N];
__shared__ int16_t S_sum[3*N+12];
__shared__ unsigned char S_intermediate[4*N];
```

Now, *S_fftOut* is $(32+2) \times 64 \times 2 = 4352$ bytes, *S_sum* is $[(3 \times 64) + 12] \times 2 = 408$ bytes and *S_intermediate* is $4 \times 64 = 256$ bytes. Therefore, Shared Memory usage is 5016 bytes in total. Dividing the Shared Memory size 96 Kib by this number leads to ≈ 19.6 blocks. Therefore, our kernel has a block limit of $min(21.3, 19.6) = 19.6$. This leads to $19 \times 64 = 1216$ threads per SM or in other words we have $1216/2048 = 0.594$, 59.4% occupancy per SM.

6.6 Methodology and Results

All results are obtained on an IntelE5410 CPU system where Linux version is 4.14.104, GNU glibc version is 2.27, CUDA version is 10.0 and NVIDIAdriver version is 415.27. The method the results are obtained is as follows. Each test round consists of 2^{14} hashes. First, there is a step to warm the CPU and the GPU up for 10 rounds. Then, we generate test data for each set of input and sequentially run the algorithm

on CPU, then on the GPU and collect the results. Generated input data is classified as follows: *(i) All zeroes:* weight 0/byte, *(ii) All ones:* weight 8/byte, *(iii) All random:* non-constant random weight/byte, *(iv) All random:* weight 4/byte.

This classification allows us to see whether or not the Hamming weight of the input does effect the performance of our kernel. All random data is generated by glibc *random(3)*. For *all random weight 4/byte* test, we employ Fisher-Yates shuffling algorithm [11]. We have generated enough random data for 2^{14} input blocks (2^{22} $bytes$) per test round. Furthermore, we set the affinity of the process via *sched_affinity(2)* to CPU 0 to get consistent results, avoid kernel rescheduling and cache invalidation.

The results are given in Table 6.1. These results are acquired using GNU gprof v2.31.1 and nvprof v10.0.130 profilers. Table 6.1 shows execution times of different implementations. First one is the x86 reference implementation, the second, GPU ported reference implementation and the last one is our parallel implementation. These results strongly indicate that the Hamming weight of the input is irrelevant, hence all data in the following tables are collected using non-constant weight *All random* data set. Table 6.2 depicts cache hit rates. Table 6.3 depicts memory throughput metrics obtained by the profiler. Figure 6.5 depicts kernel stall reasons of implementations. For the reference, test device properties are also included in Table A.1.

Table 6.1: Experimental Results, Test Round: 2^{14} hashes

	Intel XeonE5410 Reference Impl.	*NVIDIA GeForce GTX1080 Ported Reference Impl.*	*NVIDIA GeForce GTX1080 Our Parallel Impl.*	*Unit*
All zeroes	3.50×10^5	3.76×10^3	3.78×10^2	μsec
All ones	3.50×10^5	3.75×10^3	3.78×10^2	μsec
All random	3.50×10^5	3.77×10^3	3.80×10^2	μsec
All random weight 4	3.50×10^5	3.76×10^3	3.80×10^2	μsec

Table 6.1 shows almost 10x increase in speed compared to the ported reference implementation. Global memory accesses in our implementation are very efficient. For *all random* test, $nvvp$ shows global store efficiency of 70.8% and global load efficiency of 74.7%. The kernel employs a total of 5016 bytes of shared memory per block. Un-

fortunately, shared memory efficiency is only 52.2%, nevertheless, it is compensated since it is very fast.

Table 6.2: Cache Hit Rates

Metric	*Description*	*Reference Impl.*	*Parallel Impl.*
tex_cache_hit_rate	Unified Cache Hit Rate	67.05%	96.22%
l2_tex_hit_rate	Hit rate at L2 cache for all requests from texture cache	9.92%	33.75%
global_hit_rate	Hit rate for global load and store in unified l1/tex cache	92.77%	92.27%
local_hit_rate	Hit rate for local loads and stores	50.12%	0.00%

The use of shared memory makes data access very fast. However, it is limited only upto 96 KiB per SM therefore determines the number of warps spawned simultaneously. Parallel kernel requires only 48 registers. Decreasing this number via *__launch_bounds* leads to spill loads and stores degrading the performance. This number is sufficient for 19 warps to be spawned simultaneously. Our kernel performes L2 Cache hit rate of 33.7% and Unified Cache hit rate of 96.2% in the very same test (Table 6.2). These numbers indicate caches are efficiently utilized. Also, measured occupancy per SM is 57.4%. Since this is above 50%, it is enough to hide the arithmetic latency of the ALU inside GP104. This is discussed in detail in [36].

In Table 6.3, memory throughput metrics are given. First of all, system memory access is negligible in both kernels since data is copied to the device memory beforehand. Next, the device memory usage is reduced making it a bottleneck no more. Since our kernel employs a less number of registers, there is no local load or store. Global load throughput is increased by five times and stores are reduced by a half. Similarly, it is possible to observe Shared Memory throughput which makes a big difference in our parallel implementation. Unified Cache throughput is increased since we instruct the assembler to cache everything via the flag *(-Xptxas -dlcm=ca)*. This is also the reason for the reductions in L2 throughputs.

Figure 6.5 depicts the percentage of stall reasons per kernel. In the reference implementation, kernel stalls 29% percent due the memory dependencies. Also loading

Table 6.3: Memory Throughput Metrics

Metric	*Description*	*Reference Impl.*	*Parallel Impl.*
sysmem_read_throughput	System Memory Read	0.00000B/s	0.00000B/s
sysmem_write_throughput	System Memory Write	41.824KB/s	475.05KB/s
dram_read_throughput	Device Memory Read	132.83GB/s	11.888GB/s
dram_write_throughput	Device Memory Write	70.031GB/s	6.5270GB/s
local_load_throughput	Local Memory Load	265.30GB/s	0.00000B/s
local_store_throughput	Local Memory Store	72.083GB/s	0.00000B/s
gld_throughput	Global Load	117.50GB/s	580.49GB/s
gst_throughput	Global Store	8.4957GB/s	4.4536GB/s
shared_load_throughput	Shared Memory Load	0.00000B/s	1644.9GB/s
shared_store_throughput	Shared Memory Store	0.00000B/s	914.47GB/s
tex_cache_throughput	Unified Cache	327.27GB/s	1101.5GB/s
l2_tex_read_throughput	L2 (Texture Reads)	138.68GB/s	71.209GB/s
l2_tex_write_throughput	L2 (Texture Writes)	80.579GB/s	4.4536GB/s
l2_read_throughput	L2 (Reads)	139.02GB/s	71.643GB/s
l2_write_throughput	L2 (Writes)	80.579GB/s	4.4548GB/s

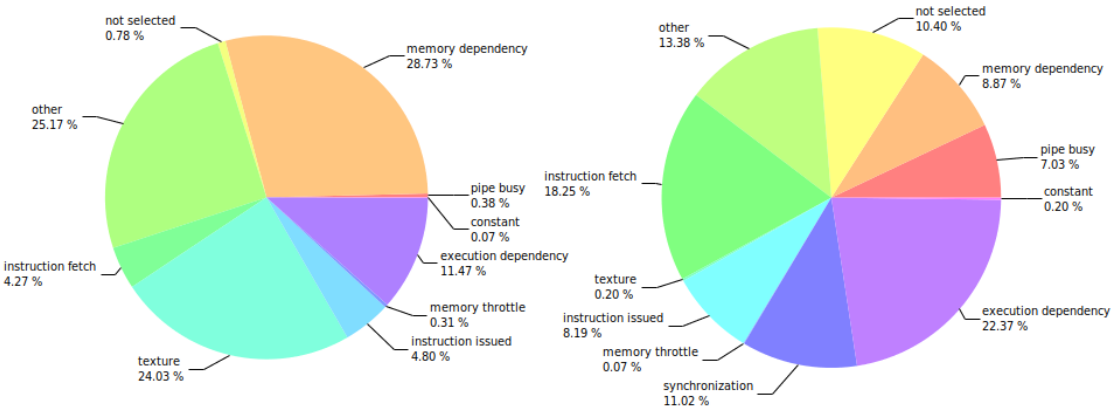
constants from Texture Cache generates a lot of traffic (24%). Moreover, it is not possible to learn what is included in *stall_other* (29%) so, it is better to keep it low under normal circumstances. In the parallel kernel, the largest percentage is owned by the execution dependency. In order to lower this value, it is possible to unroll loops so that compiler can move instructions around and optimize execution dependency. However, all of the loops in our kernel other than the one wrapping the NTT iterations are already unrolled so there is nothing that can be done. Unrolling that particular loop leads to register spills so we left it as is. The actual unrolling effect can also be observed by the 18% *stall_inst_fetch* metric. Too much unrolling is likely to overload the instruction fetch queue. In our case, the above configuration works well. Other metrics are around 10% and almost equally distributed. This is an indication of a balance between trade-offs.

Additional user-space benchmarking shows that we can calculate 2^{14} hashes in 420 milliseconds on a single x86 thread. The same can be achieved only in 4 milliseconds on the test device. This data also includes the duration of copying input to the device memory and getting it back to system memory over PCIe bus. According to these indicators, the throughput of x86 implementation is $2^{14} \times 2^8\ bytes/420\ msecs\ =$

$4\ MiB/420\ msecs \approx 10\ MiB/sec$ per thread while the throughput of our CUDA implementation is $2^{14} \times 2^8\ bytes/4\ msecs = 4\ MiB/4\ msecs \approx 1\ GiB/sec$ where $2^8\ bytes$ is the hash input block size.

Power consumption metrics have also been collected using nvprof profiler (Table B.1, B.2). The program is run for only a single test round without a warmup stage. Collected data shows on *adaptive* power mode, our board consumes 40W per test round on Reference Implementation and 35W on Parallel Implementation. This suggests our implementation consumes almost 5 Watts less on *adaptive* mode per test round. These values become 56W to 43W when test round hash count is increased to 2^{19}. The difference is almost 13 Watts.

Figure 6.5: Kernel Stall Reasons, Reference Impl. (left) vs Our Parallel Impl. (right)



CHAPTER 7

CONCLUSION

The NIST's PQC standardization process is a response to advances in the development of quantum computers. These quantum computers exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If we assume the existence of large-scale quantum computers then, they will be able to break current public-key cryptosystems. These systems consist of digital signatures and key-establishment schemes. Quantum computers will have no a drastic impact on symmetric-key cryptosystems, as said before doubling the keyspace would restore the original security level.

Round 1 of the standardization process is over and the report can be found in [3]. In this report, it is mentioned that 82 candidates were submitted and 69 first-round candidates were found eligible for processing. The criteria for eligibility includes provisions for reference and optimized C code implementations, known-answer tests, a written specification and required intellectual property statements. In addition, algorithms are required to be implementable in a wide range of hardware and software platforms.

NIST selected 26 of those 69 candidates to be qualified for the second round. The security, cost and performance, and algorithm and implementation characteristics of a candidate in selecting the second-round candidate is considered in this process.

In the report, NIST not only considered the attacks that directly demonstrated that a candidate fell short of NIST's stated security targets but also the attacks that brought the candidate's underlying security assumptions into question.

Figure 7.1: Second Round Candidates

BIKE	LEDAcrypt	Rainbow
Classic McEliece	LUOV	ROLLO
CRYSTALS-DILITHIUM	MQDSS	Round5
CRYSTALS-KYBER	NewHope	RQC
FALCON	NTRU	SABER
FrodoKEM	NTRU Prime	SIKE
GeMSS	NTS-KEM	SPHINCS+
HQC	Picnic	Three Bears
LAC	qTESLA	

The second round qualifiers are given in Table 7.1. The list contains 17 public-key encryption and key-establishment schemes and 9 digital signature schemes. In this thesis, one of the PKE scheme candidates NewHope is discussed.

The conference for the second round was held in August 2019, at the University of California Santa Barbara. In 2020, it is planned to either select finalists for a final round or select a small number of candidates for standardization.

Another subject discussed in this thesis is SWIFFTX. It is one of the lattice based hash function that provides provable collision resistance and pseudo-randomness. We have presented an efficient parallel implementation of SWIFFTX on GPU. Our tests have showed that the proposed implementation is approximately 1000 times faster than the single-thread x86 implementation and 10 times faster than the ported reference implementation. Moreover, the throughput is also increased by 100 times. In terms of power consumption, our implementation performs 5 Watts better per 2^{16} hashes and 13 Watts better per 2^{19} hashes.

It should be noted that there are newer architectures such as Volta and Turing than Pascal, a member of the sixth generation CUDA. These newer generations have higher memory bandwidth and more computation capabilities. Furthermore, the technology called Independent Thread Scheduling (ITS) is built into those new architectures. This technology will probably allow GPU's to utilize resources more efficiently in terms of scheduling and synchronization and deliver more speed and throughput if properly implemented. First idea basically aims to increase the occupancy. It might

be possible to implement a version of the algorithm that does not employ any shared memory instead, passes data across threads via Warp Shuffling. Consequently, this new implementation and the one that uses shared memory can be run simultaneously in the presence of ITS and hence a higher occupancy will be achieved. On the other hand, this might not lead to a significant improvement since we are still facing the burden of field arithmetic assigned to ALU. The second idea might target the time lost during synchronization. Figure 6.5 indicates that our kernel is stalled by synchronization primitives by 11% percent. This situation might be improved by ITS and defining explicit memory reads and writes using volatile keyword.

REFERENCES

[1] M. Ajtai, The shortest vector problem in l2 is np-hard for randomized reductions, in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 10–19, 1998.

[2] S. Akleylek, Ö. Dağdelen, and Z. Y. Tok, On the efficiency of polynomial multiplication for lattice-based cryptography on gpus using cuda, in *International Conference on Cryptography and Information Security in the Balkans*, pp. 155–168, Springer, 2015.

[3] G. Alagic, G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, et al., *Status report on the first round of the NIST post-quantum cryptography standardization process*, US Department of Commerce, National Institute of Standards and Technology, 2019.

[4] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, Newhope without reconciliation., IACR Cryptology ePrint Archive, 2016, p. 1157, 2016.

[5] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, Post-quantum key exchange—a new hope, in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 327–343, 2016.

[6] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, Swifftx: A proposal for the sha-3 standard, Submission to NIST, 2008.

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

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

[9] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*, CRC Press, 1999.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT press, 2009.

[11] R. Durstenfeld, Algorithm 235: random permutation, Communications of the ACM, 7(7), p. 420, 1964.

[12] R. P. Feynman, Simulating physics with computers, International journal of theoretical physics, 21(6), pp. 467–488, 1982.

[13] P. FIPS, Secure hash algorithm-3 (sha-3) standard: Permutation-based hash and extendable-output functions, National Institute for Standards and Technology (NIST), 202(0), 2014.

[14] C. for Research on Cryptography and B. C. R. Security, Masaryk University, Tool for generation of data from cryptoprimitives (block and stream ciphers, hash functions), `https://github.com/crocs-muni/CryptoStreams`, [Online; accessed December-2018].

[15] L. K. Grover, A fast quantum mechanical algorithm for database search, arXiv preprint quant-ph/9605043, 1996.

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

[17] J. Hoffstein, J. Pipher, J. H. Silverman, and J. H. Silverman, *An introduction to mathematical cryptography*, volume 1, Springer, 2008.

[18] W.-K. Lee, S. Akleylek, W.-S. Yap, and B.-M. Goi, Accelerating number theoretic transform in gpu platform for qtesla scheme, in *International Conference on Information Security Practice and Experience*, pp. 41–55, Springer, 2019.

[19] H. W. Lenstra, A. K. Lenstra, L. Lovfiasz, et al., Factoring polynomials with rational coeficients, 1982.

[20] R. Lidl and H. Niederreiter, *Finite fields*, volume 20, Cambridge university press, 1997.

[21] V. Lyubashevsky and D. Micciancio, Generalized compact knapsacks are collision resistant, in *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, pp. 144–155, 2006.

[22] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, Swifft: A modest proposal for fft hashing, in *International Workshop on Fast Software Encryption*, pp. 54–72, Springer, 2008.

[23] NVIDIA, GeForce GTX 1080 Whitepaper, `https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce%2FGTX%2F1080%2FWhitepaper%2FFINAL.pdf`, [Online; accessed December-2018].

[24] NVIDIA, Parallel Thread Execution ISA, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, [Online, accessed April 2018].

[25] NVIDIA, Pascal Tuning Guide, https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html, [Online, accessed April 2018].

[26] NVIDIA, Visual Profiler, https://docs.nvidia.com/cuda/profiler-users-guide/index.html%23getting-started, [Online; accessed April 2018].

[27] C. Nvidia, Nvidia cuda c programming guide, Nvidia Corporation, 120(18), p. 8, 2011.

[28] C. Peikert, Public-key cryptosystems from the worst-case shortest vector problem, in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 333–342, ACM, 2009.

[29] C. Peikert and A. Rosen, Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices, in *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pp. 145–166, 2006.

[30] O. Regev, New lattice-based cryptographic constructions, Journal of the ACM (JACM), 51(6), pp. 899–942, 2004.

[31] O. Regev, On lattices, learning with errors, random linear codes, and cryptography, Journal of the ACM (JACM), 56(6), p. 34, 2009.

[32] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM review, 41(2), pp. 303–332, 1999.

[33] N. P. Smart, *Cryptography made simple*, volume 481, Springer, 2016.

[34] M. E. Ulu and M. Cenk, A parallel gpu implementation of swifftx, Mathematical Aspects of Computer and Information Sciences, 2019.

[35] P. van Emde Boas, Another np-complete partition problem and the complexity of computing short vectors in a lattice. mathematics department, university of amsterdam, Technical report, TR 81-04, 1981.

[36] V. Volkov, Better performance at lower occupancy, in *Proceedings of the GPU technology conference, GTC*, volume 10, p. 16, San Jose, CA, 2010.

APPENDIX A

TEST DEVICE PROPERTIES

Table A.1: Test Device Properites

Property	*Value*	*Unit*
Name, Brand	AsusNVIDIA GTX1080	
Architecture	NVIDIA Pascal	
Total amount of global memory	8120	Mbytes
Number of Stream Multiprocessors	20	
Number of cores per SM	128	
Total Number of cores	2560	
GPU / Memory Clock	1734 / 5005	MHz
L2	2097152	bytes
Total amount of constant memory	65536	bytes
Total amount of shared memory per block	49152	bytes
Total number of registers available per block	65536	
Warp size	32	
Maximum number of threads per multiprocessor	2048	
Maximum number of threads per block	1024	

APPENDIX B

POWER CONSUMPTION DATA

Table B.1: Ported Reference Implementation Power Consumption Data

Data/PowerMizer Mode	*Adaptive (Min/Avg/Max)*	*Max. Perf. (Min/Avg/Max)*	*Unit*
SM Clock	139.00/1313.40/1607.00	1607.00/1607.00/1607.00	MHz
Memory Clock	405.00/3789.80/5005.00	4513.00/4709.80/5005.00	MHz
Temperature	51.00/51.56/52.00	53.00/53.00/53.00	C
Power	10039.00/39879.00/53144.00	46522.00/50178.67/53231.00	mW
Fan	0.00/0.00/0.00	0.00/0.00/0.00	%

Table B.2: Our Parallel Implementation Power Consumption Data

Data/PowerMizer Mode	*Adaptive (Min/Avg/Max)*	*Max. Perf. (Min/Avg/Max)*	*Unit*
SM Clock	139.00/1019.80/1607.00	1607.00/1607.00/1607.00	MHz
Memory Clock	405.00/2968.20/5005.00	4513.00/4759.00/5005.00	MHz
Temperature	52.00/52.33/53.00	53.38/53.00/54.00	C
Power	9995.00/34898.33/53135.00	47012.00/49984.00/53718.00	mW
Fan	0.00/0.00/0.00	0.00/0.00/0.00	%

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: ULU, Metin Evrim
Nationality: Turkish (TC)
Date and Place of Birth: April 11-th, 1980, Ankara
Marital Status: Single
E-mail: evrimulu _at_ gmail.com

EDUCATION

Degree	Institution	Year of Graduation
M.S.	M.E.T.U. Cryptography	2011
B.S.	M.E.T.U. Mechanical Engineering	2002
High School	M.E.T.U. Development Foundation School	1997

PUBLICATIONS

International Conference Publications

M.E. Ulu, [Core-serveR] - A Common Lisp Application Server, International Lisp Conference, October 2012, Kyoto, Japan

M.E. Ulu, M. Cenk, A Parallel GPU Implementation of SWIFFTX, MACIS, November 2019, Gebze, Turkey

Consent to Publish

Lecture Notes in Computer Science

Title of the Book or Conference Name: Mathematical Aspects of Computer and Information Sciences 2019.

Volume Editor(s) Name(s): Daniel Slamanig, Elias Tsigaridas, Zafeirakis Zafeirakopoulos.

Title of the Contribution: A Parallel GPU Implementation of SWIFFTX

Author(s) Full Name(s): Metin Cenk Ulu, Murat Cenk

Corresponding Author's Name, Affiliation Address, and Email:
Institute of Applied Mathematics, Cryptography Dept., Middle East Technical University, Dumlupinar Bulvari No:1, ANKARA, TURKEY. evrimulu@gmail.com mcenk@metu.edu.tr

When Author is more than one person the expression "Author" as used in this agreement will apply collectively unless otherwise indicated.

The Publisher intends to publish the Work under the imprint **Springer**. The Work may be published in the book series **Lecture Notes in Computer Science (LNCS, LNAI or LNBI).**

§ 1 Rights Granted

Author hereby grants and assigns to **Springer Nature Switzerland AG, Gewerbestrasse 11, 6330 Cham, Switzerland** (hereinafter called **Publisher**) the exclusive, sole, permanent, world-wide, transferable, sub-licensable and unlimited right to reproduce, publish, distribute, transmit, make available or otherwise communicate to the public, translate, publicly perform, archive, store, lease or lend and sell the Contribution or parts thereof individually or together with other works in any language, in all revisions and versions (including soft cover, book club and collected editions, anthologies, advance printing, reprints or print to order, microfilm editions, audiograms and videograms), in all forms and media of expression including in electronic form (including offline and online use, push or pull technologies, use in databases and data networks (e.g. the Internet) for display, print and storing on any and all stationary or portable end-user devices, e.g. text readers, audio, video or interactive devices, and for use in multimedia or interactive versions as well as for the display or transmission of the Contribution or parts thereof in data networks or search engines, and posting the Contribution on social media accounts closely related to the Work), in whole, in part or in abridged form, in each case as now known or developed in the future, including the right to grant further time-limited or permanent rights. Publisher especially has the right to permit others to use individual illustrations, tables or text quotations and may use the Contribution for advertising purposes. For the purposes of use in electronic forms, Publisher may adjust the Contribution to the respective form of use and include links (e.g. frames or inline-links) or otherwise combine it with other works and/or remove links or combinations with other works provided in the Contribution. For the avoidance of doubt, all provisions of this contract apply regardless of whether the Contribution and/or the Work itself constitutes a database under applicable copyright laws or not.

The copyright in the Contribution shall be vested in the name of Publisher. Author has asserted his/her right(s) to be identified as the originator of this Contribution in all editions and versions of the Work and parts thereof, published in all forms and media. Publisher may take, either in its own name or in that of Author, any necessary steps to protect the rights granted under this Agreement against infringement by third parties. It will have a copyright notice inserted into all editions of the Work and on the Contribution according to the provisions of the Universal Copyright Convention (UCC).

The parties acknowledge that there may be no basis for claim of copyright in the United States to a Contribution prepared by an officer or employee of the United States government as part of that person's official duties. If the Contribution was performed under a United States government contract, but Author is not a United States government employee, Publisher grants the United States government royalty-free permission to reproduce all or part of the Contribution and to authorise others to do so for United States government purposes. If the Contribution was prepared or published by or under the direction or control of the Crown (i.e., the constitutional monarch of the Commonwealth realm) or any Crown government department, the copyright in the Contribution shall, subject to any

agreement with Author, belong to the Crown. If Author is an officer or employee of the United States government or of the Crown, reference will be made to this status on the signature page.

§ 2 Rights Retained by Author

Author retains, in addition to uses permitted by law, the right to communicate the content of the Contribution to other research colleagues, to share the Contribution with them in manuscript form, to perform or present the Contribution or to use the content for non-commercial internal and educational purposes, provided the original source of publication is cited according to the current citation standards in any printed or electronic materials. Author retains the right to republish the Contribution in any collection consisting solely of Author's own works without charge, subject to ensuring that the publication of the Publisher is properly credited and that the relevant copyright notice is repeated verbatim. Author may self-archive an author-created version of his/her Contribution on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation. He/she may not use the Publisher's PDF version, which is posted on the Publisher's platforms, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgment is given to the original source of publication and a link is inserted to the published article on the Publisher's website. The link must be provided by inserting the DOI number of the article in the following sentence: "The final authenticated version is available online at https://doi.org/[insert DOI]." The DOI (Digital Object Identifier) can be found at the bottom of the first page of the published paper.
Prior versions of the Contribution published on non-commercial pre-print servers like ArXiv/CoRR and HAL can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgment needs to be given to the final publication and a link must be inserted to the published Contribution on the Publisher's website, by inserting the DOI number of the article in the following sentence: "The final authenticated publication is available online at https://doi.org/[insert DOI]".
Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgment is given to the original source of publication. Author also retains the right to use, without having to pay a fee and without having to inform the Publisher, parts of the Contribution (e.g. illustrations) for inclusion in future work. Authors may publish an extended version of their proceedings paper as a journal article provided the following principles are adhered to: a) the extended version includes at least 30% new material, b) the original publication is cited, and c) it includes an explicit statement about the increment (e.g., new results, better description of materials, etc.).

§ 3 Warranties

Author agrees, at the request of Publisher, to execute all documents and do all things reasonably required by Publisher in order to confer to Publisher all rights intended to be granted under this Agreement. Author warrants that the Contribution is original except for such excerpts from copyrighted works (including illustrations, tables, animations and text quotations) as may be included with the permission of the copyright holder thereof, in which case(s) Author is required to obtain written permission to the extent necessary and to indicate the precise sources of the excerpts in the manuscript. Author is also requested to store the signed permission forms and to make them available to Publisher if required.
Author warrants that Author is entitled to grant the rights in accordance with Clause 1 "Rights Granted", that Author has not assigned such rights to third parties, that the Contribution has not heretofore been published in whole or in part, that the Contribution contains no libellous or defamatory statements and does not infringe on any copyright, trademark, patent, statutory right or proprietary right of others, including rights obtained through licences. Author agrees to amend the Contribution to remove any potential obscenity, defamation, libel, malicious falsehood or otherwise unlawful part(s) identified at any time. Any such removal or alteration shall not affect the warranty given by Author in this Agreement.

§ 4 Delivery of Contribution and Publication

Author agrees to deliver to the responsible Volume Editor (for conferences, usually one of the Program Chairs), on a date to be agreed upon, the manuscript created according to the Publisher's Instructions for Authors. Publisher will undertake the reproduction and distribution of the Contribution at its own expense and risk. After submission of the Consent to Publish form signed by the Corresponding Author, changes of authorship, or in the order of the authors listed, will not be accepted by the Publisher.

§ 5 Author's Discount for Books

Author is entitled to purchase for his/her personal use (if ordered directly from Publisher) the Work or other books published by Publisher at a discount of 40% off the list price for as long as there is a contractual arrangement between Author and Publisher and subject to applicable book price regulation.
Resale of such copies is not permitted.

§ 6 Governing Law and Jurisdiction

If any difference shall arise between Author and Publisher concerning the meaning of this Agreement or the rights and liabilities of the parties, the parties shall engage in good faith discussions to attempt to seek a mutually satisfactory resolution of the dispute. This agreement shall be governed by, and shall be construed in accordance with, the laws of Switzerland. The courts of Zug, Switzerland shall have the exclusive jurisdiction.

Corresponding Author signs for and accepts responsibility for releasing this material on behalf of any and all Co-Authors.

Signature of Corresponding Author: Metin Ekim Ulu **Date:** December 6th 2019

.....

☐ I'm an employee of the US Government and transfer the rights to the extent transferable (Title 17 §105 U.S.C. applies)

☐ I'm an employee of the Crown and copyright on the Contribution belongs to the Crown

For internal use only:
Legal Entity Number: 1128 Springer Nature Switzerland AG
Springer-C-CTP-07/2018