

Performance Analysis of FHE Libraries

Master's Thesis
in Partial Fulfillment of the Requirements for the
Degree of
Master of Science

by
NISHA PREMANAND KALE

submitted to:
Prof. Dr. Johannes Blömer
and
Prof. Dr. Somorovsky

Paderborn, March 17, 2021

Eidesstattliche Versicherung

Nachname: Kale Vorname: Nisha Premanand

Matrikelnr.: 6845740 Studiengang: Computer Engineering

☐ Bachelorarbeit ☒ Masterarbeit

Titel der Arbeit: Performance Analysis of FHE Libraries

☒ Die elektronische Fassung ist der Abschlussarbeit beigelegt.

☒ Die elektronische Fassung sende ich an die/den erste/n Prüfenden bzw. habe ich an die/den erste/n Prüfenden gesendet.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit (Ausarbeitung inkl. Tabellen, Zeichnungen, etc.) selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Abschlussarbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung entspricht der gedruckten und gebundenen Fassung.


Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist die Vizepräsidentin / der Vizepräsident für Wirtschafts- und Personalverwaltung der Universität Paderborn. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz NRW in der aktuellen Fassung).

Die Universität Paderborn wird ggf. eine elektronische Überprüfung der Abschlussarbeit durchführen, um eine Täuschung festzustellen.

Ich habe die oben genannten Belehrungen gelesen und verstanden und bestätige dieses mit meiner Unterschrift.

Ort: Paderborn Datum: 3/17/2021

Unterschrift: 

Datenschutzhinweis

Die o.g. Daten werden aufgrund der geltenden Prüfungsordnung (Paragraph zur Abschlussarbeit) i.V.m. § 63 Abs. 5 Hochschulgesetz NRW erhoben. Auf Grundlage der übermittelten Daten (Name, Vorname, Matrikelnummer, Studiengang, Art und Thema der Abschlussarbeit) wird bei Plagiaten bzw. Täuschung der/die Prüfende und der Prüfungsausschuss Ihres Studienganges über Konsequenzen gemäß Prüfungsordnung i.V.m. Hochschulgesetz NRW entscheiden. Die Daten werden nach Abschluss des Prüfungsverfahrens gelöscht. Eine Weiterleitung der Daten kann an die/den Prüfende/n und den Prüfungsausschuss erfolgen. Falls der Prüfungsausschuss entscheidet, eine Geldbuße zu verhängen, werden die Daten an die Vizepräsidentin für Wirtschafts- und Personalverwaltung weitergeleitet. Verantwortlich für die Verarbeitung im regulären Verfahren ist der Prüfungsausschuss Ihres Studienganges der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.

Notations

\mathbb{Z}	Set of integers
\mathbb{N}	Set of Natural numbers
\mathbb{R}^+	Set of positive real numbers
R	Ring
$R[X]$	Polynomial ring in variable X
$R = \mathbb{Z}[x]/f(x)$	Polynomial ring in variable x , coefficients belong to \mathbb{Z}
$\mathbb{Z}_q = [-q/2, q/2]$	Finite set of integers within range $[-q/2, q/2]$ obtained by taking modulo q for all the integers
R_q	Polynomial ring in variable x , coefficients belong to \mathbb{Z}_q
$\phi_m(x)$	cyclotomic polynomial
$\mathcal{N}(\mu, \sigma^2)$	Gaussian distribution(Normal distribution) with mean μ and variance σ^2
λ	Security parameter
1^λ	value that can be represented with λ ones
$ \text{sk} , \text{pk} $	represents length of sk and pk
$m \in (0, 1)^*$	denotes m can be any value that can be represented in a binary form with arbitrary bit-length
$(\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda)$	denotes the process of obtaining (sk, pk) from a probabilistic algorithm KeyGen on input (1^λ)

$\Pr[\mathbf{m}' \neq \mathbf{m}]$	denotes the probability of \mathbf{m}' not equal to \mathbf{m}
$D_{\mathbb{Z},\sigma}^d$	Discrete Gaussian distribution defined over d-dimensional vector in integers with σ standard deviation
$erf(\beta)/\sqrt{2}$	It is a Gauss error function that is defined as $erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$ where $z = \beta/\sqrt{2}$
$a \leftarrow R_q$	select a uniformly at random from R_q
$\ \mathbf{a}\ $	For $\mathbf{a} \in R$, $\mathbf{a} = \sum_{i=0}^{d-1} a_i \cdot x^i$, infinity norm of ring element \mathbf{a} is defined as $\max_i a_i $
δ_R	Expansion factor of ring R given as $\max\{\ a \cdot b\ /\ a\ \cdot \ b\ : a, b \in R_q\}$
$e \leftarrow \chi$	choose $e \in R$ via distribution χ
$\lfloor x \rfloor$	rounding the x value to closest lower integer
$\lceil x \rceil$	rounding the x value to closest higher integer
$\lfloor x \rfloor$	rounding x value to closest integer
$a \cdot b$	multiply a with b
$[x]_q$	x modulo q operation
$\langle a, b \rangle$	inner product of polynomials a and b

Contents

1	Introduction	1
2	Fundamentals	3
2.1	Ring	3
2.2	Monic Irreducible Polynomial	3
2.3	Cyclotomic Polynomial	4
2.4	Polynomial Ring	4
2.5	Discrete Gaussian Distribution	4
2.6	LWE Problem	4
2.7	RLWE Problem	5
2.7.1	Chi Distribution	5
2.7.2	RLWE Distribution	6
2.7.3	Search-RLWE	6
2.7.4	Decision-RLWE	6
2.8	Public Key Encryption Scheme	6
2.8.1	Negligible Function	7
2.8.2	Definition	7
2.8.3	Adversary Attacks	7
2.8.4	Properties Of Public Key Encryption Scheme	8
2.9	Approximate Arithmetic	8
2.9.1	Floating-point representation	9
2.9.2	Types of approximate arithmetic	9
3	Homomorphic Encryption Schemes	11
3.1	Evaluation Circuit	11
3.2	Homomorphic Encryption Correctness	11
3.3	Types Of Homomorphic Encryption Scheme	12
3.4	Circular Security	13
3.5	Bootstrapping technique	13
3.6	Squashing decryption circuit	14
3.7	Modulus Switching:	15
4	BFV Scheme	17
4.1	Encryption Scheme	17
4.1.1	Decryption Correctness	18
4.1.2	Semantic security	20

4.2	Somewhat Homomorphic Encryption	20
4.2.1	Addition:	20
4.2.2	Multiplication	22
4.2.3	Relinearisation	22
4.2.4	BFV SHE scheme	24
4.2.5	Maximum depth of a SHE multiplication circuit	25
4.3	Fully Homomorphic Encryption	26
5	CKKS Scheme	29
5.1	Encoding and Decoding	29
5.2	Encryption	29
5.3	Decryption	30
5.4	Rescaling	30
5.5	Probability Distributions	31
5.6	Encryption Scheme	31
6	Microsoft SEAL	33
6.1	Encryption Parameters	33
6.2	Context Data	34
6.3	Number Theoretic Transform(NTT)	34
6.4	Encryption	35
6.4.1	Probability distributions	35
6.4.2	Keys	36
6.4.3	Ciphertext	36
6.4.4	Ciphertext structure	36
6.5	Evaluator	37
6.6	BFV Implementation	37
6.6.1	Noise budget	38
6.6.2	BFV Decryption	38
6.6.3	Relinearization	38
6.7	CKKS implementation	38
6.7.1	Rescaling	39
6.7.2	Modulus switching	39
6.7.3	Normalisation of ciphertext scale:	39
6.8	Advantages And Disadvantages	39
6.8.1	BFV scheme	39
6.8.2	CKKS scheme	40
7	Performance Evaluation	43
7.1	Experimental Setup	43
7.2	Evaluation Criteria	44
7.3	Test Functions	44
7.4	General Results	44
7.4.1	Storage space	44

7.4.2	Computation time	46
7.5	Sigmoid function	47
7.5.1	Test setup and results	47
7.6	Trained deep neural network evaluation	48
7.6.1	Test setup and results	49
7.7	Sign function lookup table	50
7.7.1	Test setup and results	51
7.7.2	Remarks	53
7.8	Trained discrete neural network	53
7.9	Discussion	54
8	Conclusion and Future Work	57
	Bibliography	59

List of Figures

3.1	Evaluation circuit	12
7.1	Public key size	45
7.2	Secret key size	46
7.3	Ciphertext size	47
7.4	Plaintext size	48
7.5	Comparison between security levels	49
7.6	Comparison between schemes	50

1 Introduction

Homomorphic encryption is a form of encryption that allows performing computations on encrypted data similar to performing computations on plain data. Nowadays, most of the confidential data is being stored on the cloud to outsource the costly computational tasks to be performed on this data to third-party service providers. With the increase in the applications of artificial intelligence in medical, finance, and other sectors, which involve dealing with confidential user data homomorphic encryption schemes provide the necessary tools to ensure security and privacy while still being able to perform costly computations on the data. The idea of homomorphic encryption schemes first came into existence when RSA was found to be multiplicatively homomorphic when it was invented[RAD78]. Since then many schemes have been proposed that have exhibited more properties of homomorphic schemes in terms of their ability to evaluate complex mathematical functions with greater performance and increased practical applications.

The main goal of this master thesis is to give an overview of different Homomorphic encryption schemes that have been implemented in Microsoft SEAL and discuss their practical applications by evaluating their performance for some selected complex mathematical functions. The primary focus of this thesis is to give an overall idea on the practical aspects of the chosen software library(Microsoft SEAL) with some explanation on the theoretical aspects of the schemes that help in better understanding of our evaluation results.

The structure of this thesis is as follows: In Chapter 2 we briefly discuss the mathematical concepts in our homomorphic encryption schemes like Polynomial rings, RLWE problem, approximate arithmetic, and basic concepts in Public Key encryption schemes. In Chapter 3 we explain some concepts in Homomorphic encryption schemes. In Chapter 4 we explain the first encryption scheme implemented in the SEAL library, the BFV scheme. We define its two versions- Somewhat Homomorphic Encryption scheme and Fully Homomorphic Encryption scheme and discuss in detail its decryption correctness property and Relinearisation methods. In Chapter 5 we discuss the second encryption scheme implemented in the SEAL library, the CKKS scheme, and explain briefly the concepts in this scheme. Chapter 6 outlines the various methods available in the SEAL library for both BFV and CKKS schemes to implement the algorithms defined in the encryption schemes in Chapter 4 and Chapter 5. In Chapter 7 we define the mathematical functions evaluated by the library, the metrics for our performance testing, discuss the results of the evaluation, and briefly explain the issues faced during the course of work and their solutions implemented.

2 Fundamentals

In this chapter we give the definitions of fundamental structures and concepts used in the encryption schemes implemented in Microsoft SEAL library.

2.1 Ring

A ring R is a non-empty set in which we can perform binary operations addition(+) and multiplication(\cdot) that satisfy the below properties,

1. $\forall (a, b, c) \in R$ the $+$ operation on Ring elements
 - is associative, if $(a + b) + c = a + (b + c)$
 - is commutative, if $(a + b) = (b + a)$
 - has an additive identity '0', such that $a + 0 = a$
 - has an additive inverse $-a$ for each a such that $a + (-a) = 0$
2. $\forall (a, b, c) \in R$ the \cdot operation on Ring elements
 - is associative, if $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
 - has multiplicative identity '1' such that $a \cdot 1 = 1 \cdot a = a$
 - is left distributive with respect to $+$, if $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
 - is right distributive with respect to $+$, if $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$

Ring can be defined on a set of integers, rational numbers, complex values, polynomials etc. Ring structures have application in various areas of cryptography such as public-key encryption schemes, private-key encryption schemes, digital signatures etc.

2.2 Monic Irreducible Polynomial

A polynomial that cannot be reduced into the product of 2 polynomial factors and the coefficient of the term with the highest degree is equal to 1 is referred to as monic irreducible polynomial. For example, a polynomial in variable x with the highest degree n can be written as : $x^n + c_{n-1}x^{n-1} + \dots + c_2x^2 + c_1x + c_0$ where $c_0, c_1, \dots, c_{n-1}, c_n$ are the coefficients with $c_n = 1$.

2.3 Cyclotomic Polynomial

It is a monic irreducible polynomial in x . It is defined as $\phi_m(x) = \prod_{1 \leq k < m} (x - e^{2\pi i k/m})$ where m is a positive integer, $\gcd(k, m) = 1$ and i is an imaginary unit. The cyclotomic polynomial is a divisor of polynomial $x^m - 1$ hence it is also called as m -th cyclotomic polynomial. The degree of $\phi_m(x)$ is totient of integer m , denoted by $\varphi(m)$ which is equal to the total number of positive integers up to m which are prime w.r.t m .

2.4 Polynomial Ring

A Polynomial ring is defined as a set of polynomials in one or more variables whose coefficients also belong to another ring. Consider a ring R with $p_0, p_1, \dots, p_m \in R$ and p is a polynomial in X , $p(X) = p_0 + p_1X + p_2X^2 + p_3X^3 + \dots + p_mX^m$, where $p(X) \in R[X]$ such that X is an external constant value that is added to ring R . We say polynomial ring $R[X]$ is derived from ring R as the coefficients of the polynomial $p(X)$ are ring elements of R and X is commutative in multiplication with every element of the ring R . Hence if R is a ring then $R[X]$ is also a ring that satisfies the same properties similar to R . We will further discuss the applications of polynomial rings in the hardness assumption of lattice-based cryptosystems.

2.5 Discrete Gaussian Distribution

Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ is a continuous probability distribution that determines the probability of a real random variable taking a range of values in an interval belonging to a set of real numbers. Discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z}}(\mu, \sigma^2)$ is the probability distribution of real random variables taking a discrete value in the set \mathbb{Z} . Discrete Gaussian distributions have pdf support in \mathbb{Z} which is proportional to $\exp(-\pi \|x\|^2 / \sigma^2)$. This type of distribution is used in most lattice-based cryptosystems.

2.6 LWE Problem

In this section, we discuss the LWE hardness problem that is the key element of security of most of the current **FHE** schemes. Lattice-based cryptosystems have many applications as their security is based on worst-case hardness problems such as shortest vector problem (SVP). Most of the **LFHE** and **FHE** schemes security is based on the hardness problems in lattices such as Learning with error(LWE), Ring learning with error(RLWE), etc.

Learning with error(LWE) problem stands for learning the secret in the presence of error. Given an integer $n \geq 1$, a prime number $q \geq 2$ polynomial in n , and a probability distribution χ on \mathbb{Z}_q . For a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ we define a distribution $A_{\mathbf{s}, \chi}$ on $\mathbb{Z}_q^n \times \mathbb{Z}_q$ that samples pairs (\mathbf{a}, \mathbf{b}) where vector \mathbf{a} is sampled uniformly at random from field \mathbb{Z}_q^n and $\mathbf{b} = \langle \mathbf{a}, \mathbf{s} \rangle + e$, $\langle \mathbf{a}, \mathbf{s} \rangle$ is the inner product and error $e \leftarrow \chi$. The LWE problem has two versions. Search LWE problem and decision LWE problem.

- Search LWE problem is, for any $\mathbf{s} \in \mathbb{Z}_q^n$ (same for all samples) given independent pairs of vectors $(\mathbf{a}_i, \mathbf{b}_i)$ for $i = 1, 2, \dots, m$ sampled from $\mathbb{Z}_q^n \times \mathbb{Z}_q$ by distribution $A_{\mathbf{s}, \chi}$ an adversary has to find the \mathbf{s} value with q and error distribution χ .
- The decision LWE problem is, for any $\mathbf{s} \in \mathbb{Z}_q^n$ given m samples of $(\mathbf{a}_i, \mathbf{b}_i)$ sampled from distribution $A_{\mathbf{s}, \chi}$, an adversary has to distinguish them from m samples (\mathbf{a}, \mathbf{b}) sampled uniformly at random from the set $\mathbb{Z}_q^n \times \mathbb{Z}_q$ with non-negligible probability.

LWE problem is as hard as solving the worst-case hardness problem such as *GapSVP* (the decision version of SVP) and *SIVP* (the shortest independent vector) by [Reg09]. It is also found to be an extension of the LPN problem (Learning parity with noise problem) for a specific case where $q = 2$ and error distribution χ is defined by a single parameter error probability $\varepsilon > 0$.

2.7 RLWE Problem

In this section, we explain the construction of the RLWE problem. RLWE problem is an extension of the LWE problem. To define n samples of LWE we require $O(n^2)$ values in the group \mathbb{Z}_q . Informally, the public keys of LWE based cryptosystem require n samples of LWE distribution due to which the key size will be of order $O(n^2)$ size that can cause huge runtime and storage costs in practical applications. Hence we define a different version of the LWE problem which can reduce the key size to a linear value in n . We choose LWE samples that have a special structure that allows multiple samples to be packed into a single sample hence reducing the key size to a linear size. For n samples of LWE, $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{Z}_q^n$ where each sample $\mathbf{a}_i = (a_1, a_2, \dots, a_n \in \mathbb{Z}_q)$. We replace \mathbb{Z}_q^n with a polynomial ring structure $R_q = \mathbb{Z}_q[x]/\langle x^n + 1 \rangle$, where single sample $a_i \in R_q$ packs n LWE samples such that $\mathbf{a}_1 = (a_1, \dots, a_n \in \mathbb{Z}_q)$ and other samples are $\mathbf{a}_i = (a_i, \dots, a_n, -a_1, \dots, -a_{i-1} \in \mathbb{Z}_q)$. This ring structure allows n LWE samples to be represented with $O(n)$ number of values in the group \mathbb{Z}_q . We will further discuss in detail the RLWE problem definition and its variants.

2.7.1 Chi Distribution

The error distribution used for RLWE problems is random distribution sampled uniformly from the family of error distributions on input λ . χ distribution is statistically indistinguishable from bounded discrete Gaussian distribution $D_{\mathbb{Z}, \sigma}^d$. The basic idea of a bounded discrete Gaussian distribution is it excludes few random values from the original distribution by slicing the tails of Gaussian distribution at discrete value B to obtain B -bounded discrete Gaussian distribution. To bound the distribution so that it always samples random values within $[-B, +B]$ range, we select a negligible value ε such that the $Pr[|x| > B] = \varepsilon$. This probability can be achieved in Gaussian distribution, where for a value $\beta > 0$ we have $Pr_{x \leftarrow \mathcal{N}(0, \sigma^2)}[|x| > \beta \cdot \sigma] = \text{erf}(\beta/\sqrt{2})$. Hence we can define β value as $\beta(\varepsilon) = \min(\beta | \text{erf}(\beta/\sqrt{2}) < \varepsilon)$ such that $Pr[|x| > \beta \cdot \sigma] = 1 - \varepsilon$ and $B = \beta \cdot \sigma$.

2.7.2 RLWE Distribution

For security parameter λ , let

- $f(x) \in \mathbb{Z}[x]$ is a cyclotomic polynomial $\phi_m(x)$ with $m = 2n$ and $f(x) = x^d + 1$ with $d = 2^n$
- $R = \mathbb{Z}[x]/f(x)$ be the polynomial ring
- $q = q(\lambda) \geq 2$ be an integer that is congruent to $1 \pmod{2n}$
- $R_q = \mathbb{Z}_q[x]/f(x)$
- distribution $\chi = \chi(\lambda)$ over R

For random $s \in R_q$, we define RLWE distribution $A_{s,\chi}^{(q)}$ that samples pairs $(a, a \cdot s + e)_q$ by selecting uniformly at random $a \in R_q$ and $e \leftarrow \chi$. RLWE problem also has 2 versions just like LWE problem. The Search RLWE problem and Decision RLWE problem.

2.7.3 Search-RLWE

Search-RLWE $_{q,d,\chi}$ problem is defined as, for any $s \in R_q$, $e \leftarrow \chi$ given m independent samples of RLWE distribution $A_{s,\chi}^q$, a ppt adversary has to find s .

2.7.4 Decision-RLWE

The *Decision-RLWE* $_{q,d,\chi}$ problem is defined as, for random $s \in R_q$, distinguishing m independent pseudorandom samples of RLWE distribution $A_{s,\chi}^q$ from the m independent random samples of uniform distribution $R_q \times R_q$.

For random $s \in R_q$, we say that the hardness of the *Decision-RLWE* $_{q,d,\chi}$ problem is due to a ppt adversary, not being able to distinguish the m independent pseudorandom samples of RLWE distribution $A_{s,\chi}^q$ from m independent random samples of uniform distribution $R_q \times R_q$ with non-negligible probability. When compared to LWE, RLWE has more practical applications [MR09], [LP11] because each sample $(a, b) \in R_q \times R_q$ from the R-LWE distribution can replace n samples from $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^n$ from the standard LWE distribution thus reducing the size of the public key (and often secret key also) by a factor of n . This is especially beneficial because key size has probably been the main barrier to practical lattice-based cryptosystems enjoying rigorous security analysis.

2.8 Public Key Encryption Scheme

In this section, we discuss briefly the concepts in public-key encryption schemes. We will later see that the public key encryption schemes act as the foundation for homomorphic encryption schemes.

2.8.1 Negligible Function

In provable security, we always measure the security strength of an encryption scheme by defining a security level that depends on the probability of success of security attack. We define the upper bound on this probability using a negligible function, to ensure that even if the security can be broken with a probability smaller than a negligible value, the scheme is still considered secure because it is too small. A function $\mu : \mathbb{N} \rightarrow \mathbb{R}^+$ is called a negligible function if, $\forall c > 0$, $\exists n_0 \in \mathbb{N}$, $\forall n \geq n_0$ such that

$$\mu(n) \leq \frac{1}{n^c}$$

2.8.2 Definition

Public Key Encryption scheme \mathcal{E} is an asymmetric encryption scheme that uses different keys to encrypt and decrypt a message. The encryption scheme comprises three algorithms.

- $\text{KeyGen}_{\mathcal{E}}(1^\lambda)$: On input 1^λ , outputs a key pair (sk, pk) with $|\text{sk}|, |\text{pk}| \geq \lambda$ where sk is secret key and pk is a public key.
- $\text{Enc}_{\mathcal{E}}(\text{m}, \text{pk})$: On input 1^λ , public key pk and for integer $t > 1$ message space $\mathcal{M} \rightarrow (0, 1)^t$ message $\text{m} \in \mathcal{M}$ outputs a ciphertext ct randomly. We assume $\text{Enc}_{\mathcal{E}}$ is a random algorithm denoted as $\text{ct} \leftarrow \text{Enc}_{\mathcal{E}}(\text{m}, \text{pk})$.
- $\text{Dec}_{\mathcal{E}}(\text{ct}, \text{sk})$: On input 1^λ , secret key sk and ciphertext ct , outputs message m or a special symbol \perp denoting failure. We assume $\text{Dec}_{\mathcal{E}}$ is a deterministic algorithm denoted as $\text{m} := \text{Dec}_{\mathcal{E}}(\text{ct}, \text{sk})$.

2.8.3 Adversary Attacks

An adversary is a probabilistic polynomial time (ppt) algorithm that tries to break the security of an encryption scheme (Public Key encryption scheme) by decrypting the ciphertext without enough information about the secret key required for decryption polynomial in time. If we define the security attack (on an encryption scheme) by a ppt adversary, as a game between the challenger and the adversary, then we say that the adversary is successful in breaking the security if it wins the game with non-negligible probability that is a polynomial in security parameter. We classify the adversarial attacks into 3 types based on the information available to the adversary to break the security. They are Eavesdropping attack, Chosen plaintext attack (CPA) and Chosen ciphertext attack (CCA). Further we describe the chosen plaintext attack against the public key encryption scheme.

Chosen Plaintext Attack

Consider the public-key encryption scheme \mathcal{E} and a ppt adversary \mathcal{A} . The CPA security game $\text{PubK}_{\mathcal{A}, \mathcal{E}}^{\text{cpa}(\lambda)}$ between the challenger and the adversary is given as follows. On input

$\text{pk} \leftarrow \text{KeyGen}_{\mathcal{E}}(1^\lambda)$, the adversary

1. Outputs a pair of messages m_0, m_1 from message space \mathcal{M} with $|\text{m}_0| = |\text{m}_1|$
2. The challenger chooses a random bit $b \in \{0, 1\}$ and outputs ciphertext $\text{ct} \leftarrow \text{Enc}_{\mathcal{E}}(\text{m}_b, \text{pk})$.
3. Adversary obtains encryption of more messages from the challenger
4. Adversary outputs a bit b'
5. Finally if $b = b'$ the challenger outputs 1 otherwise it outputs 0.

We say that the adversary wins the above game if the challenger outputs 1. Therefore the above public key encryption scheme \mathcal{E} is said to have indistinguishable encryptions under chosen-plaintext attacks, if

$\forall \text{ ppt } \mathcal{A}, \exists \text{ negligible function } \mu(\lambda) \text{ such that}$

$$\Pr[\text{PubK}_{\mathcal{A}, \mathcal{E}}^{\text{cpa}(\lambda)} = 1] \leq \frac{1}{2} + \mu(\lambda)$$

2.8.4 Properties Of Public Key Encryption Scheme

The 2 important properties of Public key encryption scheme are,

1. **Decryption Correctness:** Given a public key encryption scheme \mathcal{E} , the Encryption algorithm on input public key pk and message m generate a ciphertext $\text{ct} \leftarrow \text{Enc}_{\mathcal{E}}(\text{m}, \text{pk})$. Decryption correctness states that the decryption algorithm on input secret key sk and ciphertext ct should output message m such that,

$$\forall \lambda \in \mathbb{N}, (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda), \text{m} \in \mathcal{M}, \text{ct} \leftarrow \text{Enc}_{\mathcal{E}}(\text{m}, \text{pk}) :$$

$$\Pr[\text{Dec}_{\mathcal{E}}(\text{Enc}_{\mathcal{E}}(\text{m}, \text{pk}), \text{sk}) = \perp] \leq \mu(\lambda)$$

2. **Semantic security:** A Public Key encryption scheme is said to be semantically secure if for all ppt algorithms of adversary \mathcal{A} and for all probability distributions over the message space \mathcal{M} no negligible information about the secret key and the plaintext can be obtained from the ciphertext. Informally we say that a public key encryption scheme is semantically secure if it has indistinguishable encryptions against the chosen plaintext attacks for all ppt adversary \mathcal{A} .

2.9 Approximate Arithmetic

In this section, we briefly explain the concept of approximate arithmetic. Some homomorphic encryption schemes such as CKKS perform homomorphic operations on approximate values. We will discuss these operations in detail in CKKS scheme chapter.

2.9.1 Floating-point representation

In floating-point number system, a real number is represented by a product of significand(fractional part) and scaling factor(base to the power of exponent). For example, 5.6554 can be represented as 565.54×10^{-2} . In modern computers, we can only represent an approximate value of the real number in floating-point number system as we use fixed width datatype such as 32-bit or 64-bit to represent floating-point values. To represent an accurate real number we need larger bit-width datatypes that can represent any real number with bigger exponent values. Hence a floating-point number carries an implicit error after the decimal point due to which there is a loss of precision.

2.9.2 Types of approximate arithmetic

Approximate arithmetic is arithmetic of approximate numbers. As approximate numbers are represented using floating-point values, we have two different ways of performing these operations. Floating-point Arithmetic and fixed-point arithmetic. In Floating-point Arithmetic, the scaling factor is modified based on the required precision or efficiency. In Fixed-point arithmetic, we use fixed scaling factor to perform computations. Before performing the computations we fix the desired scaling factor and the same scaling factor is used throughout the computations. For Homomorphic computations, we use fixed-point arithmetic as we can control the error magnitude in the floating-point value.

3 Homomorphic Encryption Schemes

In this chapter, we study the various concepts in Homomorphic encryption schemes and also discuss types of homomorphic encryption schemes. Homomorphic encryption schemes are the encryption schemes that allow a user to perform mathematical operations on the encrypted values which are homomorphic to the operations that are performed on the decrypted data.

3.1 Evaluation Circuit

We can derive a homomorphic evaluation circuit from the mathematical operations performed on ciphertexts in a homomorphic encryption scheme. The Homomorphic evaluation circuit computes an arithmetic operation on n input ciphertexts and outputs a single ciphertext. It is an arithmetic circuit whose basic operations are addition and multiplication that can be used to build complex mathematical operations. It can be represented in the form of a directed acyclic graph, where the addition or multiplication operation forms the vertices of the graph and the input ciphertexts form the edges of the graph. The depth of the evaluation circuit is the maximum length of the path from the input ciphertext to the output ciphertext in the graph. The size of the evaluation circuit is defined as the total number of vertices (operations) in the circuit. In the figure 3.1 we define a homomorphic evaluation circuit that evaluates the function f on 3 input ciphertexts to compute ciphertext c_4 given by, $f(c_1, c_2, c_3) = c_4 = c_1 \cdot c_2 + c_2 \cdot c_3$. The depth of the circuit is 2 and the size of the circuit is 3.

3.2 Homomorphic Encryption Correctness

Consider a public key encryption scheme \mathcal{E} that comprises 3 algorithms $\text{KeyGen}_{\mathcal{E}}$, $\text{Enc}_{\mathcal{E}}$, and $\text{Dec}_{\mathcal{E}}$. A Homomorphic encryption scheme is a public-key encryption scheme with an additional algorithm $\text{Evaluate}_{\mathcal{E}}$ which performs mathematical operations on the ciphertexts generated by the $\text{Enc}_{\mathcal{E}}$ algorithm. The inputs to the $\text{Evaluate}_{\mathcal{E}}$ algorithm are a set of ciphertexts $\Psi = (ct_1, ct_2, \dots, ct_n)$ where $ct_i \leftarrow \text{Enc}_{\mathcal{E}}(m_i, pk)$, circuit $C_{\mathcal{E}}$ from a set of evaluation circuits \mathcal{C} and public key pk . The output of the $\text{Evaluate}_{\mathcal{E}}$ algorithm is $ct \leftarrow \text{Evaluate}_{\mathcal{E}}(\Psi, C_{\mathcal{E}}, pk)$. Thus the homomorphic encryption scheme correctness property is given by

$$\forall C_{\mathcal{E}} \in \mathcal{C}, (sk, pk) \in \text{KeyGen}_{\mathcal{E}} : \text{Dec}_{\mathcal{E}}(\text{Evaluate}_{\mathcal{E}}(\Psi, pk, sk)) = C_{\mathcal{E}}(m_1, m_2, \dots, m_n)$$

From the correctness, we conclude that a user that does not know the secret key can

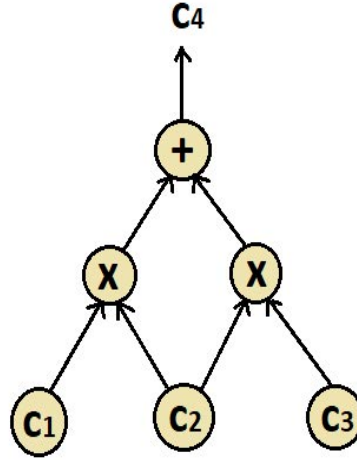


Figure 3.1: Evaluation circuit

perform a mathematical operation on a set of ciphertexts(modify the ciphertexts) which is equivalent to performing the same operation on the decrypted values of the ciphertexts using homomorphic encryption scheme.

3.3 Types Of Homomorphic Encryption Scheme

Homomorphic encryption schemes can be classified into 3 types based on the depth of their evaluation circuit.

1. **Somewhat homomorphic encryption schemes(SHE)** The Homomorphic encryption schemes that can evaluate a small, fixed depth circuit are called Somewhat homomorphic encryption schemes. The depth of the evaluation circuit depends on the encryption parameters of the scheme. Most of the first generation homomorphic schemes, for example [G⁺09],[VDGHV10] introduced noise in the ciphertexts to ensure security with a condition that the noise does not cross the pre-defined noise bound to ensure decryption correctness. This noise grew rapidly with the depth of the evaluation circuit that prevented the scheme from further evaluation hence limiting the evaluation circuit depth.
2. **Leveled Fully Homomorphic encryption schemes(LFHE)** This is an extension of SHE schemes. Like SHE schemes, Leveled FHE schemes can also evaluate a predefined, finite depth circuit based on the encryption parameters of the scheme. However unlike SHE schemes, Leveled FHE schemes allow the user to evaluate higher depth circuits by reducing the noise level in the ciphertext through some

noise reduction techniques such as Modulus switching , Key switching [BGV14]. In leveled FHE schemes the maximum depth of the circuit is a fixed polynomial in the security parameter of the encryption scheme. We will further discuss the practical implementations of LFHE schemes such as BFV, CKKS and determine their performance of evaluating functions in different scenarios.

3. **Fully homomorphic encryption schemes(FHE)** Fully homomorphic encryption schemes can evaluate a circuit of arbitrary depth. We can obtain FHE schemes FHE from SHE schemes with a special technique called bootstrapping (3.5) that resets the noise in the ciphertexts such that any circuit of arbitrary depth can be evaluated. The third generation encryption schemes such as [DM15], [CGGI16] have shown promising results in the practical applications of FHE schemes using bootstrapping technique.

3.4 Circular Security

Circular security is an important property for LFHE and FHE schemes. Homomorphic encryption scheme \mathcal{E} is said to be circularly secure if we can reveal the encryption of secret key $sk \leftarrow \text{KeyGen}_{\mathcal{E}}(1^\lambda)$ that is encrypted as $sk' \leftarrow \text{Enc}_{\mathcal{E}}(sk, pk)$. In further sections, we learn about techniques such as bootstrapping(3.5), Relinearisation4.2.3 in the schemes where the encryptions of secret key are revealed, theoretically we assume that the scheme possesses circular security that ensures the security of schemes in these scenarios.

3.5 Bootstrapping technique

Bootstrapping was introduced by Craig Gentry in 2009 to solve the problem of error growth in the ciphertexts of the SHE scheme so that an FHE scheme can be implemented that can evaluate a circuit of arbitrary depth. Bootstrapping reduces the error in a ciphertext by generating a new ciphertext that has a smaller error. Normally to generate a new ciphertext with small error, we need to decrypt the large error ciphertext and encrypt the plaintext with a small error. For decryption we have to expose the secret key hence it is not a feasible option. Thus bootstrapping allows us to obtain new ciphertext with reduced error without actually decrypting the ciphertext but by evaluating the decryption circuit homomorphically where we use encryption of secret key instead.

Consider an SHE scheme \mathcal{E} that comprises 4 algorithms- $\text{KeyGen}_{\mathcal{E}}$, $\text{Enc}_{\mathcal{E}}$, $\text{Dec}_{\mathcal{E}}$, and $\text{Evaluate}_{\mathcal{E}}$. We evaluate the $\text{Dec}_{\mathcal{E}}$ algorithm homomorphically using bootstrapping technique as follows. For $(sk, pk) \leftarrow \text{KeyGen}_{\mathcal{E}}$, $ct \leftarrow \text{Enc}_{\mathcal{E}}(m, pk)$ and $m \leftarrow \text{Dec}_{\mathcal{E}}(ct, sk)$.

- Generate a bootstrapping key $bsk \leftarrow \text{Enc}_{\mathcal{E}}(sk, pk)$
- this bootstrapping key is used to generate new encryptions of our secret key sk and ciphertext ct i.e. $sk' \leftarrow \text{Enc}_{\mathcal{E}}(sk, bsk)$ $ct' \leftarrow \text{Enc}_{\mathcal{E}}(ct, bsk)$

- the new encryption of message m is obtained from homomorphic evaluation of decryption circuit $\mathcal{D}_{\mathcal{E}}$, which is given as $ct'' \leftarrow \text{Evaluate}_{\mathcal{E}}(ct', sk', \mathcal{D}_{\mathcal{E}}, bsk)$
- This new ciphertext is also the encryption of the original message m using bootstrapping key bsk with reduced noise, $ct'' \leftarrow \text{Enc}_{\mathcal{E}}(m, bsk)$.

We consider the SHE scheme \mathcal{E} is secure under the assumption of circular security even in the case of bootstrapping where we reveal the encryption of a secret key to an adversary. Furthermore we discuss the bootstrappable property of homomorphic encryption schemes. An SHE scheme \mathcal{E} is considered bootstrappable, if the scheme can evaluate its decryption circuit. Let $\mathcal{C}_{\mathcal{E}}$ represent the set of evaluation circuits of \mathcal{E} , $\mathcal{D}_{\mathcal{E}}$ represents the decryption circuit of \mathcal{E} , we say the scheme is bootstrappable if $\mathcal{D}_{\mathcal{E}} \in \mathcal{C}_{\mathcal{E}}$. Hence, any SHE scheme that is bootstrappable can be converted to an FHE scheme because if it can evaluate its own decryption circuit, it can evaluate arbitrary depth circuits.

3.6 Squashing decryption circuit

To transform the SHE scheme to the FHE scheme we need to apply bootstrapping where the SHE scheme evaluates its decryption circuit homomorphically. This is possible if the decryption circuit is less complex or has a smaller depth. So ‘squashing’ a decryption circuit means, we reduce the depth of the decryption circuit that makes the encryption scheme bootstrappable. Craig Gentry first introduced the concept of squashing using the server-client notion. He has explained a 2 step method of squashing that not only minimizes the decryption circuit depth but also maintains the same maximum evaluation circuit depth of the scheme. In [G⁺09] he has described squashing for an abstract SHE scheme that is based on ideal lattices. In this scheme, the decryption circuit computes modulus operation on the ciphertext with the secret key such that if ψ is the ciphertext and \mathbf{B}_J^{sk} represents the secret key matrix and \mathbf{B}_I is a basis in Ring R then the decryption circuit is

$$\pi = (\psi \bmod \mathbf{B}_J^{sk}) \bmod \mathbf{B}_I$$

which is written as

$$\pi = \psi - \mathbf{B}_J^{sk} \cdot \lfloor (\mathbf{B}_J^{sk})^{-1} \cdot \psi \rfloor \bmod \mathbf{B}_I$$

This decryption circuit after some tweaks results in

$$\pi = \psi - \lfloor \mathbf{v}_J^{sk*} \times \psi \rfloor \bmod \mathbf{B}_I$$

The above decryption circuit performs a matrix-vector multiplication operation, whose complexity can be lowered by squashing the decryption circuit. Further, he has explained that the squashing is achieved by splitting the decryption algorithm into 2 phases.

- Pre-processing ciphertext: This step is performed by the server who has only access to the public key. The client hides some secret key (\mathbf{v}_J^{sk*}) information in the public key in the form of a set of vectors referred as ‘tag’ which is a finite

set of length $\gamma_{setsize}(n)$. This tag allows the server to perform the computation step of multiplication of ciphertexts with the secret key in the original decryption algorithm. This new set of product of ciphertext and ‘tag’ is then sent to the client along with the original ciphertext.

- Decryption : This step is performed by the server who owns the original secret key \mathbf{v}_J^{sk*} . On input ciphertext(ψ^*) and set of vectors (\mathbf{c}_i for $i = 1, \dots, \gamma_{setsize}(n)$), the client then performs decryption as

$$\pi = \psi^* - \lfloor \sum_i^{\gamma_{setsize}(n)} \mathbf{c}_i \rfloor \mod \mathbf{B}_I$$

Here the previous vector multiplication circuit is replaced with summation circuit. Hence reducing the decryption circuit’s complexity by squashing.

The squashing step is entirely based on the ‘tag’ information in reducing the complexity of the decryption circuit to obtain a simple summation circuit performing XOR operation on bits of small depth.

3.7 Modulus Switching:

Modulus switching is a general noise reduction technique in homomorphic encryption schemes based on LWE type problems. It is also used in bootstrapping to convert SHE schemes to LFHE schemes in homomorphic encryption schemes[FV12] . In this technique, we reduce or scale down the absolute magnitude of the ciphertext in-turn reducing the noise in the ciphertext, by switching the ciphertext modulus to a new smaller modulus. This is performed as follows. Consider a ciphertext $\mathbf{ct} \in \mathbb{Z}_{q_0}$ with noise level \mathbf{e} . We multiply the ciphertext with a fraction $\frac{q_1}{q_0}$ where $q_1 < q_0$ to obtain a ciphertext with reduced size $\mathbf{ct} \in \mathbb{Z}_{q_1}$. When we multiply the ciphertext with the fraction the noise in the ciphertext is also multiplied such that the new noise in the ciphertext will be reduced to $\mathbf{e} \cdot \frac{q_1}{q_0}$. This technique of reducing noise in LFHE schemes is referred as Modulus switching because we switch the modulus from higher level q_0 to lower level q_1 .

4 BFV Scheme

BFV scheme is a Homomorphic encryption scheme proposed by Jungeng Fan and Frederik Vercauteren. They have extended the work of Zvika Brakerski's Fully Homomorphic Encryption scheme[Bra12] as Zvika Brakerski's scheme handles the increase in noise levels in ciphertexts due to the multiplication of ciphertexts better than other LWE based schemes [Bra12] and its methods are found to be easy to be able to port to an RLWE based encryption scheme. The basic framework for most of the LWE based HE schemes is Gentry's implementation of the FHE scheme from the SHE scheme. In Gentry's implementation, he converted the SHE scheme to the FHE scheme by squashing the decryption circuit and performing bootstrapping on the ciphertext. In theory, the BFV scheme also implements FHE through bootstrapping technique except we do not squash the decryption circuit but use the modulus switching technique to reduce the complexity of the decryption circuit. However further we will see that practically FHE version of the BFV scheme is inefficient and, we can only implement the limited version of FHE that is leveled FHE scheme using the Modulus switching technique.

4.1 Encryption Scheme

In this section, we explain the construction of the BFV encryption scheme. The BFV scheme's security is due to the hardness assumption of the *Decision-RLWE* $_{q,d,\chi}$ problem. Hence the necessary ingredients for encryption, the public-key, and the secret key are obtained from pseudorandom samples of RLWE distribution. The encryption algorithm takes input messages from plaintext space R_t and public key from $R_q \times R_q$ and outputs ciphertext from the ciphertext space $R_q \times R_q$. The plaintext modulus t should be smaller than q to maintain lower noise levels in the ciphertext. We also scale our plaintext by $\Delta = \lfloor q/t \rfloor$ during encryption. We will understand why we perform the encryption in this way when we define the decryption algorithm and also understand why is it required to regulate noise levels in a ciphertext.

For security parameter λ , let

- $f(x) = x^d + 1$ be a m -th cyclotomic polynomial where d is power of 2.
- $R = \mathbb{Z}[x]/f(x)$ be the polynomial ring
- $q = q(\lambda) \geq 2$ be an integer
- $R_q = \mathbb{Z}_q[x]/f(x)$
- $R_t = \mathbb{Z}_t[x]/f(x)$

- $t > 1$ be an integer
- distribution $\chi = \chi(\lambda)$ over R that samples values in range $[-B, B]$

The BFV encryption scheme comprises following algorithms.

- $\text{FV.SH.SecretKeyGen}(1^\lambda)$: On input 1^λ sample $s \leftarrow \chi$ and output secret key $\text{sk} = s$
- $\text{FV.SH.PublicKeyGen}(\text{sk})$: On input sk , set $\text{sk} = s$, sample $a \leftarrow R_q$, $e \leftarrow \chi$ and output public key $\text{pk} = (-[a \cdot s + e]_q, a)$
- $\text{FV.SH.Encrypt}(\text{pk}, m)$: On input public-key pk , and message $m \in R_t$, let $p_0 = \text{pk}[0]$, $p_1 = \text{pk}[1]$, $u, e_1, e_2 \leftarrow \chi$, $\Delta = \lfloor q/t \rfloor$ and output ciphertext $ct = (ct[0], ct[1]) = ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e_2]_q)$
- $\text{FV.SH.Decrypt}(\text{sk}, ct)$: On input secret key sk and ciphertext ct set $c_0 = ct[0]$, $c_1 = ct[1]$, we compute the inner product of ciphertext and the secret key and after some simplifications output message $m \in R_t$.

4.1.1 Decryption Correctness

Decryption algorithm computes inner product ct, sk . The plaintext is scaled by Δ during encryption. Hence the decryption algorithm scales down the ciphertext by Δ leaving the plaintext with some noise. We will further see how the decryption algorithm removes the noise through rounding operation and achieve decryption correctness if defining an upper bound on the noise in the ciphertext to $\Delta/2$.

Explanation

The input to the decryption circuit is given as inner product of secret key and the ciphertext elements $\langle ct, \text{sk} \rangle$ where $ct = [c_0, c_1]$ and $\text{sk} = [1, s]$. Here $c_0, c_1, s \in R_q$.

$$\begin{aligned}
 \langle ct, \text{sk} \rangle &= [c_0 + c_1 \cdot s]_q = p_0 \cdot u + e_1 + \Delta \cdot m + p_1 \cdot u \cdot s + e_2 \cdot s \\
 &= -a \cdot s \cdot u + e \cdot u + e_1 + \Delta \cdot m + a \cdot u \cdot s + e_2 \cdot s \\
 &= \Delta \cdot m + e \cdot u + e_1 + e_2 \cdot s
 \end{aligned} \tag{4.1}$$

Substitute $v = e \cdot u + e_1 + e_2 \cdot s$ in the above equation

$$[c_0 + c_1 \cdot s]_q = \Delta \cdot m + v$$

Re-writing the above equation as,

$$c_0 + c_1 \cdot s = [c_0 + c_1 \cdot s]_q + q \cdot r$$

where r is the quotient obtained when we take $c_0 + c_1 \cdot s$ modulo q

In order to obtain the message, we further multiply this inner product with plain modulus t and divide with ciphertext modulus q -

$$\begin{aligned} \frac{t}{q} \cdot (c_0 + c_1 \cdot s) &= \frac{t}{q} \cdot [c_0 + c_1 \cdot s]_q + t \cdot r \\ &= \frac{t}{q} \cdot (\Delta \cdot m + v) + t \cdot r \end{aligned} \quad (4.2)$$

Since $\Delta = \lfloor \frac{q}{t} \rfloor = \frac{q}{t} - \varepsilon$ where $\varepsilon < 1$.

After substituting the Δ we get-

$$\begin{aligned} \frac{t}{q} \cdot (c_0 + c_1 \cdot s) &= \frac{t}{q} \cdot ((\frac{q}{t} - \varepsilon) \cdot m + v) + t \cdot r \\ &= \frac{t}{q} \cdot \frac{q}{t} \cdot m - \frac{t}{q} \cdot \varepsilon \cdot m + \frac{t}{q} \cdot v + t \cdot r \\ &= m + \frac{t}{q}(v - \varepsilon \cdot m) + t \cdot r \end{aligned} \quad (4.3)$$

In the final equation of decryption, we get message $m \in R_t$ and a noise term. Hence in order to separate the message from the noise we perform rounding operation which removes the noise in the ciphertext.

$$\lfloor m + \frac{t}{q}(v - \varepsilon \cdot m) + t \cdot r \rfloor$$

This rounding operation will result in correct decryption of the ciphertext only if following condition is met-

$$\frac{t}{q} \cdot \|(v - \varepsilon \cdot m)\| < \frac{1}{2}$$

Since $\varepsilon < 1$, the noise bound is

$$v < \frac{\Delta}{2}$$

Once the noise is removed, we apply modulo t on the plaintext and obtain a plaintext polynomial in R_t . Hence we say that message is decrypted correctly only if $v < \frac{\Delta}{2}$. The decryption correctness of BFV scheme is given as

$$\begin{aligned} \forall \lambda \in \mathbb{N}; \text{ sk} \leftarrow \text{FV.SH.SecretKeyGen}(1^\lambda), \text{ pk} \leftarrow \text{FV.SH.PublicKeyGen}(\text{sk}); \text{ m} \in R_t; v < \frac{\Delta}{2} \\ \Pr[\text{FV.SH.Dec}(\text{sk}, \text{FV.SH.Encrypt}(\text{pk}, \text{m})) = \text{m}] = 1 \end{aligned} \quad (4.4)$$

Noise bound of the fresh ciphertext-

We know that noise of a fresh ciphertext is given by

$$v = e \cdot u + e_1 + e_2 \cdot s$$

Since e, u, e_1, e_2, s are sampled using χ distribution and the upper bound on χ distribution samples is $\|\chi\| < B$. The upper bound on each term in v is

$$\|e \cdot u\| = \|e_2 \cdot s\| = \delta_R \cdot B^2$$

where δ_R is the expansion factor of R and the upper bound on e_1 is $\|e_1\| < B$. Therefore the upper bound on the noise in a ciphertext is-

$$\|v\| \leq 2 \cdot \delta_R \cdot B^2 + B < \frac{\Delta}{2}$$

4.1.2 Semantic security

Semantic security of the BFV scheme is ensured,

- by not revealing the secret key information by the public key. We know that by the hardness assumption of the RLWE problem, a public key sampled from the RLWE distribution is proven to be pseudorandom. Hence a ppt adversary cannot obtain secret key information from a public key.
- by not revealing any information about the plaintext. Since the ciphertexts generated by the pseudorandom public key are also pseudorandom, the scheme generates indistinguishable encryptions. Hence by the definition of semantic secure cryptosystem it is not possible for a ppt adversary to learn about the plaintext from pseudorandom ciphertexts.

4.2 Somewhat Homomorphic Encryption

In this section, we explain the construction of the SHE version of the BFV scheme. Along with the usual encryption algorithms we define the evaluation algorithm for addition, multiplication, and relinearisation operations in a SHE scheme.

4.2.1 Addition:

The inputs for the evaluation method of addition operation are the ciphertexts, the addition circuit, and the public key and the output is the sum of the ciphertexts. We will further see that the addition operation causes growth in ciphertext noise by a maximum of t . Hence we can perform an arbitrary number of addition operations with the SHE scheme.

Let,

$$\begin{aligned} ct_1 &= ([p_0 \cdot u + e_1 + \Delta \cdot m_1]_q, [p_1 \cdot u + e_2]_q) \\ ct_2 &= ([p_0 \cdot u' + e'_1 + \Delta \cdot m_2]_q, [p_1 \cdot u' + e'_2]_q) \end{aligned} \quad (4.5)$$

Adding both the ciphertexts, we get

$$[ct_1 + ct_2]_q = ([ct_1[0] + ct_2[0]]_q, [ct_1[1] + ct_2[1]]_q)$$

where

$$\begin{aligned} ct_1[0] + ct_2[0] &= [p_0 \cdot u + e_1 + \Delta \cdot m_1]_q + [p_0 \cdot u' + e'_1 + \Delta \cdot m_2]_q \\ &= [\Delta(m_1 + m_2) + p_0 \cdot u + e_1 + p_0 \cdot u' + e'_1]_q \\ [ct_1[1] + ct_2[1]] &= [p_1 \cdot u + e_2]_q + [p_1 \cdot u' + e'_2]_q \\ &= [p_1 \cdot u + e_2 + p_1 \cdot u' + e'_2]_q \end{aligned} \quad (4.6)$$

Resulting ciphertext is

$$[ct_1(s) + ct_2(s)] = [\Delta(m_1 + m_2) + p_0 \cdot u + e_1 + p_0 \cdot u' + e'_1]_q + [p_1 \cdot u \cdot s + e_2 \cdot s + p_1 \cdot u' \cdot s + e'_2 \cdot s]_q$$

Consider

$$\begin{aligned} v_1 &= e \cdot u + e_1 + e_2 \cdot s \\ v_2 &= e \cdot u' + e'_1 + e'_2 \cdot s \end{aligned} \quad (4.7)$$

and if we write,

$$\begin{aligned} [ct_1(s) + ct_2(s)]_q &= [ct_1(s) + ct_2(s)] - q \cdot r \\ (m_1 + m_2) &= [m_1 + m_2]_t + t \cdot r \end{aligned} \quad (4.8)$$

Then the sum of ciphertexts is,

$$\begin{aligned} [ct_1(s) + ct_2(s)]_q &= [\Delta \cdot [m_1 + m_2]_t + \Delta \cdot t \cdot r + v_1 + v_2] - q \cdot r \\ &= [\Delta \cdot [m_1 + m_2]_t + v_1 + v_2] - q \cdot r + \Delta \cdot t \cdot r \end{aligned} \quad (4.9)$$

Substituting $\Delta = q/t - \varepsilon$ in the third term, we get

$$[ct_1(s) + ct_2(s)]_q = [\Delta \cdot [m_1 + m_2]_t + v_1 + v_2] - \varepsilon \cdot t \cdot r$$

The input to the decryption circuit is the sum of the ciphertexts $[ct_1(s) + ct_2(s)]_q$. The noise in the resulting ciphertext is in the order of t . Finally the decryption algorithm outputs $[m_1 + m_2]_t \in R_t$ which is the sum of $[m_1]_t$ and $[m_2]_t$. We assume that the homomorphic addition correctness is ensured by default.

4.2.2 Multiplication

The inputs for the evaluation method of multiplication operation are the ciphertexts, the multiplication circuit, the public key, and the relinearisation key and the output is the product of the ciphertexts. Homomorphic multiplication involves 2 steps-

1. Basic multiplication of ciphertexts and scaling by t/q that results in ciphertext with 3 elements.

Let the two ciphertexts be

$$\begin{aligned} ct_1(s) &= c_0 + c_1 \cdot s \\ ct_2(s) &= c'_0 + c'_1 \cdot s \\ ct_1(s) \cdot ct_2(s) &= c_0 \cdot c'_0 + c_0 \cdot c'_1 \cdot s + c'_0 \cdot c_1 \cdot s + c_1 \cdot c'_1 \cdot s^2 \end{aligned} \tag{4.10}$$

Substitute

$$\begin{aligned} C_1 &= c_0 \cdot c'_0 \\ C_2 &= (c_0 \cdot c'_1 + c'_0 \cdot c_1) \\ C_3 &= c_1 \cdot c'_1 \end{aligned} \tag{4.11}$$

The product of ciphertexts is,

$$ct_1(s) \cdot ct_2(s) = C_1 + C_2 \cdot s + C_3 \cdot s^2$$

2. Reduce the degree 2 ciphertext polynomial to a degree 1 ciphertext polynomial in s through relinearisation. We will further discuss relinearisation in detail.

4.2.3 Relinearisation

In this section, we explain the construction of Relinearisation in the BFV scheme. Relinearisation is used to reduce the number of ciphertext elements of the ciphertext polynomial in s to 2 elements. When we perform homomorphic multiplication on 2 ciphertexts the resultant ciphertext will have 3 elements, which leads to increased storage space consumption and time consumption for further homomorphic operations on the ciphertext and hence reducing the overall performance of the scheme. To solve this problem we use the relinearisation method to reduce the number of elements in the ciphertext. We define two different methods of performing relinearisation.

Relinearisation Version 1

This relinearisation method can be used after we evaluate a multiplication circuit of small depth as this method adds a very small amount of relinearisation noise to the ciphertext. We eliminate the third element of the ciphertext by adding its information into the first 2 elements. This is achieved by decomposition of the third element of ciphertext into

$l + 1 = \log_T q + 1$ number of terms represented in base-T and using these base-T values to mask the secret key s^2 in the relinearisation key. Therefore we can use this key which contains a relinearisation key for each masked version of secret key $T^i \cdot s^2$ to perform relinearisation without revealing the actual secret key. Here we assume that the BFV scheme has weak circular security. Relinearisation process also adds a small noise in the ciphertext which is proportional to the value T and is independent of the existing ciphertext noise. The relinearisation is performed on a product ciphertext as follows:

- Input: A ciphertext $ct = [c_0, c_1, c_2]$ i.e. $ct(s) = [c_0 + c_1 \cdot s + c_2 \cdot s^2]_q$
- Relinearisation key: sample $a_0 \leftarrow R_q$, $e_0 \leftarrow \chi$, represent c_2 in base T as

$$c_2 = \sum_{i=0}^l T^i \cdot c_2^{(i)} \mod q$$

where $l = \lfloor \log_T(q) \rfloor$ and the coefficients $c_2^{(i)} \in R_T$. Relinearisation key is given as-

$$rlk = ([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i) \text{ where } i \in [0..l]$$

- Output: The new ciphertext elements obtained are-

$$\begin{aligned} c'_0 &= [c_0 + \sum_{i=0}^l rlk[i][0] \cdot c_2^{(i)}]_q \\ c'_1 &= [c_1 + \sum_{i=0}^l rlk[i][1] \cdot c_2^{(i)}]_q \end{aligned} \tag{4.12}$$

and compute

$$c'_0 + c'_1 \cdot s = c_0 + c_1 \cdot s + c_2 s^2 - \sum_{i=0}^l c_2^{(i)} \cdot e_i \mod q$$

- Noise level: relinearisation introduces new noise bounded by $(l + 1) \cdot B \cdot T \cdot \delta_R/2$

Relinearisation Version 2

This relinearisation method should be used when we need to perform multiplication of large depth circuits as it introduces very high relinearisation error for the first relinearisation after the first multiplication but requires small storage space and computation time when compared to relinearisation version 1. We eliminate the third element of the ciphertext by adding its information into the first 2 elements. Instead of masking s^2 with base-T values, we mask it with a single integer p to reduce the extra error that is added during relinearisation. Hence we generate a key with its elements in modulo $p \cdot q$ and sample the error $e \leftarrow \chi'$. We make sure that the new parameters used to define the

relinearisation key retain the semantic security of the scheme. The new relinearisation noise is inversely proportional to p , hence it can be reduced by choosing greater values for p . Since we do not split the ciphertext or the relinearisation key does not have multiple elements, the least amount of time and space is required to perform the relinearisation compared to version 1.

- Input: A ciphertext $ct = [c_0, c_1, c_2]$ i.e. $ct(s) = [c_0 + c_1 \cdot s + c_2 \cdot s^2]_q$
- Relinearisation key: for some integer p , sample $a \leftarrow R_{p,q}$, $e \leftarrow \chi'$, Relinearisation key is given as-

$$rlk = [(- (a \cdot s + e) + p \cdot s^2)_{p,q}, a]$$

- Output: After relinearisation, the last term c_2 is represented with a tuple of 2 elements where

$$(c_{2,0}, c_{2,1}) = ([\lfloor \frac{c_2 \cdot rlk[0]}{p} \rfloor]_q, [\lfloor \frac{c_2 \cdot rlk[1]}{p} \rfloor]_q)$$

and the relinearised ciphertext is

$$(c'_0, c'_1) = (c_0 + c_{2,0}, c_1 + c_{2,1})$$

- Noise level: relinearisation introduces new noise bounded by

$$\|r\| < \frac{q \cdot B_k \cdot \delta_R}{p} + \frac{\delta_R \cdot \|s\| + 1}{2}$$

4.2.4 BFV SHE scheme

BFV Somewhat Homomorphic Encryption scheme is defined with following algorithms,

- FV.SH.SecretKeyGen(1^λ) : On input 1^λ sample $s \leftarrow R_2$ and output secret key $sk = s$
- FV.SH.PublicKeyGen(sk) : On input sk , set $s = sk$, sample $a \leftarrow R_q$, $e \leftarrow \chi$ and output public key $pk = (-[a \cdot s + e]_q, a)$
- FV.SH.EvalKeyGen:
 - Version 1: On input (sk, T) output relinearisation key

$$rlk = ([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i) \text{ where } i \in [0..l]$$

- Version 2: On input (sk, p) output relinearisation key

$$rlk = ([-(a \cdot s + e) + p \cdot s^2]_{p,q}, a)$$

- $\text{FV.SH.Encrypt}(\text{pk}, m)$: On input public-key pk , and message $m \in R_t$, let $p_0 = \text{pk}[0]$, $p_1 = \text{pk}[1]$, and $u, e_1, e_2 \leftarrow \chi$, $\Delta = \lfloor q/t \rfloor$ output ciphertext

$$\text{ct} = ([p_0 \cdot u + e_1 + \Delta \cdot m]_q, [p_1 \cdot u + e_2]_q)$$

- $\text{FV.SH.Add}(\text{ct}_1, \text{ct}_2)$: On input of two ciphertexts ct_i for $i = 1, 2$ compute and return

$$[\text{ct}_1 + \text{ct}_2]_q = ([\text{ct}_1[0] + \text{ct}_2[0]]_q, [\text{ct}_1[1] + \text{ct}_2[1]]_q)$$

- $\text{FV.SH.Mul}(\text{ct}_1, \text{ct}_2, \text{rlk})$: On input ct_1, ct_2 and rlk
 - compute $[\text{ct}_1 \cdot \text{ct}_2]_q = ([c_0]_q, [c_1]_q, [c_2]_q)$ where

$$c_0 = [(\text{ct}_1[0] \cdot \text{ct}_2[0])]_q$$

$$c_1 = [(\text{ct}_1[0] \cdot \text{ct}_2[1] + \text{ct}_1[1] \cdot \text{ct}_2[0])]_q$$

$$c_2 = [(\text{ct}_1[1] \cdot \text{ct}_2[1])]_q$$
 - If applied relinearisation version 1 on the new product ciphertext, we obtain relinearised ciphertext $\text{ct} = (c'_0, c'_1)$ where $c'_0 = [c_0 + \sum_{i=0}^l \text{rlk}[i][0] \cdot c_2^{(i)}]_q$ and $c'_1 = [c_1 + \sum_{i=0}^l \text{rlk}[i][1] \cdot c_2^{(i)}]_q$
 - If relinearisation using version 2 rlk then return $\text{ct} = ([c_0 + c_{2,0}]_q, [c_1 + c_{2,1}]_q)$ where

$$(c_{2,0}, c_{2,1}) = ([\lfloor \frac{c_2 \cdot \text{rlk}[0]}{p} \rfloor]_q, [\lfloor \frac{c_2 \cdot \text{rlk}[1]}{p} \rfloor]_q)$$

- $\text{FV.SH.Decrypt}(\text{sk}, \text{ct}(s))$: On input secret key sk and ciphertext $\text{ct}(s)$, set $s = \text{sk}$, $c_0 = \text{ct}[0]$, $c_1 = \text{ct}[1]$ compute inner product $\langle \text{ct}, \text{sk} \rangle$ and output message $m \in R_t$

4.2.5 Maximum depth of a SHE multiplication circuit

The maximum depth of the multiplication circuit depends on the ciphertext coefficient modulus q and standard deviation σ of χ distribution and can be calculated as -

The noise level of a product ciphertext defined in [FV12] is given by ,

$$v_{mul} = E \cdot t \cdot \delta_R \cdot (\delta_R + 1.25) + E_{relin}$$

Here E is the noise of a fresh ciphertext which is bounded by $2 \cdot \delta_R \cdot B$ where δ_R is the expansion factor of R and B is the bound of the error distribution. E_{relin} is relinearisation noise that is smaller than the multiplication noise before relinearisation hence it can be ignored. That implies

$$v_{mul} = 2 \cdot \delta_R \cdot B \cdot t \cdot \delta_R \cdot (\delta_R + 1.25)$$

If we perform depth L levels multiplication on a ciphertext then the noise is given by,

$$v_{mul} = E \cdot t \cdot \delta_R^L \cdot (\delta_R + 1.25)^L$$

Since the noise of the ciphertext should be less than $\Delta/2$ for correct decryption, we write the bound for the noise level as,

$$\begin{aligned} v_{mul} &= 2 \cdot \delta_R \cdot B \cdot t \cdot \delta_R^L \cdot (\delta_R + 1.25)^L < \lfloor \frac{q}{t} \rfloor \cdot \frac{1}{2} \\ v_{mul} &= 4 \cdot \delta_R^L \cdot (\delta_R + 1.25)^{L+1} \cdot t^{L-1} < \lfloor q/B \rfloor \end{aligned} \quad (4.13)$$

If we solve the above equation for L we can determine the maximum depth of a multiplication circuit that can be evaluated homomorphically by SHE scheme. Also, note that the depth of the circuit depends on 2 parameters- ciphertext modulus q and error distribution bound B .

4.3 Fully Homomorphic Encryption

In this section, we explain the construction of the FHE part of the BFV scheme. If the SHE scheme decryption circuit is of depth D then the SHE scheme can be converted to an FHE scheme by bootstrapping technique if the maximum circuit depth evaluated by the SHE scheme from equation 4.13 is much greater than $D + 1$. The bootstrapping technique resets the noise in the ciphertext to some constant minimum noise required for correct decryption such that **after** if before bootstrapping the scheme could evaluate a depth D circuit then after bootstrapping it can evaluate plus 1 level higher depth circuit. Hence every time we have to perform a homomorphic evaluation of circuit depth greater than $D+1$, we compute ciphertexts for the depth D circuit, perform bootstrapping on the ciphertexts and use the new ciphertexts for further homomorphic computations. This is possible if we take the encryption parameters such that the maximum depth L defined in 4.13 is greater than depth $D+1$. In the BFV scheme instead of squashing the decryption circuit to reduce its complexity, we perform modulus switching on the ciphertexts. In modulus switching, we switch the ciphertext modulus q to smaller ciphertext modulus $2^{S_R} < q$, this reduces the ciphertext size by eliminating more bits from the ciphertext as noise.

If the ciphertext before bootstrapping is,

$$ct = (ct_0, ct_1)$$

then the ciphertext after bootstrapping is,

$$ct' = (ct'_0 + e_0, ct'_1 + e_1)$$

such that

$$ct'(s) = ct'_0 + ct'_1 \cdot s = \Delta \cdot m + v + e_0 + e_1 \cdot s + q \cdot r$$

The upper bound on the new noise elements in each ciphertext element $\|e_i\| < \Delta/\nu$. Hence the new noise bound will be reduced from $\Delta/2$ to Δ/μ for $\mu > 2$. The new noise

bound of a bootstrapped ciphertext as defined in [FV12] is

$$\|v\| < \left\| \frac{\Delta}{\mu} + (H(f) \cdot h + 1) \cdot \frac{\Delta}{\nu} \right\|$$

where

$$H(f) = \max \left\| \sum_{i=0}^{d-1} (x \| i + j \bmod f(x)) \| \text{ for } j = 0, \dots, d-1 \right\|$$

and h = Hamming weight of s

Here we describe the general case where $q \neq 2^n$. Hence we prepare our ciphertext for bootstrapping by scaling the ciphertext by $\frac{2^n}{q}$ to obtain a ciphertext with modulus 2^n instead of modulus q where $n = \lfloor \log_2(q) \rfloor$ and define $\Delta = 2^k$.

Definition:

BFV FHE scheme is an extension of SHE scheme. The Key generation and encryption algorithms are same as SHE scheme. But the decryption algorithm of FHE scheme is designed to accommodate bootstrapping and is defined as follows.

- FV.FH.Decrypt(ct(s), sk) : On input ct(s) and secret key sk perform modulus switching as follows and return a constant coefficient of message $m \in R_t$ -
 - Scale down the ciphertext to S_R bits by switching the modulus from 2^n to 2^{S_R} .
 - To switch the modulus, perform right shift operation by $(n - S_R)$ bits on $[ct_0]_q$ and $[ct_1]_q$. Let the ciphertext elements with new modulus be d_0 and d_1 .
 - the coefficient of each ciphertext elements d_0 and d_1 are of length S_R bits. Each ciphertext polynomial has $d + 1$ integer coefficients and hence we define $(d + 1) \times S_R$ matrix for each element.
 - The final decryption step is to add the $(d + 1)$ rows modulo 2^{S_R} that result in an integer w such that $0 \leq w \leq 2^{S_R}$
 - Define rounding bit $w_b = w[k - n + S_R - 1]$ and output the decryption of constant coefficient of message $m_0 = [w \gg (k - n + S_R) + w_b]_t$.
- FV.FH.BootKeyGen(sk, pk): on input secret key and public key return bootstrapping key $bsk = [FV.FH.Encrypt(s_i, pk) : i \in [0 \dots d - 1]]$
- FV.FH.Bootstrapping(ct'(s), bsk) : On input bootstrapping key and ciphertext return a new ciphertext by evaluating the FV.FH.Decrypt (ct'(s), bsk) algorithm. This new ciphertext can be used to evaluate homomorphic functions of greater depth.

As we can see from the decryption algorithm, the decryption circuit is simplified from complex polynomial multiplication to simple bit level addition of carry-save adder circuit with depth $S_R - 1$.

5 CKKS Scheme

CKKS scheme is proposed by Jung Hee Cheon, Andre Kim, Miran Kim, and Yongsoo Song[CKKS17]. This scheme is an extension of the BGV scheme[BGV14] and it is also based on the Decision-RLWE hardness problem. We can perform approximate arithmetic computations homomorphically on real numbers with the CKKS scheme. The construction of the scheme is as shown in [CKKS17].

5.1 Encoding and Decoding

For a positive integer M , let $\phi_M(X)$ be the M -th cyclotomic polynomial of degree $N = \pi(M)$ with M -th roots of unity ζ_M . Let $\mathcal{R} = \mathbb{Z}[X]/\phi_M(X)$ be a polynomial ring of integers of number field $\mathbb{Q}[X]/\phi_M(X)$ and $\mathcal{S} = \mathbb{R}[X]/\phi_M(X)$ be the polynomial ring of real numbers. CKKS plaintext space is polynomials in \mathcal{R} . CKKS scheme allows the user to encrypt a vector of complex numbers into a single ciphertext by encoding the vector of complex numbers in \mathbb{C}^N into a polynomial of \mathcal{R} . This encoding comprises various steps, one of them requires complex canonical embedding map. Canonical embedding maps a real polynomial $a \in \mathcal{S}$ to a vector of complex numbers in \mathbb{C}^N where the vector elements are the evaluation values of a at ζ_M . Hence CKKS encoding uses inverse canonical embedded mapping to obtain the polynomial in \mathcal{S} from complex number vector. Since the polynomials in \mathcal{S} have real coefficients we multiply the polynomial with a scaling factor $\Delta \geq 1$ and later perform the rounding operation to obtain plaintext polynomials with integer coefficients in \mathcal{R} . CKKS Decoding uses the canonically embedded mapping $\sigma : \mathcal{S} \rightarrow \mathbb{C}^N$ to obtain the complex vector in \mathbb{C}^N from plaintext polynomials.

5.2 Encryption

For integer $q > 0$, $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ be the residual integer ring \mathcal{R} modulo integer q . During encryption the plaintext is added with some random noise to ensure security of hardness assumption RLWE problem. Therefore the encryption algorithm on input public key $pk \in \mathcal{R}_q^2$ and plaintext $a(X) \in \mathcal{R}$ generates a pair of polynomial ciphertext $ct = (c_0(X), c_1(X)) \in \mathcal{R}_q^2$.

The composition of a ciphertext is combination of plaintext and error, the plaintext bits and error bits are situated on the LSB side of the ciphertext modulus, so that after every homomorphic computation the resulting plaintext bits are moved to the MSB side and the new error bits are accommodated on the LSB side. Hence the noise in the ciphertext can be treated as an approximation error in approximate arithmetic which

can be removed by a rounding operation that eliminates the LSBs of the ciphertext modulus while preserving the plaintext bits.

5.3 Decryption

Due to the presence of noise in the ciphertext, the decryption structure of CKKS scheme is given by $\langle ct, s \rangle = c_0 + c_1 \cdot s = m + e \pmod{q}$ for a small error e . The decrypted plaintext is in the form of $m' = m + e$. The plaintext m' is treated as an approximate value of plaintext m with higher precision. Therefore for correct decryption, we should keep the decryption structure size smaller than the ciphertext modulus q and this can be realized by maintaining the noise in the ciphertext very small compared to the plaintext.

5.4 Rescaling

Rescaling is a form of modulus switching where we switch a larger ciphertext modulus q to a smaller ciphertext modulus q' . It is required to implement leveled homomorphic encryption which allows homomorphic computations of predefined circuit depth. When we perform homomorphic multiplication of certain depth on the ciphertext, the size of the ciphertext modulus grows linearly with the depth of the circuit, in turn causing an increase in the noise of ciphertext leading to a point where no more computations are possible because the noise magnitude is much larger than the plaintext. Hence we use rescaling where the ciphertext is divided with an integer $q' < q$ which in-turn divides the plaintext by q' that eliminates the error bits in the LSB of plaintext, keeping the original message the same.

When we multiply a ciphertext of scale Δ , for l number of times, the resulting ciphertext will have a scale equal to Δ^l . The decryption structure of the product ciphertext is given as

$$\langle ct_{mul}, s \rangle = m + e \pmod{q}$$

. To perform rescaling on it, we multiply the ciphertext with new modulus q' and divide it with q to obtain a new ciphertext $ct' \in R_{q'}$. The decryption structure of the new ciphertext will be

$$\langle ct', s \rangle = \frac{q'}{q}m + \frac{q'}{q}e + e_{scale} \pmod{q'}$$

where e_{scale} is scaling noise. By reducing the ciphertext modulus from q to q' , we actually reduce the scale of the ciphertext by Δ times because $q = \Delta \cdot q'$, which causes the ciphertext to lose $\log(\Delta)$ number of LSBs of ciphertext, leaving the original message intact. Due to this, during decryption, the plaintext can be easily recovered by rounding operation. Hence the new ciphertext is said to be an encryption of message $\frac{q'}{q}m$. The input ciphertext and output ciphertext are both encryptions of the same message with different representations.

5.5 Probability Distributions

We define some distributions that are required in CKKS encryption scheme in this section. For a positive integer M , let $\phi_M(X)$ be the M -th cyclotomic polynomial of degree $N = \pi(M)$ with M -th roots of unity ζ_M . Let $\mathcal{R} = \mathbb{Z}[X]/\phi_M(X)$ be polynomial ring of integers of number field $\mathbb{Q}[X]/\phi_M(X)$.

- $\mathcal{DG}(\sigma^\epsilon)$ - For a real $\sigma > 0$, $\mathcal{DG}(\sigma^\epsilon)$ is a discrete gaussian distribution with variance σ^2 over \mathbb{Z}^N .
- $\mathcal{HWT}(h)$ - For integer $h > 0$, $\mathcal{HWT}(h)$ is the set of signed binary vectors in $[0, \pm 1]^N$ with its hamming weight equals to h .
- $\mathcal{ZO}(\rho)$ - For real $0 \leq \rho \leq 1$, $\mathcal{ZO}(\rho)$ samples ± 1 values with probability $\rho/2$ and samples 0 with probability $1 - \rho$ in $\mathcal{HWT}(h)$.

5.6 Encryption Scheme

We define CKKS leveled fully homomorphic encryption scheme for levels $0 \leq l \leq L$ where L is the maximum depth of the evaluation circuit. For scale $\Delta > 0$, let the ciphertext modulus at a level l be $q_l = \Delta^l \cdot q_0$. The HE scheme comprises following algorithms.

1. **KeyGen**(1^λ): On input 1^λ , choose a power of two $M = M(\lambda, q_L)$, integer $h = H(\lambda, q_L)$, integer $P = P(\lambda, q_L)$ and real value $\sigma = \sigma(\lambda, q_L)$
 - Sample $s \leftarrow \mathcal{HWT}(h)$ and set secret key $\mathbf{sk} = (1, s)$
 - Sample polynomial $a \leftarrow \mathcal{R}_{q_L}$ and error $e \leftarrow \mathcal{DG}(\sigma^2)$. Set public key $\mathbf{pk} = (b, a) \in \mathcal{R}_{q_L}^2$ where $b = [-a \cdot s + e]_{q_L}$
 - Sample $a' \in \mathcal{R}_{P \cdot q_L}$ and $e' \in \mathcal{DG}(\sigma^2)$. Set evaluation key $\mathbf{evk} = (b', a') \in \mathcal{R}_{P \cdot q_L}^2$ where $b' = [-a \cdot s + e']_{P \cdot q_L}$

output secret key \mathbf{sk} , public key \mathbf{pk} and evaluation key \mathbf{evk} .
2. **Encode**(z, Δ): On input $N/2$ dimensional vector $z \in \mathbb{C}^{N/2}$ and scale Δ output plaintext polynomial $m \in \mathcal{R}$.
3. **Decode**(m, Δ): On input plaintext polynomial $m \in \mathcal{R}$ and scale Δ output $N/2$ dimensional vector $z \in \mathbb{C}^{N/2}$.
4. **Enc**(\mathbf{pk}, m): On input $\mathbf{pk} = (b, a)$ and plaintext $m \in \mathcal{R}$,
 - sample $v \leftarrow \mathcal{ZO}(0.5)$, $e_0, e_1 \leftarrow \mathcal{DG}(\sigma)^2$
 - output ciphertext $\mathbf{ct} = (c_0, c_1) \in \mathcal{R}_{q_L}^2$ where $c_0 = [v \cdot b + m + e_0]_{q_L}$ and $c_1 = [v \cdot a + e_1]_{q_L}$

To keep track of the scale and the modulus level of ciphertext we represent the ciphertext as $\mathbf{cipher} = (\mathbf{ct}, \Delta, L)$

5. $\text{Dec}(\text{sk}, \text{cipher})$: On input sk , ciphertext $\text{cipher} = (\text{ct}, \Delta, L)$ output an approximate plaintext $\langle \text{ct}, \text{sk} \rangle = c_0 + c_1 \cdot s = m + e \pmod{q_L}$
6. $\text{Add}(\text{cipher}_1, \text{cipher}_2)$: On input $\text{cipher}_1 = (\text{ct}_1, \Delta, l)$, $\text{cipher}_2 = (\text{ct}_2, \Delta, l)$ output ciphertext $(\text{cipher}_{\text{add}}, \Delta, l)$ where $\text{cipher}_{\text{add}} = \text{ct}_1 + \text{ct}_2 \pmod{q_l}$
7. $\text{Mult}(\text{cipher}_1, \text{cipher}_2, \text{evk})$: On input $\text{cipher}_1 = (\text{ct}_1, \Delta, l)$, $\text{cipher}_2 = (\text{ct}_2, \Delta, l)$
 - $\text{ct}_1 = (c_0, c_1)$ and $\text{ct}_2 = (c'_0, c'_1)$ then compute $(c_0, c_1) \cdot (c'_0, c'_1) = (d_0, d_1, d_2) \in \mathcal{R}_{q_l}$ where $d_0 = c_0 \cdot c'_0$, $d_1 = c_1 \cdot c'_0 + c_0 \cdot c'_1$, $d_2 = c_1 \cdot c'_1$.
 - compute $\text{ct}_{\text{mult}} = (d_0, d_1) + \lfloor P^{-1} \cdot d_2 \cdot \text{evk} \rfloor \pmod{q_l}$output ciphertext $\text{cipher}_{\text{mult}} = (\text{ct}_{\text{mult}}, \Delta^2, l)$.
8. $\text{Rescale}(\text{ct}_{\text{mult}}, \Delta^2, l)$: On input ciphertext $(\text{ct}_{\text{mult}}, \Delta^2, l)$ output ciphertext $(\text{ct}', \Delta, l-1)$ where $\text{ct}' = \lfloor \frac{q_{l-1}}{q_l} \text{ct}_{\text{mult}} \rfloor \pmod{q_{l-1}}$. Here $q_{l-1} < q_l$ is the modulus of next lower level.

6 Microsoft SEAL

In this chapter, we will discuss the Microsoft SEAL library [SEA20] and its features. Microsoft SEAL is available in C++ programming language along with a binding for .NET(C#) and can be used in different environments such as Linux, Windows and, Android. Along with these qualities, the library is designed in such a way that it is very user friendly and can be used by anyone with very less knowledge of the complex mathematical background of the schemes. It implements leveled fully homomorphic encryption for the BFV scheme and CKKS scheme. BFV scheme allows homomorphic modular arithmetic on positive integers whereas the CKKS scheme enables homomorphic operations on real numbers. In theory, it is possible to construct FHE over polynomial rings for the BFV scheme [FV12]. However, it is observed that it is impractical to implement FHE for the BFV scheme because of the problem of plaintext datatype overflow while performing calculations on big integer values that currently limit SHE computations as well. We will further discuss this problem in the Performance Evaluation chapter. Therefore the library currently implements only the Leveled FHE scheme for both BFV and CKKS schemes. In the below sections we will discuss the different classes and methods available in the library for performing the homomorphic operations.

6.1 Encryption Parameters

In SEAL, we have different classes and methods defined for both schemes that help in Encryption, Decryption, and Evaluation. Most importantly, to create ciphertexts we define the encryption scheme parameters such as polynomial modulus degree, coefficient modulus and plain modulus (Plain modulus is not required for the CKKS scheme) with the base class EncryptionParameters. The library has a predefined set of values for the parameters based on the security level and performance of the scheme which are proven to be secure by the HomomorphicEncryption.org security standard.

- **poly_modulus_degree**: It represents the total number of coefficients a plaintext polynomial can have i.e. the degree d in the ring polynomial $\mathbb{R} = \mathbb{Z}[x]/(x^d + 1)$. This value is usually a power of 2.
- **coeff_modulus**: It represents the modulus in the ciphertext space. It is represented using the Residue number system(RNS) as a set of pairwise co-prime factors of the ciphertext modulus q . If the total coefficient modulus is integer q then it is stored as a vector of co-prime factors $(q_0, \dots, q_L, q_{L+1})$. L represents the maximum depth of the evaluation circuit. q_0 is the base modulus. q_{L+1} is the special modulus.

We define the co-efficient modulus with c++ vector data type, each value in the vector is the bit length of the prime numbers. The criteria for a valid co-efficient modulus are

- special modulus (q_{L+1}) bit length should be greater than all other moduli,
 - (q_1, \dots, q_L) bit-length should be close to each other,
 - base modulus (q_0) bit length should be same as special modulus,
 - bit length of each prime number in the vector should be at most 60 bits.
- `plain_modulus`: It represents the plaintext modulus in BFV scheme.

6.2 Context Data

After selecting the parameters for the encryption scheme, we create an instance of base class `SEALContext`. It validates the encryption parameters for correctness, evaluates their properties, performs and stores the results of several costly pre-computations. This class along with its child class `ContextData` stores the information of encryption parameters for different coefficient modulus, that will be passed on to the `Encryptor`, `Decryptor`, and `Evaluator` classes. The features of `ContextData` class are as follows.

- Modulus switching chain: The `SEALContext` on input encryption parameters create a chain of `ContextData` instances for each coefficient modulus called "modulus switching chain". Each `ContextData` in the chain can be accessed through its unique id called `parms_id()`. The `parms_id()` is a 256-bit hash of encryption parameters.
- The class also provides 2 special methods that can access context data information at 2 levels in the modulus chain directly. The `key_context_data()` corresponds to a special modulus in the modulus switching chain which stores the encryption parameters used to generate keys and `last_context_data()` corresponds to the base modulus in the chain.
- Each context data in the chain corresponds to the encryption parameters that are derived from each `coeff_modulus` in the vector. This feature of `ContextData` class is especially beneficial when we need to perform rescaling (in CKKS scheme) or modulus switch operation on ciphertexts. It is also useful to access the encryption parameters of a ciphertext through the ciphertext's `parms_id()`.

6.3 Number Theoretic Transform(NTT)

In BFV and CKKS scheme, many operations require the user to perform multiplication of polynomials with a large degree as the plaintexts and the ciphertexts are ring polynomials in the subsets of $R = \mathbb{Z}[x]/f(x)$. Hence it can be very complex when multiplied using the regular polynomial multiplication technique. Thus we transform the polynomials

to NTT form for multiplication and then apply INTT(Inverse NTT) to the result and obtain the resulting product of polynomials.

- Number Theoretic Transform is a special type of discrete Fourier transformation technique applied over the quotient rings $\mathbb{Z}/q\mathbb{Z}$ instead of complex numbers.
- In SEAL, NTT based computations are used in encryption, decryption, multiplication, relinearisation, rescaling, etc. In the CKKS scheme, both plaintext and ciphertext are stored in NTT form whereas in the BFV scheme only during encryption ciphertexts are stored in NTT form.
- Evaluator class also provides methods to transform a ciphertext or a plaintext polynomials to NTT form and also to perform Inverse NTT on the ciphertexts. This can be very useful in the case of plaintext-ciphertext multiplication if the plaintext is used for many multiplication operations.
- There are different algorithms to perform NTT based multiplication, we mainly use the Modified Iterative NTT algorithm [LN16] or four-step Cooley-Tukey algorithm [CT65].

6.4 Encryption

In SEAL, the encryption functions are defined in base class `Encryptor`. Since both the schemes are based on the RLWE problem, the encryption methods are the same except for the last step where the plaintext is added to the zero encrypted ciphertext tuple (C_0, c_1) . In the case of the BFV scheme, we need to scale the plaintext with $\lfloor q/t \rfloor$ to be added to the zero encrypted ciphertext whereas in the CKKS scheme the plaintext is directly added to the zero encrypted ciphertext.

6.4.1 Probability distributions

In this section, we discuss the classes that implement the probability distributions required to construct public key, secret key, and relinearisation key and also the distributions used in encryption.

- Random generators- Since we need to randomly sample values from distributions, the library uses a random generator defined in the base class `UniformRandomGeneratorFactory` that creates instances of `UniformRandomGenerator` class for randomly sampled seeds. This class returns a 64-bit random integer on a seed input.
- Distributions - The base class `util` defines two methods `sample_poly_normal` and the `sample_poly_ternary`
 - `sample_poly_normal` method defines the χ distribution. The parameters for this distribution are as follows

1. `noise_standard_deviation` = 3.20
 2. `noise_dist_width_mult` = 6
 3. `noise_max_deviation` = `noise_standard_deviation` \times `noise_dist_width_mult`
- `sample_poly_ternary` method defines a uniform integer distribution in $[-1, 1]$.

6.4.2 Keys

Both the BFV scheme and CKKS scheme use the same method to generate a public-key and secret key. The secret key is sampled from `sample_poly_ternary` method. The public key is obtained by sampling a from `sample_poly_ternary` method and e from `sample_poly_normal` method. Both public key and secret key are stored in NTT form.

6.4.3 Ciphertext

The ciphertext is constructed using the public key and secret key as defined above. As we already know the ciphertext of BFV scheme is $ct = ([u \cdot b + e_1 + \Delta \cdot m]_q, [u \cdot a + e_2]_q)$ and the ciphertext for CKKS scheme is $ct = ([u \cdot b + e_1 + m]_{q_L}, [u \cdot a + e_2]_{q_L})$. On input public key (b, a) and plaintext m , the `encryptor()` performs below steps to generate a ciphertext,

- The error $u \leftarrow R_3$ is sampled uniformly at random from `sample_poly_ternary` method and $e_1 \leftarrow \chi$, $e_2 \leftarrow \chi$ are sampled randomly from `sample_poly_normal` method.
- Generate a zero encryption ciphertext which is given by $ct = (c_0, c_1) = ([u \cdot b + e_1]_q, [u \cdot a + e_2]_q)$
- BFV scheme adds $\Delta \cdot m$ to c_0 to obtain the final ciphertext.
- CKKS scheme adds its plaintext m to c_0 to obtain the final ciphertext.

6.4.4 Ciphertext structure

In SEAL, the encryption parameter co-efficient modulus is represented using the Residue number system i.e. in the form of a vector of its pairwise co-prime factors. Therefore the ciphertext polynomial in ciphertext space R_q will also have multiple components corresponding to each prime factor of the co-efficient modulus. For example, a ciphertext polynomial's integer co-efficient $x \in \mathbb{Z}_q$ will have multiple components $x_i = x \bmod q_i \in \mathbb{Z}_{q_i}$, since $q = \prod_i q_i$ where q_i is a prime factor of q .

To perform homomomorphic operations on such polynomial we use the Chinese Remainder Theorem(CRT) technique to represent the polynomial coefficients corresponding to each modulus. Therefore CRT representation of a ciphertext polynomial allows the

user to perform any operation on a ciphertext by applying the operation to all CRT components simultaneously. Hence it is faster to perform computations on large polynomial rings with large co-efficient modulus through CRT technique.

The internal structure of ciphertexts in SEAL is a backing array of data type `IntArray < ct_coeff_type >`. The array elements are of type `ct_coeff_type` which is `std::uint64_t` datatype in c++. If the total number of co-efficient moduli is K , the `poly_modulus_degree` is N then the size of the backing array is $N \times K \times 8$ bytes. The additional 8 bytes are due to the size of the datatype `std::uint64_t`. If the ciphertext is $ct = (c_0, c_1)$ then it comprises 2 ciphertext polynomials. Hence its total size will be 2 times the size of the backing array.

6.5 Evaluator

The homomorphic operations on ciphertexts are performed using the class `Evaluator`. There are many methods available in this class that can perform computations based on the type of scheme or type of plaintext encoding. It defines all the methods that are required by a server to perform computations on ciphertexts such as addition, multiplication, subtraction, negation, modulus switching, rescaling, relinearisation, etc for both BFV and CKKS scheme.

6.6 BFV Implementation

For BFV scheme, the plaintext space is polynomial ring R_t and homomorphic operations on the ciphertexts are the operations in ring R_t . In practical applications of homomorphic encryption, user inputs are integers instead of polynomials. Hence the library uses encoders to convert the integers into polynomials in R_t . There are two different encoders called `IntegerEncoder` and `BatchEncoder` available for the BFV scheme in SEAL.

- `IntegerEncoder` encodes individual integers into plaintext polynomials by placing its binary digits as the coefficients of the polynomial and during decoding the plaintext polynomial is converted to an integer by evaluating the plaintext polynomial in $x=2$. For integer encoding, we use the default `plain_modulus` for the selected `poly_modulus_degree` available in the SEAL.
- `BatchEncoder` encodes a matrix of integers into a single plaintext polynomial using CRT batching technique as implemented in [SV14], [BGH13] that allows the user to operate on a matrix of integers in a SIMD (Single Instruction, Multiple Data) manner. The Batch encoder input matrix size is $2 \times N/2$ where N is the polynomial modulus degree. Therefore depending on the polynomial modulus degree, we can encode and encrypt 2 rows and $N/2$ columns of integers simultaneously. For batch encoding, we can define the bit-length of the `plain_modulus` and the library sets the plain modulus to a prime number that is congruent to $(1 \bmod 2 * \text{poly_modulus_degree})$.

6.6.1 Noise budget

Based on the encryption parameters selected, the BFV scheme has a pre-defined maximum noise level that enables correct decryption. We measure the amount of noise that is available in a ciphertext before it reaches the maximum noise for that ciphertext by a method called `invariant_noise_budget(ct)`. This method returns the available noise budget in the input ciphertext in bits. The initial noise budget available for performing computations is the maximum noise level in a freshly encrypted ciphertext and gradually decreases when computations are performed.

6.6.2 BFV Decryption

In SEAL, the implementation of the decryption algorithm for the BFV scheme is different from the original algorithm defined in [FV12]. In the original decryption algorithm, we multiply the ciphertext with t/q and perform the rounding operation, which is not possible to implement directly on the CRT components of ciphertext. Therefore we use an RNS variant implementation for decryption as shown in [HPS19] which introduces a special operation called Fast Basis Conversion which allows multiplication of t/q and rounding operation on CRT components.

6.6.3 Relinearization

The relinearization implementation is slightly different from the techniques mentioned in [FV12]. In SEAL, they have implemented a method that uses relinearization version 2 described in section 4.2.3 when `poly_modulus_degree` ≥ 4096 and uses relinearization version 1 described in 4.2.3 when `poly_modulus_degree` ≤ 2048 from [FV12]. This implementation is a combination of various techniques such as full RNS variant implementation of Relinearization as shown in paper [BEHZ16],[HPS19], and also a special prime technique introduced in Helib[Hel].

6.7 CKKS implementation

For the CKKS scheme, we use `CKKSEncoder` that encodes individual complex or real numbers and also a vector of complex or real numbers into a plaintext polynomial. The encoder method is given as `CKKSEncoder.encode(input, scale, plaintext)`. Since we do not define the plaintext modulus in the CKKS scheme, we use a scale to set the maximum bit size of the plaintext polynomial co-efficients. The CKKS encoder allows the user to perform homomorphic operations on the plaintexts in a SIMD(Single Instruction, Multiple Data) manner. If the input is a single complex or real number, then the number is repeated for $N/2$ times in the vector and is encoded in a plaintext. The CKKS scheme plaintext is stored in the NTT form for each of the prime numbers in the `coeff_modulus`. Based on the position of a prime value in the co-efficient modulus vector, the plaintext is assigned a `parms_id`.

6.7.1 Rescaling

In Evaluator class there are different methods available to perform rescaling. All these methods call one common method `mod_switch_scale_to_next` which performs the actual computations in rescaling. We extract ContextData information of the ciphertext that has to be rescaled and determine the new co-efficient modulus and the encryption parameters of the resulting ciphertext. This context data information is passed to a method `divide_and_round_q_last_ntt_inplace` in base class RNSTool where we divide our ciphertext with the last modulus in the co-efficient modulus vector and perform a rounding operation to obtain a new rescaled ciphertext in NTT form. Finally, we also set a new scale for the rescaled ciphertext. The new scale is obtained by dividing the old scale with the last prime number in the co-efficient modulus vector before it was rescaled.

6.7.2 Modulus switching

There are various methods to perform modulus switching in SEAL. We can either define a particular `parms_id` in the modulus chain to switch or switch the modulus to the next lower level in the modulus chain. Modulus switching computation is implemented in the method `mod_switch_drop_to_next`. It drops or removes one of the prime numbers in `coeff_modulus` and switches the ciphertext to the next level in the modulus chain without modifying the plaintext like in rescaling. We also do not modify the scale of the switched ciphertext in this case.

6.7.3 Normalisation of ciphertext scale:

Addition and Subtraction of ciphertexts in the CKKS scheme require the input ciphertext's encryption parameters to match i.e. `parms_id` and scale of the ciphertexts be same. Even if the moduli in the coefficient modulus vector have the same bit-length, the prime numbers are not equal. Hence when we perform rescaling, the resulting scale will be an approximate scale that is close to the expected scale. Hence we cannot add the ciphertexts that were rescaled. To solve this problem, we normalize the scales of the ciphertexts by manually setting the scale of all ciphertexts to a constant scale that is closest to all the ciphertexts scale.

6.8 Advantages And Disadvantages

In this section we explain what are some of the limitations of BFV and CKKS scheme that we have observed.

6.8.1 BFV scheme

Advantages

1. We can perform modular arithmetic homomorphically.

2. We can perform exponentiation operation on the ciphertext easily with the available evaluator methods in SEAL.
3. We can perform parallel evaluation on many plaintexts simultaneously using batch encoding.
4. We can reduce the ciphertext size in turn reducing the noise level in the ciphertext by modulus switch operation.

Limitations

1. Cannot perform division operation on the ciphertexts.
2. Can only work with integer values. Hence limiting the scheme's practical applications as most of the mathematical computations are performed on real numbers.
3. Since plaintexts are represented in modulus t polynomial ring, there is a possibility of data type overflow when we perform operations on large integer values. Hence we can only work with limited range of input values. The current implementation of BFV scheme in SEAL does not support a plaintext modulus larger than 61 bits.
4. We can only perform Leveled fully homomorphic evaluations with BFV scheme currently as FHE scheme has not been implemented in the library because of the issue of the plaintext datatype overflow which results in failure of correct decryption even if there is plenty of noise budget available in the ciphertext.

6.8.2 CKKS scheme**Advantages**

1. We can perform approximate arithmetic homomorphially.
2. We can perform parallel evaluation on many plaintexts simultaneously by encoding them with CKKSEncoder.
3. Precision loss in the decrypted plaintext due to homomorphic evaluation is bounded by the maximum depth of evaluation of the scheme.

Limitations

1. Loss of significance- Due to increase in the noise in the ciphertext, we perform rescaling and rounding of the decrypted ciphertext, which leads to loss of significant digits of a decrypted plaintext. Eg: If message $m = 5.0000456$ then the approximate plaintext $m' = 5.000$, here we lose the significant digits "456". Hence we lose precision of results for very small scales.

2. It is difficult to implement FHE scheme in current CKKS scheme, as it relies on rescaling to perform leveled homomorphic evaluation. Due to this there is not enough co-efficient modulus left at the end of the LFHE evaluation that it can perform an expensive computation such as bootstrapping where it has to evaluate its own decryption circuit homomorphically.
3. Currently in the SEAL library, we can only perform evaluations in CKKS scheme for only 128 bit security level.

7 Performance Evaluation

In this chapter, we discuss the homomorphic functions that we were able to evaluate in BFV and CKKS schemes. Based on the metrics used in our testing we will compare the performances of both schemes and explain the results of our evaluation. Initially, we compare the general elements of both the schemes such as plaintexts, ciphertexts, the public key, and the secret key for different security level encryption parameters and present our results for each metric. Further, we also compare our performance evaluation results for both the schemes for each function and present our results.

7.1 Experimental Setup

In this section we discuss about the initial setup of our testing. We use Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Core(s), 8 Logical Processor(s) with a Linux OS. The Microsoft SEAL C++ library is built with cmake 3.15.2 and a Clang++ 6.0 compiler. We have used below defined encryption parameters for our testing.

- The standard deviation(σ) for the χ distribution is 3.20
- Bound on the distribution is $B = 6$
- The secret key s is a ternary polynomial whose coefficients $(-1, 0, 1)$ are sampled uniformly at random.

<i>poly_modulus_degree(n)</i>	1024	2048	4096	8192	16384	32768
<i>plain_modulus(t)</i>	12289	786433	786433	786433	786433	786433
<i>max_coeff_modulus_bit_length(q)</i>	27	54	109	218	438	881

Table 7.1: Security level 128

<i>poly_modulus_degree(n)</i>	1024	2048	4096	8192	16384	32768
<i>plain_modulus(t)</i>	12289	786433	786433	786433	786433	786433
<i>max_coeff_modulus_bit_length(q)</i>	19	37	75	152	300	600

Table 7.2: Security level 192

$poly_modulus_degree(n)$	1024	2048	4096	8192	16384	32768
$plain_modulus(t)$	12289	786433	786433	786433	786433	786433
$max_coeff_modulus_bit_length(q)$	14	29	58	118	237	476

Table 7.3: Security level 256

7.2 Evaluation Criteria

We measure the performance of the SEAL library with two metrics computation time (in seconds) and storage space (in KB). Computation time is the average time taken to evaluate a single homomorphic operation or to execute a single function in Evaluator class calculated over 10 iterations. Storage space is computed by calculating the size of the backing array of the ciphertext.

7.3 Test Functions

Based on the available methods in Evaluator class , we have tested the performance of SEAL for below functions

- Evaluate sigmoid function in CKKS scheme.
- Evaluate a simple Deep neural network in CKKS scheme
- Evaluate sign function in BFV scheme.
- Evaluate single neuron of discrete neural network in BFV scheme.

7.4 General Results

In this section we discuss the results of our general performance evaluation for each metric.

7.4.1 Storage space

We measure the storage space for the public key, secret key, ciphertext, and plaintext data for BFV and CKKS schemes for each security level and compare their results. The public key, secret key, ciphertext, and plaintext data are stored in the form of a special backing array called `IntArray`. Hence their storage space is determined by the size of the backing array.

Results

Plots 7.1, 7.2,7.3, 7.4 show the results of our test. Since we use the same encryption parameters (Table 7.1, 7.2, 7.3) for both BFV and CKKS scheme, we obtain similar storage space for both schemes.

1. However, all of them follow a common trend which is an increase in storage space with an increase in polynomial modulus degree. The reason for this is,
 - The size of the backing array of ciphertext is obtained by computing the product of `poly_modulus_degree`, number of primes in `coeff_modulus` and the number of ciphertext elements in a ciphertext.
 - The size of the backing array of the public key, secret key, and plaintext are also derived similarly using `poly_modulus_degree` and `coeff_modulus`.
2. Their storage space decreases with an increase in the security level because the `max_coeff_modulus_bit_length` becomes smaller for every higher security level.
3. Plot 7.4 compares the plaintexts obtained from the batch encoding technique of the BFV scheme and the CKKS encoding technique of the CKKS scheme. The BFV scheme plaintext size is less than 32KB and is the same for all security levels because
 - We have chosen same `plaintext_modulus` for all security levels whereas the CKKS scheme plaintexts are stored in NTT form for each `coeff_modulus` primes.
 - Their size is obtained by multiplying the `poly_modulus_degree` and number of primes in `coeff_modulus`. Hence their plaintext size is different for each `poly_modulus_degree`.

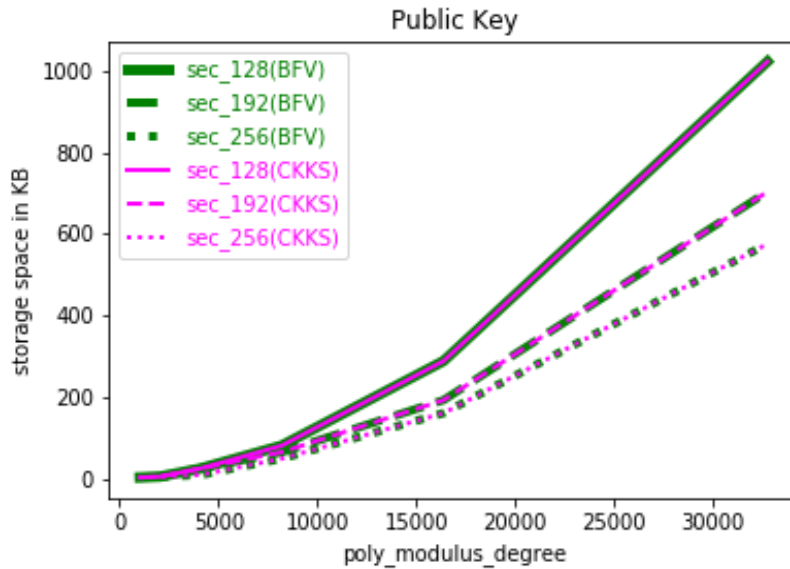


Figure 7.1: Public key size

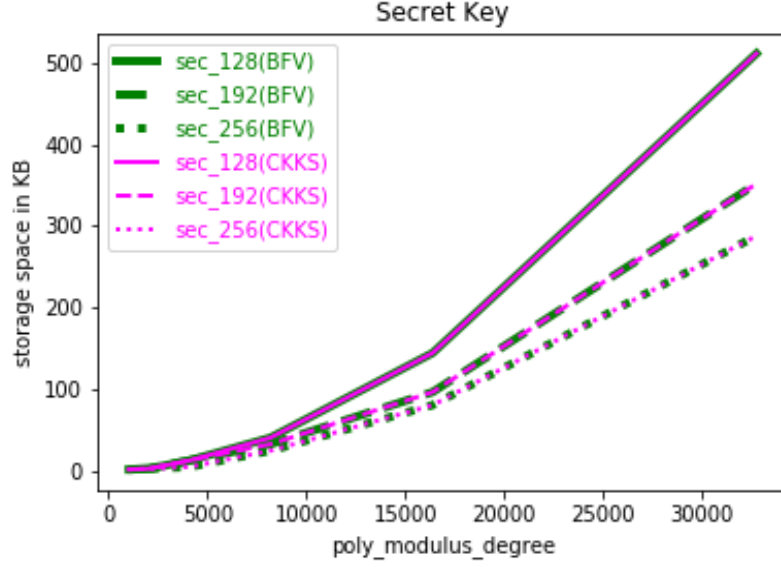


Figure 7.2: Secret key size

7.4.2 Computation time

We measure the computation time for both schemes at different security levels and compare their results. We perform two tests, the first test is to compare the average computation time for the operations such as encryption, decryption, addition, etc between the schemes and the second test is comparing the average computation time for the operations between the security levels.

Results

1. Average computation time for all the operations decreases gradually with an increase in the security level for both the schemes. That implies the highest computation time is for security level 128 bits and the lowest computation time is taken for highest security level 256 bits. This behavior can be seen in figure 7.5. This is because `max_coeff_modulus_bit_length` of security level 128 bits is greater than the `max_coeff_modulus_bit_length` of security level 256 bits which imply the coefficient modulus of 128-bits security level is bigger than the coefficient modulus of 256-bits security level, hence the coefficient values of former ciphertext polynomial are bigger than the co-efficient values of latter ciphertext polynomial.
2. We also observe that the computation time for encryption, decryption, multiplication, relinearisation, and rotation operations is higher for the BFV scheme compared to the CKKS scheme in all security levels whereas computation time for encoding and decoding operations is higher for the CKKS scheme. This behavior can be seen in figure 7.6.

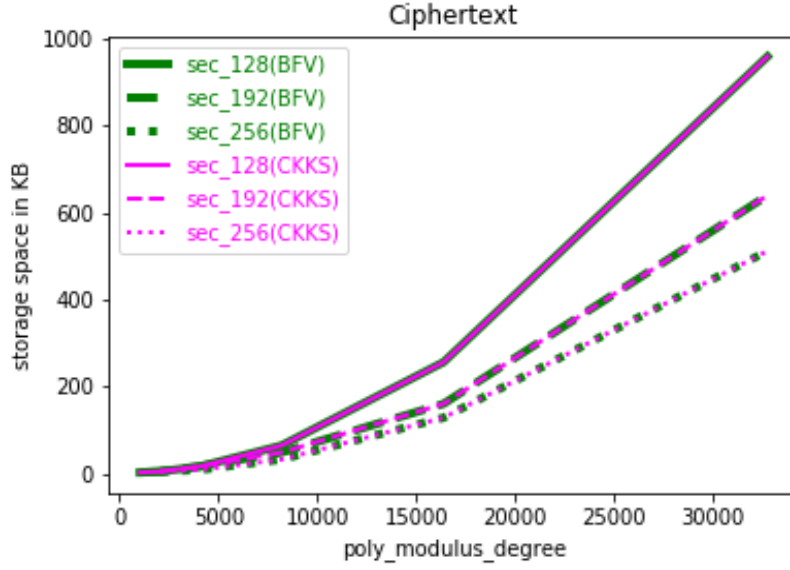


Figure 7.3: Ciphertext size

7.5 Sigmoid function

In this section, we discuss the performance evaluation results of evaluating sigmoid function in the CKKS scheme. The Sigmoid function is a mathematical function that has an S-shaped curve. It is a continuous function that takes input real values in the range $[-\infty, +\infty]$ and returns a value in the range 0 to 1. Many functions are sigmoidal in nature for example logistic function, hyperbolic tangent function, etc. We are interested in the logistic function as it is very popular among machine learning algorithms. It is used as a non-linear activation function in the training and validation of different neural networks. The logistic function is defined as- $\sigma(x) = \frac{e^x}{1+e^x}$

7.5.1 Test setup and results

As the input to the sigmoid function is a real number, we were able to evaluate the sigmoid function using the CKKS scheme. Since we need to compute exponential values in the sigmoid function, we observed that it is not feasible to implement the actual function in the CKKS scheme because only a fixed depth multiplication circuit can be evaluated. Hence we have implemented an approximate sigmoid function using the least square approximation method that uses a fixed depth multiplication circuit for evaluating a sigmoid function.[KSW⁺18]. (We can perform non-linear classification using a sigmoid function). The results of our tests are given in 7.4.

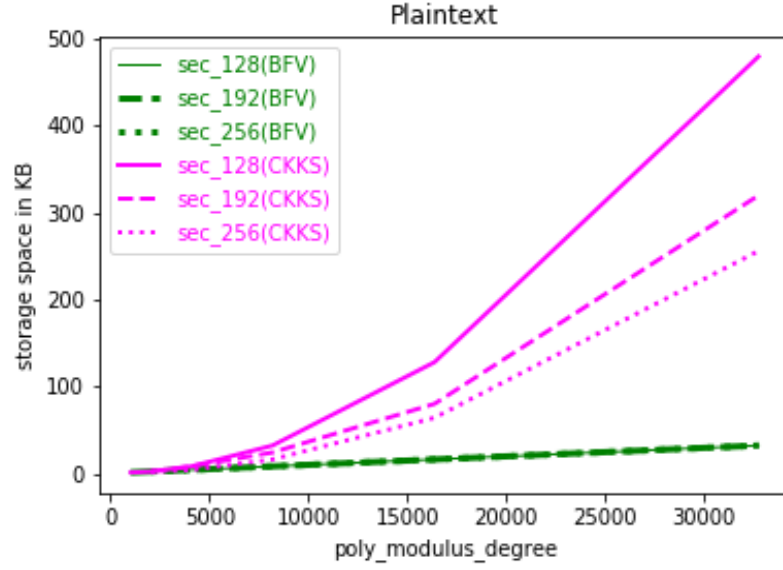


Figure 7.4: Plaintext size

Input range	Computation time	Storage space	HE average Error	poly_modulus_degree
$[-8, 8]$	0.98s	256KB	2.34943×10^{-15}	32768
$[-8, 8]$	0.54 s	128 KB	-1.86763×10^{-10}	16384
$[-8, 8]$	0.14s	48KB	0.000123001	8192

Table 7.4: sigmoid function

To implement a sigmoid function of input range greater than $[-8, 8]$, we need to use the least square approximation polynomial of degree greater than 7 which implies we need bigger encryption parameters that will allow evaluation of polynomial of degree greater than 7.

7.6 Trained deep neural network evaluation

Machine learning is a branch of artificial intelligence that deals with training machines or systems to perform a certain task with minimal human intervention without explicitly programming the task. There are various ways to train a machine to obtain artificial intelligence. Deep learning is a field in machine learning which deals with algorithms such as deep neural network, convolutional neural network, recurrent neural networks that are inspired from the human brain that aids a machine with intelligence.

The deep neural network comprises three types of layers- An Input layer, an output layer, and Hidden layers. The Hidden layer and output layer each comprise of Perceptron. Perceptron is an artificial neuron(node) based on the model of biological neuron which

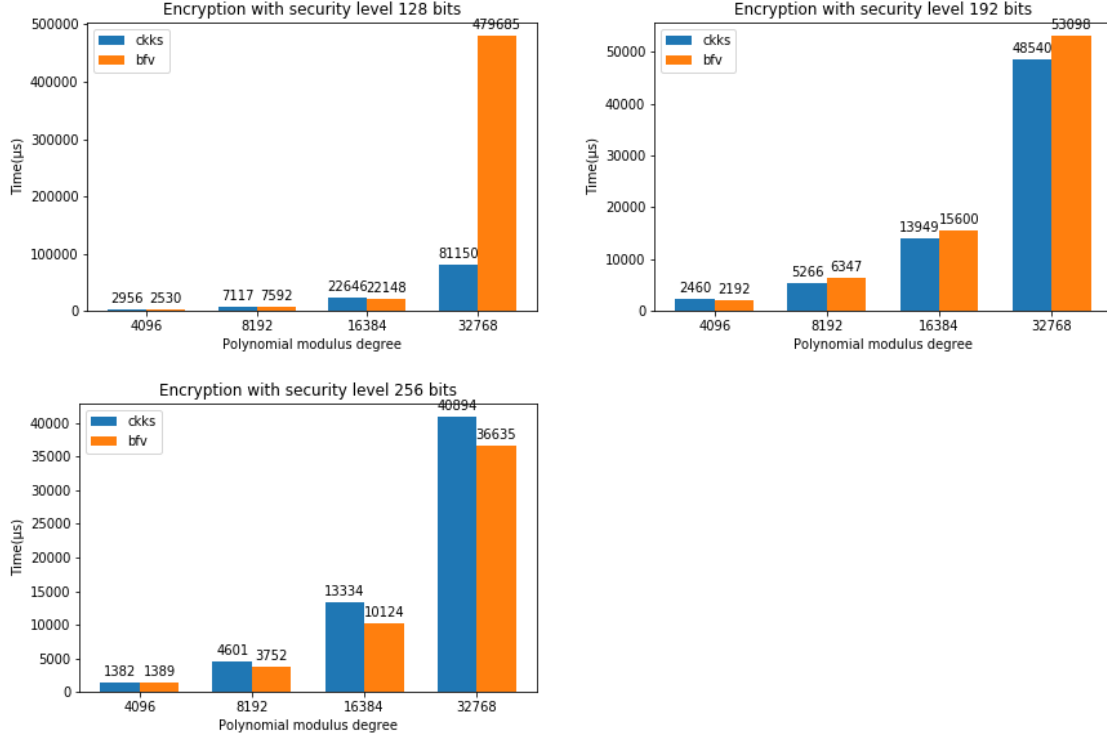


Figure 7.5: Comparison between security levels

takes inputs from input layer or previous layer, weights, and biases. It computes the product of inputs and weights and adds them to the bias and passes the sum to a non-linear activation function that generates an output that is helpful to decide for that particular task. Deep neural networks are modeled to make a decision or classify data through the training process. The neural network acts as a mathematical function that is trained to map a set of input values to output values. During the training process, the weights and biases of the network are updated in multiple iterations such that the network produces an output with a minimized error. Once the training is completed, we test the network performance on test data using the trained weights and biases. The network performance can be measured with a metric validation accuracy. It is a measure of the total number of input samples from the test dataset that were correctly mapped to the output.

7.6.1 Test setup and results

In our implementation in the CKKS scheme, we evaluated a simple pre-trained deep neural network that consists of 1 input layer with 2 inputs and 1 hidden layer with 2 nodes, and 1 output layer with 1 node. This network is trained to classify input data into 2 classes. The output layer generates the posterior probability of the given input

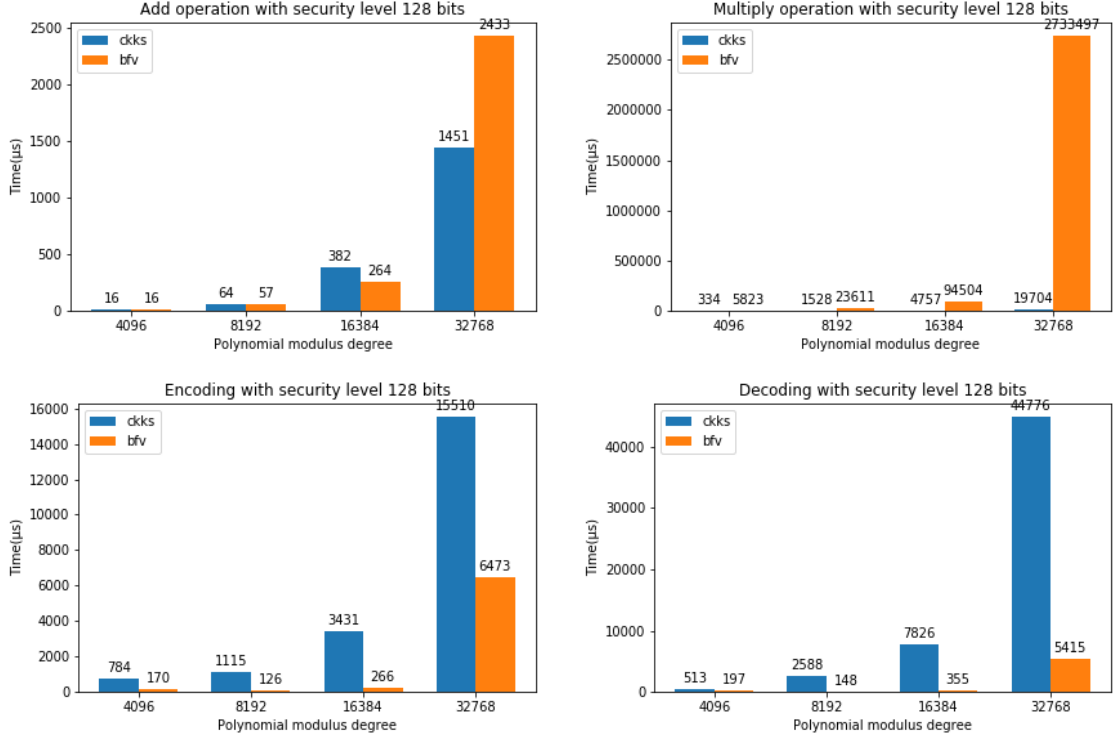


Figure 7.6: Comparison between schemes

belonging to a certain class. In our case, the decision boundary is 0.5. If the network output for any input is greater than or equal to 0.5, it is classified as a class 2 input sample and if the output is less than 0.5 it is classified as a class 1 input sample. We have used the sigmoid function defined in section 7.5 as the non-linear activation function in the network. The non-encrypted validation accuracy of this dnn is found to be equal to the encrypted validation accuracy which is equal to 91 percent. The results of our tests are given in 7.5.

It is possible to evaluate a deep neural network with more hidden layers, by selecting different encryption parameters that can enable the scheme to evaluate a larger depth multiplication circuit.

7.7 Sign function lookup table

We have implemented a function lookup table that is used to approximate a bounded activation function. To implement this function lookup table we have used the algorithm from [TGP19]. This algorithm can be used to implement any function over a defined bounded interval. It can be used to implement sign function, sigmoid function, Rectified

polynomial modulus degree	coefficient modulus / modulus length	N/w Inputs		N/w Outputs		N/w evalua- tion
		Time	Storage space	Time	Storage space	Time
16384	25-bits / 17	1.04 s	4096 KB	5.30 s	96 KB	2.04 s
32768	50-bits / 17	2.36 s	8192 KB	11.01 s	192 KB	4.39 s

Table 7.5: Deep neural network

Linear Units(ReLU), tanh, and other bounded functions. The idea of this algorithm is, the function lookup table is a vector where we store the pre-computed output of our activation function for a range of input values, such that when given an encrypted input the lookup table returns the encrypted output of our activation function.

7.7.1 Test setup and results

We have implemented the sign function in the BFV scheme. The sign function returns 3 values $[-1, 0, 1]$ depending on the input.

$$\text{sign}(x) = \begin{cases} -1 & \text{if, } x < 0 \\ 0 & \text{if, } x = 0 \\ 1 & \text{if, } x > 0 \end{cases}$$

In our implementation we create a plaintext function lookup vector $\vec{F} = [f(x_1), \dots, f(x_{N+1})]$ that stores $\text{sign}(x_i)$ at each index i in the input range $[-N/2, N/2]$. The size of our vector is $N + 1$, where $N = 2^n$ for $n > 0$. We also define an encrypted vector $E[\vec{x}]$ that contains the encrypted test index v . It is an index in the \vec{F} that contains the output of sign function for the test input in $[-N/2, N/2]$. We have shown 2 ways of implementing the lookup table.

Algorithm 1

The inputs to algorithm 1 are an encrypted vector $E[\vec{x}]$, integer n , plaintext vector \vec{F} . We perform the right rotation operation (r_{2^s}) on the encrypted vector $E(\vec{x})$ s number of times to obtain a vector with all values equal to 0 except for test index v . This algorithm returns 2 ciphertexts numerator and denominator. If we decrypt the ciphertexts and

divide the numerator by denominator we obtain the result of $f(x_v) = \text{sign}(x_v)$ stored at test index v . We were able to implement this algorithm using both Integer and Batch encoder. Table 7.6 displays the results of algorithm 1. It is possible to implement a sign function of a higher range like $[-64, 64]$ with batch encoder whereas with integer encoder we can only implement for input range $[-8, 8]$. For greater input range, we do not get the expected results in the Integer encoder because of the plaintext overflow issue. Because the noise budget is lacking in the ciphertext, we cannot implement this algorithm for lower polynomial modulus degree 8192 or 4096 whose coefficient modulus is less than 400 bits with the batch encoder.

Algorithm 1 Encrypted lookup table with division

Require: $E(\vec{x}), n, \vec{F}$
for $s = 0$ to n **do**
 $E(\vec{x}') = E(r_{2^s}(x))$
 $E(\vec{x}) = E(\vec{x}') \times E(\vec{x})$
end for
return $\frac{\vec{F} \cdot E(\vec{x})}{\sum_j E(\vec{x}_j)}$

Algorithm 2

The inputs to algorithm 2 are the encrypted vector $E[\vec{x}]$, integer n , plaintext vector \vec{F} and a plaintext vector \vec{P} . The plaintext vector \vec{P} contains input values to perform modular multiplication with $E[\vec{x}]$ that will result in a vector of zeros except for the test index v which contains integer 1 as a result of modular multiplication. To perform modular multiplication we use plaintext modulus and based on this modulus we define our vector \vec{P} . We were able to implement the sign function of range $[-32, 32]$ using a plain modulus 786433 set by the library. As a result of this algorithm, we obtain a single ciphertext that outputs $E(f(x_v))$, the encrypted result of the sign function. The results of this implementation are given in Table 7.7.

Algorithm 2 Encrypted lookup table with modulus operation

Require: $E(\vec{x}), n, \vec{F}, \vec{P}$
for $s = 0$ to n **do**
 $E(\vec{x}') = E(r_{2^s}(x))$
 $E(\vec{x}) = E(\vec{x}') \times E(\vec{x})$
end for
 $E(\vec{x}) = E(\vec{x}) \times \vec{P}$
return $E(\vec{x}) \times \vec{F}$

Encoder	polynomial modulus	Input range	Computation time	Input storage	output storage space
Integer	32768	[-8, 8]	48.38 s	15.93 MB	64 KB + 64 KB
Batch	32768	[-8, 8]	50 s	8.43 MB	64 KB + 64 KB
Batch	16384	[-16, 16]	21.45 s	8.25 MB	32 KB + 32 KB
Batch	32768	[-16, 16]	106 s	15.93 MB	64 KB + 64 KB
Batch	16384	[-32, 32]	48.858 s	16.25 MB	32 KB + 32 KB
Batch	16384	[-64, 64]	118.248 s	32.25 MB	32 KB + 32 KB

Table 7.6: Algorithm1 results

Security level	polynomial modulus	Computation time	Input storage	output storage space	Noise budget
128	16384	51 s	17.265 MB	32KB	20 bits
128	32768	262.22 s	62.96 MB	64KB	26 bits
192	16384	33.96 s	11.17 MB	32 KB	0 bits
192	32768	148.87 s	42.65 MB	64 KB	25 bits
256	16384	25.09 s	9.14 MB	32 KB	0 bits
256	32768	117.26 s	34.53 MB	64 KB	23 bits

Table 7.7: Algorithm 2 results

7.7.2 Remarks

The original storage space of output ciphertext without modulus switch operation was 960 KB. After performing modulus switch operation on the resultant ciphertexts, the storage space of ciphertexts is reduced to 64 KB for a polynomial modulus degree 32768 and 32 KB for polynomial modulus degree 16384. However the computation time is same in both the cases.

7.8 Trained discrete neural network

Normally the inputs, weights, and biases used in neural networks are real numbers. On the other hand we can only perform modular arithmetic operations with integer values in the BFV scheme. Therefore we have implemented a neural network that can perform its computations on integer values while performing classification like regular networks. Our implementation is inspired by the discrete neural network implementation of [BMMP18]. To implement a discrete neural network we need to convert the real weights, biases, and inputs into discrete values. We have used the processWeights method from [BMMP18] to perform discretization.

$$\text{processWeights}(w, \tau) = \tau \cdot \left\lfloor \frac{w}{\tau} \right\rfloor$$

This method takes 2 inputs. A real-valued weight or bias or network inputs w and a scaling value $\tau \in \mathbf{N}$. The values returned by this method are discretized values that can be used to perform the perceptron computations. The discrete perceptron output is sent to the activation function. We have chosen the sign function as the activation function since it generates integer output in range $-1, 1$. It is implemented using the encrypted lookup table algorithm 2. We were able to evaluate a single neuron in a Discrete neural network using the BFV scheme to perform linear classification of 10 input samples into 2 classes. (We can only perform linear classification with sign activation function). The results of our tests are given in Table 7.8.

Security level	Polynomial modulus	HE accuracy	input computation time	neuron run-time	Input storage space	Output storage space	Noise budget
128	16384	100%	1.4 s	55 s	1280 KB	32 KB	20 bits
128	32768	100%	4.64 s	274.5 s	4800 KB	64 KB	26 bits
192	16384	0%	1 s	79.44 s	800 KB	32 KB	0 bits
192	32768	100%	3.16 s	156.17 s	3200 KB	64 KB	25 bits
256	16384	0%	1.88 s	65.26 s	640 KB	32 KB	0 bits
256	32768	100%	3.28 s	110.61 s	2560 KB	64 KB	23 bits

Table 7.8: Discrete neuron results

7.9 Discussion

In this section we will discuss about the issues that we faced while working on the library and how we resolved them.

- **Scaling problem-** This is a very common problem when we perform operations in the CKKS scheme. Since addition, multiplication, or subtraction operations between ciphertexts or between ciphertext and plaintext with different scales will generate false results because of different floating-point values, the library throws `std :: invalid_argument : scales mismatch`. While implementing the sigmoid function in the CKKS scheme we add all the product ciphertexts in the last step. To overcome this issue, we normalized the scales of ciphertext by choosing a new scale that is closest to the scales of all the ciphertexts to be added. This can be achieved, if we set the same bit-length for all the primes in the co-efficient modulus and also use the scale equal to the 2^p where p is the co-efficient modulus prime bit length to generate a new ciphertext. So that, the post multiplication scales are always multiples of the co-efficient modulus bit-length and we can perform rescaling such that all the product ciphertexts will have nearly equal scales. This method can also be used to regulate the allowed multiplication circuit depth of the implementation so that it can evaluate a maximum depth circuit. Even when we implemented

the general version of the deep neural network we had to fix the scale of the bias ciphertext such that its scale will be nearly equal to the scale of product ciphertext of weights and inputs ciphertexts. If we know the depth of the network, we can calculate the multiplication depth at each layer and fix the scale of bias before computations.

- **Transparent ciphertext-** Transparent ciphertexts are the ciphertexts that do not require a secret key to decrypt the ciphertext. In these ciphertexts, the second polynomial(c_1) is zero. When we perform addition, subtraction, multiplication, or modulus switch operations the library checks if the resulting ciphertexts are transparent and throws the **std:: logic error : "resulting ciphertext is transparent"**. In our sign function lookup table implementation of the BFV scheme, when we multiply the plaintext vector of a function lookup table that has an encoding of integer value '0' with the encrypted vector, the library throws this error. There are 2 ways to override this problem.
 - When we build the SEAL library we can set the `SEAL_THROW_ON_TRANSPARENT_CIPHERTEXT` to 'OFF' in the advanced CMake options.
 - Instead of using `multiply_plain` to perform multiplication of '0' encoded plaintext with a ciphertext, we can encrypt the plaintext and perform regular multiplication of 2 ciphertexts.
- **Parameters mismatch-** This is also a very common problem in CKKS scheme. When we add, subtract or multiply two ciphertexts or a ciphertext and a plaintext, if their `parms_id` does not match library throws the error **std::invalid argument : "encrypted1 and encrypted2 parameter mismatch"**. This error can be resolved by performing modulus switch operation on the ciphertext or the plaintext which is at a higher level in the modulus chain to the `parms_id` which is at a lower level. Therefore we obtain both the inputs at the same modulus level with same `parms_id`.
- **Incorrect decryption in a called function-** When we call a function in the main program and pass a ciphertext as an argument in this function and decrypt the ciphertext, we observe that the decrypted results are different from the original input value. This problem can be resolved by defining the public key, secret key, and the relinearisation key as a *unique_ptr* outside the main program and then use the pointers to the defined keys in the called function to perform encryption, decryption, or relinearisation on the ciphertext. This technique always generates correct results in the called function.
- **Plaintext overflow in Integer encoder:** It is the case of integer overflow where an arithmetic operation attempts to create a big integer value that cannot be represented by its data type. In the BFV scheme, we define our Plaintexts datatype using a fixed-width integer datatype `int64_t` whose bit-length is 64 bits. The

library throws the exception **std::invalid argument: "output out of range"** when the `decode_int64()` method of `IntegerEncoder` tries to decode a plaintext value whose coefficients bit length is greater than 64 bits. This is a limitation of the Integer encoder in BFV scheme. Since we perform operations only on the integer data type. Hence irrespective of the noise budget in the ciphertext, if there is an integer overflow, we cannot obtain correct results after decryption. Thus Integer encoder has been removed from recent versions of SEAL as it is found to be inefficient for practical applications.

- **Scale out of bounds:** In the CKKS scheme, when we multiply 2 ciphertexts or a ciphertext and a plaintext the scale of the resulting ciphertext will be the sum of the scales of the 2 inputs. Hence to make sure that the scale of the resulting ciphertext does not exceed the total co-efficient modulus bit length, the library throws the exception **invalid argument: "scale out of bounds"** when the scale of the ciphertext exceeds the current total co-efficient modulus bit length. This issue is found when we have rescaled the ciphertext many times and reached level 1 in the modulus switching chain and we try to perform multiplication of ciphertexts at this level. Consider the chosen coefficient modulus chain is $Q = (30, 30, 30, 30, 30, 30, 30, 30)$ and we are trying to evaluate a depth 6 circuit on a fresh ciphertext with initial scale equals 30 bits. After every multiplication, the ciphertext scale is doubled hence we rescale it to the next level to reset the scale back to 30 bits. Once we reach level 5 in the chain(excluding the special modulus) and we still have one more multiplication to perform the library throws this error to warn the user that the current available total co-efficient modulus bit length is 30 bits and the resulting scale of our multiplication will be 60 bits which are greater than total co-efficient modulus bit length. This issue can be resolved by choosing a coefficient modulus chain of length $D + 2$ levels for a depth D circuit. This is because the first modulus in the chain is the special modulus that is used for key generation purposes and the last modulus in the chain is the base modulus that is used for decryption purposes hence only the remaining modulo are available for rescaling.

8 Conclusion and Future Work

In conclusion, BFV and CKKS schemes have many applications. We can also evaluate basic statistical operations such as mean, variance, standard deviation, covariance, correlation on n ciphertexts. Since we can perform matrix operations such as scalar matrix multiplication, we can evaluate linear transformation, transpose, and inverse of matrix(in CKKS) and also evaluate AES algorithm. We can also compute many steps in Neural network training such as forward differentiation, reverse differentiation, gradient descent and complex operations such as Principal component analysis, linear discriminant analysis and linear regression in CKKS scheme with bigger encryption parameters that can evaluate larger depth multiplication circuit. The open problem in both BFV and CKKS schemes currently is finding a solution to implement FHE scheme, to benefit more from both the schemes practically.

Bibliography

- [BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.
- [BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–13. Springer, 2013.
- [BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
- [BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.
- [Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.
- [CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*, pages 3–33. Springer, 2016.
- [CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.
- [CT65] 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.
- [DM15] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

- [FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.
- [G⁺09] Craig Gentry et al. *A fully homomorphic encryption scheme*, volume 20. Stanford university Stanford, 2009.
- [Hel] Helib.
- [HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.
- [KSW⁺18] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e19, 2018.
- [LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.
- [LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Cryptographers' Track at the RSA Conference*, pages 319–339. Springer, 2011.
- [MR09] Daniele Micciancio and Oded Regev. *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [RAD78] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.
- [Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
- [SEA20] Microsoft SEAL (release 3.6). <https://github.com/Microsoft/SEAL>, November 2020. Microsoft Research, Redmond, WA.
- [SV14] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.
- [TGP19] Patricia Thaine, Sergey Gorbunov, and Gerald Penn. Efficient evaluation of activation functions over encrypted data. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 57–63. IEEE, 2019.
- [VDGHV10] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.