Improving the efficiency of RLWE-based IPFE and its application on privacy-preserving biometrics

Submitted by

Supriya Adhikary

ROLL NO. CRS1903

M.TECH IN CRYPTOLOGY & SECURITY

INDIAN STATISTICAL INSTITUTE

KOLKATA

Primary Supervisor

Dr. Ir. Bart Preneel

ELECTRICAL ENGINEERING DEPARTMENT

KATHOLIEKE UNIVERSITEIT LEUVEN

BELGIUM

Secondary Supervisor

Dr. Bimal Kumar Roy

APPLIED STATISTICS UNIT

INDIAN STATISTICAL INSTITUTE

KOLKATA

Mentors

Angsuman Karmakar & Jose M. Bermudo Mera

KATHOLIEKE UNIVERSITEIT LEUVEN

BELGIUM

INDIAN STATISTICAL INSTITUTE

203, B.T. Road, Kolkata-700035

15'th June, 2021

Declaration

I hereby declare that the project entitled **"Improving the efficiency of RLWE-based IPFE and its application on privacy-preserving biometrics"** submitted in partial fulfillment for the award of the degree of Master of Technology in Cryptology and Security completed under the supervision of Prof. Dr. Ir. Bart Preneel and Prof. Dr. Bimal Kumar Roy, at ISI Kolkata is an authentic work. Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Signature and name of the student with date

It is certified that the above statement made by the student is correct to the best of my knowledge.

Signature and designation with date

(Primary supervisor)

(Secondary supervisor)

Abstract

Encryption is a method with which one can securely share data over an insecure channel. The traditional public-key encryption follows an all-or-nothing approach where the receiver is either able to get the whole message using a key or nothing. In *functional encryption* (FE) it is possible to control the amount of information revealed to the receiver. The emerging use of cloud computing and a massive amount of collected data leaves us with a question of data privacy. For many applications, the regular notion of public-key encryption may be insufficient. For example, a hospital may want to share patients' private healthcare data with researchers for analytics without disclosing patients' private information. *Functional Encryption* can be very useful in such a scenario, where the authority(hospital) can provide a secret key $\mathbf{sk}_f$ to the researchers corresponding to a function f and the researcher can only get the evaluation $f(x)$, so the researchers can compute on patients' data without violating the privacy of the patients.

The idea functional encryption was first introduced in terms of identity-based encryption [6, 41], attribute-based encryption [38] and predicated encryption [22]. All of these extensions and their variants can be unified under the name *Functional Encryption* for an arbitrary function f. *Inner Product Functional Encryption* (IPFE) is one of the variants of FE. IPFE has been instantiated based on different assumptions like decisional Diffie-Hellman(DDH), learning with errors (LWE) assumptions [3, 4]. The first IPFE scheme based on RLWE assumption has recently been introduced by Mera et al. [29]. RLWE schemes tend to be efficient but the main bottlenecks in any RLWE scheme are *Gaussian sampling* and large *polynomial multiplication*. These are the reasons concerning performance loss in these schemes. Improvements are required to these operations for better performance.

Our primary objective in this thesis is two fold

(a) **Improving the efficiency of RLWE-based IPFE [29]:** One of the basic observations that we can have here is that we can run most of the sections in the scheme parallelly without getting any changes in the result. We have used OpenMP to implement a multi-threaded implementation of the scheme. This allows this code to run parallelly on multiple cores simultaneously and improve the performance.

Another aspect of performance optimization is AVX2 implementation. Intel Advanced Vector Extensions (AVX) is a vector processor for doing single instruction multiple data (SIMD) operations on Intel architecture CPUs. They were first supported by

Intel with the Haswell processor, which shipped in 2013. We propose a fast vectorized polynomial multiplication using intel AVX2.

(b) **Privacy preserving biometric authentication :** We introduce an IPFE-based privacy-preserving biometric authentication protocol. We use the optimized IPFE library developed in this work. We then show the difference between using this IPFE-based protocol and a similar HE-based approach of the protocol.

List of Abbreviations

AVX Advanced Vector Extensions

CRT Chinese Remainder Theorem

CVP Closest Vector Problem

DDH Decisional Diffie-Hellman

ECC Elliptic Curve Cryptography

FE Functional Encryption

HD Hamming Distance

HE Homomorphic Encryption

IPFE Inner Product Functional Encryption

LWE Learning With Errors

NHD Normalized Hamming Distance

NTT Number Theoretic Transformation

PKC Public-Key Cryptography

RLWE Ring-Learning With Errors

SEAL Simple Encrypted Arithmetic Library

SIMD Single Instruction Multiple Data

SIS Short Integer Solution

SVP Shortest Vector Problem

List of symbols

$\mathbb{Z}$	Set of all integers.
$\mathbb{R}$	Set of all real numbers.
$\mathbb{Z}_n$	Set of all integers under modulo n.
$[n]$	Set of all natural numbers upto n.
$\mathcal{L}(B)$	Lattice generated by the basis B.
$\|\|v\|\|$	Euclidean norm of the vector v.
$\|\|v\|\|_\infty$	Infinity norm or the max norm of the vector v.
$\mathbb{Z}^{m\times n}$	set of all matrices of dimension $m \times n$ with elements from $\mathbb{Z}$.
$\langle x, y\rangle$	Inner-product of teo vectors x and y.
$\xleftarrow{R} S$	Sampled randomly from S.
R	Ring of polynomials modulo a polynomial f with coefficients in $\mathbb{Z}$.
R_q	Ring of polynomials modulo a polynomial f with coefficients in $\mathbb{Z}_q$.
σ	Standard deviation of a discrete Gaussian distribution.
$\mathcal{D}_\sigma$	Discrete Gaussian distribution with standard deviation σ.
ζ	primitive root of unity in a prime order field.
$\hat{f}$	NTT transformation of the polynomial f.
ζ_{rev}	List of all the powers of ζ in bit-reversed order.
$\mathrm{NTT}_\ell(f)$	NTT transformation on a polynomial f where last ℓ levels are skipped.
$\mathrm{NTT}_\ell^{-1}(f)$	Inverse-NTT transformation on a polynomial f where first ℓ levels are skipped.
$\oplus$	bit-wise `"XOR"` operation between two binary strings.
$\odot$	point-wise multiplication of two vectors with elements from $\mathbb{Z}$.
$\|A\|$	The *Hamming weight* of a binary string A.

Contents

Abstract i

List of Abbreviations iii

List of Symbols iv

Contents v

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Objective . . . 2
- 1.2 Summary of the thesis . . . 2

2 Preliminaries 4
- 2.1 Lattice . . . 4
- 2.2 Computational problems . . . 6
  - 2.2.1 Shortest vector problem (SVP) [34] . . . 6
  - 2.2.2 Closest vector problem (CVP) [34] . . . 7
- 2.3 Learning with errors . . . 8
- 2.4 Functional encryption [3] . . . 9
- 2.5 RLWE-based IPFE scheme [29] . . . 10
  - 2.5.1 Selectively secure IPFE based on RLWE . . . 11
  - 2.5.2 Adaptively secure IPFE based on RLWE . . . 12
  - 2.5.3 Parameters of the scheme . . . 15

3 Improving efficiency of RLWE based IPFE 16
- 3.1 Primitives used for implementation . . . 17
  - 3.1.1 Chinese remainder theorem (CRT) . . . 17
  - 3.1.2 Number theoretic transformation (NTT) . . . 17
  - 3.1.3 Choice of primes . . . 19
  - 3.1.4 Modular reductions . . . 20
- 3.2 OpenMP optimization . . . 22
  - 3.2.1 Introduction to OpenMP [12] . . . 22

3.2.2 Parallelization with OpenMP . . . 26
3.2.3 Scheduling . . . 31
3.2.4 Scalability . . . 32
3.2.5 Experimental results . . . 33
3.3 AVX2 Optimization . . . 34
3.3.1 8-point NTT . . . 35
3.3.2 NTT with AVX2 . . . 36
3.3.3 Incomplete NTT . . . 41
3.3.4 Reducing loads and stores . . . 43
3.3.5 Experimental results . . . 43
3.4 Conclusion . . . 44

4 Privacy preserving biometric protocol 45
4.1 Preliminaries . . . 45
4.1.1 Transformation of the binary vector and some results . . . 46
4.1.2 Calculating NHD using inner product . . . 47
4.2 Privacy preserving biometric protocol using IPFE . . . 48
4.2.1 Protocol . . . 48
4.2.2 Correctness . . . 50
4.2.3 Change in parameters of the RLWE based IPFE scheme [29] . . . 50
4.3 HE based privacy preserving biometric protocol . . . 51
4.3.1 Protocol . . . 51
4.3.2 Correctness . . . 53
4.4 Modifications . . . 54
4.4.1 Modifications in IPFE-based protocol . . . 54
4.4.2 Modifications in HE-based protocol . . . 55
4.5 Experimental results . . . 56
4.6 Conclusion . . . 58

5 Conclusion and future work 59
5.1 Conclusion . . . 59
5.2 Future work . . . 60

A Assembly code for first three levels of NTT 62

B Assembly code last two levels of incomplete-NTT 64

C Permutations π_1, π_2 and π_4 66

D Assembly code for multiplication and Montgomery reduction 67

E Assembly code for 2×2 base multiplication for incomplete-NTT 68

F Iris matching algorithm [13, 43] **70**
F.1 Localizing and isolating iris 70
F.2 Iris feature encoding by 2D wavelet demodulation 71
F.3 Matching 71
F.4 Rotations 72

Bibliography **73**

List of Figures

2.1 Examples of lattices with different basis 5
2.2 This is a lattice generated by the basis $\{b_1, b_2\}$. The shortest vector in the lattice is c and for the point v in the span have the point v' as the closest vector in the lattice. 7
2.3 Functional encryption [9] 9

3.1 Fork-Join model in OpenMP 22
3.2 OpenMP core elements 23
3.3 Static scheduling of 16 iterations with `chunk_size`=2 24
3.4 Dynamic scheduling of 16 iterations with `chunk_size`=2 25
3.5 Guided scheduling of 16 iterations with `chunk_size`=2 25
3.6 Visualization of the operations done in *Setup* operation 26
3.7 Visualization of the operations done in *Encrypt* operation 27
3.8 The cpucycles taken for different optimization for different security levels . . 30
3.9 The cpucycles required for different operations under different scheduling parameters 32
3.10 Performance of different operations with increasing number of threads 33
3.11 Performance optimization for different security levels 34
3.12 8-point NTT 35
3.13 Visualizing 8192 size array as a 8×1024 matrix 36
3.14 The first 64 coefficients as 8×8 matrix 37
3.15 Permutation π_4 38
3.16 Permutation π_2 38
3.17 Permutation π_1 38
3.18 First of the last three levels of NTT 39
3.19 Second of the last three levels of NTT 39
3.20 Third of the last three levels of NTT 40
3.21 cpucycles taken for AVX2 implementation and normal C code 40
3.22 Result for incomplete NTT 41
3.23 Comparision of cpucycles for C, OpenMP and AVX2 with OpenMP 43

4.1 *Privacy preserving biometric protocol using IPFE when the data saved in server's database is encrypted* 49
4.2 Privacy preserving biometric protocol using *Homomorphic Encryption* 52
4.3 Protocol based on HE with $2r + 1$ number of shift operations 56

List of Tables

2.1 The parameters for different security levels 15

3.1 Performance increased with different layers of optimization 44

4.1 The table corresponding to two binary bits a and b 46
4.2 The parameters for different security levels 51
4.3 The *cpucycles* for each of the operations done by client and server for the HE based protocol implemented in *Microsoft SEAL* 57
4.4 The *cpucycles* for each of the operations done by client and server in the IPFE based protocol 57
4.5 The *cpucycles* with the shift operations for the HE based protocol implemented in *Microsoft SEAL* 57

Chapter 1

Introduction

With the emerging use of data and communication through the internet, one basic need that comes along is security. The rapid advancement in the field of cryptography has made it possible. While symmetric cryptography is widely used for encrypting a large volume of data, public-key cryptography (PKC) takes care of decentralized key distribution, authenticity, and confidentiality. The security of PKC relies on some of the computationally hard problems such as integer factorization, discrete logarithm problems. The most popular PKC schemes such as RSA [37], ECC [30] both rely on the hard problems of integer factorization and discrete logarithm respectively.

The famous physicist Richard Feynman observed that it appeared to be impossible to efficiently simulate the evolution of a quantum system on a classical computer. He proposed a basic model for a quantum computer [16]. Initially, it wasn't really a threat to the field of cryptography until Shor [42], Proos and Zalka [32]. They proposed polynomial-time quantum algorithms for integer factorization and discrete logarithm in the elliptic curve. A quantum computer with a sufficiently large number of qubits can break RSA and ECC. The requirements of such quantum computers to break the current PKC protocol are still far fetched [18]. In spite of that, with the rapid advancement in the field of quantum computation in the past few years, it is not highly unlikely to achieve "*quantum supremacy*" soon.

This is where post-quantum cryptography gains the spotlight. There are many different approaches of post-quantum cryptography e.g., *lattice-based cryptography, code-based cryptography, hash-based cryptography* and *isogeny-based cryptography*. All of these approaches are based on some computationally hard problems which are still believed to be secure against quantum computers. We solely focus on the *lattice-based* cryptography here. The security of lattice-based schemes relies on some underlying hard problems like shortest vector problem (SVP), closest vector problem (CVP) and shortest integer solution (SIS). Regev's LWE [33] or one of its variants RLWE [27] assumptions are the most popular choices for lattice-based cryptography schemes.

Apart from security in the modern world, another important area that people have become sensitive about is their privacy. Cryptology has taken care of protecting data from the very beginning. The last few years have been the witness of cloud computing, where millions

of people store their data in public servers in order to be able to access it from anywhere in the world. In addition, people now want to be able to compute, perform analytics or searches in the cloud. For secure computation on encrypted data, there are three technologies we can use : secure multiparty computation, fully homomorphic encryption and functional encryption.

Functional encryption (FE) is a public key construction, on which it is possible to produce a functional key corresponding to a function f and evaluate the function on encrypted input data. The decryption only gives an evaluation of a function on data, nothing else. The lattice-based approach can be very useful to this primitive. Although the current works and advancement in this direction are not practical due to their large key sizes and slow speed. Therefore, we need advancement in lattice-based cryptography for efficient design and implementation. The RLWE keys are roughly the square root of keys in LWE [28, see page-3]. This makes RLWE schemes achieve greater security with respectively small key sizes. Also, RLWE schemes are much faster than LWE as RLWE uses quasi-linear time algorithms such as NTT for polynomial multiplication. The recent IPFE scheme by Mera et al. [29] is based on the RLWE assumption.

1.1 Objective

In this thesis, our main objective is to optimize the implementation of the RLWE based IPFE scheme [29]. We improve the current implementation of polynomial multiplication and use OpenMP [12] to increase the performance for better performance. We also introduce a secure privacy-preserving biometric protocol based on IPFE. In the next section, we discuss the summary of this work.

1.2 Summary of the thesis

The work in this thesis is mainly focused on improving the performance of the RLWE based IPFE scheme [29]. The chapters of this thesis are briefly described below.

- **Chapter 2 :** The second chapter starts with a short description of post-quantum cryptography. Since this thesis is focused on a lattice-based scheme, so here we introduce the concept of lattice and we discuss the problems that are at the heart of lattice based cryptography. We also describe the *functional encryption* and its security notion. Since this work focused on optimizing the implementation of the RLWE-based IPFE scheme mentioned earlier, so we give a short description of the construction of the scheme and the parameter set that is used in this scheme.
- **Chapter 3 :** In the third chapter, we describe the optimization work for the scheme. It starts with some preliminary descriptions of primitives that are to be used in the work. There are two parts of the work, (i) OpenMP and (ii) AVX2 optimization. First of all, we use OpenMP to distribute the works among threads. The construction of the scheme allows us to do that very easily. Based on some experimental results

we set proper scheduling method, chunk size and a number of threads for optimal performance for the scheme. Next, we use AVX2 instructions to optimize the code. Our main target is to optimize the polynomial multiplication. NTT has been used for polynomial multiplication but yet it is one of the main bottleneck of the scheme. In this chapter, we describe an AVX2 implementation of NTT which significantly improves the performance of the scheme.

- **Chapter 4 :** In this chapter, we propose a privacy-preserving biometric protocol using the mentioned IPFE scheme. We implement the protocol using our optimized IPFE library and compared the result with a similar HE protocol which we implement using *simple encrypted arithmetic library* (SEAL [39]).
- **Chapter 5 :** In the final chapter of this thesis, we summarize the work and the future improvements that we can do.

Chapter 2

Preliminaries

The security of post-quantum cryptography depends upon the hardness of some underlying computationally hard problems. There are several problems that are still presumed to be difficult to solve even with the help of quantum computers. These problems are at the heart of all the current post-quantum cryptography. *Lattice-based cryptography* is one such post-quantum cryptography that is widely used today. Since this work focuses on the scheme based on *lattice* problems, we discuss the *lattice-based* problems only.

2.1 Lattice

Definition 2.1.1. (Lattice [34]) *Let, $b_1, b_2, \cdots b_n \in \mathbb{R}^m$ are n linearly independent vectors, the lattice generated by these vectors is defined as*

$$\mathcal{L}(b_1, b_2, \cdots b_n) = \left\{ \sum_{i=1}^{n} x_i b_i \mid x_i \in \mathbb{Z} \right\}$$

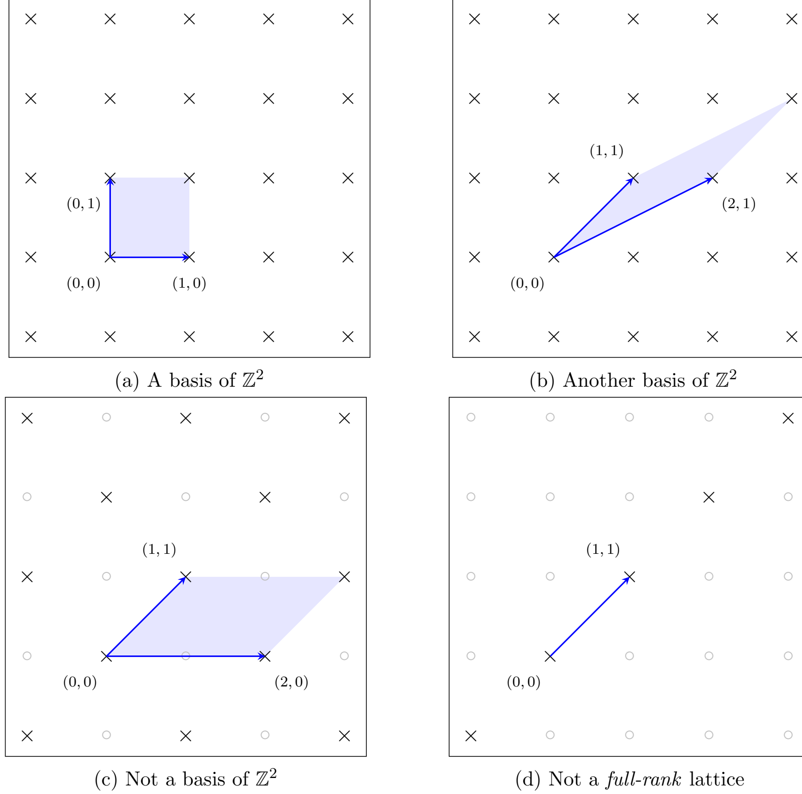
The set $\{b_1, b_2, \cdots b_n\}$ is the *basis* of the lattice. The *rank* of this lattice is n and the *dimension* of the lattice is m. If $m = n$ then the lattice is called a *full-rank lattice*. Equivalently, If we define a $m \times n$ matrix B with columns as $b_1, b_2, \cdots, b_n$, then the lattice generated by B is

$$\mathcal{L}(B) = \left\{ Bx \mid x \in \mathbb{Z}_n \right\}$$

Here are some examples of lattices. The vectors $(0,1)^T$ and $(1,0)^T$ forms a basis of $\mathbb{Z}^2$ (Figure 2.1a). This is not a unique basis of $\mathbb{Z}^2$. In Figure 2.1b we can see that the vectors $(1,1)^T$ and $(2,1)^T$ also forms a basis of $\mathbb{Z}^2$. On the other hand the vectors $(1,1)^T$ and $(2,0)^T$ also forms a lattice (Figure 2.1c), but this is not $\mathbb{Z}^2$. All these lattices are *full-rank* lattices. Now lattice generated by only one vector like in Figure 2.1d is not a *full-rank* lattice.

Definition 2.1.2. (Span [34]) *The span of a lattice $\mathcal{L}$ is the linear space generated by its basis vectors,*

$$\text{span}(\mathcal{L}) = \left\{ \sum_{i=1}^{n} y_i b_i \mid y_i \in \mathbb{R} \right\}$$



(a) A basis of $\mathbb{Z}^2$ (b) Another basis of $\mathbb{Z}^2$ (c) Not a basis of $\mathbb{Z}^2$ (d) Not a *full-rank* lattice

Figure 2.1: Examples of lattices with different basis

Here $b_1, b_2, \cdots b_n$ are the basis vectors of the lattice $\mathcal{L}$

Definition 2.1.3. (Fundamental parallelepiped [34]) *For any lattice with basis $\{b_1, b_2, \cdots b_n\}$ we define*

$$\mathcal{P}(b_1, b_2, \cdots, b_n) = \left\{ \sum_{i=1}^{n} x_i b_i \mid 0 \leq x_i < 1 \right\}$$

In the Figure 2.1 we can see the *fundamental parallelepiped*, which are the filled sections in the figures. Also it is clear that if we place one parallelepiped for each lattice point in $\mathcal{L}$ then we obtain the entire span of the lattice.

Another parameter of *lattice* is the shortest non-zero vector in the lattice, the shortest is considered in terms of the *Euclidean norm*. The length of the shortest vector is denoted by λ_1. The alternate definition of λ_1 is the smallest r, such that the lattice points inside the closed ball of radius r span a space of dimension 1. The following definition is the generalization of λ_1, known as *successive minima*.

Definition 2.1.4. (Successive minima [34]) *Let, $\mathcal{L}$ be a lattice of rank n, for each $i \in \{1, 2, \cdots, n\}$ we define the i'th successive minima as*

$$\lambda_i = \inf \left\{ r \mid \dim \left(\text{span}(\mathcal{L} \cap \bar{B}(0, r)) \right) \geq i \right\}$$

where, $\bar{B}(0,r) = \left\{ x \in \mathbb{R}^m \mid ||x|| \leq r \right\}$, *a closed ball of radius* r *centered at origin.*

2.2 Computational problems

Here we discuss the basic computational problems involving lattices.

2.2.1 Shortest vector problem (SVP) [34]

In the shortest vector problem, we are given a lattice and we are supposed to find the shortest non-zero lattice point. There are three variants of the **SVP**.

1. **Search SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find $v \in \mathcal{L}(B)$, such that $||v|| = \lambda_1(\mathcal{L}(B))$.

2. **Optimization SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find $\lambda_1(\mathcal{L}(B))$.

3. **Decisional SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ and a rational $r \in \mathbb{Q}$, determine if $\lambda_1(\mathcal{L}(B)) \leq r$ or not.

The relations among the above three variants is that **decisional SVP** is not harder than the **optimization SVP**, and **optimization SVP** is not harder than **search SVP**. In fact, the converse is also true. To summarize, the above three varients are equivalent. So, from no on **SVP** is considered as **search** variant of SVP.

Another variant of SVP is the approximate SVP. In this problem, we are interested in finding an approximation of the shortest vector. The approximation factor is given by some parameter $\alpha \geq 1$. Similar to the SVP problem this has also three variants.

1. **Search α-SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find $v \in \mathcal{L}(B)$, such that $v \neq 0$ and $||v|| \leq \alpha\lambda_1(\mathcal{L}(B))$.

2. **Optimization α-SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ find d such that $d \leq \lambda_1(\mathcal{L}(B)) \leq \alpha d$.

3. **Promise α-SVP :** Given a lattice basis $B \in \mathbb{Z}^{m \times n}$ and a rational $r \in \mathbb{Q}$, determine if $\lambda_1(\mathcal{L}(B)) \leq r$ or $\lambda_1(\mathcal{L}(B)) > \alpha \cdot r$.

The *promise problem* is usually denoted by **α-GapSVP**. In this problem, there are two disjoint sets of inputs and we have to determine from which set of inputs has the pair (B, r) been taken. Unlike the decisional problem, the union of the set of inputs does not contain all possible inputs.

As before, for any $\alpha \geq 1$ **promise α-SVP** is not harder than the **optimization α-SVP**, and the **optimization α-SVP** is not harder than the **search α-SVP**. It is also known that the **optimization α-SVP** is not harder than **promise α-SVP**, but it is still unknown if **search α-SVP** is not harder than **optimization α-SVP**.

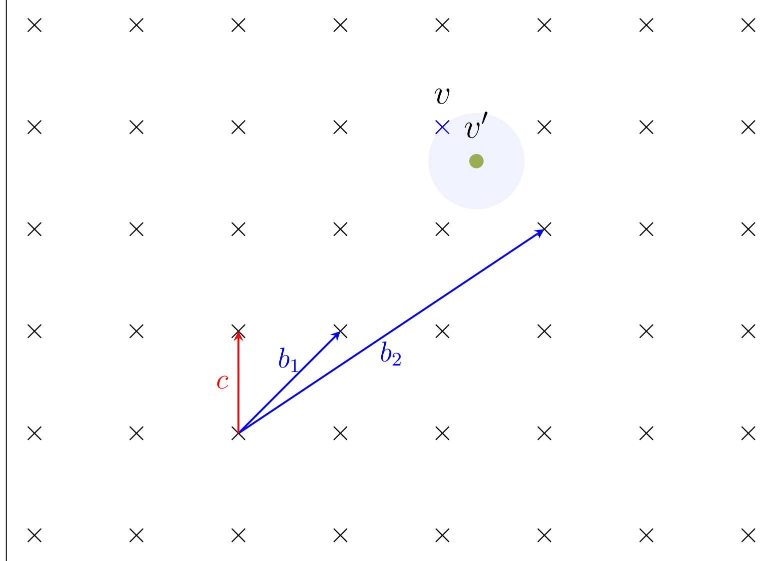


Figure 2.2: This is a lattice generated by the basis $\{b_1, b_2\}$. The shortest vector in the lattice is c and for the point v in the span have the point v' as the closest vector in the lattice.

2.2.2 Closest vector problem (CVP) [34]

Another fundamental lattice problem is closest vector problem or CVP. Just like SVP here we have three varients of CVP.

1. **Search CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$ and a vector $t \in \mathbb{Z}^m$ find $v \in \mathcal{L}(B)$, such that $||v - t|| \leq ||y - t||$ for all $y \in \mathcal{L}(B)$.

2. **Optimization CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$ and a point $t \in \mathbb{Z}^m$ find $dist(t, \mathcal{L}(B))$.

3. **Decisional CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$, $t \in \mathbb{Z}^m$ and a rational $r \in \mathbb{Q}$, determine if, $dist(t, \mathcal{L}(B)) \leq r$ or not.

As before for an approximation factor $\alpha \geq 1$ there are three variants of apporximate CVP.

1. **Search α-CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$ and a vector $t \in \mathbb{Z}^m$ find $v \in \mathcal{L}(B)$, such that $||v - t|| \leq \alpha \cdot ||y - t||$ for all $y \in \mathcal{L}(B)$.

2. **Optimization α-CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$ and a point $t \in \mathbb{Z}^m$ find d such that $d \leq dist(t, \mathcal{L}(B)) \leq \alpha \cdot d$.

3. **Decisional α-CVP :** Given a lattice basis $B \in \mathbb{Z}^{m\times n}$, $t \in \mathbb{Z}^m$ and a rational $r \in \mathbb{Q}$, determine if, $dist(t, \mathcal{L}(B)) \leq r$ or $dist(t, \mathcal{L}(B)) > \alpha \cdot r$.

CVP is a generalization of SVP. Goldreich et al. [19] has shown that SVP problem is not harder than CVP problem.

2.3 Learning with errors

Learning with the error was first introduced by Regev [35] in 2004. The LWE problem is one of the most popular choices for the construction of lattice-based cryptographic schemes. The LWE problem is assumed to be hard.

The LWE problem is defined as follows. Let $\mathbb{Z}_q$ be the set of integers modulo q. Consider the uniform random vectors $a_i \in \mathbb{Z}_q^n$, a small vector $s \in \mathbb{Z}_q^n$ and the sampled error vectors $e_i \in \mathbb{Z}_q$ chosen according to the distribution χ. The LWE distribution $\mathcal{A}_{n,q,\chi}$ is defined by $(a_i, b_i = \langle a_i, s \rangle + e_i \in \mathbb{Z}_q^n \times \mathbb{Z}_q$. There are two variants of LWE problem.

Definition 2.3.1. (Search LWE [33]) *Given polynomially many samples from the LWE distribution $\mathcal{A}_{n,q,\chi}$ the problem is to recover the secret s with non-negligible probability.*

Definition 2.3.2. (Decisional LWE [33]) *Given polynomially many samples of (a_i, b_i) from LWE distribution $\mathcal{A}_{n,q,\chi}$ and equal number of samples (a_i, b_i') sampled uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}$, the problem is to distinguish between these two distributions with non-negligible probability.*

The above two problems are equivalent. The hardness of the above problems are derived from the computational hardness of lattice problems α-SVP and SVP [36].

Another variant of LWE problem is Ring-LWE problem introduced by Lyubashevsky et al. [27]. Ring-LWE is the LWE problem over ring. The ring that is considered here is $R = \mathbb{Z}[x]/\langle f(x) \rangle$ where $f(x)$ is an irreducible polynomial over $\mathbb{Z}$. We consider $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$. The polynomials $a_i \in R_q$ are sampled uniformly, secret $s \in R_q$ is a small polynomial sampled uniformly or from a distribution χ and the error polynomials $e_i \in R_q$ are sampled from a distribution χ. Similar to LWE we define the Ring-LWE distribution as $(a_i, b_i = a_i \cdot s + e_i) \in R_q \times R_q$. Here we also have two variants of Ring-LWE problems.

Definition 2.3.3. (Search Ring-LWE [27]) *Given polynomially many samples from the Ring-LWE distribution, the problem is the find the secret $s \in R_q$ with non-negligible probability.*

Definition 2.3.4. (Decisional Ring-LWE [27]) *Given polynomially many samples (a_i, b_i) from Ring-LWE distribution and equal number of samples (a_i, b_i') sampled uniformly from $R_q \times R_q$, the problem is to distinguish between these two distributions with non-negligible probability.*

The LWE-based cryptosystems are inefficient in terms of computation, as they need to perform matrix vector multiplication in $\mathbb{Z}_q$ to encrypt each bit. So the key size for LWE problem is very large. One of the main advantage of RLWE-based cryptosystems is that the key size is smaller with respect to LWE-based cryptosystem. The RLWE keys are roughly the square root of keys in LWE [28, see page 3]. Also the matrix-vector multiplication in LWE assumption is replaced by polynomial multiplication in RLWE assumption.

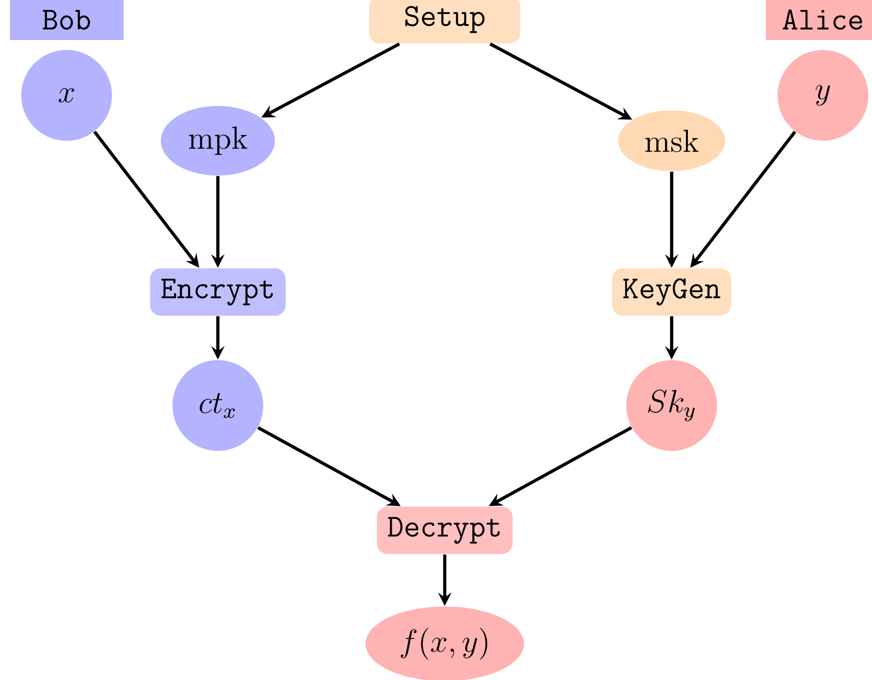


Figure 2.3: Functional encryption [9]

2.4 Functional encryption [3]

This section contains the syntax of *functional encryption* scheme and one of its variant *inner product functional encryption.*

Definition 2.4.1. (Functional encryption [3]) *A functional encryption is parameterized by $\rho = (X, Y, Z, f)$ for functionality $f : X \times Y \rightarrow Z$, is defined by the following four algorithms*

- $\texttt{Setup}(1^\kappa)$ *:* `Setup` *receives a security parameter κ and outputs a pair of master public key and master secret key, which is* $(\mathbf{mpk}, \mathbf{msk})$*.*

- $\texttt{Enc}(\mathbf{mpk}, x)$ *:* `Enc` *receives a message $x \in X$ and the master public key* **mpk** *and returns a ciphertext* `ct`*.*

- $\texttt{KeyGen}(\mathbf{msk}, y)$ *:* `KeyGen` *receives a $y \in Y$ and the master secret key* **msk** *and returns a functional key* $\texttt{sk}_y$*.*

- $\texttt{Dec}(\texttt{ct, sk}_y)$ *:* `Dec` *receives a ciphertext* `ct` *and a functional key* $\texttt{sk}_y$ *and outputs the value $f(x, y)$ or $\perp$ if the input is invalid.*

Correctness : For a correct execution of the above encryption system, $\texttt{Dec}(\texttt{ct}, \texttt{sk}_y)$ would return $f(x, y)$ where $\texttt{ct}\leftarrow\texttt{Enc}(\mathbf{mpk}, x)$ and $\texttt{sk}_y\leftarrow\texttt{KeyGen}(\mathbf{msk}, y)$.

For *inner product functional encryption* the functionality f is defined by $f(x, y) = \langle x, y \rangle = \sum_{i=1}^{\ell} x_i \cdot y_i$ where $x, y \in \mathcal{M}^\ell$

Security notion [3] **:** Following the standard security notion for FE, the game $\texttt{IND}^b_{\mathcal{A}}(1^\kappa)$ between the adversary $\mathcal{A}$ and challenger is defined as follows, where $b \overset{R}{\leftarrow} \{0,1\}$.

- *Initialize:* The challenger runs ($\mathbf{mpk}$, $\mathbf{msk}$)$\leftarrow\texttt{Setup}(1^\kappa)$ and send $\mathbf{mpk}$ to $\mathcal{A}$.

- *Query:* The adversary adaptively submits queries y and receivesthe reponse $\texttt{sk}_y\leftarrow\texttt{KeyGen}(\mathbf{msk},y)$ from the challenger.

- *Challenger:* The adversary sends messages $x^{(0)}, x^{(1)}$ and the challenger runs $\texttt{ct}\leftarrow\texttt{Enc}(\mathbf{mpk}, x^{(b)})$ and returns it to adversary $\mathcal{A}$. The challenge should satisfy the constraint $f_y(x^{(0)}) = f_y(x^{(1)})$ for all previous queries y.

- *Query:* The adversary adaptively submits queries y and receivesthe reponse $\texttt{sk}_y\leftarrow\texttt{KeyGen}(\mathbf{msk},y)$, where the queries y should satisfy the constraint $f_y(x^{(0)}) = f_y(x^{(1)})$.

- *Finalize:* The adversary outputs a bit b' as its guess for the bit b.

We say that a FE scheme is (adaptively) indistinguishable-secure (IND-secure), if for any PPT adversary $\mathcal{A}$,

$$\mathbf{Adv}^{\mathrm{FE}}_{\mathcal{A}}(\mathrm{IND}^b_{\mathcal{A}}) = \left| \Pr\left[\mathrm{IND}^1_{\mathcal{A}}(1^\kappa) = 1\right] - \Pr\left[\mathrm{IND}^0_{\mathcal{A}}(1^\kappa) = 1\right] \right| \leq \mathbf{neg}(\kappa)$$

$\mathbf{neg}$ is a negligible function. Again, a FE scheme is selectively secure if the adversary submits its challenges $(x^{(0)}, x^{(1)})$ at the very beginning of the game before seeing the public-key.

2.5 RLWE-based IPFE scheme [29]

As we have stated before in the introduction of this thesis the main objective is to develop an IPFE library based on the scheme introduced by Mera et al. [29]. Before going into detail about the implementation and optimization methods we first describe the scheme in this chapter. The scheme is the first IPFE scheme based on the RLWE assumption, although it is inspired by the LWE-based IPFE scheme from [3, 4]. We only discuss the construction of the scheme, for more details of the scheme the reader may check the original publication [29].

As we have mentioned in Section-2.4 there are two notions of the security of *functional encryption*, based on that here we have two constructions of IPFE which are *selectively secure* and *adaptively secure* IPFE. We discuss both.

Before continuing to the next section here are some notations that are to be used in this chapter. R is the ring $\mathbb{Z}[x]/\left\langle x^n+1\right\rangle$, also R_q is the ring $\mathbb{Z}_q[x]/\left\langle x^n+1\right\rangle$. $\mathcal{D}_\sigma$ and $\mathcal{D}_{\sigma I_n}$ are the *Gaussian distribution* over $\mathbb{Z}$ and $\mathbb{Z}^n$ respectively with standard deviation σ.

2.5.1 Selectively secure IPFE based on RLWE

The construction allows us to encrypt a vector of dimension ℓ, where infinity norms of the message x and the functional vector y are bounded by B_x and B_y respectively. Let K be greater than the maximum value of the resulting inner product *i.e.*, $K > \ell B_x B_y$. We first describe the construction of the scheme.

Construction

Setup

- First sample $a \in R_q$ uniformly at random
- Sample $s_i, e_i \in R$ for $i \in \{1, 2, \cdots, \ell\}$, whose coefficients are from the distribution $\mathcal{D}_{\sigma_1}$
- Compute $\mathbf{pk}_i = a \cdot s_i + e_i \in R_q$ for $i \in \{1, 2, \cdots, \ell\}$
- Set, $\mathbf{msk} = \{s_i \mid i \in [\ell]\}$ and $\mathbf{mpk} = \big(a, \{\mathbf{pk}_i \mid i \in [\ell]\}\big)$

Encryption Given a vector $x = (x_1, x_2, \cdots, x_\ell) \in \mathbb{Z}^\ell$ with $||x||_\infty \leq B_x$, below is the encryption algorithm.

- First sample $r, f_0 \in R_q$ with coefficients from the distribution $\mathcal{D}_{\sigma_2}$
- Sample $f_i \in R_q$ independently with coefficients from the distribution $\mathcal{D}_{\sigma_3}$, for all $i \in \{1, 2, \cdots, \ell\}$
- We fix 1_R to be the polynomial with all the coefficients equal to $1 \in \mathbb{Z}_q$
- Calculate $\mathbf{ct}_0 = a \cdot r + f_0, \quad \mathbf{ct_i} = \mathbf{pk}_i \cdot r + f_i + \lfloor q/K \rfloor x_i 1_R$ for all $i \in [\ell]$
- Output $\big(\mathbf{ct}_0, \{\mathbf{ct}_i \mid i \in [\ell]\}\big)$ as encryption of x.

KeyGen Given a vector $y = (y_1, y_2, \cdots, y_\ell) \in \mathbb{Z}^\ell$ such that $||y||_\infty \leq B_y$, We need to find a decryption key corresponding to y.

We simply calculate

$$\mathbf{sk}_y = \sum_{i=1}^{\ell} y_i s_i \in R$$

Decryption To decrypt the ciphertext $\big(\mathbf{ct}_0, \{\mathbf{ct}_i \mid i \in [\ell]\}\big)$ using the functional key $\mathbf{sk}_y$ and y we calculate

$$d = \left(\sum_{i=1}^{\ell} y_i \mathbf{ct}_i\right) - \mathbf{ct}_0 \cdot \mathbf{sk}_y$$

This d should be close to $\lfloor q/K \rfloor \langle x, y \rangle 1_R$ and we can extract $\langle x, y \rangle$ easily.

Correctness

We can write the decryption d as

$$\begin{aligned} d &= \left(\sum_{i=1}^{\ell} y_i \mathbf{ct}_i\right) - \mathbf{ct}_0 \cdot \mathbf{sk}_y \\ &= \sum_{i=1}^{\ell} y_i(\mathbf{pk}_i \cdot r + f_i + \lfloor q/K \rfloor x_i 1_R) - \sum_{i=1}^{\ell} y_i s_i (a \cdot r + f_0) \\ &= \sum_{i=1}^{\ell} y_i((a \cdot s_i + e_i) \cdot r + f_i + \lfloor q/K \rfloor x_i 1_R) - \sum_{i=1}^{\ell} y_i s_i (a \cdot r + f_0) \\ &= \sum_{i=1}^{\ell} (y_i e_i r + y_i f_i - y_i s_i f_0) + \lfloor q/K \rfloor \sum_{i=1}^{\ell} y_i x_i 1_R \\ &= \sum_{i=1}^{\ell} (y_i e_i r + y_i f_i - y_i s_i f_0) + \lfloor q/K \rfloor \langle x, y \rangle 1_R \end{aligned}$$

the term $\sum_{i=1}^{\ell}(y_i e_i r + y_i f_i - y_i s_i f_0)$ is the "noise". For correctness we need the condition $||\texttt{noise}||_\infty < \lfloor q/2K \rfloor$. For the security parameter κ, with non-negligible probability we have, $||e_i||_\infty, ||s_i||_\infty \leq \sqrt{\kappa}\sigma_1$, also $||r||_\infty, ||f_0||_\infty \leq \sqrt{\kappa}\sigma_2$ and $||f_i||_\infty \leq \sqrt{\kappa}\sigma_3$. Thus the noise is

$$\left|\left| \sum_{i=1}^{\ell} (y_i e_i r + y_i f_i - y_i s_i f_0) \right|\right|_\infty \leq \ell(2n\kappa\sigma_1\sigma_2 + \sqrt{\kappa}\sigma_3)B_y$$

So for correctly extracting the result we need $\ell(2n\kappa\sigma_1\sigma_2 + \sqrt{\kappa}\sigma_3)B_y < \lfloor q/2K \rfloor$.

2.5.2 Adaptively secure IPFE based on RLWE

Here we give the description of the modified construction of the scheme which is adaptively secure. The main difference from the selectively secure construction is that, here each secret keys s_i and the public key parameter a are vectors of polynomials which is two single polynomials in the selective case. Here also we have ℓ dimensional message x and functional vector y in $\mathbb{Z}^\ell$, satisfying $||x||_\infty \leq B_x$ and $||y||_\infty \leq B_y$. K be greater than the maximum value that the inner-product may take *i.e.*, $K > \ell B_x B_y$.

Construction

Setup

- For each $i \in [\ell]$ sample $s_i = (s_{i1}, s_{i2}, \cdots, s_{im}) \in R^m$, where each $s_{ij} \in R$ are sampled from $\mathcal{D}_{\sigma_1 I_n}$.

- Sample $a = (a_1, a_2, \cdots, a_m) \in R_q^m$ uniformly at random. make sure that at least one a_i is invertible in R_q.

- Compute $\mathbf{pk}_i = \langle a, s_i \rangle = \sum_{j=1}^{\ell} a_j s_{ij}$.
- Set $\mathbf{msk} = \{s_i \mid i \in [\ell]\}$ and $\mathbf{mpk} = \big(a, \{\mathbf{pk}_i \mid i \in [\ell]\}\big)$.

Encryption Given a vector $x = (x_1, x_2, \cdots, x_\ell) \in \mathbb{Z}^\ell$ with $||x||_\infty \leq B_x$, below is the encryption algorithm.

- Sample $r \in R_q$ and $f_0 = (f_{01}, f_{02}, \cdots, f_{0m}) \in R_q^m$ from the distributions $\mathcal{D}_{\sigma_2 I_n}$ and $\mathcal{D}_{\sigma_2 I_{nm}}$ respectively.
- Sample $f_i \in R_q$ from the distribution $\mathcal{D}_{\sigma_3 I_n}$, for all $i \in \{1, 2, \cdots, \ell\}$.
- Compute

$$\begin{aligned}\mathbf{ct}_0 &= ar + f_0 = (a_1 r + f_{01}, a_2 r + f_{02}, \cdots, a_m r + f_{0m})\\ \mathbf{ct}_i &= \mathbf{pk}_i r + f_i + \lfloor q/K \rfloor x_i 1_R\end{aligned}$$

- Check if at least one of the components of $\mathbf{ct}_0$ is invertible. if not then resample r and f_0 and recompute $\mathbf{ct}_0$ and $\mathbf{ct}_i$.
- The encryption of x is $\big(\mathbf{ct}_0, \{\mathbf{ct}_i \mid i \in [\ell]\}\big)$

KeyGen Given a vector $y = (y_1, y_2, \cdots, y_\ell) \in \mathbb{Z}^\ell$ such that $||y||_\infty \leq B_y$, We need to find a decryption key corresponding to y.

We simply calculate

$$\mathbf{sk}_y = \sum_{i=1}^{\ell} y_i s_i = \left(\sum_{i=1}^{\ell} y_i s_{i1}, \sum_{i=1}^{\ell} y_i s_{i2}, \cdots, \sum_{i=1}^{\ell} y_i s_{im} \right) \in R^m$$

Decryption To decrypt the ciphertext $\big(\mathbf{ct}_0, \{\mathbf{ct}_i \mid i \in [\ell]\}\big)$ using the functional key $\mathbf{sk}_y$ and y we calculate

$$d = \left(\sum_{i=1}^{\ell} y_i \mathbf{ct}_i \right) - \langle \mathbf{ct}_0, \mathbf{sk}_y \rangle$$

This d should be close to $\lfloor q/K \rfloor \langle x, y \rangle 1_R$ and we can extract $\langle x, y \rangle$ easily.

Correctness

In decryption we have,

$$\begin{aligned}
d &= \left(\sum_{i=1}^{\ell} y_i \mathbf{ct}_i\right) - \langle \mathbf{ct}_0, \mathbf{sk}_y \rangle \\
&= \sum_{i=1}^{\ell} y_i(\mathbf{pk}_i r + f_i + \lfloor q/K \rfloor x_i 1_R) - \sum_{j=1}^{\ell}\left((a_j r + f_{0j}) \cdot \sum_{i=1}^{\ell} y_i s_{ij}\right) \\
&= \sum_{i=1}^{\ell} y_i\big(\sum_{j=1}^{\ell} a_j s_{ij} r + f_i + \lfloor q/K \rfloor x_i 1_R\big) - \sum_{j=1}^{\ell}\sum_{i=1}^{\ell} y_i s_{ij} a_j r - \sum_{j=1}^{\ell}\sum_{i=1}^{\ell} y_i s_{ij} f_{0j} \\
&= \sum_{i=1}^{\ell} y_i\left(f_i - \langle f_0, s_i \rangle\right) + \lfloor q/K \rfloor \langle x, y \rangle 1_R
\end{aligned}$$

Here $\texttt{noise} = \sum_{i=1}^{\ell} y_i\big(f_i - \langle f_0, s_i\rangle\big)$. So we can extract the inner product correctly if $||\texttt{noise}||_\infty < \lfloor q/2K \rfloor$. Now again, for security parameter κ, $||y||_\infty \leq B_y$, $||s_{ij}||_\infty \leq \sqrt{\kappa}\sigma_1$, $||f_{0i}||_\infty \leq \sqrt{\kappa}\sigma_2$ and $||f_i||_\infty \leq \sqrt{\kappa}\sigma_3$. Thus,

$$\left|\left|\sum_{i=1}^{\ell} y_i\left(f_i - \langle f_0, s_i \rangle\right)\right|\right|_\infty \leq \ell B_y(\sqrt{\kappa}\sigma_3 + mn\kappa\sigma_1\sigma_2)$$

So, for correctly extracting the result we need $\ell B_y(\sqrt{\kappa}\sigma_3 + mn\kappa\sigma_1\sigma_2) < \lfloor q/2K \rfloor$

We have so far discussed the construction of the IPFE scheme that this work is focused on. Here in this paper, we have not discussed the security proof and parameters selection. The readers may refer to the paper [29] for a detailed description of the security proof and parameter selections. Although we are not discussing how the parameters are chosen, in the next section we have discussed what are the parameters that we have to choose for the implementation of the scheme.

2.5.3 Parameters of the scheme

There are three sets of parameters that are set depending on the values of ℓ, B_x and B_y. Here we consider the selectively secure scheme. In Table 2.1, the parameters for the IPFE scheme has been deduced which offers $\mathcal{S}$ bits of security. As we optimize the scheme for all different security levels, we would need this table of parameters in the later chapters.

Security Level	PQ Security	FE Bounds		Gaussian Parameters		Ring Parameters	CRT moduli	
SEC_LEVEL_0	80	B_x	2	σ_1	33	n : 2048	q_1	$2^{16}-2^{12}+1$
		B_y	2	σ_2	64880641		q_2	$2^{17}-2^{14}+1$
		ℓ	64	σ_3	129761280	$\lceil \log q \rceil$: 64	q_3	$2^{31}-2^{17}+1$
SEC_LEVEL_1	129	B_x	4	σ_1	226	n : 4096	q_1	$2^{24}-2^{14}+1$
		B_y	16	σ_2	258376413		q_2	$2^{26}-2^{16}+1$
		ℓ	785	σ_3	516752823	$\lceil \log q \rceil$: 81	q_3	$2^{31}-2^{24}+1$
SEC_LEVEL_2	267	B_x	32	σ_1	2049	n : 8192	q_1	$2^{31}-2^{17}+1$
		B_y	32	σ_2	5371330561		q_2	$2^{31}-2^{19}+1$
		ℓ	1024	σ_3	10742661120	$\lceil \log q \rceil$: 94	q_3	$2^{32}-2^{20}+1$

Table 2.1: The parameters for different security levels

In this chapter, we have seen how the IPFE scheme is constructed. In the next chapter we discuss the implementation aspects and the optimization techniques we use for this work. One thing that we can observe by now is that the **Setup** and **Encrypt** are the most expensive operations with respect to the other operations in the scheme as it includes a large number of polynomial multiplications and sampling, our goal here is to build an efficient implementation of the scheme by reducing overheads in these sections.

We focus on the implementation of the selectively secure implementation of the scheme but the techniques used in this work can be extended to the adaptively secure version of the scheme.

Chapter 3

Improving efficiency of RLWE based IPFE

The RLWE assumption lets us build lattice-based encryption schemes using the smaller key sizes and it also lets more bits of data be encrypted, so the ciphertext length also reduces. This is indeed an advantage of RLWE assumption over LWE assumption but now we have to deal with rings of polynomials. In RLWE based schemes, we have to deal with costly operations such as polynomial multiplication. Although NTT and NTT^{-1} gives us a quasi-linear time algorithm for polynomial multiplication with some certain parameters, it still can be one of the bottlenecks which decreases performance.

OpenMP [12] is an API that supports multi-threading programming in C, C++ and Fortran over many different platforms. It includes compiler directives, library routines and environment variables that influence run-time behaviour. It uses a portable, scalable model that gives programmers a simple interface to build parallel codes for many different platforms. There are large sections of codes in RLWE-based IPFE scheme, where we use OpenMP to parallelize the application.

Advanced Vector Extension also known as AVX is one of the most popular vector processor that is used for fast constant-time programming. It supports registers of size 128 bits, AVX2 can support up to 256 bits registers. We can load eight 32 bits integers or four 64 bits integers and the instructions can be applied to all of these integers simultaneously. There have also been AVX-512 extensions that support 512 bits registers but the instructions are not supported by as many systems as AVX, AVX2 does. We would be using the AVX2 in this work.

All the experiments in this chapter are performed on 4 cores of Intel(R) Core(TM) i5-8265U processor running at 1.60GHz with hyper-threading enabled, on Ubuntu 20.04.2 running on a HP 15-da1041TU laptop. All codes have been compiled using `gcc-9.3.0` with flags `-O3 -fomit-frame-pointer -march=native`.

3.1 Primitives used for implementation

In this section, we will discuss the choice of model and parameters for the implementation of the scheme. We will later discuss the optimizations and finally, we apply this scheme for building the privacy-preserving biometric authentication protocol.

3.1.1 Chinese remainder theorem (CRT)

Due to the correctness and security constraints of this scheme, the modulus q here has to be large *i.e.*, more than 64 bits. Therefore, calculations in $\mathbb{Z}_q$ will require expensive and in-efficient multi-precision arithmetic. Similar to the implementation of SEAL [39] here the residual number system based polynomial arithmetic has been used. Here a chain of moduli $q_0, q_1, \cdots, q_t$ are chosen such that $q = q_0 q_1 \cdots q_t$. So all the inputs, intermediate values and outputs are stored as the elements of the ring $\mathbb{Z}_{q_i}$ instead of $\mathbb{Z}_q$. At the end of the scheme we use Garner's Algorithm [25] and GNU multi-precision library [20] for converting the elements in $\mathbb{Z}_{q_i}$ back to $\mathbb{Z}_q$.

The moduli q_i's are all at most 32 bits. So the all calculations are now reduced to 32 bits integer arithmetic which is in fact much better than multi-precision arithmetic.

Algorithm 1: Garner's algorithm [25]

Input: A positive integer $q = \prod_{i=1}^{t} q_i > 1$, with $\texttt{gcd}(q_i, q_j) = 1$ for all $i \neq j$ and $v(x) = (v_1, v_2, \cdots, v_t)$ such that $X \equiv v_i \mod q_i$ for all i.
Output: x such that $X \equiv x \mod q$

```
1  for ( i = 2; i ≤ t; i++ ) do
2      C_i ← 1;
3      for ( j = 1; j ≤ i − 1; j++ ) do
4          u ← q_j^{-1} mod q_i;
5          C_i ← u · C_i mod q_i;
6  u ← v_1;
7  x ← u;
8  for ( i = 2; i ≤ t; i++ ) do
9      u ← (v_i − x) · C_i mod q_j;
10     x ← x + u · ∏_{j=1}^{i-1} q_j;
11 return x
```

3.1.2 Number theoretic transformation (NTT)

A very efficient way to perform multiplication in $R_q = \mathbb{Z}_q[x] / \langle x^n + 1 \rangle$ is NTT, where n is a power of 2 and q is a prime number. If q is a prime of the form $q \equiv 1 \mod 2n$ then there exists a *primitive* $2n$*'th root of unity* in $\mathbb{Z}_q^*$, say it is ζ. Now, $x^{2n} - 1 = (x^n + 1) \cdot (x^n - 1)$ and all even powers of ζ are the n'th root of unity *i.e.*, the the roots of $(x^n - 1)$. Therefore, we have

$$x^n + 1 = \prod_{i=0}^{n-1} \left(x - \zeta^{2i+1}\right)$$

Algorithm 2: Forward NTT using Cooley-Tukey method [11]

Input: A vector $a = (a_0, a_1, \cdots, a_{n-1}) \in \mathbb{Z}_q^n$ in standard ordering, where q is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\zeta_{rev} \in \mathbb{Z}_q^n$, storing the powers of ζ in bit-reversed order.

Output: $a \leftarrow NTT(a)$ in bit-reversed ordering

```
1  t ← n;
2  for ( m = 1; m < n; m = 2m ) do
3  |  t ← t/2;
4  |  for ( i = 0; i < m; i++ ) do
5  |  |  j1 ← 2 · i · t;
6  |  |  j2 ← j1 + t − 1;
7  |  |  S = ζrev[m + i];
8  |  |  for ( j = j1; j ≤ j2; j++ ) do
9  |  |  |  U ← a_j;
10 |  |  |  V ← a_{j+t} · S;
11 |  |  |  a_j ← U + V  mod q;
12 |  |  |  a_{j+t} ← U − V  mod q;
13 return a
```

If we recall the chinese remainder theorem for rings, we can now write

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle \cong \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/\langle x - \zeta^{2i+1}\rangle$$

where the isomorphism is $\Phi(f) = \big(f(\zeta), f(\zeta^3), \cdots, f(\zeta^{2n-1})\big)$. The NTT computes this isomorphism. Therefore, NTT is a mapping from R_q to $\mathbb{Z}_q^n$.
Let $\mathrm{NTT}(f) = (\hat{f}_0, \hat{f}_1, \cdots, \hat{f}_{n-1})$ then,

$$\hat{f}_i = \sum_{j=0}^{n-1} f_j \cdot \zeta^{(2i+1)\cdot j}$$

where $f(x) = \sum_{j=0}^{n} f_j \cdot x^j$ is a polynomial in R_q. The inverse of NTT denoted as NTT^{-1} is defined by $\mathrm{NTT}^{-1}(\hat{f}) = (f'_0, f'_1, \cdots, f'_{n-1})$ where,

$$f'_i = \sum_{j=0}^{n-1} \hat{f}_j \cdot \zeta^{-(2i+1)\cdot j}$$

Using NTT and its inverse NTT^{-1} we can very efficiently multiply the product of two polynomials $f, g \in R_q$ as $\mathrm{NTT}^{-1}\big(\mathrm{NTT}(f) \circ \mathrm{NTT}(g)\big)$. Here $(\mathrm{NTT}(f) \circ \mathrm{NTT}(g) = \hat{f} \circ \hat{g} = \hat{h}$ this is the pointwise multiplication of $\hat{f}$ and $\hat{g}$. In other words $\hat{h}_i = \hat{f}_i \cdot \hat{g}_i$ for all $i \in \{0, 1 \cdots, n-1\}$.

Therefore, $\text{NTT}^{-1}(\hat{h}) = f \cdot g$. So the polynomial multiplication is now pointwise multiplication of vectors in NTT domain.

We are using Cooley-Tukey [11] algorithm for NTT forward transformation and Gentleman-Sande [17] butterfly algorithm NTT^{-1} transformation. We have stated the algorithms of NTT and NTT^{-1} in Algorithm 2 and Algorithm 3 respectively.

Algorithm 3: NTT^{-1} using Gentleman-Sande butterfly method [17]

Input: A vector $a = (a_0, a_1, \cdots, a_{n-1}) \in \mathbb{Z}_q^n$ in bit-reversed ordering, where q is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\zeta_{rev}^{-1} \in \mathbb{Z}_q^n$, storing the powers of ζ^{-1} in bit-reversed order.

Output: $a \leftarrow NTT(a)$ in standard ordering

```
1  t ← 1;
2  for ( m = n; m > 1; m = m/2 ) do
3  |   j1 ← 0;
4  |   h ← m/2;
5  |   for ( i = 0; i < h; i++ ) do
6  |   |   j2 ← j1 + t − 1;
7  |   |   S = ζ_rev^{-1}[h + i];
8  |   |   for ( j = j1; j ≤ j2; j++ ) do
9  |   |   |   U ← a_j;
10 |   |   |   V ← a_{j+t};
11 |   |   |   a_j ← U + V  mod q;
12 |   |   |   a_{j+t} ← (U − V) · S  mod q;
13 |   |   j1 ← j1 + 2t;
14 |   t ← 2t;
15 for ( j = 0; j < n; j++ ) do
16 |   a_j ← a_j · n^{-1}  mod q;
17 return a
```

3.1.3 Choice of primes

As we have stated in Section 3.1.1, a sequence of primes $q_1, q_2, \cdots, q_t$ are chosen such that $q = q_1 q_2 \cdots q_t$. All the polynomials in $\mathbb{Z}_q$ are now converted into polynomials in $\mathbb{Z}_{q_i}$ for all $i \in [t]$. Now the polynomial arithmetic in $\mathbb{Z}_{q_i}$ involves NTT transformation, which requires primes $q_i \equiv 1 \mod 2n$. Here primes have been selected in a way to accelerate NTT.

The primes chosen here for the implementation, are of the form $2^i - 2^j + 1$. Clearly, if $i > j > \log_2 n$ where n is in powers of 2, then $2^i - 2^j + 1 \equiv 1 \mod 2n$. Therefore, we can easily perform NTT for such primes. Also we can perform fast modular reductions for such special primes.

3.1.4 Modular reductions

Modular reductions play one of the most important roles in constant time programming. Divisions need to be avoided for fast modular reductions. We this section we will be discussing some of the most popular modular reduction techniques used. Also for vectorized NTT implementation, we need techniques like Montgomery or Barret reductions.

Specialized reduction

As we have mentioned before, the primes chosen for the implementation are of the form $q = 2^i - 2^j + 1$. Modular reductions for such primes are very efficient.
As we can see that,

$$2^i \equiv 2^j - 1 \mod q$$

Therefore, for any number of the form $z = z_1 2^i + z_2$ we have,

$$\begin{aligned} z &\equiv z_1 2^i + z_2 \mod q \\ &\equiv z_1(2^j - 1) + z_2 \mod q \\ &\equiv z_1 2^j + z_2 - z_1 \mod q \end{aligned}$$

So as we can see that for fixed i, j the modular reductions under modulus q are just a series of additions and shift operations which is much more efficient than multiplication or division.

Barret reduction [5]

The following barret reduction method is taken from [21]. The actual algorithm considers a reduction in an arbitrary base b, here we will consider only the reduction base 2. For the algorithm we have a moduli q , $\mu = \lfloor \frac{2^{2k}}{q} \rfloor$ where $k = \lfloor \log_2 q \rfloor + 1$ and an integer $z \in \mathbb{Z}$ such that $0 \leq z < 2^{2k}$. Let $t = \lfloor \frac{z}{q} \rfloor$ then, the result should be $r = z - t \cdot q$. Now,

$$\frac{z}{q} = \frac{z}{2^{k-1}} \cdot \frac{2^{2k}}{q} \cdot \frac{1}{2^{k+1}}$$

We approximate t by $\hat{t}$ in the following way

$$\hat{t} = \left\lfloor \frac{\lfloor \frac{z}{2^{k-1}} \rfloor \cdot \mu}{2^{k+1}} \right\rfloor \leq \left\lfloor \frac{z}{q} \right\rfloor = t$$

The value μ can be computed in advance, so the cost of computation of $\hat{q}$ is one multiplication and some shift operations on integers.

Let $\alpha = \frac{z}{2^{k-1}} - \lfloor \frac{z}{2^{k-1}} \rfloor$ and $\beta = \frac{2^{2k}}{q} - \lfloor \frac{2^{2k}}{q} \rfloor$ then,

$$\begin{aligned} t &= \left\lfloor \frac{\left(\lfloor \frac{z}{2^{k-1}} \rfloor + \alpha\right)\left(\lfloor \frac{2^{2k}}{q} \rfloor + \beta\right)}{2^{k+1}} \right\rfloor \\ &\leq \left\lfloor \frac{\lfloor \frac{z}{2^{k-1}} \rfloor \cdot \mu + \alpha \cdot \mu + \beta \cdot \lfloor \frac{z}{2^{k-1}} \rfloor + \alpha \cdot \beta}{2^{k+1}} \right\rfloor \\ &\leq \left\lfloor \frac{\lfloor \frac{z}{2^{k-1}} \rfloor \cdot \mu}{2^{k+1}} + \frac{\lfloor \frac{2^{2k}}{q} \rfloor + \lfloor \frac{z}{2^{k-1}} \rfloor + 1}{2^{k+1}} \right\rfloor \end{aligned}$$

Since $0 \leq z < 2^{2k}$ and $2^{k-1} \leq q < 2^k$, we have

$$\begin{aligned} \left\lfloor \frac{2^{2k}}{q} \right\rfloor + \left\lfloor \frac{z}{2^{k-1}} \right\rfloor + 1 &\leq 2^{k+1} + (2^{k+1} - 1) + 1 \\ &= 2 \cdot 2^{k+1} \end{aligned}$$

So we have,

$$\begin{aligned} t &\leq \left\lfloor \frac{\lfloor \frac{z}{2^{k-1}} \rfloor \cdot \mu}{2^{k+1}} \right\rfloor + 2 \\ &= \hat{t} + 2 \end{aligned}$$

Therefore, $\hat{t} \leq t \leq \hat{t} + 2$. Now if we compute $\hat{r} = z - \hat{t} \cdot q$ then, $r = \hat{r} - (t - \hat{t}) \cdot q$. Since, $(t - \hat{t}) \leq 2$ so we need at most 2 substractions to get the result r from $\hat{r}$.

The Barret reduction that we have showed here contains only one multiplication, three subtractions at most and rest are binary shift operations. One the other hand specialized reductions has no multiplications only shifts and additions where the number of iterations may vary. Specialized reduction is not easily vectorizable for AVX2, so we may use Barret reduction.

Again, in Barret reduction μ is k bits, z is at most $2k$ bits, so $\left(\lfloor \frac{z}{2^{k-1}} \rfloor \cdot \mu\right)$ is at most $2k + 1$ bits. Now, the primes that we are using for this implementation are at most 32 bits, so $2k + 1 \leq 65$. Therefore, it is possible that we encounter a overflow when we are working with 32 bits prime. Since we have 32 bits prime used in the implementation of the scheme, so it would not be a good idea to use Barret reduction too.

Therefore, we implement reductions in NTT multiplication using Mongomery's method as it gives us a constant time implementation and it can be easily vectorized.

Montgomery reduction [31]

The method was first introduced by Montgomery [31]. Let β be an integer such that $q < \beta$ and $\gcd(\beta, q) = 1$. Let $a \in \mathbb{Z}$ be a integer such that $0 \leq a < q\beta$. Then the algorithm returns a integer r such that $r \equiv a\beta^{-1} \mod q$. For our work, β is taken as some powers of 2 and q

is some prime. The algorithm is as below.

Algorithm 4: Montgomery reduction [31]

Input: A moduli q, an integer β such that $0 < q < \beta$ and $\gcd(q, \beta) = 1$, an integer $a \in \mathbb{Z}$ such that $0 \le a < q\beta$

Output: r such that $r \equiv a\beta^{-1} \mod q, \quad 0 \le r < q$

1 $m \leftarrow -aq^{-1} \mod \beta$;
2 $t \leftarrow \lfloor \frac{a+mq}{\beta} \rfloor$;
3 **if** $t \ge q$ **then**
4   **return** $t - q$;
5 **else**
6   **return** t;

Although Montgomery reduction outputs $r \equiv a\beta^{-1} \mod q$ instead of the desired value $r' \equiv a \mod q$ this will not be a problem since the reductions in NTT algorithm are required when the coefficients are multiplied with the powers of ζ which are precomputed. For this reason, we can have precomputed values of powers of ζ multiplied with $\beta \mod q$ so that at the end of reduction we will have the desired result.

3.2 OpenMP optimization

Before we get into details about the work we need to understand some basic operations of OpenMP that we have used here.

3.2.1 Introduction to OpenMP [12]

OpenMP is an implementation of multi-threading. In this method, the main thread forks a specified number of sub-threads and the system divides tasks among them. It is possible that the sub-threads may further forks more threads recursively until a certain task granularity is reached. The sub-threads joins into the main thread when the desired task is completed. This model is called *fork-join* model and we have the depiction of the model in Figure 3.1.

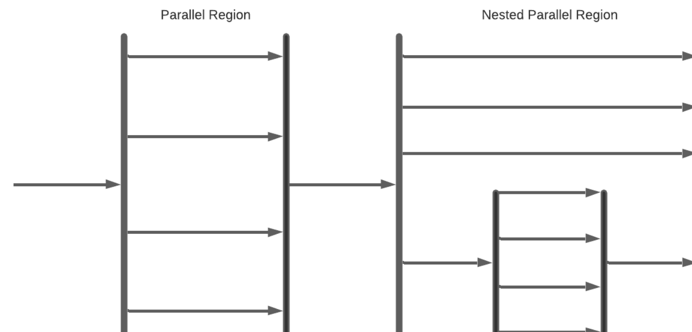


Figure 3.1: Fork-Join model in OpenMP

When more than one thread is running in parallel, there is a lot of things to take into

account e.g., data sharing, scheduling, task distribution, race conditions etc. Now we will discuss some core elements of OpenMP that is required for multi-threaded programming. The core elements of OpenMP are the constructs for thread creation, workload distribution (work-sharing), data-environment management, thread scheduling, user-level runtime routines and environment variables. Figure 3.2 depicts these core elements.

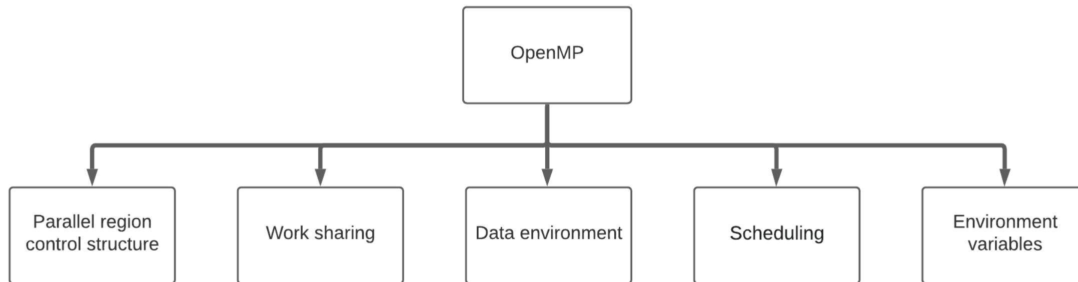


Figure 3.2: OpenMP core elements

Parallel region control structure [2]

In OpenMP, we use `#pragma omp parallel` to specify the main thread to fork into additional threads and execute the work enclosed in a region in parallel. This enclosed region is called the *parallel region* and the main thread has the thread ID as 0. After the enclosed region is executed the threads join with the original/main thread again, as we see in Figure 3.1.

Work sharing [2]

The work-sharing constructs specify how to assign independent works to all of the threads. The constructs are as follows

(i) `omp for` : This construct is used to split a for loop iterations and distribute them into different threads.

(ii) `sections` : This construct assigns consecutive but independent code blocks to different threads. The code enclosed by each section is distributed among different threads.

(iii) `single` : This is used to specify a code block that is to be executed by only one thread. A barrier is implied in the end i.e., every other thread will wait at the end of this block until the section specified by `single` is executed.

(iv) `master` : This is similar to `single`, the only difference is that the block of code specified by `master` is to be executed by only the main thread. There is no implicit barrier in this case.

Data environment [2]

Since OpenMP is a shared memory programming model, most variables are visible to all threads by default. These are the shared memory. Sometimes private variables are also necessary to avoid race conditions, so we need some way to specify a variable as shared or private. We use the following to specify the data environments of the variables.

(i) `shared :` the data declared outside a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work-sharing region are shared except the loop iteration counter.

(ii) `private :` the data declared within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.

(iii) `default :` allows the programmer to state that the default data scoping within a parallel region will be either shared or none. The none option forces the programmer to declare each variable in the parallel region using the data-sharing attribute clauses.

One can specify variables to be shared or private in the following manner `shared(var1, var2, ···)` or `private(var1, var2, ···)`. These clauses are appended to the OpenMP directives to specify the data environment.

Scheduling [2]

The scheduling clauses are used in the following manner `schedule(type, chunk_size)`. If we are using a work-sharing construct for a for-loop then the threads are scheduled according to the specified method by this clause. OpenMP divides the iterations into chunks of size `chunk_size`. The three types of scheduling are

(i) `static :` Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. OpenMP divides the iterations into chunks of size `chunk_size` and distributes the chunks to threads in a circular order.

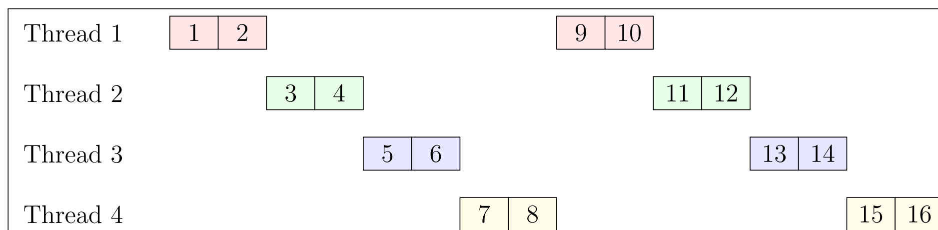


Figure 3.3: Static scheduling of 16 iterations with `chunk_size`=2

(ii) `dynamic` : Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter `chunk_size` defines the number of contiguous iterations that are allocated to a thread at a time. There is no particular order in this scheduling.

Thread	Iterations
Thread 1	5 6
Thread 2	7 8 9 10, 15 16
Thread 3	3 4, 11 12
Thread 4	1 2, 13 14

Figure 3.4: Dynamic scheduling of 16 iterations with `chunk_size`=2

(iii) `guided` : A large chunk of contiguous iterations is allocated to each thread dynamically. The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter `chunk_size`.

Thread	Iterations
Thread 1	12 13 14
Thread 2	7 8 9 10 11
Thread 3	15 16
Thread 4	1 2 3 4 5 6

Figure 3.5: Guided scheduling of 16 iterations with `chunk_size`=2

Environment variables [1]

We can alter the execution feature of the OpenMP application using environment variables. Here are some of the examples

(i) `OMP_SCHEDULE` : We can use this environment variable to control the schedule kind and chunk size of all loop directives.

(ii) `OMP_NUM_THREADS` : This environment variable sets the number of threads to use for parallel regions.

(iii) `OMP_PROC_BIND` : if it is set to `true` then execution environment binds the threads to the cores, if it is set to `false` then environment may move the threads among cores.

There are also other environment variables that are used for different requirements in the application.

3.2.2 Parallelization with OpenMP

As we have discussed in the previous chapter, the construction of the scheme has expensive multiplications in **Setup** and **Encrypt** operations. As depicted in Figure 3.6 the sampled polynomial $a \in R_q$ is multiplied with all the polynomials s_i and added with the error polynomial e_i for all $i \in [\ell]$. All these multiplications are independent of each other, by independent we mean that the outcome of one multiplication does not directly affect the other. Therefore, all these multiplications can be distributed among multiple cores and computed in parallel. Since, all these operations are writing in different memory locations so we don't have any race conditions, even though all these operations depends on the polynomial $a \in R_q$ but this polynomial is not changed throughout the whole operation.

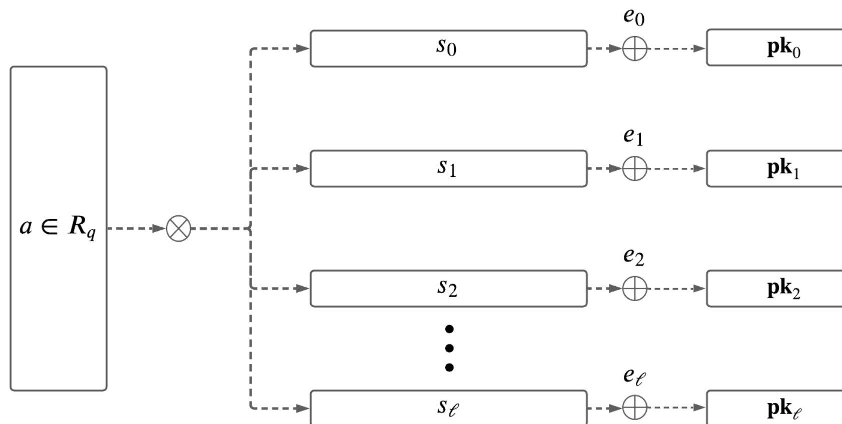
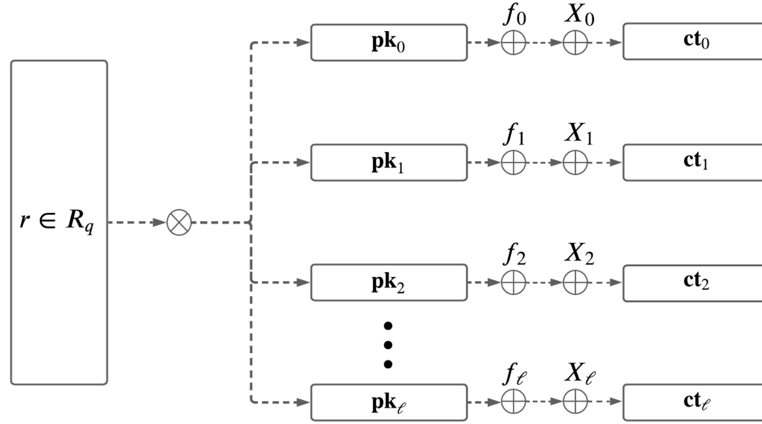


Figure 3.6: Visualization of the operations done in *Setup* operation

Just like the setup algorithm, the encryption operation also have the similar type of construction where the sampled polynomial $r \in R_q$ is multiplied with all the $\mathbf{pk}_i$ and added with $e_i + \lfloor q/K \rfloor x_i 1_R$ for all $i \in [\ell]$, where e_i are the error polynomials and the message $x = (x_1, x_2, \cdots, x_\ell) \in \mathbb{Z}^\ell$. The Figure 3.7 will give the idea of the **Encrypt** operation.



$X_i = \lfloor \frac{q}{K} \rfloor x_i 1_R$, where $x = (x_1, x_2, \cdots, x_\ell)$ is the message

Figure 3.7: Visualization of the operations done in *Encrypt* operation

---

Algorithm 5: Setup algorithm in RLWE-based IPFE [29]

Data: moduli q_1, q_2, q_3 with $q = q_1 q_2 q_3$ and q_i's are primes.
Output: The master public-key and master secret-key pair $(\mathbf{mpk}, \mathbf{msk})$

```
1  for ( i = 1 ; i ≤ 3 ; i++ ) do
2      a_i ←R R_{q_i};
3  for ( i = 1 ; i ≤ ℓ ; i++ ) do                      } parallel
4      s' ← D_{σ1};
5      e' ← D_{σ1};
6      s_i ← CRTconvert(s');
7      e_i ← CRTconvert(e');
8  for ( i = 1 ; i ≤ ℓ ; i++ ) do                      } parallel
9      for ( j = 1 ; j ≤ 3 ; j++ ) do
10         e_ij ← NTT(e_ij);
11         pk_ij ← NTT(a_j) ∘ NTT(s_ij);
12         pk_ij ← pk_ij + e_ij;
13
14 return ((s_i | i ∈ [ℓ]), (pk_i | i ∈ [ℓ]));
```

The Algorithm 5 is the `Setup` algorithm for the RLWE-based IPFE scheme. Each for loops in the algorithm can be parallelized (See Algorithm 5) with OpenMP as each iteration of the loops in `Setup` algorithm are independent of each other. Similarly, the `Encrypt` algorithm (Algorithm 6) can also be parallelized by distributing all the iterations of the loops among multiple threads.

Algorithm 6: Encrypt algorithm in RLWE-based IPFE [29]

Data: moduli q_1, q_2, q_3 with $q = q_1 q_2 q_3$ and q_i's are primes, S_1, S_2, S_2 which are the scaling factors under moduli q_1, q_2, q_3 respectively
Input: a message $x \in \mathbb{Z}^\ell$, public-key $\mathbf{mpk} = \big(a, \{\mathbf{pk}_i \mid i \in [\ell]\}\big)$ in NTT domain
Output: Encryption of the message x

```
1  m ← CRTconvert(x);
2  for ( j = 1 ; j ≤ 3 ; j++ ) do
3      for ( i = 1 ; i ≤ ℓ ; i++ ) do
4          m_ji ← m_ji · S_j  mod q_j;
5  r' ← D_σ2;
6  f' ← D_σ2;
7  r ← CRTconvert(r');
8  f ← CRTconvert(f');
9  for ( j = 1 ; j ≤ 3 ; j++ ) do
10     r_j ← NTT(r_j);
11 for ( j = 1 ; j ≤ 3 ; j++ ) do
12     ct_0j ← a_j · r_j;
13     ct_0j ← NTT^-1(ct_0j);
14     ct_0j ← ct_0j + f_j
15 for ( j = 1 ; j ≤ ℓ ; j++ ) do
16     f' ← D_σ3;
17     f''_j ← CRTconvert(f');
18 for ( j = 1 ; j ≤ 3 ; j++ ) do
19     for ( i = 1 ; i ≤ ℓ ; i++ ) do
20         ct_ij ← pk_ij · r_j;
21         ct_ij ← NTT^-1(ct_ij);
22         ct_ij ← ct_ij + f''_ij;
23         ct_ij ← ct_ij + m_ji 1_R  mod q_j;
```

The above algorithms (Algorithm 5, Algorithm 6) are discussed in Section 2.5.1. The `KeyGen` and `Encrypt` can be similarly parallelized as these algorithms.

Now another approach would be parallelizing the NTT multiplication. In the NTT-forward algorithm (Algorithm 7) we have shown the sections that we can parallelize.

Algorithm 7: Forward NTT transformation

Input: A vector $a = (a_0, a_1, \cdots, a_{n-1}) \in \mathbb{Z}_q^n$ in standard ordering, where q is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\zeta_{rev} \in \mathbb{Z}_q^n$, storing the powers of ζ in bit-reversed order.

Output: $a \leftarrow NTT(a)$ in bit-reversed ordering

```
1  t ← n;
2  for ( m = 1; m < n; m = 2m ) do
3  |   t ← t/2;
4  |   for ( i = 0; i < m; i++ ) do
5  |   |   j1 ← 2 · i · t;
6  |   |   j2 ← j1 + t − 1;
7  |   |   S = ζrev[m + i];
8  |   |   for ( j = j1; j ≤ j2; j++ ) do
9  |   |   |   U ← a_j;
10 |   |   |   V ← a_{j+t} · S;
11 |   |   |   a_j ← U + V mod q;
12 |   |   |   a_{j+t} ← U − V mod q;
13 |   |
14 return a
```

parallel section (line 4-12)

The for loop in line-4 of Algorithm 7 is split among multiple threads. Now the problem is when m is smaller than the number of threads, the number of iteration will be less than the number of threads. Therefore, some threads will be idle which is not something that we would want.

In OpenMP, we can collapse two nested for loops together using the `collapse(n)` clause with the work-sharing construct for a for-loop. Here `n` is the number of nested for loops. Now we modify the Algorithm 7 so that we can use this `collapse` clause. Algorithm 8 is the modified version of NTT. In this algorithm, the for-loops in lines 4 and 5 can be collapsed together giving us a total of $m \times t = n/2$ many iterations. Therefore, at each level, we will have the same number of iterations and the threads will be occupied with the same amount of work. The NTT^{-1} can be similarly parallelized.

Now we have three options for OpenMP optimization

(i) Parallelize the `Encrypt` algorithm and `Setup` algorithm by splitting the for-loops as shown in Algorithm 5.

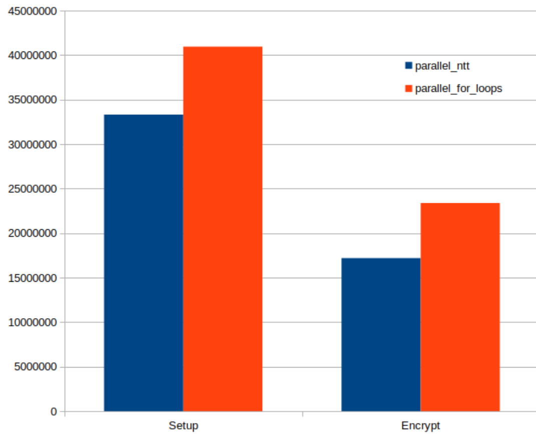
(ii) Parallelize the NTT and NTT^{-1} as we showed in Algorithm 8.

(iii) We can parallelize by using nested threading approach i.e., we will parallelize using

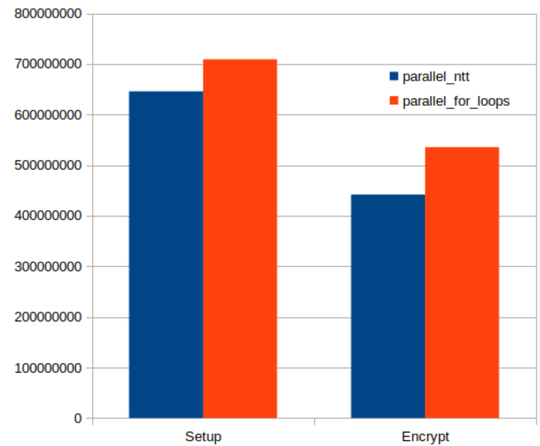
method (i) with m many threads and then each thread will spawn n many threads for NTT transformation as in (ii).

Option (iii) will decrease the performance as spawning threads have some overheads and if each thread spawns more threads for all iterations then the overhead just for spawning threads will be huge. Therefore, we will avoid using nested threading and spawn all the threads at the start of parallel region and distribute work among these threads.

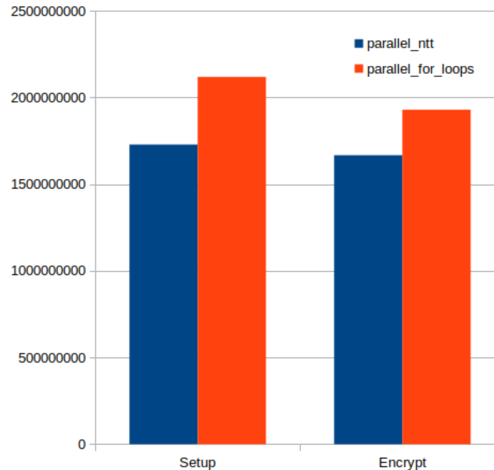
Now, we have to choose between (i) and (ii). The `Decrypt` and `KeyGen` algorithms do not have polynomial multiplications (See 2.5.1), so we will stick to the option (i) for these two algorithms.



(a) `SEC_LEVEL_0`



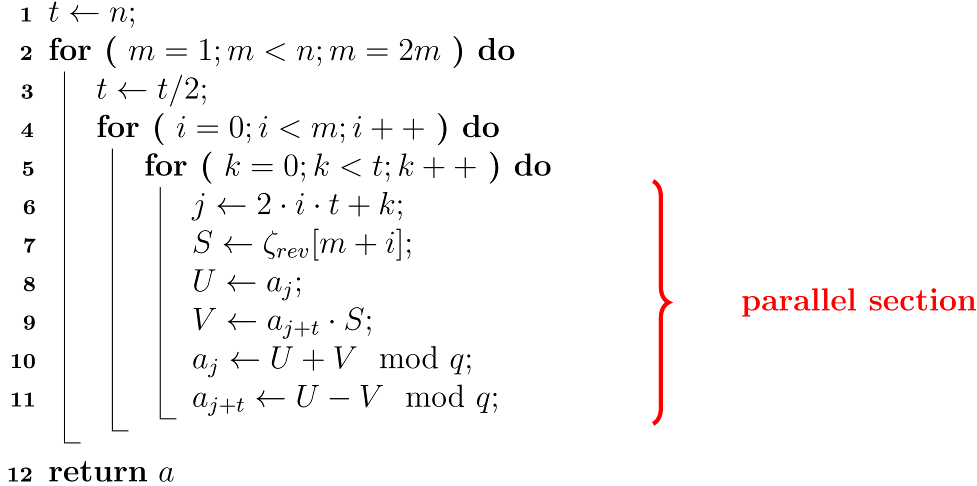
(b) `SEC_LEVEL_1`



(c) `SEC_LEVEL_2`

Figure 3.8: The cpucycles taken for different optimization for different security levels

In the Figure 3.8, we can see that parallelizing NTT using the Algorithm 8 have a better performance for `Setup` and `Encrypt` algorithms. After we have distributed the work among the threads we will now see how the thread scheduling works in the next section.



Algorithm 8: Modified forward NTT transformation

Input: A vector $a = (a_0, a_1, \cdots, a_{n-1}) \in \mathbb{Z}_q^n$ in standard ordering, where q is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\zeta_{rev} \in \mathbb{Z}_q^n$, storing the powers of ζ in bit-reversed order.

Output: $a \leftarrow NTT(a)$ in bit-reversed ordering

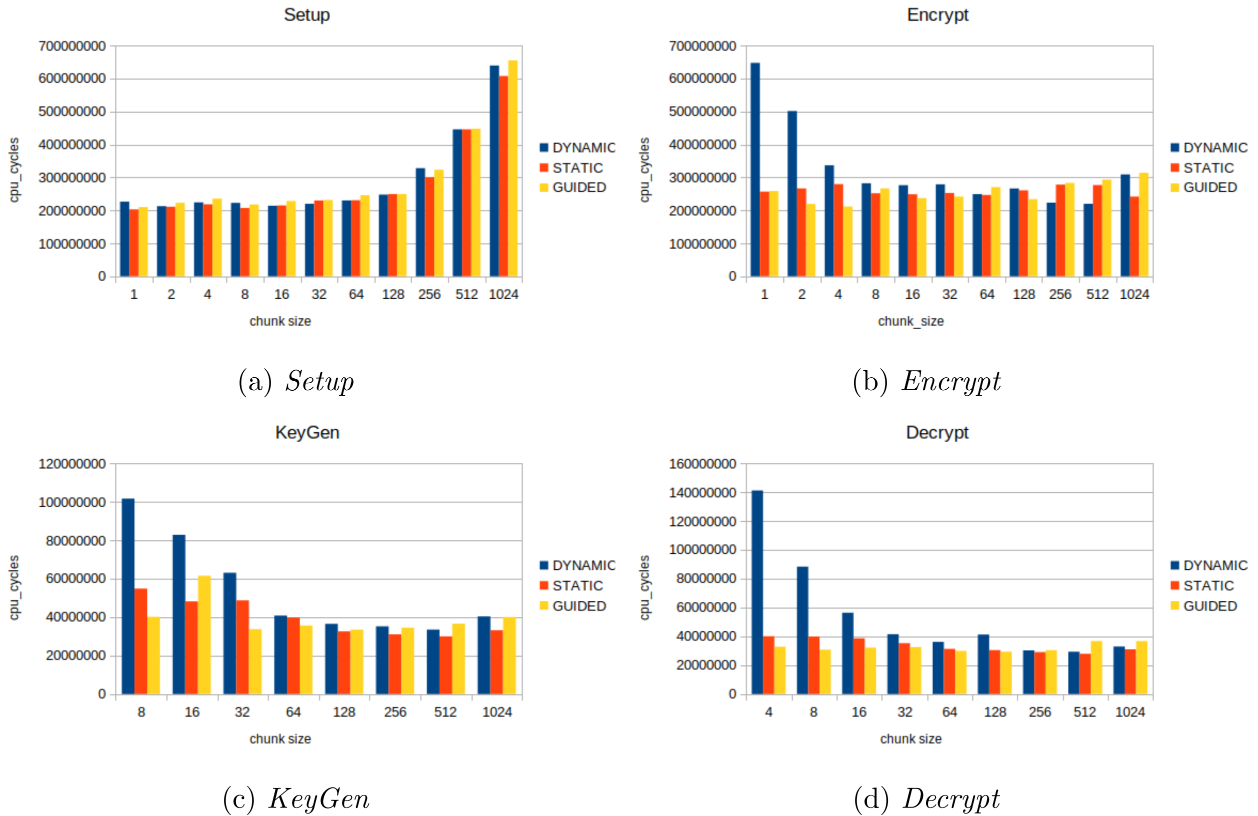
1 $t \leftarrow n$;
2 **for** ($m = 1; m < n; m = 2m$) **do**
3 $\quad t \leftarrow t/2$;
4 $\quad$ **for** ($i = 0; i < m; i++$) **do**
5 $\quad\quad$ **for** ($k = 0; k < t; k++$) **do**
6 $\quad\quad\quad j \leftarrow 2 \cdot i \cdot t + k$;
7 $\quad\quad\quad S \leftarrow \zeta_{rev}[m + i]$;
8 $\quad\quad\quad U \leftarrow a_j$;
9 $\quad\quad\quad V \leftarrow a_{j+t} \cdot S$;
10 $\quad\quad\quad a_j \leftarrow U + V \mod q$;
11 $\quad\quad\quad a_{j+t} \leftarrow U - V \mod q$;
12 **return** a

3.2.3 Scheduling

In OpenMP, we can specify `schedule(type, chunk size)` clause with a specific type and chunk size (See 3.2.1) for different types of scheduling, in a work-sharing construct. The `chunk_size` specifies the number of iterations and `type` is the scheduliing type. There are mainly three types of scheduling that we can use in OpenMP which are `static`, `dynamic` and `guided`, we can also specify types as runtime or auto (without any chunk size) which let the compiler or some external variables to deal with the scheduling.

Now we have used different scheduling and different chunk-sizes to see which is the most optimal setup for RLWE-based IPFE. As we can observe from the Figure 3.9a the *setup* operation works better with smaller chunk sizes, as the chunk size gets larger the operation starts using more cpucycles. The type of scheduling doesn't seems to make a difference in terms of performance. So we can use scheduling with a small chunk size *e.g.*, 8 or 16 for this operation.

Again, the *Encrypt*, *Keygen* and *Decrypt* operations on the other hand have optimal performance with larger chunk sizes if we are using dynamic scheduling. The static, guided scheduling have almost the same performance for different chunk sizes, although the static scheduling does have slight performance increase for chunk sizes of about 128-256.



(a) *Setup* (b) *Encrypt*

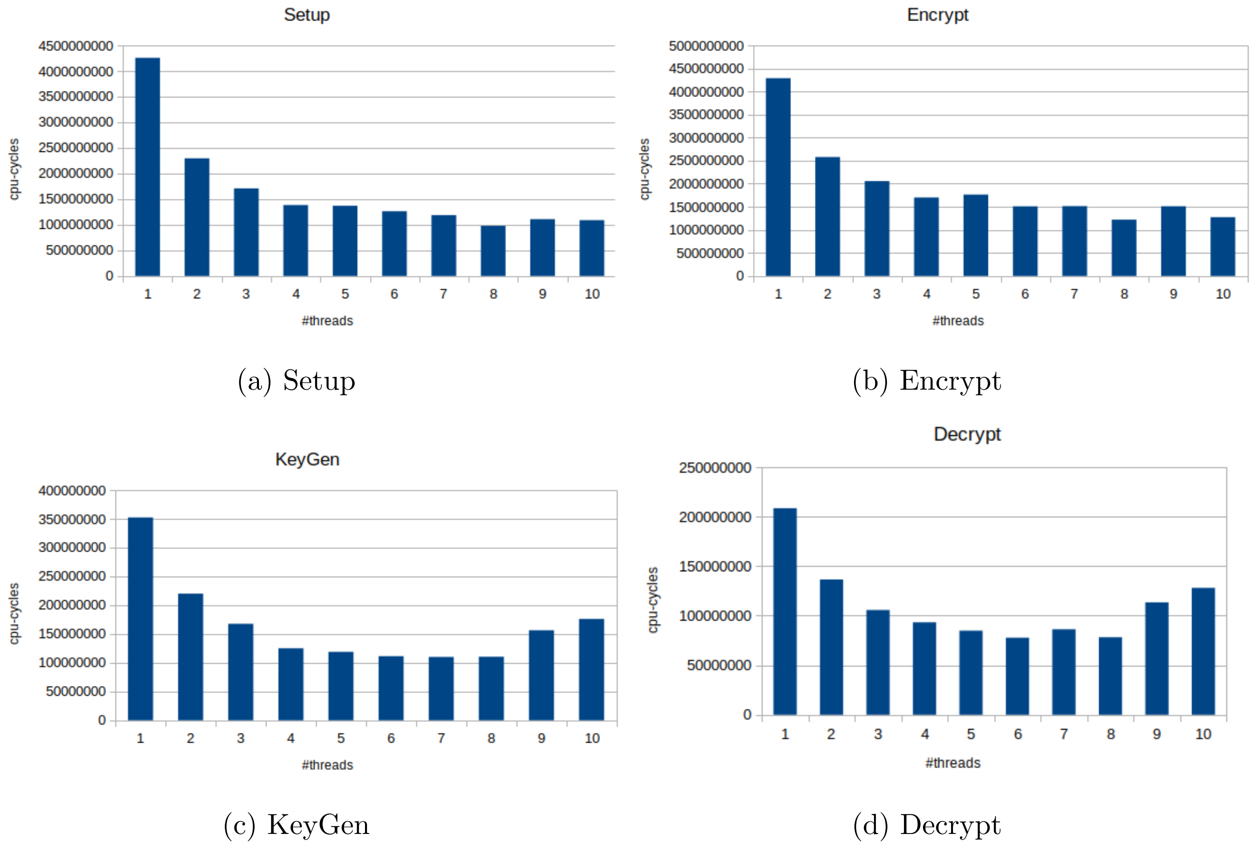
(c) *KeyGen* (d) *Decrypt*

Figure 3.9: The cpucycles required for different operations under different scheduling parameters

3.2.4 Scalability

In any parallel application, we distribute works among multiple threads and these threads run simultaneously on different cores. So it is expected that a parallel application should work better with an increasing number of threads but it also increases the amount of context switching, false sharing which in fact decreases the performance.

In the Figure 3.10 we can see how the performance of the operations increases with the increasing number of threads. The maximum number of threads that we can use without any conflicts depends on the system and the number of cores in the CPU that we are using. In our case, we have 8 threads without any conflicts but using more threads seems to increase the cpucycles for `KeyGen` and `Decrypt`.



(a) Setup

(b) Encrypt

(c) KeyGen

(d) Decrypt

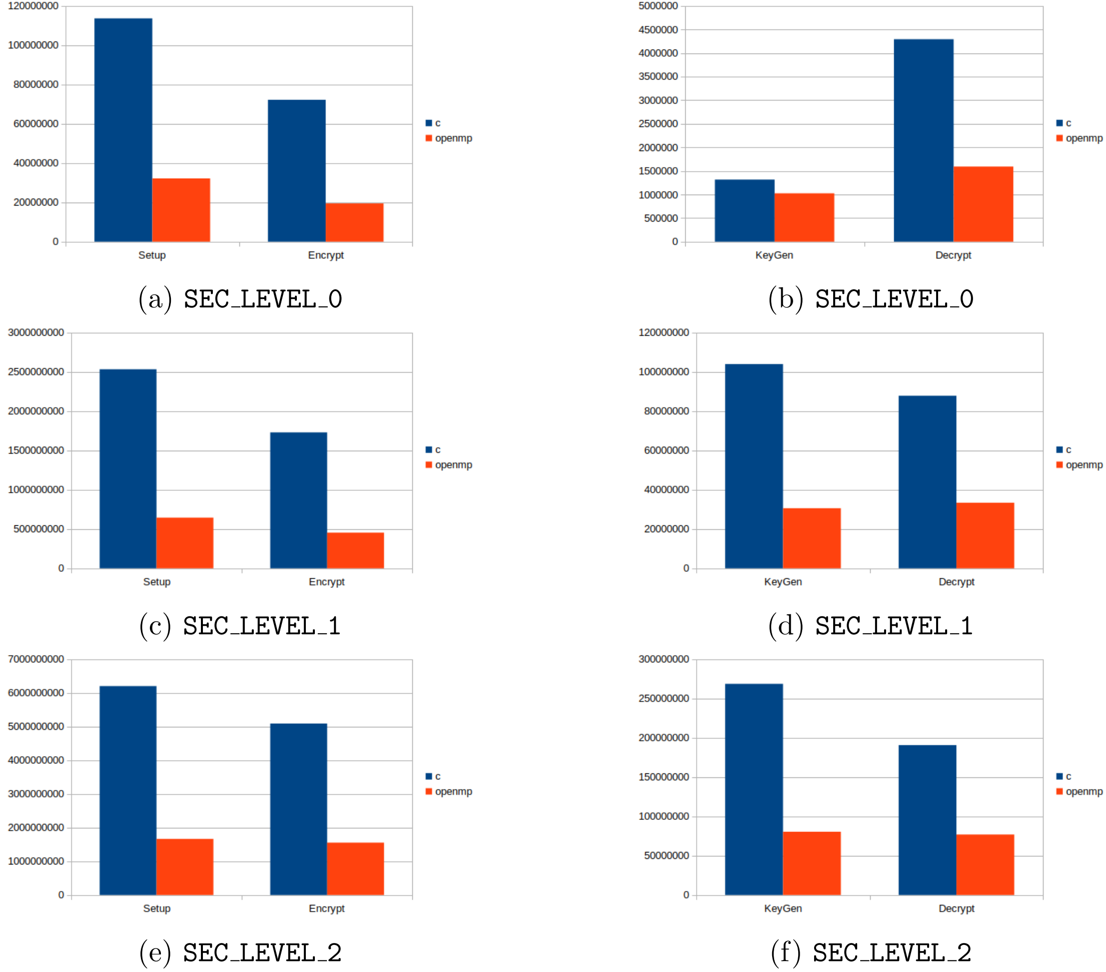
Figure 3.10: Performance of different operations with increasing number of threads

3.2.5 Experimental results

After choosing proper number of threads and scheduling method we have implemented the scheme using with and without OpenMP. We have the results after the implementation. The result for *Setup*, *Encrypt*, *KeyGen* and *Decrypt* is in the Figure 3.11.

It is clear from the Figure 3.11, that we surely have optimized performance with OpenMP implementation. Below we have the table containing results with and without OpenMP optimization for different operations. We have the same result depicted in the Figure 3.11.

Operation	SEC_LEVEL 0		SEC_LEVEL 1		SEC_LEVEL 2	
	no OpenMP	OpenMP	no OpenMP	OpenMP	no OpenMP	OpenMP
Setup	113622133	32144297	2532139210	643970197	6199926817	1664594004
Encrypt	72165295	19432165	1729108772	453515506	5089390687	1552127926
Key_pair	1315350	1024138	103941056	30565113	268592841	80375688
Decrypt	4291258	1590337	87830552	33343998	190676512	76918169



(a) `SEC_LEVEL_0` (b) `SEC_LEVEL_0`

(c) `SEC_LEVEL_1` (d) `SEC_LEVEL_1`

(e) `SEC_LEVEL_2` (f) `SEC_LEVEL_2`

Figure 3.11: Performance optimization for different security levels

OpenMP optimization has surely given a significant speed-up to the implementation of the scheme but we have parallelized NTT multiplication instead of parallelizing the for-loop in `Setup` and `Encrypt` operations. Now we make another change in the implementation. Instead os using OpenMP to parallelize the NTT, we use AVX2 to vertorize the implementation of NTT. After that we can easily parallelize the for-loops in `Setup` and `Encrypt` as it is shown in Algorithm 5.

3.3 AVX2 Optimization

One of the most expensive operations in RLWE based schemes is polynomial multiplication. As we have discussed in 3.1.3, the primes q, that are chosen for the implementation are of the form $q \equiv 1 \mod 2n$. So we can use NTT transformation for polynomial multiplication which is a quasi-linear time algorithm. We will discuss the AVX2 implementation of NTT in this section.

3.3.1 8-point NTT

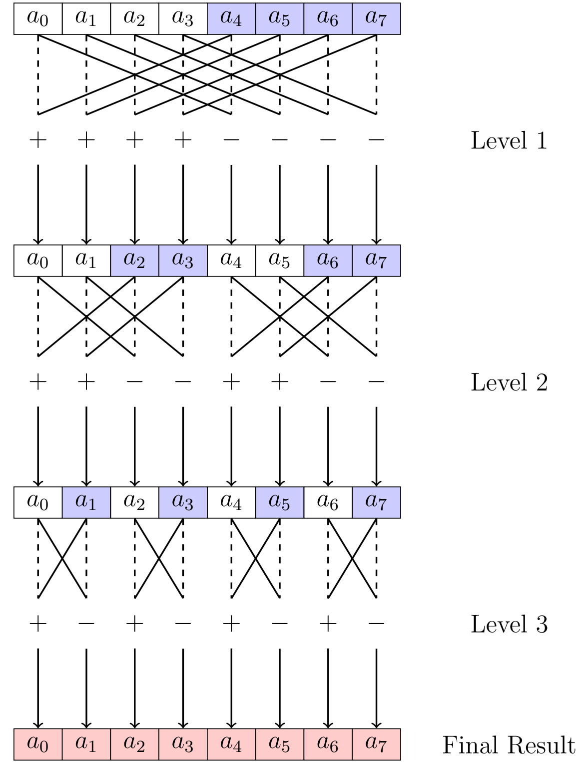


Figure 3.12: 8-point NTT

In the Figure 3.12, it is depicted how the 8-point NTT transformation works. First, in level-1 the coefficients with blue colour are multiplied with the powers of ζ, which are called the twiddle factors. The coefficients a_0 to a_7 are updated using the following update rule.

(i) At level-1 for each $0 \leq j < 4$, the coefficients are updated as

$$a_j = a_j + a_{j+4} \cdot \zeta_{rev}[1]$$
$$a_{j+4} = a_j - a_{j+4} \cdot \zeta_{rev}[1]$$

(ii) At level-2 for each $0 \leq j < 2$, the coefficients are updated as

$$a_j = a_j + a_{j+2} \cdot \zeta_{rev}[2]$$
$$a_{j+4} = a_j - a_{j+2} \cdot \zeta_{rev}[2]$$

And for each $4 \leq j < 6$, the coefficients are updated as

$$a_j = a_j + a_{j+2} \cdot \zeta_{rev}[3]$$
$$a_{j+4} = a_j - a_{j+2} \cdot \zeta_{rev}[3]$$

(iii) At level-3 for each $0 \leq i < 4$, the coefficients are updated as

$$a_{2i} = a_{2i} + a_{2i+1} \cdot \zeta_{rev}[4+i]$$
$$a_{2i+1} = a_{2i} - a_{2i+1} \cdot \zeta_{rev}[4+i]$$

The update method for 8-point NTT is depicted in Figure 3.12.

This 8-point NTT is going to be useful for the implementation of full-NTT for the scheme. The number of coefficients in the polynomials that we will be dealing with are 2048, 4096 or 8192 depending on the security level.

3.3.2 NTT with AVX2

In this section, we will discuss the NTT and how to deal with the transformation with 8192 many coefficients. The cases with 4096 or 2048 many coefficients are similar. For $N = 8196$ there is total 13 levels in NTT transformation.

Levels 1-3 of NTT

Initially, we have an array of size $N = 8192$. First, we will complete the first three levels of NTT. From the Algorithm 2, we can observe that for any coefficient a_j for $j < N$, the first three levels of NTT transformation of this coefficient is influenced by $a_{j+N/2}$, $a_{j+N/4}$ and $a_{j+N/8}$. Therefore taking the eight points $A_j = \left\{a_{j+\frac{kN}{8}} \mid 0 \leq k < 8\right\}$, we can perform the 8-point NTT transformation on it to obtain the first 3 levels of NTT transformation of A_j for each $0 \leq j < N/8$.

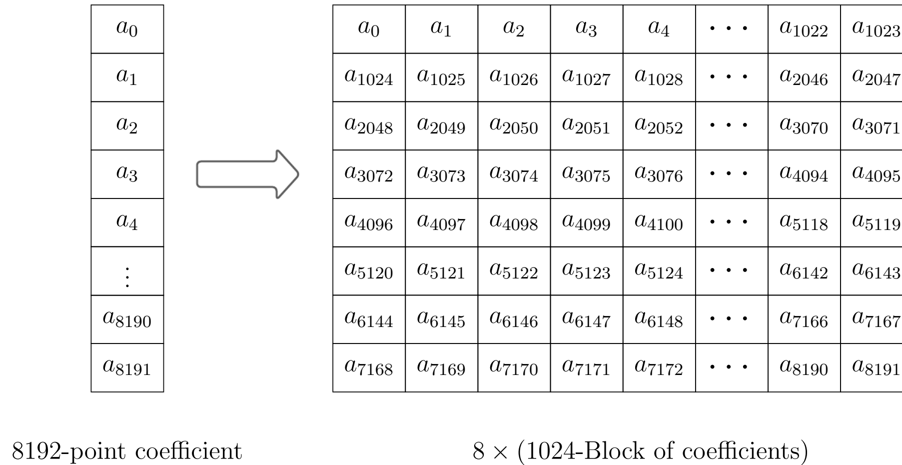


Figure 3.13: Visualizing 8192 size array as a 8×1024 matrix

In other words, we consider this array of size $N = 8192$ as an 8×1024 matrix by putting all the elements row-wise in the matrix. We can see the order of the elements in Figure 3.13. Each columns of the matrix is A_j for $0 \leq j < 1024$. Therefore 8-point NTT transformation

on each of these columns of the matrix perform the first three levels of NTT transformation of the whole array of size 8192.

In AVX2 we can use 256-bit registers, so in one register we can pack eight 32 bit consecutive integers. Now we can take eight such registers and in each of these registers, we can pack eight consecutive elements from each row of the matrix in Figure 3.13. After the coefficients loaded in these registers we can apply the 8-point NTT transformation. At each level half of the registers are to be multiplied with the twiddle factors. We pack the twiddle factors in one register and multiply it with eligible registers. The AVX2 code for the first three levels of transformation is given in Appendix A.

Levels 4-7 of NTT

After the first three levels are done, we have to transform the polynomials with 1024 coefficients. Each of these 1024 coefficients can be considered as a matrix of size 8×128. Once we transform each column of this matrix of size 8×128, three more levels of transformation of NTT will be done. In AVX2, these three levels of transformation are similar to the first three levels.

Now, we conduct only one level of transformation on the polynomials of size 128. We consider $a_0, a_1, \cdots, a_{127}$ to be the 128 many coefficients. First we load a_0 to a_{31} into first four registers and a_{64} to a_{95} in the rest of four registers. Now multiplying twiddle factors to the second four registers, we update the values and store them. Next a_{32} to a_{63} is loaded into the first four registers and a_{96} to a_{127} is loaded into the second four registers and similarly, the values are updated and stored. Thus we complete level-7 of NTT transformation. Now we have to transform each polynomial with 64 coefficients.

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{16}	a_{17}	a_{18}	a_{19}	a_{20}	a_{21}	a_{22}	a_{23}
a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}	a_{30}	a_{31}
a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{48}	a_{49}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}
a_{56}	a_{57}	a_{58}	a_{59}	a_{60}	a_{61}	a_{62}	a_{63}

Figure 3.14: The first 64 coefficients as 8×8 matrix

Level 8-13 of NTT

Now we describe how the polynomial with 64 many coefficients can be transformed using NTT. The 64 many coefficients can be considered as an 8×8 matrix as shown in Figure 3.14. All 64 coefficients can be loaded into eight AVX2 registers. Now level-8 to level-11, these three levels of the transformation is similar to level-1 to level-3 transformation. Now our target is to complete the last three levels.

For the last three levels of transformation, we have adapted the method from [40]. They have worked with 16-bit integers for Kyber [8],here we are dealing with 32 bits of integers. First of all, we require some permutations on 16 points. More specifically, we require three permutations namely π_1, π_2 and π_4. The description of the permutations is in the figures below. Also the AVX codes of these permutation are in Appendix C.

The permutation π_4 is depicted in Figure 3.15

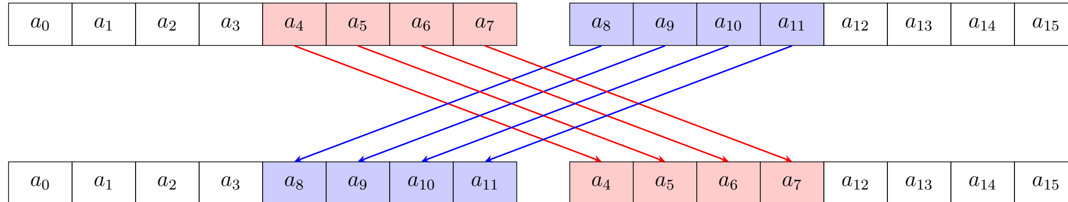


Figure 3.15: Permutation π_4

The permutation π_2 is depicted in Figure 3.16

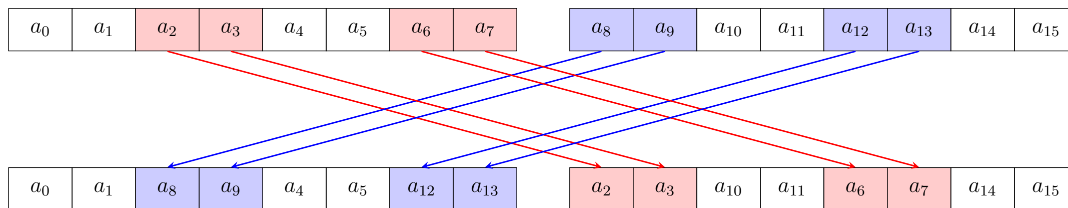


Figure 3.16: Permutation π_2

The permutation π_1 is depicted in Figure 3.17

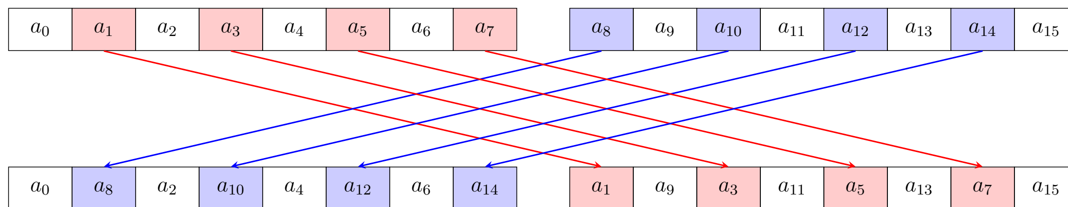


Figure 3.17: Permutation π_1

Starting from level-11 of NTT each of the polynomials have eight coefficients which are packed into one register. Now half of the coefficients in this one register needs to be multiplied with the twiddle factors and updated. Thus we would apply these permutation on two registers such it will group all the coefficients which are to be multiplied into one register and rest are in another.

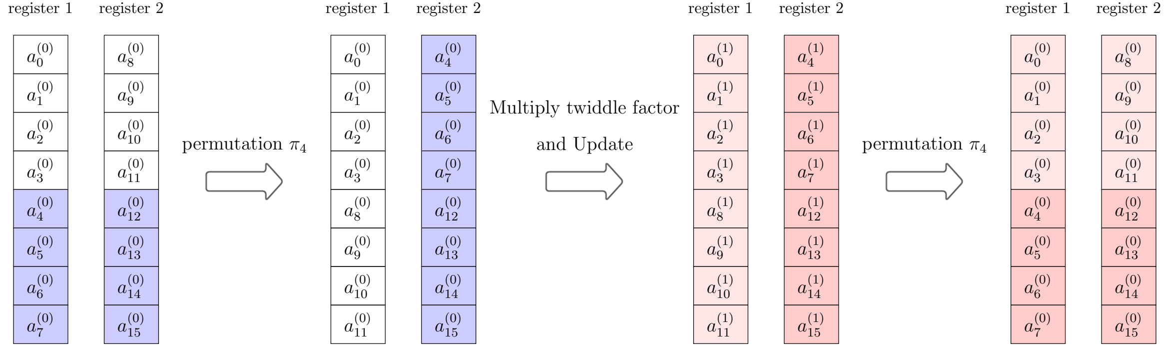


Figure 3.18: First of the last three levels of NTT

At the start of level-11 we take a pair of two registers containing polynomials with eight coefficients as seen in Figure 3.18. The high four coefficients (coloured blue in the figure) are to be multiplied with twiddle factors. Therefore, we first apply the permutation π_4 and take all the high coefficients of the polynomials in register-2 and rest in register-1. Now we can multiply the twiddle factors to the register-2 and update. In the end, we apply the permutation π_4 again to put the transformed polynomials in their own place again. Doing this for all 4 pairs of registers, perform the level-11 of the NTT.

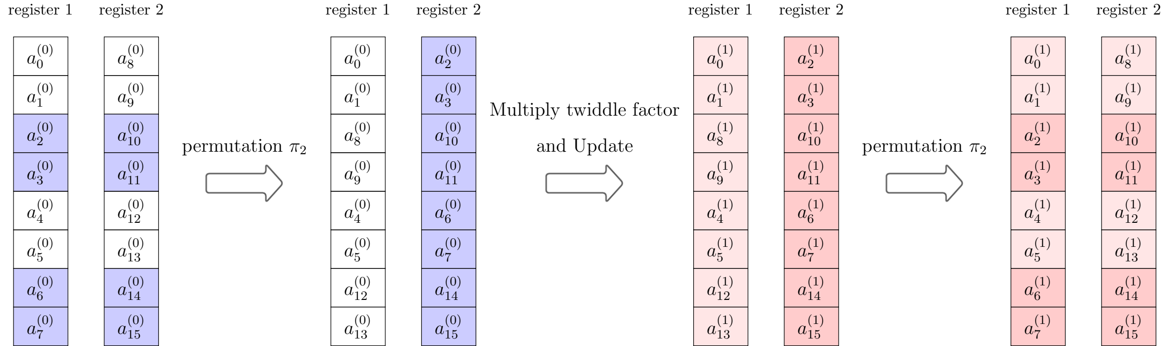


Figure 3.19: Second of the last three levels of NTT

At the start of level-12 we take a pair of two registers, each containing two polynomials with four coefficients as seen in Figure 3.19. The high two coefficients of each polynomial (coloured blue in the figure) are to be multiplied with twiddle factors. Therefore, we first apply the permutation π_2 and take all the high coefficients of the polynomials in register-2 and rest in register-1. Now we can multiply the twiddle factors to the register-2 and update. At the end, we apply the permutation π_2 again to put the transformed polynomials in their own place again. Doing this for all 4 pairs of registers, perform the level-12 of the NTT.

At the last level we take a pair of two registers, each containing four polynomials with four coefficients as seen in Figure 3.20. The alternative coefficients of each register (coloured blue in the figure) are to be multiplied with twiddle factors. Therefore, we first apply the permutation π_1 and take all the odd indexed coefficients of the two registers in register-2 and even indexed coefficients in register-1. Now we can multiply the twiddle factors to the register-2

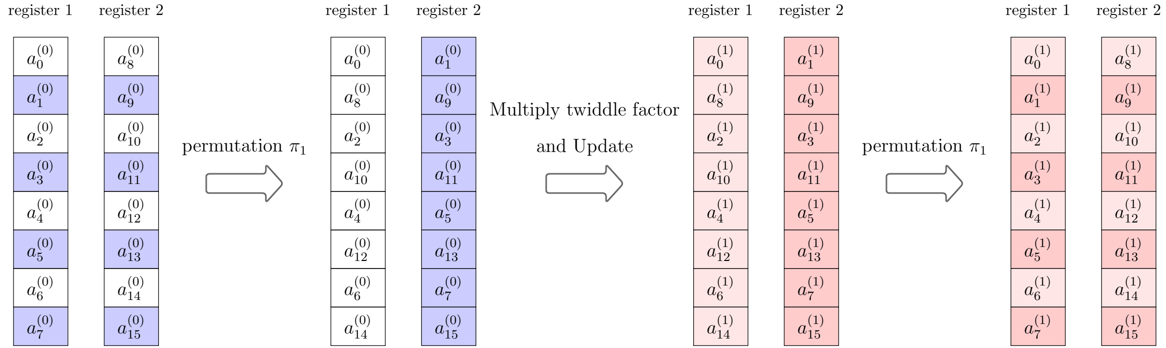


Figure 3.20: Third of the last three levels of NTT

and update. In the end, we apply the permutation π_1 again to put the transformed polynomials in their own place again. Doing this for all 4 pairs of registers, perform the level-13 or the last level of the NTT.

Each of the `ymm` registers supported by AVX2 instructions has a length of 256 bits which can load eight 32 bits integers at once. So we can load 8 many coefficients in one `ymm` register and we are using a total of 8 registers to store the coefficients from the array. We have reserved two registers to pack the constants q, $q^{-1} \mod 2^{32}$. One register is used to pack the twiddle factors. The rest of the five registers are used to store temporary results during the computation.

Thus we have completed full-NTT of 8192 points. Similarly, for 2048 or 4096 points NTT we can use the same method. The inverse-NTT or NTT^{-1} can be implemented using same technique, just all the steps for NTT are reversed. We have implemented the *Cooley-Tukey* forward NTT (Algorithm 2) and *Gentleman-Sande* (Algorithm 3) in C and using the AVX2 instructions as described in the above section and there is a significant speedup we can observe in the Figure 3.21.

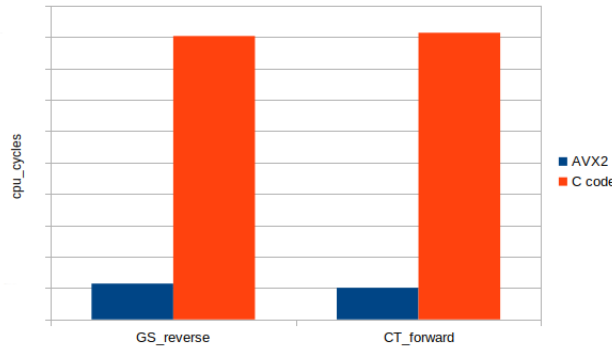


Figure 3.21: cpucycles taken for AVX2 implementation and normal C code

3.3.3 Incomplete NTT

In this section, we describe incomplete NTT. In a recently published paper by Chung et al. [10] and also in the paper by Lyubashevsky and Seiler [26], they have used this technique to optimize the implementation of NTT in unfriendly rings where the full NTT is not possible. The idea here is to skip the last few levels of NTT and apply standard polynomial multiplications instead of point-wise multiplication and apply inverse-NTT skipping the first few levels. Say that we skip the last ℓ many levels of NTT, then at the end, we have polynomials with 2^ℓ coefficients. These polynomial are multiplied using the schoolbook multiplication method and the NTT^{-1} is applied on this resultant polynomial skipping the first ℓ many levels. We denote NTT_ℓ and NTT_ℓ^{-1} to represent incomplete NTT and NTT^{-1} respectively with ℓ many levels skipped.

Now the big question is how many levels to skip. We can skip the last level and settle for a 2×2 schoolbook multiplication or we can skip the last 2 levels and do a 4×4 multiplication. But increasing the number of levels skipped increases the size of the polynomials to be multiplied later.

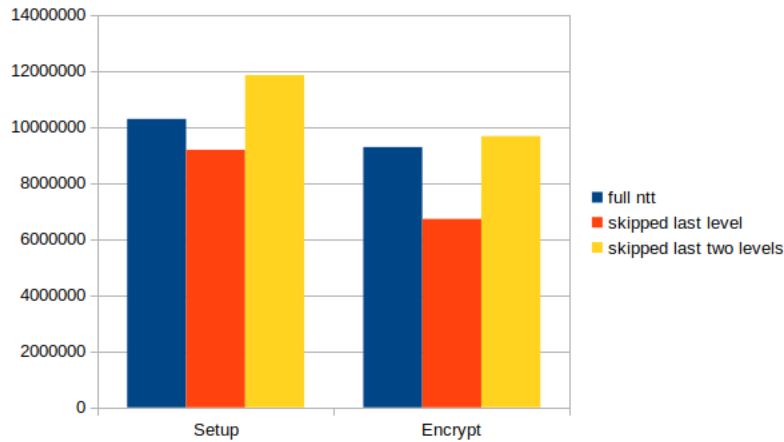


Figure 3.22: Result for incomplete NTT

As we can see in the Figure 3.22 there is indeed a slight improvement in the performance of *Setup* and *Encrypt* algorithm when we skip the last level of NTT and opt for a 2×2 multiplication to complete the polynomial multiplication.

Now we have to multiply two one degree polynomials. We already know that

$$x^n + 1 = \prod_{i=0}^{n-1} (x - \zeta^{2i+1})$$

Where n is some power of 2, ζ is the $2n$'th primitive root of unity in $\mathbb{Z}_q$. Now for any $t < n$

we have, $\zeta^{t+n} = -\zeta^t$. So we can write

$$x^n + 1 = \prod_{i=0}^{n/2-1} (x - \zeta^{2i+1})(x + \zeta^{2i+1})$$

Now observe that for any number $k < n$, if k is odd then $\texttt{brv}(k) > n/2$. $\texttt{brv}$ maps a $\log_2 n$ bit number to its bit-reversal form. So for $i < n/2$, $2i + 1 = \texttt{brv}(n/2 + t)$ for some $t < n/2$. Thus we can write

$$\begin{aligned} x^n + 1 &= \prod_{i=0}^{n/2-1} (x - \zeta^{\texttt{brv}(n/2+i)})(x + \zeta^{\texttt{brv}(n/2+i)}) \\ &= \prod_{i=0}^{n/2-1} (x^2 - \zeta^{2\cdot\texttt{brv}(n/2+i)}) \end{aligned}$$

Algorithm 9: 2×2 base multiplication for incomplete-NTT

Input: Two arrays a and b of n many coefficients in iNTT domain, an array with precomputed values of $\zeta^{2\texttt{brv}(n/2+i)}$ for all $0 \leq i < n/2$, a prime module q

Output: An array c of size n with each components c_i for $0 \leq i < n$ such that $c_{2j} + c_{2j+1}x = (a_{2j} + a_{2j+1}x) \cdot (b_{2j} + b_{2j+1}x) \mod (x^2 - \zeta^{2\texttt{brv}(n/2+j)})$ for $0 \leq j < n/2$.

```
1  for ( i = 0 ; i < n/2 ; i++ ) do
2      ω ← ζ^{2brv(n/2+i)};
3      A ← a_{2i} + a_{2i+1}  mod q;
4      B ← b_{2i} + b_{2i+1}  mod q;
5      C ← a_{2i} · b_{2i}  mod q;
6      D ← a_{2i+1} · b_{2i+1}  mod q;
7      A ← A · B  mod q;          /* A = (a_{2i} + a_{2i+1}) · (b_{2i} + b_{2i+1}) */
8      B ← C + D  mod q;          /* B = a_{2i} · b_{2i} + a_{2i+1} · b_{2i+1} */
9      D ← D · ω  mod q;          /* D = a_{2i+1} · b_{2i+1} · ζ^{2brv(n/2+i)} */
10     c_{2i} ← C + D  mod q;
11     c_{2i+1} ← A − B  mod q;
```

As we see in the Algorithm 2, we store the powers of ζ in the bit-reversed order. So if we skip the last level of NTT then a polynomial $f \in R_q$ will be transformed into

$$\left(f \mod (x^2 - \zeta^{2\cdot\texttt{brv}(n/2)}), f \mod (x^2 - \zeta^{2\cdot\texttt{brv}(n/2+1)}), \cdots, f \mod (x^2 - \zeta^{2\cdot\texttt{brv}(n-1)})\right)$$

NTT_1 and NTT_1^{-1} are the incomplete versions of NTT and NTT^{-1} respectively with one skipped level. Then the multiplication of two polynomials $f, g \in R_q$ is calculated as

$$fg = \text{NTT}_1^{-1}\big(\text{NTT}_1(f) \circ \text{NTT}_1(g)\big)$$

If $h = \text{NTT}_1(f) \circ \text{NTT}_1(g)$, then $h_i = fg \mod (x^2 - \zeta^{2\cdot\texttt{brv}(n/2+i)})$ for all $0 \le i < n/2$. Therefore, $fg = \text{NTT}_1^{-1}(h)$.

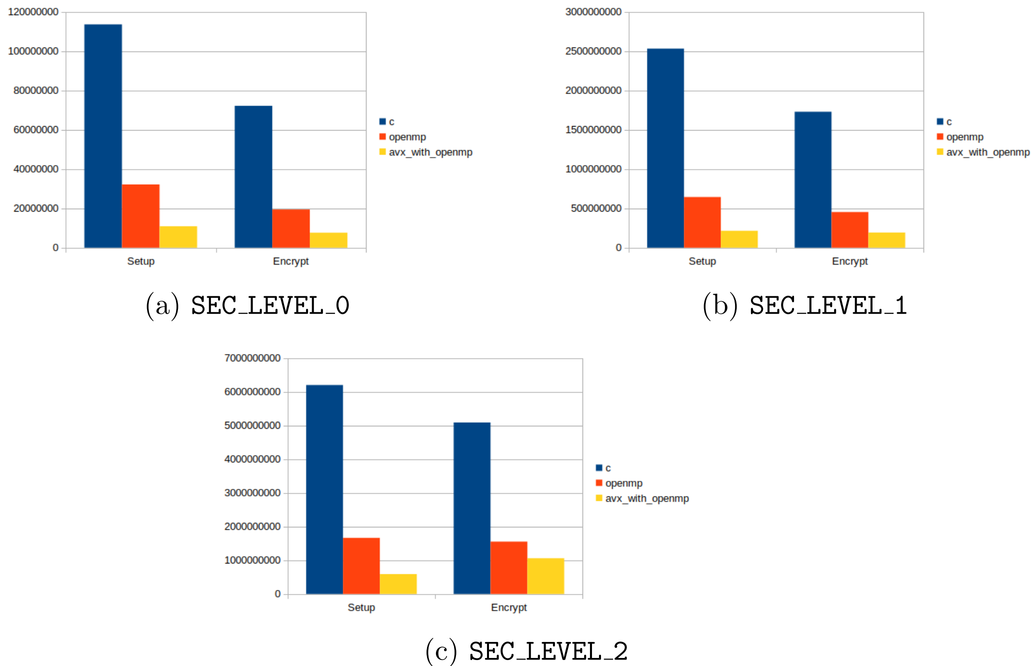
For the implementation purpose the list of pre-computed values of $\zeta^{2\texttt{brv}(n/2+i)}$ for each $0 \le i < n/2$, is stored in an array. The computation is done according to the Algorithm 9.

The assembly code of this multiplication in Algorithm 9 is given in Appendix E.

3.3.4 Reducing loads and stores

The `ymm` registers in AVX2 vector instructions have a width of 256 bits. So we can pack eight 32 bit integers into one register. In our implementation of NTT, before the last 6 levels, we have densely packed 64 coefficients into eight registers and completed three levels of transformation without any further load or store of the coefficients. So it takes 8 loads and 8 stores for every 64 coefficients for 3 levels of transformation. Again, in the last six levels of transformation, we can load an entire polynomial of degrees under 64 in these eight registers and complete six levels of transformation without any further load or store. This way we have reduced the total number of load and store for the AVX2 implementation of the NTT.

3.3.5 Experimental results



(a) `SEC_LEVEL_0` (b) `SEC_LEVEL_1`

(c) `SEC_LEVEL_2`

Figure 3.23: Comparision of cpucycles for C, OpenMP and AVX2 with OpenMP

Since we have optimized the polynomial multiplication using AVX2 so we only see a performance increase in *Setup* and *Encrypt* algorithm. As we can see the construction of

the *selectively secure* version of the IPFE scheme we are implementing here (See 2.5.1), the *KeyGen* and *Decrypt* operations do not have many polynomial multiplications. So we cannot expect these operations to be optimized after the AVX2 optimization. In the Figure 3.23 we have presented how cpucycles are decreased with each layers of optimization for different security levels, Table 3.1 contains the values that has been plotted in the Figure 3.23.

Security Levels	operations	Optimizations				
		C-code	OpenMP	Optimization	AVX+OpenMP	Optimization
SEC_LEVEL_0	Setup	113622133	32144297	71.71	10886303	94.42
	Encrypt	72165295	19432165	73.07	7594374	89.48
	KeyGen	1315350	1024138	22.14	1076530	18.17
	Decrypt	4291258	1590337	62.94	1104984	74.25
SEC_LEVEL_1	Setup	2532139210	643970197	74.57	214679687	91.52
	Encrypt	1729108772	453515506	73.77	192252489	88.88
	KeyGen	103941056	30565113	70.59	31185826	70.00
	Decrypt	87830552	33343998	62.04	27810494	68.34
SEC_LEVEL_2	Setup	6199926817	1664594004	73.15	590560069	90.47
	Encrypt	5089390687	1552127926	69.50	1060621261	79.16
	KeyGen	268592841	80375688	70.08	84117554	68.68
	Decrypt	190676512	76918169	59.66	60222755	68.42

Table 3.1: Performance increased with different layers of optimization

Table 3.1 shows how the performance is increased (in %) with OpenMP and AVX2 optimizations. As we can see that after AVX2 optimization only the *Setup* and *Encrypt* operations have shown significant speed-up as they are heavily equipped with polynomial multiplication.

3.4 Conclusion

In this chapter, we have shown how we have optimized the implementation of the RLWE-based IPFE. We have shown results that how the optimization has increased the performance of the scheme for different security levels. In the next chapter, we introduce a privacy-preserving biometric protocol based on this IPFE scheme.

Chapter 4

Privacy preserving biometric protocol

Here we discuss one application of IPFE. We propose a secure privacy-preserving biometric authentication protocol. There are mainly two types of biometric (i) fingerprint, (ii) iris image. Here we are using iris as our biometric data. We use the iris matching algorithm by Daugman [13], Wildes [43] have also mentioned similar matching algorithm. We have briefly discussed about this iris matching algorithm from [13] in Appendix F. In this algorithm an iris image is transformed into a 2048 bit binary string, this is called the *feature vector*. A *mask* of same length is also generated which has "*0*" in all the places having invalid bit. Before getting into the privacy preserving biometric protocol, we discuss some preliminaries first.

4.1 Preliminaries

Let X be our iris feature vector, a k-bit binary array. The feature extraction algorithm also outputs a binary mask vector with "*1*" at the positions with valid bits and "*0*" at the positions with invalid bits. To match two different sets of biometric data we use *Normalized Hamming Distance*.

Definition 4.1.1 (Normalized Hamming distance/fractional Hamming distance [13]). *Let X, M and X', M' be pairs of feature vector and mask of two biometric data then the* normalized Hamming distance *between the two biometric data is defined by*

$$NHD = \frac{|(X \oplus X') \cdot M \cdot M'|}{|M \cdot M'|}$$

Since we are trying to implement the protocol using *Inner Product Functional Encryption*, so first we need to find a method of computing NHD which is suitable with the IPFE protocol i.e., which can be easily implemented using the IPFE protocol. We have discussed a method in the next section.

4.1.1 Transformation of the binary vector and some results

In the paper by Kim et al. [23], They have given a method to calculate Hamming distance of two binary vectors using inner product. We use this method and extend it to calculate the *normalized Hamming distance*. First of all, we consider the vector $\widetilde{X}$, which is some transformation of the binary vector X defined as below :

$$\widetilde{X}_i = \begin{cases} 1 & if \quad X_i = 0 \\ -1 & if \quad X_i = 1 \end{cases}$$

Let us consider the operation "$\odot$" be the pointwise multiplication operation between two vectors of same size. Therefore for any two vectors A and B, if $C = A \odot B$ then

$$\langle A, B \rangle = \sum_{i=1}^{n} C_i$$

Before calculating the *Normalized Hamming Distance* using the inner product, we derive the following results involving the transformation to extend the calculation of *Hamming distance* to *normalized Hamming distance*.

Results

Let us consider two binary vectors A and B of length n then,

- Result 1. We claim that for A and B, we can write

$$\widetilde{A} \odot \widetilde{B} = (\widetilde{A \oplus B}) \tag{4.1}$$

Consider the following table for two binary bits a, b

a	b	$\widetilde{a}$	$\widetilde{b}$	$\widetilde{a} \odot \widetilde{b}$	$a \oplus b$	$(\widetilde{a \oplus b})$
0	0	1	1	1	0	1
0	1	1	-1	-1	1	-1
1	0	-1	1	-1	1	-1
1	1	-1	-1	1	0	1

Table 4.1: The table corresponding to two binary bits a and b

From the table it is clear that $\widetilde{A_i} \odot \widetilde{B_i} = (\widetilde{A_i \oplus B_i})$ for all $i \in \{0, 1, \cdots, n\}$
Therefore, $\widetilde{A} \odot \widetilde{B} = (\widetilde{A \oplus B})$

- Result 2. Now we claim for A and B, we can write

$$|A \cdot B| = \frac{|B| - \left\langle \widetilde{A}, B \right\rangle}{2} \tag{4.2}$$

Now if we take $C = \widetilde{A} \odot B$ then,

$$\sum_{i=1}^{n} C_i = \#\{\text{1's in } \widetilde{A} \odot B\} - \#\{\text{-1's in } \widetilde{A} \odot B\}$$

Again we have,

$$|B| = \#\{\text{1's in } \widetilde{A} \odot B\} + \#\{\text{-1's in } \widetilde{A} \odot B\}$$

So we have

$$\begin{aligned} |B| - \sum_{i=1}^{n} C_i &= 2 \times \#\{\text{-1's in } \widetilde{A} \odot B\} \\ &= 2 \times \left| \left\{ i \in \{1, 2, \cdots, n\} | B_i = 1, A_i = 1 \right\} \right| \\ &= 2 \times |A \cdot B| \end{aligned}$$

Therefore, $|A \cdot B| = \frac{|B| - \left\langle \widetilde{A}, B \right\rangle}{2}$

4.1.2 Calculating NHD using inner product

Here in this section we describe the method to find NHD, using the results in Section-4.1.1, we are now going to calculate *normalized Hamming distance* using the inner product. Let X, X' are the binary feature vectors and M, M' be the binary masks of the biometric data. We know the *Normalized Hamming Dastance* is defined as

$$NHD = \frac{|(X \oplus X') \cdot M \cdot M'|}{|M \cdot M'|}$$

Now, from (4.2) we can write

$$|(X \oplus X') \cdot M \cdot M'| = \frac{|M \cdot M'| - \left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle}{2}$$

We know that, for any pair of binary vectors A and B, $A \cdot B = A \odot B$. Now, we consider the vector

$$\begin{aligned} T &= \widetilde{(X \oplus X')} \odot (M \cdot M') \\ &= \widetilde{X} \odot \widetilde{X'} \odot M \odot M' \qquad\qquad \left[\text{ from } (4.1)\ \right] \\ &= (\widetilde{X} \odot M) \odot (\widetilde{X'} \odot M') \end{aligned}$$

Since, $\sum_{i=0}^{n} T_i = \left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle$, from the above representation of T we can write that

$$\left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle = \left\langle \widetilde{X} \odot M, \widetilde{X'} \odot M' \right\rangle$$

So, we have

$$|(X \oplus X') \cdot M \cdot M'| = \frac{|M \cdot M'| - \left\langle \widetilde{X} \odot M, \widetilde{X'} \odot M' \right\rangle}{2}$$

Again the value $|M \cdot M'| = \langle M, M' \rangle$. This gives us the value of the *normalized Hamming distance* as below

$$NHD = \frac{\langle M, M' \rangle - \left\langle \widetilde{X'} \odot M', \widetilde{X} \odot M \right\rangle}{2 \langle M, M' \rangle} \tag{4.3}$$

clearly, we are only using inner products to calculate this *Normalized Hamming Distance*, so we can use our IPFE scheme here to implement the privacy-preserving biometric authentication protocol.

4.2 Privacy preserving biometric protocol using IPFE

Here we propose a privacy-preserving biometric authentication protocol based on IPFE. Here we consider that the client has created two keys K_x, K_m for the feature vector X and mask M respectively and saves the pair $(X \oplus K_x, M \oplus K_m)$ in the server's database at the time of enrollment. At the time of verification the client has to come up with the keys K_x, K_m along with the feature vector X' and mask M'.

4.2.1 Protocol

- First client creates two pairs of *master public key* and *master secret key* namely, (msk_1, mpk_1) and (msk_2, mpk_2).
- Server sends the encrypted mask $T = M' \oplus K_m$ to the client.
- If the client has correct K_m used at the time of enrollment, then he finds $M = T \oplus K_m$ and proceeds.

- Client computes $ct_1 \leftarrow Enc(\widetilde{X' \oplus K_x} \odot (M \cdot M'), mpk_1)$ and $ct_2 \leftarrow Enc(M \cdot M', mpk_2)$ and sends these to the server.
- Server takes $y_1 \leftarrow \widetilde{X \oplus K_x}$ and $y_2 \leftarrow 1^n$ and sends to the client to get the functional keys.
- Client generates *functional keys* $\mathbf{sk}_{y_1}$ and $\mathbf{sk}_{y_2}$ using `KeyGen` algorithm and msk_1, msk_2 respectively.
- Server computes $d_1 \leftarrow Dec(ct_1, \mathbf{sk}_{y_1})$ and $d_2 \leftarrow Dec(ct_2, \mathbf{sk}_{y_2})$.
- Compute $NHD = \frac{d_2 - d_1}{2d_2}$
- Let τ be the threshold to decide. then if $NHD < \tau$ then we say that the biometrics *matched* else they do *not match*.

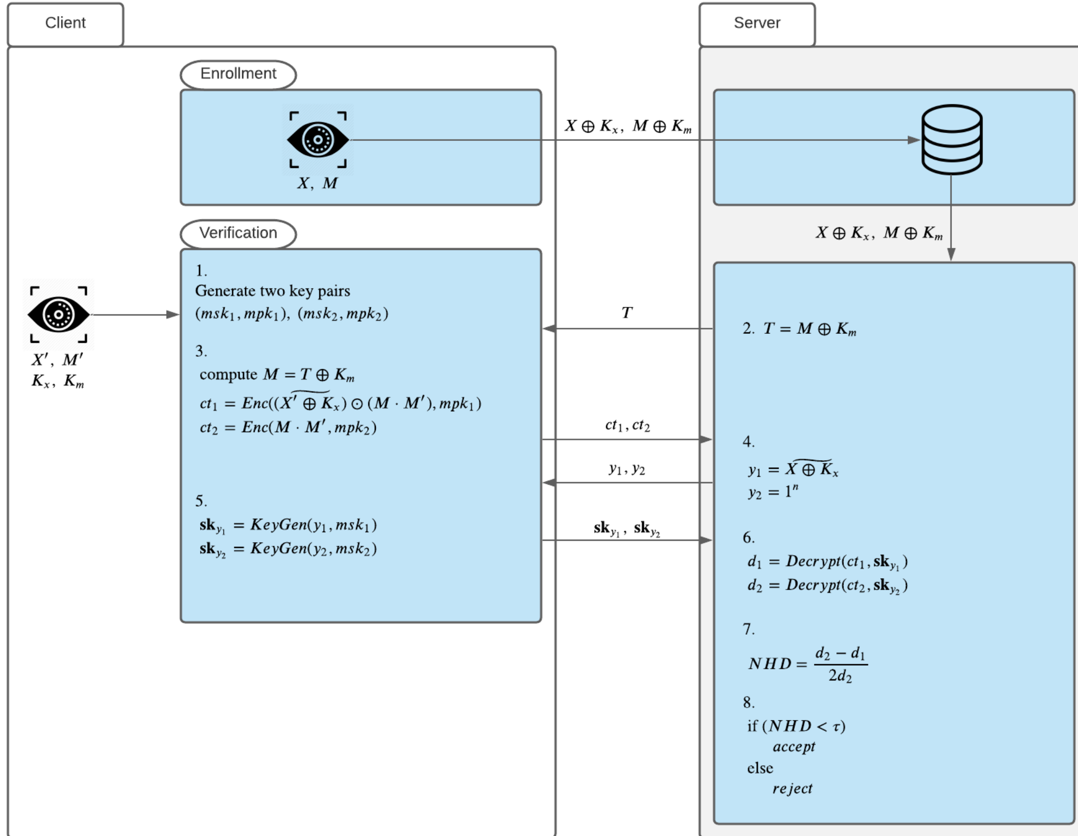


Figure 4.1: *Privacy preserving biometric protocol using IPFE when the data saved in server's database is encrypted*

4.2.2 Correctness

Here we have to verify whether the server has correctly calculated the NHD or not. In the protocol showed in Figure 4.1 we have ciphertexts

$$ct_1 = Enc((\widetilde{X' \oplus K_x}) \odot (M \cdot M'), mpk), \qquad ct_2 = Enc(M \cdot M', mpk)$$

Given $y_1 = \widetilde{X \oplus K_x}$ and $y_2 = 1^n$ we have the functional keys $\mathbf{sk}_{y_1}$ and $\mathbf{sk}_{y_2}$ using `KeyGen` algorithm. After decryption we have two Inner-Products

$$d_1 = \left\langle (\widetilde{X' \oplus K_x}) \odot (M \cdot M'), (\widetilde{X \oplus K_x}) \right\rangle, \qquad d_2 = \langle M \cdot M', 1^n \rangle$$

Clearly, $d_2 = |M \cdot M'|$. We consider, $d_1 = \sum_{i=1}^{n} T_i$ where

$$\begin{aligned} T &= (\widetilde{X' \oplus K_x}) \odot (M \cdot M') \odot (\widetilde{X \oplus K_x}) \\ &= (\widetilde{X \oplus X'}) \odot (M \cdot M') \qquad\qquad \big[\text{ from } (4.1)\big] \end{aligned}$$

So, we have $d_1 = \left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle$.
Again from (4.2) we have $|(X \oplus X') \cdot M \cdot M'| = \frac{d_2 - d_1}{2}$.

Therefore,

$$\begin{aligned} NHD &= \frac{|(X \oplus X') \cdot M \cdot M'|}{|M \cdot M'|} \\ &= \frac{(d_2 - d_1)}{2\ d_2} \end{aligned}$$

Therefore at the end the server can correctly calculate NHD between the biometric saved in the database and the biometric provided by the client at the time of verification.

In the above section, we have seen how we can use IPFE to create a privacy-preserving biometric protocol. In the next section, we create a similar protocol using *Homomorphic Encryption*

4.2.3 Change in parameters of the RLWE based IPFE scheme [29]

As the size of the biometric information is of size 2048, so the parameters of the scheme are changed to support the protocol. As the vectors, x and y both can have negative components so the value of the inner product may range between $-\ell B_x B_y$ and $\ell B_X B_y$. So we take $K > 2\ell B_x B_y$ and the scaling parameter is $\lfloor \frac{q}{K} \rfloor$. There are three sets of parameters that are set depending on the values of ℓ, B_x and B_y. Here we have considered the selectively secure scheme.

As we have seen in Section-2.5.1 for correctness we need $\ell(2n\kappa\sigma_1\sigma_2 + \sqrt{\kappa}\sigma_3)B_y < \lfloor q/2K \rfloor$.

Security Level	FE Bounds	Gaussian Parameters	Ring Parameters	CRT moduli
SEC_LEVEL_0	B_x 1	σ_1 33	n : 2048	q_1 $2^{20} - 2^{14} + 1$
	B_y 1	σ_2 64880641		q_2 $2^{23} - 2^{13} + 1$
	ℓ 2048	σ_3 129761280	$\lceil \log q \rceil$: 74	q_3 $2^{31} - 2^{17} + 1$
SEC_LEVEL_1	B_x 1	σ_1 226	n : 4096	q_1 $2^{24} - 2^{14} + 1$
	B_y 1	σ_2 258376413		q_2 $2^{26} - 2^{16} + 1$
	ℓ 2048	σ_3 516752823	$\lceil \log q \rceil$: 81	q_3 $2^{31} - 2^{24} + 1$
SEC_LEVEL_2	B_x 1	σ_1 2049	n : 8192	q_1 $2^{31} - 2^{17} + 1$
	B_y 1	σ_2 5371330561		q_2 $2^{31} - 2^{19} + 1$
	ℓ 2048	σ_3 10742661120	$\lceil \log q \rceil$: 94	q_3 $2^{32} - 2^{20} + 1$

Table 4.2: The parameters for different security levels

We can check that for this set of parameters this inequality holds.

4.3 HE based privacy preserving biometric protocol

We have used a modified version of HE-based biometric protocol proposed by Kulkarni and Namboodiri [24]. They use somewhat homomorphic encryption (SHE) encryption scheme by Boneh et al. [7], whereas we use The fully homomorphic encryption scheme by Fan and Vercauteren [15] which is known as Brakerski-Fan-Vercauteren (BFV) scheme. To implement the biometric protocol using HE, we use MS Seal library [39]. In this BFV scheme we can use the following evaluation functions to compute on encrypted data.

1. **Addition:** we can perform addition on two encrypted data or perform addition on a plaintext and a ciphertext.

2. **Multiplication:** we can perform multiplication on two encrypted data or perform multiplication on a plaintext and a ciphertext.

3. **Rotation:** Another operation that we can do here is rotation. We can rotate the columns or we can rotate rows.

Using these operations we are implementing a protocol using HE. One more operation that we need to construct is to add all the components of the vector by computing the encrypted data. We would define `add_components` function which operates on ciphertext and works as follows.

We consider the notation $\widehat{d}$ to denote the vector with all components equal to d. If $C = Enc_{pk}(D)$ and $C' =$`add_components`(C), then $C' = Enc_{pk}(D')$ where $D' = \widehat{d}$ is the plaintext with all the components equal to d, the sum of the components of the plaintext D.

We consider the operations $+_h, -_h, \times_h$ to be the operations addition, substraction and multiplication respectively on (*ciphertext, ciphertext*) pair or (*ciphertext, plaintext*) pair.

4.3.1 Protocol

As we can see in Figure 4.2 the biometric information saved in the server's database is encrypted by the client at the time of enrollment. The feature vector and the mask are

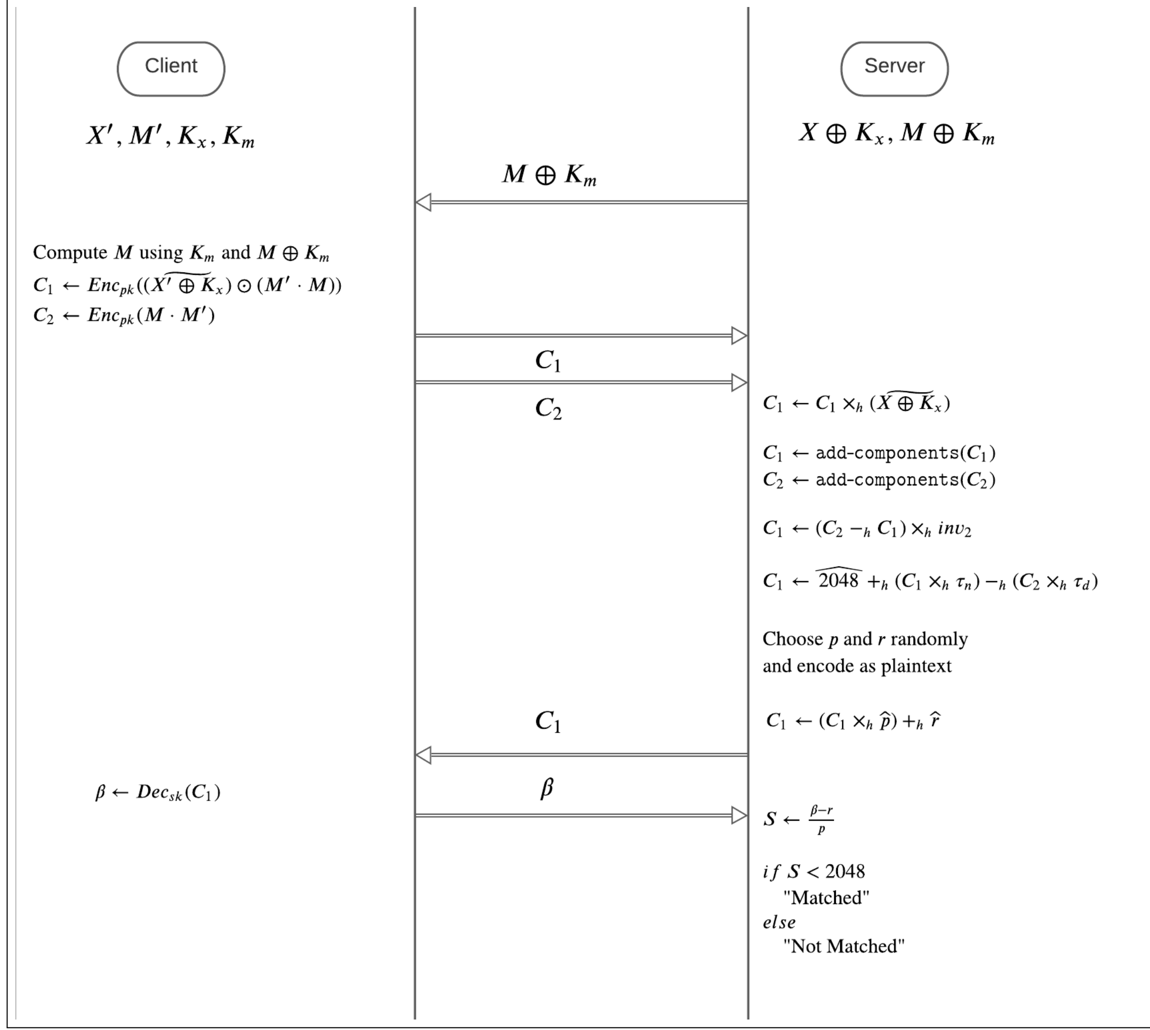


Figure 4.2: Privacy preserving biometric protocol using *Homomorphic Encryption*

encrypted using the keys K_x, K_m respectively. At the time of verification, the client has to hold the keys K_x, K_m to proceed with this protocol.

- First the client generates the public key, secret key pair (pk, sk).
- Server sends the encrypted mask $M \oplus K_m$ to the client. If the client has the valid key K_m then only the client can decrypt and proceed with the rest of the protocol.
- Now client sends the encryptions $C_1 = Enc_{pk}\big((\widetilde{X' \oplus K_x}) \odot (M \cdot M')\big)$ and $C_2 = Enc_{pk}(M \cdot M')$ to the server.
- After receiving the ciphertexts server computes $\big(C_1 \times_h (\widetilde{X \oplus K_x})\big)$ and obtains the encryption $C_3 = Enc_{pk}((\widetilde{X' \oplus X}) \odot (M \cdot M'))$
- Now server performs `add_components` operation on the resulted ciphertexts C_3, C_2 which gives server the encryptions $C = Enc_{pk}(\widehat{d_1})$ and $C' = Enc_{pk}(\widehat{d_2})$ where $d_1 =$

$\left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle$ and $d_2 = \langle M, M' \rangle$

- Now the server calculates $C \leftarrow (C' -_h C) \times_h (\widehat{inv_2})$. So, $C = Enc_{pk}(d_1')$ where $d_1' = \frac{d_2 - d_1}{2}$.
- Let the threshold be $\frac{t_d}{t_n}$ then the server computes $C'' = \widehat{2048} +_h (C \times_h \widehat{t_n}) -_h (C' \times_h \widehat{t_d})$ which is $Enc_{pk}(\widehat{S})$ where, $S = 2048 + (d_1' \times t_n) - (d_2 \times t_d)$.
- Server selects random values p, r and computes $\left((C" \times_h \widehat{p}) +_h \widehat{r}\right)$, which is $Enc_{pk}(\widehat{\beta})$ where $\beta = (p \times S) + r$ and sends it to client.
- The client decrypts the message and sends the value β to server.
- The server computes $S = \frac{\beta - r}{p}$. If $S < 2048$ then the biometrics matchs otherwise it does not match.

4.3.2 Correctness

The server computes $Enc_{pk}(\widehat{d_1})$ and $Enc_{pk}(\widehat{d_2})$ where, $d_1 = \left\langle \widetilde{X \oplus X'}, M \cdot M' \right\rangle$ and $d_2 = \langle M, M' \rangle$. As we have seen before

$$NHD = \frac{d_2 - d_1}{2d_2}$$

The server now computes $d_1' = \frac{d_2 - d_1}{2}$ and $d_2' = d_2$. So, $NHD = \frac{d_1'}{d_2'}$. Now If $\tau = \frac{t_d}{t_n}$ is the threshold then we consider the biometrics are matched if

$$\begin{aligned}
& NHD < \tau \\
i.e., \quad & \frac{d_1'}{d_2'} < \frac{t_d}{t_n} \\
i.e., \quad & d_1' \times t_n < d_2' \times t_d \\
i.e,, \quad & (d_1' \times t_n) - d_2' \times t_d) < 0 \\
i.e,, \quad & 2048 - (d_1' \times t_n) - (d_2' \times t_d) < 2048
\end{aligned}$$

This is why the server computes $Enc_{pk}(\widehat{S})$ where $S = 2048 - (d_1' \times t_n) - (d_2' \times t_d)$. At the end of the protocol server obtains the value S and clearly if $S < 2048$ then the biometrics are considered matched.

Both the protocols using IPFE and HE gives us a privacy-preserving client-server architecture for biometric authentication without any accuracy loss of the matching algorithm. In the next section, we discuss the performance of these protocols.

4.4 Modifications

In both of the above protocols, the client and server try to find the *Normalized Hamming Distance* of the pairs (X, M) and (X', M'). As we have discussed in Section-F.4, the matching algorithm first takes N many cyclic rotations of X' and M', which are the orientations of the biometric data. This involves r many right rotations, $N - r - 1$ many left rotations and one with no rotation. Then the minimum of all the *Normalized Hamming Distance* is taken into consideration to check the two iris data. If we ignore these cyclic rotations then the matching rate decreases [13], which is not desirable.

We make some modifications to both IPFE-based protocol and HE-based protocol that we have proposed here. The modifications are discussed in the following section.

4.4.1 Modifications in IPFE-based protocol

We have an advantage with the protocol based on IPFE. In the IPFE scheme by Mera et al. [29] the message $(x_1, x_2, \cdots x_\ell)$ of size ℓ is encrypted as

$$ct_0 = \mathbf{a}r + f_0$$

$$ct_i = \mathrm{pk}_i r + f_i + \lfloor q/K \rfloor x_i 1_R$$

After decryption we have a message close to $\lfloor q/K \rfloor \langle x, y \rangle 1_R$ and we can extract $\langle x, y \rangle$.

Now consider n many messages $m^{(0)}, m^{(1)}, \cdots, m^{(n-1)} \in \mathbb{Z}^\ell$. We encrypt these messages using the IPFE scheme in a different way. Let's consider the polynomials

$$m_i(x) = \sum_{k=0}^{n} m_i^{(k)} x^k \text{ for all } i \in \{0, 1, \cdots, \ell\}$$

Now we modify the encryption in the following way

$$ct_0 = \mathbf{a}r + f_0$$

$$ct_i = \mathrm{pk}_i r + f_i + \lfloor q/K \rfloor m_i(x)$$

The decryption algorithm remains same and after decryption with functional key $\mathbf{sk}_y$ we

obtain the polynomial

$$\begin{aligned} d &= \sum_{i=0}^{\ell} y_i \cdot m_i(x) + \texttt{noise} \\ &= \sum_{i=0}^{\ell} y_i \left(\sum_{j=0}^{n} m_i^{(j)} x^j \right) + \texttt{noise} \\ &= \sum_{j=0}^{n} \left(\sum_{i=0}^{\ell} y_i \cdot m_i^{(j)} \right) x^j + \texttt{noise} \\ &= \sum_{j=0}^{n} \left\langle y, m^{(j)} \right\rangle x^j + \texttt{noise} \end{aligned}$$

Therefore, the coefficient of x^j in the polynomial d is the inner product of y and the j'th message $m^{(j)}$.

We can consider all the messages $m^{(i)}$ for $i \in [n]$ as a matrix of size $\ell \times n$, the columns of the matrix are the messages. This matrix is encrypted using the modified encryption method for RLWE-based IPFE. After decryption, we the inner products $\left\langle y, m^{(i)} \right\rangle$ for all $i \in [n]$. Thus we can encrypt at most n messages at a time.

The client in the protocol first generates all N many cyclic rotations of X' and M' and for each of these rotations client creates corresponding messages $m^{(i)}, M^{(i)}$ for all $i \in [N]$, if $N < n$ then all other messages can be taken as all 1's. This messages are encrypted to generate two ciphertexts $\mathbf{ct}_1$ and $\mathbf{ct}_2$.

At the server's side, server selects $y_1 = \widetilde{X \oplus K_x}$ and $y_2 = 1^n$ and generates the corresponding functional keys $\mathbf{sk}_{y_1}$, $\mathbf{sk}_{y_2}$ and after decryption the server obtains $d_1^{(i)} = \left\langle y_1, m^{(i)} \right\rangle$ and $d_2^{(i)} = \left\langle y_2, M^{(i)} \right\rangle$ for all $i \in [N]$. Now the value $\min_i \left\{ \frac{d_2^{(i)} - d_1^{(i)}}{2d_2^{(i)}} \right\}$, minimum of all *Normalized Hamming Distances* is used to compare with the threshold.

4.4.2 Modifications in HE-based protocol

The modification in the homomorphic encryption-based protocol the change is straightforward. The Figure 4.3 depicts the protocol.

The client send four encryptions $C_1 = Enc_{pk}(\widetilde{X'} \odot M')$, $C_2 = Enc(M')$, $C_3 = Enc_{pk}(M)$ and $C_4 = Enc_{pk}(\widetilde{K_x})$ to the server.

Server computes $C = C_4 \times_h (\widetilde{X \oplus K_x})$ which is $C = Enc(\widetilde{X})$. Now with the help of C, C_1, C_2 and C_3 the server can now compute all the encryptions of score values S for each cyclic rotations of X', M'.

All these scores are sent to the client after blinding each with randomly chosen integers p_i and r_i. The decryption of all these blinded scores are sent back to the server (as shown in

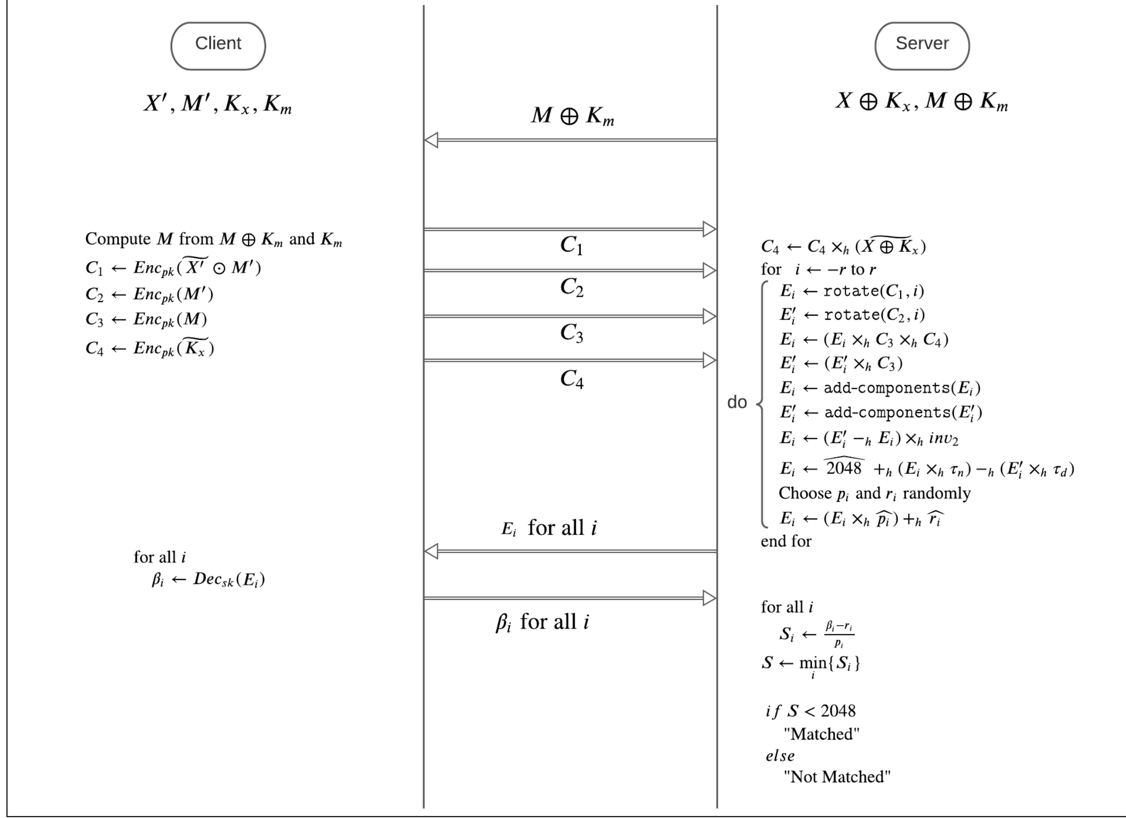


Figure 4.3: Protocol based on HE with $2r + 1$ number of shift operations

Figure 4.3), where the server takes the smallest score to decide if the biometrics are matched or not.

4.5 Experimental results

We use SEAL [39] to implement the protocol using *Homomorphic Encryption*. The protocol using IPFE is implemented using the optimized implementation of the scheme in [29]. As we have seen in the protocol description, in the IPFE based protocol the client needs to send the ciphertexts to the server, then server queries for functional keys corresponding to the vectors y_1 and y_2 and makes a decision without any further interaction with the client but in HE based protocol server have to send a ciphertext back to the client that has to be decrypted, so even after computation by the server we still have to consider the decryption cost by the client for this HE based protocol.

We have used the CASIA Iris database for the biometric data. The algorithm in [13] extracts a feature vector and a mask of 2048 bits.

All the experiments in this section are performed on 4 cores of Intel(R) Core(TM) i5-8265U processor running at 1.60GHz with hyper-threading enabled, on Ubuntu 20.04.2 running on a HP 15-da1041TU laptop. All codes have been compiled using `gcc-9.3.0` with

flags `-O3 -fomit-frame-pointer -march=native`.

In Table 4.3 we have the list of computations that client and server has to do for each of the operations in HE-based protocol (without cyclic rotations).

Operations	Client			Client's Total	Server	Server's Total
	Key Setup	Encryption	Decryption		Computation	
cpucycles	603045572	14540189	2663617	620249378	321598281	321598281

Table 4.3: The *cpucycles* for each of the operations done by client and server for the HE based protocol implemented in *Microsoft SEAL*

In Table 4.4 we have the cpucycles required for the protocol implemented based on IPFE. In this case the client has to setup keys, make encryption and generate functional keys but the server on the other hand only has to decrypt and making a decision. Hence we have the following table for three different levels of security.

cpucycles	Client			Client's Total	Server	Server's Total
	Key Setup	Encryption	KeyGen		Decryption	
SEC_LEVEL_0	625095993	477788560	72779686	1175664239	88219395	88219395
SEC_LEVEL_1	1157076708	1345270046	192166297	2694513051	174886546	174886546
SEC_LEVEL_2	2808825638	3428221626	307432325	6544479589	360046944	360046944

Table 4.4: The *cpucycles* for each of the operations done by client and server in the IPFE based protocol

Clearly, we can see that the performance of *Homomorphic Encryption* for the protocol is much more reasonable. But in reality, the matching algorithm involves a number of cyclic rotations of the feature vector and the mask. Now we see what happens when we use the modified version of the protocol.

Now, in the modified protocol based on HE, the server has to perform rotation on the ciphertext sent by the client and computes on the ciphertexts to obtain a number of computed ciphertexts. These must be sent to the client and all the decryptions must be sent back to the server. So both server and client have to perform extra computation and the communication cost also increases.

The Table 4.5 shows client and the server's cost for different number of cyclic rotations applied.

No. of shifts	Client			Client's Total	Server	Server's Total
	Key Setup	Encryption	Decryption		Computation	
1	591241330	14428934	2668379	608338643	331915119	331915119
3	597766631	14467737	7649680	619884048	870367289	870367289
5	586208777	14459451	12380866	613049094	1390614963	1390614963
9	596517308	14341751	24802356	635661415	2478505242	2478505242
17	630778312	18632230	41762560	691173102	4626583024	4626583024

Table 4.5: The *cpucycles* with the shift operations for the HE based protocol implemented in *Microsoft SEAL*

Clearly the computation in the server's end increases and also the server and client have to send a number of ciphertexts and plaintexts to each other, which also increases the communication cost. As we discussed in (4.4.1) for the modified protocol based on IPFE, the client only sends two ciphertexts and follows the same protocol as before. So the computation cost for the server and the client is the same as Table 4.4.

The tables that we have in this section only reflect how many cpucycles does the operations consume but on the other hand, if we include communication cost then we can see that IPFE-based protocol has less communication cost than the HE-based protocol.

4.6 Conclusion

In this chapter, we have presented a secure privacy-preserving biometric authentication protocol based on IPFE. In the protocol, the client never reveals its biometric information to the server. Even at the time of enrollment the server only gets some encrypted information from the client. And at the time of verification client has to present the decryption keys. Again, the accuracy of the matching algorithm by Daugman [13] depends upon the calculated NHD and here in our privacy-preserving protocol the NHD is correctly calculated. Therefore, the prtocol does not lose any accuracy of the underlying matching algorithm.

Though we have used the RLWE-based IPFE, yet we can use any IPFE scheme for this protocol. We have seen that, in our experimental results initially the HE-based protocol has better performance in terms of computation. But when we modify the protocol to increase the matching rate according to the Section 4.4, the construction of RLWE-based protocol gives us an advantage over the communication cost.

Chapter 5

Conclusion and future work

Here in this chapter we have concluded the work and also summarized the possible direction for future work.

5.1 Conclusion

OpenMP : Due to a large number of polynomials and coefficients in the polynomial operations in the RLWE based schemes, performances are affected. These operations are not complex in terms of computation and also these operations are quite similar to each other only con is that it comes on a large scale. We have used OpenMP to distribute these works among different threads giving us a scalable, fast implementation of the scheme [29] that was our primary target in this thesis. In this work, we have used the Intel Core i5-8265U processor to do all the experiments. The processor only has 4 cores that bound us to 8 threads (2 threads per core) which limits the performance of the implementation. Nowadays processors with more cores are available in the market which can potentially increase the performance of this implementation.

AVX2 optimization : Alongside OpenMP, we have also used AVX2 instructions to optimize the implementation. Polynomial multiplication is one of the most intensive parts of any RLWE-based scheme. The polynomial multiplication method used here is NTT which is the default choice for most of the design choices for RLWE-based schemes. The reason for such a popularity of NTT is that it runs in a quasi-linear time. In spite of being a quasi-linear time algorithm, the large number of polynomial multiplication creates a bottleneck and naturally it affects the performance, with or without NTT. Here we have presented an AVX2 implementation of the NTT multiplication which has indeed increased the performance of the implementation. Since NTT is used widely in different schemes for polynomial multiplication, this work can be extended to any scheme that uses NTT for polynomial multiplication. We have also used incomplete NTT which have shown a slight speed-up for this scheme than the usual full NTT.

Application : In this work, we have also presented a secure privacy-preserving biometric protocol where the client can securely authenticate to the server without revealing their own

biometric information to the server. Even at the time of enrollment the server only stores data that has been encrypted by the client with some symmetric-key cryptosystem. We have also shown how a similar HE-based protocol outperforms the IPFE based protocol at first but IPEE-based protocol does have some advantages in terms of communication overhead when we try to implement the modified protocol to increase the matching rate for the biometric matching algorithm.

5.2 Future work

Large discrete Gaussian sampling Apart from the polynomial multiplication another big challenge for RLWE-based schemes are discrete Gaussian sampling. The errors and secrets are sampled using discrete Gaussian sampling. Just like polynomial multiplication, the large scale of this operation creates a bottleneck degrading the performance. So efficient implementation of this sampler can optimize the performance. We have optimized the NTT multiplication but we have not explored the Gaussian sampler in this work. An efficient implementation of the Gaussian sampler is not only useful for this scheme but also can be useful for most of the RLWE-based schemes.

AVX-512 optimization Our current implementation is optimized with AVX2 instructions. The AVX-512 instructions support more instructions and the width of the registers are double the size of AVX2. So we can pack more coefficients. Also, we can get 32 registers instead of 16 in AVX2. So optimizing with AVX-512 can have a huge impact on the performance. Although it is not as widely available as AVX2 or AVX it's only a matter of time.

Appendix A

Assembly code for first three levels of NTT

Here is the macro defined for the first three levels of NTT transformation with `off` as the offset. q , $q^{-1} \mod 2^{32}$ are already packed in `ymm0` and `ymm1` respectively. `ymm15` is used to load the twiddle factors.

The way these first three levels are done the same is applied to all consecutive three levels except the final three levels.

```
.macro level0_2 off

#level 0
vmovdqa      (8*\off+1024)*4(%rdi),%ymm7
vmovdqa      (8*\off+1280)*4(%rdi),%ymm8
vmovdqa      (8*\off+1536)*4(%rdi),%ymm9
vmovdqa      (8*\off+1792)*4(%rdi),%ymm10

vpbroadcastd     (%rdx),%ymm15

mul          15, 7
mont         7
mul          15, 8
mont         8
mul          15, 9
mont         9
mul          15, 10
mont         10

vmovdqa      (8*\off+0)*4(%rdi),%ymm3
vmovdqa      (8*\off+256)*4(%rdi),%ymm4
vmovdqa      (8*\off+512)*4(%rdi),%ymm5
vmovdqa      (8*\off+768)*4(%rdi),%ymm6

update  2, 3, 4, 5, 6, 7, 8, 9, 10

#level 1
vpbroadcastd     4(%rdx),%ymm15

```

```
mul          15, 4
mont         4
mul          15, 5
mont         5

vpbroadcastd    8(%rdx),%ymm15

mul          15, 9
mont         9
mul          15, 10
mont         10

update  6, 2, 3, 7, 8, 4, 5, 9, 10

#level 2
vpbroadcastd    12(%rdx),%ymm15
mul          15, 2
mont         2

vpbroadcastd    16(%rdx),%ymm15
mul          15, 5
mont         5

vpbroadcastd    20(%rdx),%ymm15
mul          15, 7
mont         7

vpbroadcastd    24(%rdx),%ymm15
mul          15, 10
mont         10

update  8, 6, 4, 3, 9, 2, 5, 7, 10

vmovdqa     %ymm8, (8*\off+0)*4(%rdi)
vmovdqa     %ymm2, (8*\off+256)*4(%rdi)
vmovdqa     %ymm6, (8*\off+512)*4(%rdi)
vmovdqa     %ymm5, (8*\off+768)*4(%rdi)
vmovdqa     %ymm4, (8*\off+1024+0)*4(%rdi)
vmovdqa     %ymm7, (8*\off+1024+256)*4(%rdi)
vmovdqa     %ymm3, (8*\off+1024+512)*4(%rdi)
vmovdqa     %ymm10,(8*\off+1024+768)*4(%rdi)

.endm
```

Appendix B

Assembly code last two levels of incomplete-NTT

Although the actual implementation performs the last 5 levels for incomplete-NTT (6 levels for full-NTT) in one go but for the simplicity we have only presented the code for last 2 levels previous 3 levels are done exactly similar as `level0_2` in Appendix A.

```
.macro final2_level

vmovdqa     (%rdi),%ymm8
vmovdqa     32(%rdi),%ymm2
vmovdqa     64(%rdi),%ymm6
vmovdqa     96(%rdi),%ymm5
vmovdqa     128(%rdi),%ymm4
vmovdqa     160(%rdi),%ymm7
vmovdqa     192(%rdi),%ymm3
vmovdqa     224(%rdi),%ymm10

shuffle4    8, 2, 9, 2
shuffle4    6, 5, 8, 5
shuffle4    4, 7, 6, 7
shuffle4    3,10, 4,10

vmovdqa     (%rdx), %ymm15
mul 15, 2
mont 2

vmovdqa     32(%rdx), %ymm15
mul 15, 5
mont 5

vmovdqa     64(%rdx), %ymm15
mul 15, 7
mont 7

vmovdqa     96(%rdx), %ymm15
mul 15, 10
mont 10

```

```
33 update  3, 9, 8, 6, 4, 2, 5, 7, 10
34
35 shuffle4    3, 2, 4, 2
36 shuffle4    9, 5, 3, 5
37 shuffle4    8, 7, 9, 7
38 shuffle4    6,10, 8,10
39
40 add     $128, %rdx
41
42 shuffle2    4, 2, 6, 2
43 shuffle2    3, 5, 4, 5
44 shuffle2    9, 7, 3, 7
45 shuffle2    8,10, 9,10
46
47 vmovdqa     (%rdx), %ymm15
48 mul 15, 2
49 mont 2
50
51 vmovdqa     32(%rdx), %ymm15
52 mul 15, 5
53 mont 5
54
55 vmovdqa     64(%rdx), %ymm15
56 mul 15, 7
57 mont 7
58
59 vmovdqa     96(%rdx), %ymm15
60 mul 15, 10
61 mont 10
62
63 update  8, 6, 4, 3, 9, 2, 5, 7, 10
64
65 shuffle2    8, 2, 9, 2
66 shuffle2    6, 5, 8, 5
67 shuffle2    4, 7, 6, 7
68 shuffle2    3,10, 4,10
69
70 add     $128, %rdx
71
72 vmovdqa     %ymm9, (%rdi)
73 vmovdqa     %ymm2, 32(%rdi)
74 vmovdqa     %ymm8, 64(%rdi)
75 vmovdqa     %ymm5, 96(%rdi)
76 vmovdqa     %ymm6, 128(%rdi)
77 vmovdqa     %ymm7, 160(%rdi)
78 vmovdqa     %ymm4, 192(%rdi)
79 vmovdqa     %ymm10,224(%rdi)
80
81 add     $256, %rdi
82 .endm
```

Appendix C

Permutations π_1, π_2 and π_4

The permutaions π_1, π_2 and π_4 are implemented as `shuffle1`, `shuffle2` and `shuffle4`. The codes are taken from the implementation of Kyber [1].

```
.macro shuffle1 r0,r1,r2,r3

vmovsldup   %ymm\r1,%ymm\r2
vpblendd    $0xAA,%ymm\r2,%ymm\r0,%ymm\r2
vpsrlq      $32,%ymm\r0,%ymm\r0
vpblendd    $0xAA,%ymm\r1,%ymm\r0,%ymm\r3

.endm
```

```
.macro shuffle2 r0,r1,r2,r3

vpunpcklqdq %ymm\r1,%ymm\r0,%ymm\r2
vpunpckhqdq %ymm\r1,%ymm\r0,%ymm\r3

.endm
```

```
.macro shuffle4 r0,r1,r2,r3

vperm2i128  $0x20,%ymm\r1,%ymm\r0,%ymm\r2
vperm2i128  $0x31,%ymm\r1,%ymm\r0,%ymm\r3

.endm
```

[1] https://github.com/pq-crystals/kyber

Appendix D

Assembly code for multiplication and Montgomery reduction

For multiplication in AVX2, we use the following code. Here two packed 32-bit coefficients in registers indexed by `r0, r1` are multiplied together and the low 32-bits of multiplication are stored in register indexed by `rl` and high 32 bits of multiplication are stored in register indexed by `rh`. By default these `rh` and `rl` are set to 13 and 12 respectively.

```
.macro mul r0, r1, rl=13, rh=12 , e1=11

vpmuludq    %ymm\r0, %ymm\r1, %ymm\rh
vmovshdup   %ymm\rh, %ymm\rh
vmovshdup   %ymm\r1, %ymm\e1
vmovshdup   %ymm\r0, %ymm\rl
vpmuludq    %ymm\rl, %ymm\e1, %ymm\e1
vmovshdup   %ymm\e1, %ymm\e1
vpblendd    $0xAA, %ymm\e1, %ymm\rh, %ymm\rh
vpmulld     %ymm\r0, %ymm\r1, %ymm\rl

.endm
```

Below is the code for Montgomery reduction in AVX2. After multiplication the result must be reduced. We use the packed high and low 32-bit of the multiplication in `ymm12` and `ymm13` respectively. Also the constants q and $q^{-1} \mod 2^{32}$ are packed in `ymm0` and `ymm1` respectively.

```
.macro mont r, q=0, qi=1

vpmulld     %ymm13, %ymm\qi, %ymm13
vpmuludq    %ymm13, %ymm\q, %ymm\r
vmovshdup   %ymm\r, %ymm\r
vmovshdup   %ymm13, %ymm13
vpmuludq    %ymm13, %ymm\q, %ymm13
vpblendd    $0xAA, %ymm13, %ymm\r, %ymm\r
vpsubq      %ymm\r, %ymm12, %ymm\r
vpsrad      $31, %ymm\r, %ymm13
vpand       %ymm13, %ymm\q, %ymm13
vpaddq      %ymm\r, %ymm13, %ymm\r

.endm
```

Appendix E

Assembly code for 2×2 base multiplication for incomplete-NTT

The following is the AVX2 code for 2×2 base multiplication of two degree one polynomials of the form $f(x) = a_0 + a_1 x$ and $g(x) = b_0 + b_1 x$. Here we are packing eight pairs such polynomial in four registers indexed by `l0, l1, r0, r1`.

If the eight polynomials are of the form $f^{(i)}(x) = a_0^{(i)} + a_1^{(i)} x$ and $g^{(i)}(x) = b_0^{(i)} + b_1^{(i)} x$ for $0 \leq i < 8$, then the coefficients $a_0^{(i)}$, $a_1^{(i)}$, $b_0^{(i)}$ and $b_1^{(i)}$ for $0 \leq i < 8$ are packed in registers indexed by `l0, l1, r0` and `r1` respectively.

The final resultant polynomials are packed in two registers. The low coefficients and the high coefficients of the resultant polynomials and packed in registers indexed by `l0` and `f` respectively.

```
.macro multupdate l0, l1, r0, r1, f

vpaddq      %ymm\l0, %ymm\l1, %ymm\f
vpsubq      %ymm0, %ymm\f, %ymm\f
vpsrad      $31, %ymm\f, %ymm11
vpand       %ymm11, %ymm0, %ymm11
vpaddq      %ymm11, %ymm\f, %ymm\f

mul     15, \l0
mont    \l0
mul     15, \l1
mont    \l1
mul     15, \f
mont    \f

mul     \l0, \r0
mont    \l0
mul     \l1, \r1
mont    \l1

vpaddq      %ymm\r0, %ymm\r1, %ymm\r0
vpsubq      %ymm0, %ymm\r0, %ymm\r0
vpsrad      $31, %ymm\r0, %ymm11
```

```
24 vpand        %ymm11, %ymm0, %ymm11
25 vpaddq       %ymm11, %ymm\r0, %ymm\r0
26
27 mul      \f, \r0
28 mont     \f
29
30 vpaddq       %ymm\l0, %ymm\l1, %ymm\r0
31 vpsubq       %ymm0, %ymm\r0, %ymm\r0
32 vpsrad       $31, %ymm\r0, %ymm11
33 vpand        %ymm11, %ymm0, %ymm11
34 vpaddq       %ymm11, %ymm\r0, %ymm\r0
35
36
37 vpsubq       %ymm\r0, %ymm\f, %ymm\f
38 vpsrad       $31, %ymm\f, %ymm11
39 vpand        %ymm11, %ymm0, %ymm11
40 vpaddq       %ymm11, %ymm\f, %ymm\f
41
42 vmovdqa      (%rcx), %ymm\r1
43 add          $32, %rcx
44
45 mul      \r1, \l1
46 mont     \l1
47
48 vpaddq       %ymm\l0, %ymm\l1, %ymm\l0
49 vpsubq       %ymm0, %ymm\l0, %ymm\l0
50 vpsrad       $31, %ymm\l0, %ymm11
51 vpand        %ymm11, %ymm0, %ymm11
52 vpaddq       %ymm11, %ymm\l0, %ymm\l0
53
54 .endm
```

Appendix F

Iris matching algorithm [13, 43]

The iris matching algorithm involves an image of the eye and then extracting iris data from the image to use that data for matching the biometric data. Therefore, first, we need to find the iris in a given image.

F.1 Localizing and isolating iris

First, the target is to find the iris boundaries both outer and inner boundary, also we have to find the centre of the pupil, The algorithm by [13] make use of the first derivatives of image intensity to signal the location of edges that correspond to the borders of the iris. Here, the notion is that the magnitude of the derivative across an imaged border will show a local maximum due to the local change of image intensity.

Let I represent the iris image and $I(x, y)$ be the intensity of the image at the location (x, y). Let the circular boundaries be parameterized by center location (x_c, y_c) and radius r. The algorithm fits the circular contours via gradient ascent on the parameters (x_c, y_c, r) so as to maximize

$$\left| G_\sigma(r) * \frac{\partial}{\partial r} \oint_{r,x_c,y_c} \frac{I(x,y)}{2\pi r} ds \right| \tag{F.1}$$

Here $G(r) = \frac{1}{\sqrt{2\pi r}} e^{-\frac{(r-r_0)^2}{2\sigma^2}}$ is a radial Gaussian with center r_0 and standard deviation σ that smooths the image to select the spatial scale of edges under consideration, $*$ symbolizes convolution, ds is an element of circular arc, and division by $2\pi r$ serves to normalize the integral.

The Equation F.1 serves to find both the pupillary boundary and the outer (limbus) boundary of the iris. After this step, the centre along with radius (x_c, y_c, r_0), the outer and inner boundaries of the iris are found. In the next step, this iris data are to be transformed into binary data.

F.2 Iris feature encoding by 2D wavelet demodulation

Each isolated iris pattern is then demodulated to extract its phase information using quadrature 2D Gabor wavelets (See [14]). First of all, the coordinates (x, y) are transformed into polar coordinates (ρ, ϕ) with respect to the parameters (x_c, y_c, r_0). Now we can denote the intensities of the image as $I(\rho, \phi)$ at (ρ, ϕ) polar coordinate. Now, this data is projected onto a complex plane with the complex-valued Gabor wavelets and the bits are extracted according to which quadrant it belongs to.

$$h_{\{Re,Im\}} = \text{sgn}_{\{Re,Im\}}\left(\int_{\rho}\int_{\phi} I(\rho, \phi) e^{-i\omega(\theta-\phi)} e^{-(r-\rho)^2/\alpha^2} e^{-(\theta-\phi)^2/\beta^2} \rho \, \mathrm{d}\rho \, \mathrm{d}\phi \right) \tag{F.2}$$

Here the $\text{sgn}_{\{Re,Im\}}$ function works as follows for any complex value z

$$\text{sgn}_{\{Re,Im\}}(z) = \begin{cases} (0,0) & \text{, if } z \text{ is in first quadrant} \\ (0,1) & \text{, if } z \text{ is in second quadrant} \\ (1,0) & \text{, if } z \text{ is in third quadrant} \\ (1,1) & \text{, if } z \text{ is in fourth quadrant} \end{cases}$$

α and β are the multi-scale 2D wavelet size parameters, spanning an 8-fold range from 0.15 to 1.2 mm on the iris. ω is wavelet frequency, spanning three octaves in inverse proportion to β and (r, θ) represents the polar coordinates of each region of iris for which $h_{\{Re,Im\}}$ is computed. The value $h_{\{Re,Im\}}$ is two bis of data, this are called phase bits. Altogether 2048 such phase bits (256 bytes) are computed for each iris and an equal number of masking bits are also computed to signify whether any iris region is obscured by eyelids, contains any eyelash occlusions, specular reflections, boundary artifacts of hard contact lenses, or poor signal-to-noise ratio and thus should be ignored in the demodulation code as artifact.

F.3 Matching

The key to iris recognition is the failure of a test of statistical independence, This test of independence is determined by a computation of *normalized Hamming distance* or *fractional Hamming distance*, for two irises whose two-phase code bit vectors are denoted $\{C_A, C_B\}$ and whose mask bit vectors are denoted $\{M_A, M_B\}$ the *normalized Hamming distance* is defined by :

$$NHD = \frac{\left|(C_A \oplus C_B) \cdot M_A \cdot M_B\right|}{\left|M_A \cdot M_B\right|}$$

The threshold τ thas to be selected that will decide whether the two iris data are statistically independent or not. The method to select τ has been discussed in detail in [13].

F.4 Rotations

Robust representations for pattern recognition must be invariant under transformations in the size, position, and orientation of the patterns. Now one naive way for orientation is to rotate the iris image and compute as we discussed in previous sections. But it is better that we first transform the iris data to polar coordinate like we did in Section F.2, and now orientation is easy by making changes in the angle of the polar coordinate. Now when calculating $h_{\{Re,Im\}}$ in Section F.2, one thing that we can observe is that if the whole image was rotated with respect to the centre (x_c, y_c) then the values of the phase bits will same as before with some cyclic scrolling of its angular variable. Therefore, instead of rotating the image now, we can make a cyclic rotation on the 2048 phase bits.

Since now can calculate different orientations of iris data, now we will calculate all NHD with n orientations in one iris and the "*best of n*" is taken as the relative Hamming distance between the two iris data.

Bibliography

[1] Openmp api specification: Version 5.1. URL https://www.openmp.org/spec-html/5.1/openmp.html.

[2] Openmp, Jun 2021. URL https://en.wikipedia.org/wiki/OpenMP.

[3] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In Jonathan Katz, editor, *Public-Key Cryptography – PKC 2015*, pages 733–751, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46447-2.

[4] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 333–362, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53015-3.

[5] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. ISBN 978-3-540-47721-1.

[6] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 213–229, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44647-7.

[7] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30576-7.

[8] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. https://eprint.iacr.org/2017/634.

[9] Florian Bourse. *Functional Encryption for Inner-Product Evaluations*. PhD thesis, 12 2017.

[10] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings. Cryptology ePrint Archive, Report 2020/1397.

[11] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. ISSN 00255718, 10886842. URL http://www.jstor.org/stable/2003354.

[12] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: 10.1109/99.660313.

[13] J. Daugman. How iris recognition works. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):21–30, 2004. doi: 10.1109/TCSVT.2003.818350.

[14] J.G. Daugman. Complete discrete 2-d gabor transforms by neural networks for image analysis and compression. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(7):1169–1179, 1988. doi: 10.1109/29.1644.

[15] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[16] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6/7):467–488, 1982.

[17] W. M. Gentleman and G. Sande. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), page 563–578, New York, NY, USA, 1966. Association for Computing Machinery. ISBN 9781450378932. doi: 10.1145/1464291.1464352. URL https://doi.org/10.1145/1464291.1464352.

[18] Craig Gidney and Martin Ekeraa. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, April 2021. ISSN 2521-327X. doi: 10.22331/q-2021-04-15-433. URL https://doi.org/10.22331/q-2021-04-15-433.

[19] Oded Goldreich, Daniele Micciancio, S. Safra, and J. Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors.

[20] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, London, GBR, 2015. ISBN 9789888381968.

[21] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*.

[22] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 146–162, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78967-3.

[23] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J. Wu. Function-hiding inner product encryption is practical. Cryptology ePrint Archive, Report 2016/440, 2016. https://eprint.iacr.org/2016/440.

[24] Rohan Kulkarni and Anoop Namboodiri. Secure hamming distance based biometric authentication. In *2013 International Conference on Biometrics (ICB)*, pages 1–6, 2013. doi: 10.1109/ICB.2013.6613008.

[25] Yongnan Li, Limin Xiao, Aihua Liang, Yao Zheng, and Li Ruan. Fast parallel garner algorithm for chinese remainder theorem. In James J. Park, Albert Zomaya, Sang-Soo Yeo, and Sartaj Sahni, editors, *Network and Parallel Computing*, pages 164–171, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35606-3.

[26] Vadim Lyubashevsky and Gregor Seiler. Nttru: Truly fast ntru using ntt. Cryptology ePrint Archive, Report 2019/040.

[27] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13190-5.

[28] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. https://eprint.iacr.org/2012/230.

[29] Jose Maria Bermudo Mera, Angshuman Karmakar, Tilen Marc, and Azam Soleimanian. Efficient lattice-based inner-product functional encryption. Cryptology ePrint Archive, Report 2021/046.

[30] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. ISBN 978-3-540-39799-1.

[31] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[32] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Info. Comput.*, 3(4):317–344, July 2003. ISSN 1533-7146.

[33] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56:34:1–34:40, 2009.

[34] Oded Regev. Lecture notes: Lattices in computer science. URL https://cims.nyu.edu/~regev/teaching/lattices_fall_2009/.

[35] Oded Regev. New lattice-based cryptographic constructions. *J. ACM*, 51(6):899–942, November 2004. ISSN 0004-5411. doi: 10.1145/1039488.1039490. URL https://doi.org/10.1145/1039488.1039490.

[36] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139608. doi: 10.1145/1060590.1060603. URL https://doi.org/10.1145/1060590.1060603.

[37] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. ISSN 0001-0782. doi: 10.1145/359340.359342. URL https://doi.org/10.1145/359340.359342.

[38] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 457–473, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32055-5.

[39] SEAL. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[40] Gregor Seiler. Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039.

[41] Adi Shamir. Identity-based cryptosystems and signature schemes. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 47–53, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39568-3.

[42] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

[43] R.P. Wildes. Iris recognition: an emerging biometric technology. *Proceedings of the IEEE*, 85(9):1348–1363, 1997. doi: 10.1109/5.628669.