pyFHE - A Python Library for Fully Homomorphic Encryption

by

Saroja Erabelli

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 14, 2020

Certified by .....
Vinod Vaikuntanathan
Associate Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

pyFHE - A Python Library for Fully Homomorphic Encryption

by

Saroja Erabelli

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Fully homomorphic encryption (FHE) schemes often entail complex lattice operations and error associated with addition and multiplication, making them a challenge to implement. While a few lattice cryptography libraries exist in C++, there is no such library in Python, a language which allows simplicity and readability, making it ideal for prototyping. Many such libraries also do not include bootstrapping, the most complicated operation of FHE schemes. We present a new Python library pyFHE for fully homomorphic encryption schemes, which currently includes the Brakerski-Fan-Vercauteren (BFV) scheme, the Cheon-Kim-Kim-Song (CKKS) scheme, and bootstrapping for CKKS.

Thesis Supervisor: Vinod Vaikuntanathan
Title: Associate Professor

Acknowledgments

I would first like to thank Vinod Vaikuntanathan, my thesis advisor, and PhD candidate Leo de Castro for all of their feedback and support throughout this year. They were both very helpful in providing me guidance when I was stuck. Leo also met with me regularly to help me debug and provided many valuable insights throughout the process. I would not have been able to complete this project without his help.

I would also like to thank Chiraag Juvekar for helping me get started with this project and providing direction on the components needed to design this library.

I would like to thank my academic advisor, Ron Rivest, for his perspective and advice throughout my MIT experience. I am extremely grateful for his support and positivity.

Thank you to all my friends at MIT, for their endless support and words of encouragement.

A final thank you goes to my mom, my dad, and my brother for their unwavering love and belief in me throughout the last 23 years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the biggest problems in cryptography today is being able to perform computations on encrypted data. Encryption schemes that allow us to do this are known as *homomorphic encryption* schemes, where if we are given the encryption of a and the encryption of b, we can compute the encryption of $a + b$ and the encryption of $a \cdot b$. One example of an application is if a user wishes to search stock data, but they wish to keep their search queries private, they can do so if the search engine had a homomorphic encryption scheme in place for queries [15]. Homomorphic encryption also has many other applications in medical and genomic data. [7, 13, 15]

For many years, there existed partial homomorphic encryption schemes, such as RSA and Paillier [1], where given the encryption of a and the encryption of b, you can compute the encryption of $a + b$ or the encryption of $a \cdot b$, but not both. However, in 2009, Craig Gentry designed the first *fully homomorphic encryption* (FHE) scheme, allowing us to compute both the encryption of $a + b$ and the encryption of $a \cdot b$ [9]. Since then, a number of more efficient FHE schemes have been designed, including the widely known Brakerski-Fan-Vercauteren (BFV) and Cheon-Kim-Kim-Song (CKKS) encryption schemes [2, 6]

The pyFHE library provides Python implementations of the two well known FHE schemes, the BFV and CKKS encryption schemes [2, 6], including CKKS boot-

strapping [4]. Some existing implementations of FHE schemes include the PALISADE library, Microsoft's Simple Encrypted Arithmetic Library (SEAL), and HELib [16, 19, 12], all of which are in C++ and designed for maximal performance. These libraries also do not currently include bootstrapping for the CKKS scheme, although they may be adding it in the future. The primary purpose of our library is to provide readable code for FHE schemes that researchers can easily and quickly build on or use to test correctness for their own optimizations and variants. We designed pyFHE to achieve the following design goals:

1. *Create a readable library for lattice cryptography* that is easy to understand and make changes to.

2. *Create a modular design* to make it easy to add new encryption schemes to the library.

3. *Write reliable and correct code* with unit testing for all components of the library.

4. *Provide usable code* where it is straightforward how to use and test these encryption schemes.

With these design goals in mind, we chose to design this library in Python, to make the code as simple and readable as possible, while sacrificing the efficiency that comes with C++ implementations [16, 19, 12]. The library can be found in the py-fhe Github repository [17].

1.1 Roadmap

In Chapter 2, we introduce the mathematical background and notation assumed throughout this paper. In Chapter 3, we give an overview of the BFV encryption scheme as implemented in pyFHE. In Chapter 4, we give an overview of the CKKS encryption scheme and bootstrapping as implemented in pyFHE. In Chapter 5, we describe the library architecture and explain in more detail how to use it. In Chapter 6, we describe how to choose parameters for BFV and CKKS, and in chapter 7, we give performance results for a few of these parameter sets.

Chapter 2

Preliminaries

2.1 Basic Notation

Let $\lfloor x \rceil$ denote the nearest integer to x, and $\lfloor x \rfloor$ and $\lceil x \rceil$ denote rounding down and rounding up, respectively. For an integer q, we define $\mathbb{Z}_q$ to be the space $\mathbb{Z} \cup (-q/2, q/2]$, and we use $[z]_q$ to denote the reduction of the integer z modulo q to the interval. In pyFHE, we reduce an element $R \pmod q$ using `Polynomial::mod` and reduce to $\mathbb{Z}_q$ using `Polynomial::mod_small`. We use $x \leftarrow D$ to denote the sampling of x from a probability distribution D. Writing $x \leftarrow S$ from a set S denotes the unifory sampling from the set S. Let λ denote the security parameter, such that all known attacks against a given cryptographic scheme require $\Omega\left(2^{\lambda}\right)$ bit operations. Let $D_{\mathbb{Z},\sigma}$ denote the discrete Gaussian distribution, where the probability of choosing x is proportional to $\exp\left(-\pi|x|^2/\sigma^2\right)$. Let $\mathcal{ZO}\left(\rho\right)$ be the distribution which draws each entry from the set $[-1, 0, 1]$ with the probability distribution $\left[\frac{\rho}{2}, 1-\rho, \frac{\rho}{2}\right]$.

2.2 The Cyclotomic Ring

We will work extensively in the polynomial ring $R = \mathbb{Z}[x]/f(x)$ where $f(x) \in \mathbb{Z}[x]$ is a monic irreducible polynomial of degree N. Specifically, we will choose $f(x) = x^N + 1$ where N is a power of two. Note that $f(x) = \Phi_M(x)$, the M^{th} cyclotomic polynomial where $M = 2N$. Let R_q denote the ring R, where the coefficients of each polynomial

are reduced to elements in $\mathbb{Z}_q$. For an element $p \in R$, let $||p||_\infty$ denote the largest absolute value of its coefficients.

2.3 The Ring Learning with Errors problem

Many FHE schemes such as Brakerski-Gentry-Vaikuntanathan (BGV), Brakerski/Fan-Vercauteren (BFV), and Cheon-Kim-Kim-Song (CKKS) rely on the Learning with Errors (LWE) problem, a hard lattice problem which has been used proven to be as hard as all worst-case lattice problems, making it a prime candidate to base cryptographic protocols on [2, 8, 6]. All cryptographic protocols based on LWE are secure under the assumption that worst-case lattice problems are hard [18].

2.3.1 Problem Statement

Learning with Errors (LWE) is a problem introduced by Oden Regev in 2005 involving solving a linear system of equations with error [18]. An algorithm $\mathcal{A}$ is said to solve the LWE problem, if given several samples (x, y) where $x \in \mathbb{Z}_q^n$ (a vector of n integers modulo q), and $y \in \mathbb{Z}_q$, where for some linear function $f : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$, we have $f(x) = y$ with high probability, with some known noise model, $\mathcal{A}$ returns a close approximation of f with high probability.

2.3.2 Search LWE

More formally, given a secret vector $s \in \mathbb{Z}_q^n$ and noise distribution χ, and polynomially samples k such that

$$\begin{aligned} \langle a_1, s \rangle + e_1 &= b_1 \pmod q \\ \langle a_2, s \rangle + e_2 &= b_2 \pmod q \\ &\vdots \\ \langle a_k, s \rangle + e_k &= b_k \pmod q \end{aligned}$$

where each e_i is sampled from the noise distribution χ and each a_i is sampled uniformly at random from $\mathbb{Z}_q^n$, the **Learning with Errors (LWE)** problem is to find the vector s in probabilistic polynomial time.

Example. The LWE problem may ask you to recover the secret $s \in \mathbb{Z}_{19}^4$ given the following system of equations:

$$\begin{aligned}
2s_1 + 3s_2 + 14s_3 + 8s_4 &\approx 5 \pmod{19} \\
12s_1 + 8s_2 + s_4 &\approx 7 \pmod{19} \\
2s_1 + 8s_2 + 7s_3 + 5s_4 &\approx 1 \pmod{19} \\
13s_1 + 11s_2 + 10s_3 + 18s_4 &\approx 5 \pmod{19} \\
4s_1 + 2s_2 + 17s_3 + 5s_4 &\approx 2 \pmod{19} \\
&\vdots \\
9s_1 + 3s_2 + 12s_3 + 6s_4 &\approx 0 \pmod{19}
\end{aligned}$$

where each equation is correct up to an error of $e = \pm 1$. Without error, we could easily solve the system of equations using Gaussian elimination. However, the error makes the problem difficult. In this case, solving LWE would give the secret $s = (1, 2, 3, 4)$. Solving for s is known as the **Search LWE** problem, and is actually not as suitable for cryptography as the **Decisional LWE** problem described below.

2.3.3 Decisional LWE

The Decisional LWE problem is defined as follows. Given an oracle $\mathcal{O}_s^n$ which outputs samples of the form $(a, \langle a, s \rangle + e)$ where a is chosen uniformly at random from $\mathbb{Z}_q^n$ for each sample and e is chosen randomly according to the noise distribution χ for each sample, and an oracle $\mathcal{R}$ which outputs samples of the form $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}$ uniformly at random, then solving the Decisional LWE problem entails determining whether you are interacting with the oracle $\mathcal{O}_s^n$ or the oracle $\mathcal{R}$ based on the samples you receive.

Example. The Decisional LWE problem may give you the following samples:

$$\begin{aligned}
((2,3,14,8),5) &\pmod{19}\\
((12,8,0,1),7) &\pmod{19}\\
((2,8,7,5),1) &\pmod{19}\\
((13,11,10,18),5) &\pmod{19}\\
((4,2,17,5),2) &\pmod{19}\\
&\vdots\\
((9,3,12,6),0) &\pmod{19}
\end{aligned}$$

Solving the Decisional LWE problem would be outputting $\mathcal{O}_s^n$, since the above samples were generated with the secret $s = (1,2,3,4)$.

The error introduced in the LWE problem is fundamental to the security of FHE schemes, since it makes the recovering the secret key from the public key and ciphertexts extremely hard.

2.3.4 Ring LWE

The Ring LWE (RLWE) problem is defined very similarly to above. Given an oracle $\mathcal{O}_s$ which outputs samples of the form $(a, \langle a, s\rangle + e)$ where a is chosen uniformly at random from R_q for each sample and e is chosen randomly according to the noise distribution χ for each sample, and an oracle $\mathcal{R}$ which outputs samples of the form $(a,b) \in R_q \times R_q$ uniformly at random, then solving the Decisional LWE problem entails determining whether you are interacting with the oracle $\mathcal{O}_s$ or the oracle $\mathcal{R}$ based on the samples you receive.

Table 2.1: Notable variables in pyFHE

Parameter	Description	Name in pyFHE
q	Modulus in ciphertext space	`ciph_modulus`
Q	Large modulus in ciphertext space (CKKS only)	`big_modulus`
t	Modulus in plaintext space (BFV only)	`plain_modulus`
N	Polynomial ring degree where $f(x) = x^N + 1$ N is a power of two	`poly_degree` `ring_degree`
R	Polynomial ring $\mathbb{Z}[x]/(x^N + 1)$	
R_q	Polynomial ring $\mathbb{Z}_q[x]/(x^N + 1)$	
Δ	Scaling factor (CKKS) Quotient $\lfloor q/t \rfloor$ (BFV)	`scaling_factor`
χ	Discrete Gaussian distribution $D^N_{\mathbb{Z},\sigma}$	`sample_triangle`

2.3.5 In practice

The RLWE assumption is that the RLWE problem is infeasible. In practice for this assumption to hold, we choose the noise distribution χ according to a discrete Gaussian distribution $D^N_{\mathbb{Z},\sigma}$ [8]. Note that the output of this distribution are the coefficients of an $N-1$ degree polynomial in the ring R_q. For the rest of the paper, let χ denote the probability distribution $D^N_{\mathbb{Z},\sigma}$. For simplicity, we implement χ in pyFHE as the triangle distribution $\mathcal{ZO}\left(\frac{1}{2}\right)$, where we choose from $[-1, 0, 1]$ with the probability distribution $\left[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right]$. We summarize all the above notation in Table 2.1.

Chapter 3

The BFV Encryption Scheme

In this chapter, we give a brief overview of the BFV encryption scheme, as implemented in pyFHE.

3.1 Plaintext Space

In BFV, in addition to our ciphertext modulus, we have a plaintext modulus t. Our plaintext space is the polynomial ring $R_t = \mathbb{Z}_t/(x^N + 1)$. In order to encrypt an integer, we must first encode it into the polynomial plaintext space R_t. In pyFHE, we currently support two types of encodings: an `IntegerEncoder` and a `BatchEncoder` based on the Chinese Remainder Theorem.

3.1.1 IntegerEncoder

The `IntegerEncoder` inputs a base b (with a default value of 2), and encodes an integer $x \in \left[-\frac{b^N-1}{2}, \frac{b^N-1}{2}\right]$ by writing it in base b. The coefficients in base b become the coefficients of our encoded plaintext polynomial. We decode a plaintext polynomial by evaluating it at b. For example, for the base $b = 2$, we encode $6 = 1 \cdot 2^2 + 1 \cdot 2^1$ as $x^2 + x^1$, and we decode $p(x) = x^2 + x^1$, by evaluating $p(2) = 6$. This encoder supports homomorphic operations since

$$\texttt{IntegerDecode}\left(\texttt{IntegerEncode}(a) + \texttt{IntegerEncode}(b)\right) = a + b$$

and

$$\texttt{IntegerDecode}\left(\texttt{IntegerEncode}(a) \cdot \texttt{IntegerEncode}(b)\right) = a \cdot b$$

as long as no modular reductions occur. *An important limitation of this encoding scheme is that the result will not decode correctly if any modular reductions have occurred in ciphertext evaluations (either mod $x^N + 1$ or mod q).*

3.1.2 BatchEncoder

The BatchEncoder encodes N integers modulo t into a single plaintext polynomial. In order to use this encoding scheme, we must set the plaintext modulus t to be prime such that $t \equiv 1 \pmod{2N}$. When this is the case, there exists an element ζ such that $\zeta^{2N} \equiv 1 \pmod{t}$ and $\zeta^m \neq 1 \pmod{t}$ for all $1 \leq m < 2N$, known as a $(2N)^{th}$ primitive root of unity [19]. Then $x^N + 1 \pmod{t}$ has roots ζ^i for all odd i such that $1 \leq i < 2N$, so $x^N + 1$ factors as

$$x^N + 1 = (x - \zeta)\left(x - \zeta^3\right) \ldots \left(x - \zeta^{2N-1}\right)$$

.

Now, using the Chinese Remainder Theorem (CRT), we can uniquely express elements mod $x^N + 1$ as N different elements mod $x - \zeta^i$ for odd i where $1 \leq i < 2N$. Expressing a polynomial $p(x)$ in the ring $\mathbb{Z}[x]/\left(x - \zeta^i\right)$ is the same as evaluating $p\left(\zeta^i\right)$, since $x \equiv \zeta^i$ in this ring.

We use this CRT representation to define our decoding procedure below.

$$\texttt{Decode}(p(x)) \rightarrow \left[p\left(\zeta\right), p\left(\zeta^3\right), \ldots, p\left(\zeta^{2N-1}\right)\right]$$

We implement this function as a variation of the Number Theoretic Transform, which allows us to evaluate $p(x)$ at the odd $(2N)^{th}$ roots of unity, rather than at all of the $(2N)^{th}$ roots of unity. We use its inverse as our batch encoding function, which

takes N integers modulo t and outputs a polynomial in R_t. The plaintext polynomial is stored in a `Plaintext` object within the `poly` parameter. This encoder supports homomorphic operations component-wise since

$$\texttt{Decode}(p(x) + q(x)) = \left[(p+q)(\zeta), (p+q)\left(\zeta^3\right), \ldots, (p+q)\left(\zeta^{2N-1}\right)\right]$$

and

$$\texttt{Decode}(p(x) \cdot q(x)) = \left[(p \cdot q)(\zeta), (p \cdot q)\left(\zeta^3\right), \ldots, (p \cdot q)\left(\zeta^{2N-1}\right)\right]$$

3.2 Ciphertext Space

Our ciphertext space is the two-dimensional space R_q^2, consisting of two polynomials in R_q. These two polynomials are stored in a `Ciphertext` object within the `c0` and `c1` parameters.

3.3 Scheme

Given the security parameter λ and defining $\Delta = \lfloor q/t \rfloor$, we present the BFV encryption scheme [8] below:

- $\texttt{SecretKeyGen}\left(1^\lambda\right)$: Choose values for N, t, q, σ based on the security parameter λ, where N is a power of two as noted in Section 2.2. Sample $\mathbf{s} \leftarrow \chi$ and output
$$\texttt{sk} = \mathbf{s}$$

- $\texttt{PublicKeyGen}(\texttt{sk})$: Set $\mathbf{s} = \texttt{sk}$, sample $\mathbf{a} \leftarrow R_q$, $\mathbf{e} \leftarrow \chi$ and output
$$\texttt{pk} = \left(\left[-(\mathbf{a} \cdot \mathbf{s} + \mathbf{e})\right]_q, \mathbf{a}\right)$$

- $\texttt{RelinearizationKeyGen}(\texttt{sk}, T)$: Set $\mathbf{s} = \texttt{sk}$. Let $l = \lfloor \log_T q \rfloor$. Then for $i \in$

$\{0, 1, \ldots, l\}$, sample $\mathbf{a}_i \leftarrow R_q$, $\mathbf{e}_i \leftarrow \chi$. Output

$$\texttt{rlk} = \left(\left[-\left(\mathbf{a}_i \cdot \mathbf{s} + \mathbf{e}_i\right) + T^i \cdot \mathbf{s}^2\right]_q, \mathbf{a}_i\right) \qquad \text{for } i \in \{0, 1, \ldots, l\}$$

The above key generation methods are implemented in `BFVKeyGenerator`. For relinearization, we choose $T = \left\lceil \sqrt{q} \right\rceil$, as suggested in [8].

- $\texttt{Encrypt}\left(\texttt{pk}, \mathbf{m}\right)$: To encrypt $\mathbf{m} \in R_t$, let $\texttt{pk} = \left(\mathbf{p}_0, \mathbf{p}_1\right)$. Sample $\mathbf{u}, \mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi$ and output

$$\texttt{ct} = \left(\left[\Delta \cdot \mathbf{m} + \mathbf{p}_0 \cdot \mathbf{u} + \mathbf{e}_0\right]_q, \left[\mathbf{p}_1 \cdot \mathbf{u} + \mathbf{e}_1\right]_q\right)$$

- $\texttt{Decrypt}\left(\texttt{sk}, \texttt{ct}\right)$: Set $\mathbf{s} = \texttt{sk}$ and $\texttt{ct} = \left(\mathbf{c}_0, \mathbf{c}_1\right)$ and output

$$\left[\left\lfloor \frac{t}{q} \left[\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}\right]_q \right\rceil\right]_q$$

Encryption and decryption are implemented in `BFVEncryptor` and `BFVDecryptor`, respectively.

- $\texttt{Add}\left(\texttt{ct}_1, \texttt{ct}_2\right)$: Output

$$\texttt{ct}_{\texttt{add}} = \left(\left[\texttt{ct}_1\left[0\right] + \texttt{ct}_2\left[0\right]\right]_q, \left[\texttt{ct}_1\left[1\right] + \texttt{ct}_2\left[1\right]\right]_q\right)$$

- $\texttt{Multiply}\left(\texttt{ct}_1, \texttt{ct}_2, \texttt{rlk}\right)$: Compute

$$\begin{aligned}
\mathbf{c}_0 &= \left[\left\lfloor \frac{t}{q} \left(\texttt{ct}_1\left[0\right] \cdot \texttt{ct}_2\left[0\right]\right) \right\rceil\right]_q \\
\mathbf{c}_1 &= \left[\left\lfloor \frac{t}{q} \left(\texttt{ct}_1\left[0\right] \cdot \texttt{ct}_2\left[1\right] + \texttt{ct}_1\left[1\right] \cdot \texttt{ct}_2\left[0\right]\right) \right\rceil\right]_q \\
\mathbf{c}_2 &= \left[\left\lfloor \frac{t}{q} \left(\texttt{ct}_1\left[1\right] \cdot \texttt{ct}_2\left[1\right]\right) \right\rceil\right]_q
\end{aligned}$$

Relinearize the ciphertext by writing $\mathbf{c}_2$ in base T as $\mathbf{c}_2 = \sum_{i=0}^{l} \mathbf{c}_2^{(i)} T^i$ with $\mathbf{c}_2^{(i)} \in$

R_T. Set

$$\mathbf{c}_0' = \left[\mathbf{c}_0 + \sum_{i=0}^{l} \texttt{rlk}[i][0] \cdot \mathbf{c}_2^{(i)} \right]_q$$
$$\mathbf{c}_1' = \left[\mathbf{c}_1 + \sum_{i=0}^{l} \texttt{rlk}[i][1] \cdot \mathbf{c}_2^{(i)} \right]_q$$

and output $\texttt{ct}_{\texttt{mul}} = (\mathbf{c}_0', \mathbf{c}_1')$.

Addition, multiplication, and relinearization are implemented in `BFVEvaluator`. Currently, `BFVEvaluator::multiply` automatically relinearizes the ciphertext after each multiplication. Further evaluation functions and bootstrapping have not yet been implemented for BFV in pyFHE.

Chapter 4

The CKKS Encryption Scheme

In this chapter, we give a brief overview of the CKKS encryption scheme, as implemented in pyFHE. The most notable difference between CKKS and other FHE schemes is that CKKS is an *approximate homomorphic encryption* scheme which supports complex-number arithmetic, as opposed to just integer arithmetic. It achieves this through its encoding scheme.

4.1 Plaintext Space

Our plaintext space is the polynomial ring $R_q = \mathbb{Z}_q/(x^N + 1)$. In order to encrypt a complex number, we must first encode it into the polynomial plaintext space R_q. This encoder is implemented in pyFHE as `CKKSEncoder`.

4.1.1 CKKS Encoding Scheme

The CKKS encoding scheme is similar to the CRT batching scheme described in section 3.1.2. One notable distinction between the two schemes is that the CKKS encoding function maps $\mathbb{C}^{N/2} \to R_q$ via a function similar to the Fast Fourier Transform, whereas the batch encoding function maps $\mathbb{Z}_t^N \to R_t$ via a function similar to the Number Theoretic Transform.

A natural encoding scheme to use is an existing embedding scheme (an embedding scheme satisfies homomorphic addition and multiplication) from $\mathbb{C}$ to a polynomial ring. Thus, CKKS uses the canonical embedding map σ which maps $\mathbb{Q}[x]/(x^N+1) \rightarrow \mathbb{C}^N$ as part of the decoding function [14, 6].

Let $\zeta = \exp\left(\frac{2\pi i}{2N}\right)$, a complex $(2N)^{th}$ root of unity. The canonical embedding map for $\mathbb{Q}[x]/f(x) \rightarrow \mathbb{C}^N$ is defined as the vector of evaluations at the complex roots of $f(x)$, for any polynomial f with degree N. When $f(x) = x^N + 1$, we define σ using the odd $(2N)^{th}$ roots of unity as shown below:

$$\sigma(p(x)) \rightarrow \left[p\left(\zeta\right), p\left(\zeta^3\right), \ldots, p\left(\zeta^{2N-1}\right)\right]$$

This looks very similar to the CRT batching described in section 3.1.2. However, here $p(x)$ has rational coefficients and ζ is complex, which imposes additional constraints on the image of σ. Since, $p(x)$ has rational coefficients, for any complex number w, we have $\overline{p(w)} = p\left(\overline{w}\right)$. Thus, $\overline{p(\zeta^i)} = p\left(\overline{\zeta^i}\right) = p\left(\zeta^{2N-i}\right)$ for all $1 \leq i < 2N$. The image of σ turns out be precisely the set

$$H = \{(x_1, x_2, \ldots, x_N) \in \mathbb{C}^N \text{ such that } x_i = \overline{x_{N-i}}\}$$

[14]. Thus, we need only include the roots of unity whose power is 1 (mod 4). The elements 1 (mod 4) in $\mathbb{Z}_{2N}$ are equivalent to the powers of 5 in $\mathbb{Z}_{2N}$. so we define a variant of the canonical embedding as the first part of our decoding scheme below:

$$\texttt{Embedding}(p(x)) \rightarrow \left[p\left(\zeta_0\right), p\left(\zeta_1\right), \ldots, p\left(\zeta_{\frac{N}{2}-1}\right)\right]$$

where $\zeta_j = \zeta^{5^j}$. We define this function using powers of five to allow us to implement it using an optimized variant of the Fast Fourier Transform [3]. This function is implemented in pyFHE in `FFTContext::embedding`. Note that this function supports

component-wise homomorphic addition and multiplication.

For our encoding function, we use $\texttt{Embedding}^{-1}$, with an output in $\mathbb{Q}[x]/\left(x^N+1\right)$. Now, the most crucial part of the CKKS encoding scheme is to multiply the resulting polynomial by a scaling factor Δ to maintain precision throughout rounding errors coming from this encoding scheme and homomorphic operations. The encoding and decoding schemes are as follows:

- $\texttt{Encode}(\mathbf{z}, \Delta)$: For an $(N/2)$-dimensional vector $\mathbf{z}$, output

$$m(x) = \left\lfloor \Delta \cdot \texttt{Embedding}^{-1}\left(\mathbf{z}\right) \right\rceil$$

- $\texttt{Decode}(m)$: For a plaintext polynomial $m \in R_q$ with scaling factor Δ output

$$\mathbf{z} = \texttt{Embedding}\left(\Delta^{-1} \cdot m\right)$$

These two functions are implemented in `CKKSEncoder`. Each `Plaintext` stores a plaintext polynomial as well as a scaling factor, which is input into the encoding function as a parameter.

4.2 Ciphertext Space

Our ciphertext space, like in BFV, is the two-dimensional space R_q^2, consisting of two polynomials in R_q. These two polynomials are stored in a `Ciphertext` object within the `c0` and `c1` parameters. However, in CKKS, the values of the modulus q and the scaling factor Δ can change for each ciphertext. Each ciphertext keeps track of its current modulus and its scaling factor. We call the maximum modulus a ciphertext can have Q and refer to it as `big_modulus` in pyFHE.

4.3 Scheme

We introduce one additional sampling distribution for CKKS. For a positive integer h, we call $\mathcal{HWT}(h)$ the set of vectors in $\{0, -1, 1\}^N$ that contain exactly h non-zero values (a Hamming weight of h). We will also use χ and $\mathcal{ZO}(\rho)$ as defined in section 2.1. Given the security parameter λ, we present the CKKS encryption scheme [6] below:

- $\texttt{SecretKeyGen}\left(1^\lambda\right)$: Choose values for N, q, Q, Δ, h based on the security parameter λ. We choose powers of two for q, Q, and Δ, such that $\Delta < q < Q$ to avoid introducing error during rescaling. We must also choose N to be a power of two, as noted in Section 2.2. Sample $\mathbf{s} \leftarrow \mathcal{HWT}(h)$ and output

$$\texttt{sk} = \langle 1, \mathbf{s} \rangle$$

- $\texttt{PublicKeyGen}\left(\texttt{sk}\right)$: Set $\langle 1, \mathbf{s} \rangle = \texttt{sk}$, sample $\mathbf{a} \leftarrow R_Q$, $\mathbf{e} \leftarrow \chi$ and output

$$\texttt{pk} = \left(\left[-\mathbf{a} \cdot \mathbf{s} + \mathbf{e}\right]_Q, \mathbf{a}\right)$$

- $\texttt{SwitchKeyGen}\left(\texttt{sk}, \mathbf{s}'\right)$: Set $\langle 1, \mathbf{s} \rangle = \texttt{sk}$, sample $\mathbf{a} \leftarrow R_Q$, $\mathbf{e} \leftarrow \chi$, and output

$$\texttt{swk} = \left(\left[-\mathbf{a} \cdot \mathbf{s} + \mathbf{e} + Q \cdot \mathbf{s}'\right]_{Q^2}, \mathbf{a}\right)$$

- $\texttt{RelinearizationKeyGen}\left(\texttt{sk}\right)$: Set $\langle 1, \mathbf{s} \rangle = \texttt{sk}$, and output

$$\texttt{rlk} = \texttt{SwitchKeyGen}\left(\texttt{sk}, \mathbf{s}^2\right)$$

The above key generation methods are implemented in `CKKSKeyGenerator`.

- $\texttt{Encrypt}\left(\texttt{pk}, \mathbf{m}\right)$: To encrypt $\mathbf{m} \in R_q$, let $\texttt{pk} = \left(\mathbf{p}_0, \mathbf{p}_1\right)$. Sample $\mathbf{u} \leftarrow \mathcal{ZO}\left(\frac{1}{2}\right)$ and $\mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi$ and output

$$\texttt{ct} = \left(\left[\mathbf{m} + \mathbf{p}_0 \cdot \mathbf{u} + \mathbf{e}_0\right]_q, \left[\mathbf{p}_1 \cdot \mathbf{u} + \mathbf{e}_1\right]_q\right)$$

- $\texttt{Decrypt}\left(\texttt{sk}, \texttt{ct}\right)$: Set $\langle 1, \mathbf{s} \rangle = \texttt{sk}$, and $\texttt{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ and output

$$\left[\langle \texttt{ct}, \texttt{sk} \rangle\right]_q = \left[\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}\right]_q$$

Encryption and decryption are implemented in `CKKSEncryptor` and `CKKSDecryptor`, respectively.

- $\texttt{Add}\left(\texttt{ct}_1, \texttt{ct}_2\right)$: To add two ciphertexts, they must have the same modulus and scaling factor, which will also be the modulus and scaling factor of their ciphertext sum. Output

$$\texttt{ct}_{\texttt{add}} = \left(\left[\texttt{ct}_1\left[0\right] + \texttt{ct}_2\left[0\right]\right]_q, \left[\texttt{ct}_1\left[1\right] + \texttt{ct}_2\left[1\right]\right]_q\right)$$

- $\texttt{Multiply}\left(\texttt{ct}_1, \texttt{ct}_2, \texttt{rlk}\right)$: To multiply two ciphertexts, the must have the same modulus, which will also be the modulus of their ciphertext product. Compute

$$\begin{aligned}
\mathbf{c}_0 &= \left[\left(\texttt{ct}_1\left[0\right] \cdot \texttt{ct}_2\left[0\right]\right)\right]_q \\
\mathbf{c}_1 &= \left[\left(\texttt{ct}_1\left[0\right] \cdot \texttt{ct}_2\left[1\right] + \texttt{ct}_1\left[1\right] \cdot \texttt{ct}_2\left[0\right]\right)\right]_q \\
\mathbf{c}_2 &= \left[\left(\texttt{ct}_1\left[1\right] \cdot \texttt{ct}_2\left[1\right]\right)\right]_q
\end{aligned}$$

Relinearize the ciphertext to obtain

$$\begin{aligned}
\mathbf{c}'_0 &= \left[\mathbf{c}_0 + \left\lfloor Q^{-1} \cdot \mathbf{c}_2 \cdot \texttt{rlk}\left[0\right]\right\rceil\right]_q \\
\mathbf{c}'_1 &= \left[\mathbf{c}_1 + \left\lfloor Q^{-1} \cdot \mathbf{c}_2 \cdot \texttt{rlk}\left[1\right]\right\rceil\right]_q
\end{aligned}$$

and output $\texttt{ct}_{\texttt{mul}} = (\mathbf{c}'_0, \mathbf{c}'_1)$ with scaling factor $\Delta_{\texttt{mul}} = \Delta_1 \cdot \Delta_2$.

- $\texttt{Rescale}\left(\texttt{ct}, \Delta'\right)$: Given a ciphertext $\texttt{ct}$ with scaling factor $\Delta_{\texttt{ct}}$ and modulus $q_{\texttt{ct}}$ output

$$\texttt{ct}_{\texttt{rescale}} = \left(\left[\left\lfloor \frac{\texttt{ct}[0]}{\Delta'} \right\rceil\right]_{q/\Delta'}, \left[\left\lfloor \frac{\texttt{ct}[1]}{\Delta'} \right\rceil\right]_{q/\Delta'}\right)$$

with scaling factor $\frac{\Delta_{\texttt{ct}}}{\Delta'}$ and modulus $\frac{q}{\Delta'}$.

Addition, multiplication, relinearization, rescaling are implemented in `CKKSEvaluator`. Currently, `CKKSEvaluator::multiply` automatically relinearizes the ciphertext after each multiplication. We recommend also calling `CKKSEvaluator::rescale` with the original scaling factor after each multiplication in order to reduce the ciphertext product's scaling factor from Δ^2 to Δ, to avoid blow-up of the scaling factor after further homomorphic multiplications.

4.4 Bootstrapping

After many multiplications and rescaling operations, the ciphertext modulus will be too small compared to the scaling factor, which would cause the message to wrap around $\pmod q$ allowing the noise to corrupt the message. However, before this can occur, we can increase the modulus of the ciphertext through bootstrapping [4]. We will give a brief overview of the CKKS bootstrapping procedure, while omitting details about rotation, conjugation, and matrix multiplication. These details can be found in [4] and are implemented in `CKKSEvaluator::rotate`, `CKKSEvaluator::conjugate`, and `CKKSEvaluator::multiply_matrix`. Bootstrapping involves the following steps:

- ModRaise: Starting with a ciphertext $\mathtt{ct}$ such that $[\langle \mathtt{ct}, \mathtt{sk} \rangle]_q = m(x)$, we raise our modulus q to the big modulus Q. We also change the scaling factor Δ to $\Delta' = q$, so that our plaintext coefficients stay as integers > 1. This step does not involve any homomorphic operations, but we now would have a different plaintext if we decrypted $\mathtt{ct}$, namely, $[\langle \mathtt{ct}, \mathtt{sk} \rangle]_Q = t(x)$, such that $[t(x)]_q = m(x)$. This function is implemented in `CKKSEvaluator::raise_modulus`.

- CoeffToSlot: We homomorphically perform the $\mathtt{Embedding}^{-1}$ function on our ciphertext in order to move our plaintext polynomial coefficients into our decoded slots, with the output split into two vectors $\mathbf{t}_0 = \left(t_0, t_1, \ldots, t_{\frac{N}{2}-1}\right)$ and $\mathbf{t}_1 = \left(t_{\frac{N}{2}}, t_{\frac{N}{2}+1}, \ldots, t_{N-1}\right)$. The $\mathtt{Embedding}^{-1}$ function can be represented

through the equations $\mathbf{z}'_k = \frac{1}{N}\left(\overline{U_k}^T \cdot \mathbf{z}' + U_k^T \cdot \overline{\mathbf{z}}'\right)$ for $k = 0, 1$ where

$$U_0 = \begin{pmatrix} 1 & \zeta_0 & \dots & \zeta_0^{\frac{N}{2}-1} \\ 1 & \zeta_1 & \dots & \zeta_1^{\frac{N}{2}-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{\frac{N}{2}-1} & \dots & \zeta_{\frac{N}{2}-1}^{\frac{N}{2}-1} \end{pmatrix} \quad \text{and} \quad U_1 = \begin{pmatrix} \zeta_0^{\frac{N}{2}} & \zeta_0^{\frac{N}{2}+1} & \dots & \zeta_0^{N-1} \\ \zeta_1^{\frac{N}{2}} & \zeta_1^{\frac{N}{2}+1} & \dots & \zeta_1^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \zeta_{\frac{N}{2}-1}^{\frac{N}{2}} & \zeta_{\frac{N}{2}-1}^{\frac{N}{2}+1} & \dots & \zeta_{\frac{N}{2}-1}^{N-1} \end{pmatrix}.$$

We precompute these matrices in `CKKSBootstrappingContext`, and perform this function in `CKKSEvaluator::coeff_to_slot`.

Now that we have the coefficients of $t(x)$ in our plaintext slots, we wish to compute $m(x) = [t(x)]_q$ homomorphically. In order to do so, we use $\frac{q}{2\pi}\sin\left(\frac{2\pi t}{q}\right)$ as an approximation for $[t]_q$ with the assumption that m is much smaller than q.

- EvalExp: To compute sin, we use the Taylor series approximation for $e^{\frac{2\pi i \mathbf{t}_k}{q \cdot 2^r}}$ for $k = 0, 1$ in `CKKSEvaluator::exp_taylor` and square it r times in `CKKSEvaluator::exp` to reduce error. Increasing the value of the parameter r reduces the error, but increases the number of multiplications performed.

- ImagExt: To extract the imaginary part of the exponential function, we compute

$$\mathbf{m}_k = [\mathbf{t}_k]_q \approx \frac{q}{2\pi}\sin\left(\frac{2\pi \mathbf{t}_k}{q}\right) = \frac{\exp\left(\frac{2\pi i \mathbf{t}_k}{q}\right) - \exp\left(-\frac{2\pi i \mathbf{t}_k}{q}\right)}{2i}$$

  for $k = 0, 1$.

- SlotToCoeff: In the final step, we homomorphically perform the `Embedding` function to put our decoded slots back into the plaintext polynomial. To do so, we compute the equation $\mathbf{z} = U_0\mathbf{m}_0 + U_1\mathbf{m}_1$ to obtain back our original message $\mathbf{z}$ in a larger modulus Q_1 such that $q < Q_1 < Q$. We do not end up with the modulus Q due to multiple rescaling operations that occur during bootstrapping. This function is implemented in `CKKSBootstrapping::slot_to_coeff`.

The bootstrapping procedure along with more details about how the scaling factor changes is summarized in Table 4.1. The procedure is implemented in

Table 4.1: CKKS Bootstrapping

	Decoded Slots	Encoded Plaintext	Scaling Factor	Modulus
Before	$\mathbf{z}$ $\texttt{Embedding}\left(m/\Delta\right)$	$m(x) = m_0(x) + x^{N/2} m_1(x)$ $\Delta \cdot \texttt{Embedding}^{-1}\left(\mathbf{z}\right)$	Δ	q
ModRaise	$\mathbf{z}'$	$t(x) = t_0(x) + x^{N/2} t_1(x)$ $\left[t(x)\right]_q = m(x)$ $\Delta' \cdot \texttt{Embedding}^{-1}\left(\mathbf{z}'\right)$	Δ'	Q
CoeffToSlot $k = 0, 1$	$\texttt{Embedding}^{-1}\left(\mathbf{z}'\right)$ t_k/Δ'	$\Delta' \cdot \texttt{Embedding}^{-1}\left(t_k/\Delta'\right)$	Δ'	Q'
Scale $(k = 0, 1)$	$\frac{2\pi i t_k}{q}$	$\Delta' \cdot \texttt{Embedding}^{-1}\left(\frac{2\pi i t_k}{q}\right)$	Δ'	Q'
EvalExp $k = 0, 1$	$e^{\frac{2\pi i t_k}{q}}$, $e^{-\frac{2\pi i t_k}{q}}$	$\Delta' \cdot \texttt{Embedding}^{-1}\left(e^{\frac{2\pi i t_k}{q}}\right)$, $\Delta' \cdot \texttt{Embedding}^{-1}\left(e^{-\frac{2\pi i t_k}{q}}\right)$	Δ'	Q''
ImagExt $k = 0, 1$	$\frac{q}{2\pi} \sin\left(\frac{2\pi t_k}{q}\right)$ $\left[t_k\right]_q = m_k$	$\Delta' \cdot \texttt{Embedding}^{-1}\left(m_k\right)$	Δ'	Q''
Scale $(k = 0, 1)$	m_k/Δ'	$\Delta' \cdot \texttt{Embedding}^{-1}\left(m_k/\Delta'\right)$	Δ'	Q''
SlotToCoeff	$\texttt{Embedding}\left(m/\Delta'\right)$	$m(x)$	Δ'	Q_1
Reset scaling factor	$\texttt{Embedding}\left(m/\Delta\right)$ $\mathbf{z}$	$m(x)$	Δ	Q_1

`CKKSEvaluator::bootstrap`.

4.5 RNS Representation

In order to speed up multiplication, we use the Residue Number System (RNS) representation of elements in R_Q [3]. To use this representation, we choose a modulus $P > NQ^4$ such that $P = p_1 p_2 \dots p_n$ for distinct primes $p_i \equiv 1 \pmod{2N}$ for $1 \leq i \leq 2N$. For a single ciphertext multiplication, instead of performing a naive $O(N^2)$ polynomial multiplication in $\mathbb{Z}_q$, we perform n polynomial multiplications in $\mathbb{Z}_{p_i}$ for $1 \leq i \leq n$, and reconstruct the product in $\mathbb{Z}_P$ using the Chinese Remainder Theorem. Since $p_i \equiv 1 \pmod{2N}$, we can perform these n polynomial multiplications using the NTT, making our runtime $O(nN \log N)$. In pyFHE, we reconstruct to $\mathbb{Z}_P$ after every multiplication, and since the largest modulus we perform multiplication in is Q^2, as long as $P > NQ^4$, we will not overflow the modulus P. We manage the RNS representation through `CRTContext`, and include this optimization as an optional parameter `num_primes` in `CKKSParameters`, which is input to `Polynomial::multiply`. We recommend turning off this parameter for $N \leq 2048$, since it only provides a speedup for larger N.

Future work includes supporting rescaling in the RNS representation, which would allow us to remain in RNS representation until decryption.

Chapter 5

Library Architecture and Capabilities

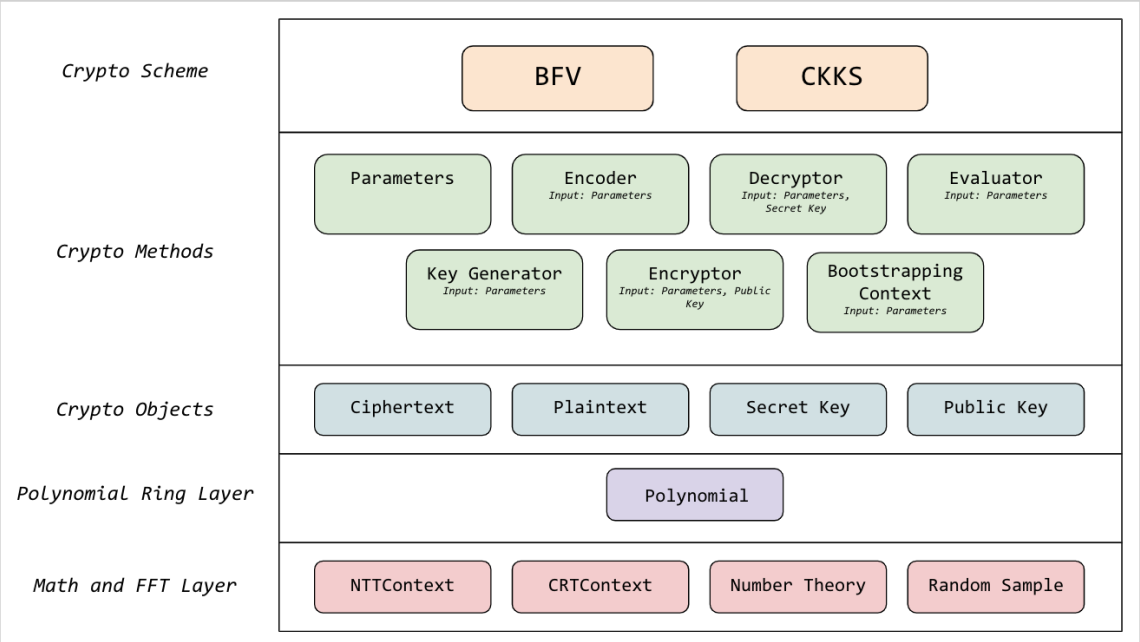
5.1 Library Architecture

pyFHE is designed to have a common set of classes for every fully homomorphic encryption scheme. The high-level architecture is illustrated in Figure 5-1. The layers are organized as follows:

1. **Crypto Scheme:** The existing schemes in pyFHE are BFV and CKKS.
2. **Crypto Methods:** These include all cryptographic protocols in a scheme.
3. **Crypto Objects:** These objects are common to all cryptographic schemes.
4. **Polynomial Ring Layer:** This allows us to perform operations such as addition and multiplication in the ring $\mathbb{Z}[x]/(x^N + 1)$.
5. **Math and FFT Layer:** This includes all basic math and number theory operations, including FFT, NTT, CRT, and random sampling.

To add a new FHE scheme to the library, one simply needs to add a Parameters class, a Key Generator class, various Encoder classes, Encryptor and Decryptor classes, an Evaluator class, and a Bootstrapping Context class if implementing bootstrapping. These classes can use the existing `Ciphertext`, `Plaintext`, `SecretKey`, and `PublicKey` modules to implement the scheme. If the new FHE scheme is based

Figure 5-1: pyFHE Library Architecture



on the Ring LWE problem, as most FHE schemes are, we can use the `Polynomial` module to implement ring operations. Finally, we have basic math and number theory operations any scheme can use as necessary.

5.2 Capabilities

We summarize the functions currently supported in pyFHE in Table 5.1. We omit functions that are only useful for bootstrapping in this table. Future work includes adding more functionality to BFV, adding more FHE schemes to the library, and adding more optimizations including full-RNS for both schemes [10, 5].

5.3 Sample Implementation

A sample implementation of CKKS is shown in Figures 5-2 and 5-3. Figure 5-2 illustrates how to initialize various components of the CKKS scheme, and figure 5-3 illustrates how to encode and encrypt two messages, perform a homomorphic multi-

Table 5.1: Capabilities of pyFHE

Functions Supported	BFV	CKKS
`Add`	✓	✓
`AddPlain`		✓
`Subtract`		✓
`Multiply`	✓	✓
`MultiplyPlain`		✓
`Relinearize`	✓	✓
`Rescale`		✓
`LowerModulus`		✓
`KeySwitch`		✓
`Rotate`		✓
`Conjugate`		✓
`MultiplyMatrix`		✓
`Exponentiate`		✓
`Bootstrap`		✓

plication, and decrypt the result.

Figure 5-2: CKKS Scheme Initialization

```
poly_degree = 8
ciph_modulus = 1 << 600
big_modulus = 1 << 1200
scaling_factor = 1 << 30
params = CKKSParameters(poly_degree=poly_degree,
                        ciph_modulus=ciph_modulus,
                        big_modulus=big_modulus,
                        scaling_factor=scaling_factor)
key_generator = CKKSKeyGenerator(params)
public_key = key_generator.public_key
secret_key = key_generator.secret_key
relin_key = key_generator.relin_key
encoder = CKKSEncoder(params)
encryptor = CKKSEncryptor(params, public_key, secret_key)
decryptor = CKKSDecryptor(params, secret_key)
evaluator = CKKSEvaluator(params)
```

Figure 5-3: CKKS Multiplication

```
message1 = [0.5, 0.3 + 0.2j, 0.78, 0.88j]
message2 = [0.2, 0.11, 0.4 + 0.67j, 0.9 + 0.99j]
plain1 = encoder.encode(message1, scaling_factor)
plain2 = encoder.encode(message2, scaling_factor)
ciph1 = encryptor.encrypt(plain1)
ciph2 = encryptor.encrypt(plain2)
ciph_prod = evaluator.multiply(ciph1, ciph2, relin_key)
decrypted_prod = decryptor.decrypt(ciph_prod)
decoded_prod = encoder.decode(decrypted_prod)
```

Chapter 6

Parameter Selection

Before we can use BFV or CKKS, we must pick a number of parameters including:

1. The polynomial degree N (a power of two)
2. The ciphertext modulus q
3. The plaintext modulus t (BFV only)
4. The large ciphertext modulus Q (CKKS only)
5. The scaling factor Δ (CKKS only)

6.1 Polynomial degree

For both schemes, a larger choice of N will give a higher security level and less efficiency. Since we are currently using this library primarily to test correctness rather than to use in secure applications, we often choose smaller values of N such as $N = 16$ so that we can test functions quickly.

6.2 Moduli

For both schemes, a larger ciphertext modulus q allows us to perform more homomorphic operations before the noise gets too large. For testing, we have used values

of q anywhere from 40 bits to 1200 bits. However, q and Q are upper bounded based on the security parameter λ and polynomial degree N [7, 19].

For BFV, decryption works correctly as long as the noise is less than $\frac{\Delta}{2}$ where $\Delta = \lfloor q/t \rfloor$. Thus, a smaller value of t allows more homomorphic operations. One should choose t to be exactly as large as needed for the largest plaintext value.

6.3 Scaling Factor (CKKS)

For CKKS, a larger value of the scaling factor Δ yields more precision, but allows less homomorphic operations. More specifically, for a plaintext polynomial $m(x) \in R_q$ where $m(x) = \texttt{Encode}(\mathbf{z}, \Delta)$, we will compute the error in the result of decoding $m(x) + e(x)$ for some accumulated error e.

Since we have $m = \lfloor \Delta \cdot \texttt{Embedding}^{-1}(\mathbf{z}) \rceil$, we introduce $e_{\texttt{round}}$ such that

$$m + e_{\texttt{round}} = \Delta \cdot \texttt{Embedding}^{-1}(\mathbf{z}) \qquad \text{where } ||e_{\texttt{round}}||_\infty \leq \frac{1}{2}.$$

Now, decoding $m + e$ gives

$$\begin{aligned}
\widetilde{\mathbf{z}} &= \texttt{Embedding}\left(\Delta^{-1}(m+e)\right) \\
&= \texttt{Embedding}\left(\Delta^{-1} \cdot m\right) + \texttt{Embedding}\left(\Delta^{-1} \cdot e\right) \\
&= \texttt{Embedding}\left(\Delta^{-1} \cdot \left(\Delta \cdot \texttt{Embedding}^{-1}(\mathbf{z}) - e_{\texttt{round}}\right)\right) + \texttt{Embedding}\left(\Delta^{-1} \cdot e\right) \\
&= \mathbf{z} - \texttt{Embedding}\left(\Delta^{-1} \cdot e_{\texttt{round}}\right) + \texttt{Embedding}\left(\Delta^{-1} \cdot e\right)
\end{aligned}$$

For any polynomial $p(x) = \sum_{i=0}^{N} p_i x^i$, for all $0 \leq k < N$, we have

$$\begin{aligned} ||\texttt{Embedding}(p)||_\infty \leq |p\left(\zeta_k\right)| &= \left|\sum_{i=0}^{N} p_i \zeta_k^i\right| \\ &\leq \sum_{i=0}^{N} \left|p_i \zeta_k^i\right| = \sum_{i=0}^{N} |p_i| \\ &\leq N \cdot ||p||_\infty \end{aligned}$$

Thus, the error $e_{\texttt{decode}} = \widetilde{\mathbf{z}} - \mathbf{z}$ can be bounded by

$$\begin{aligned} ||e_{\texttt{decode}}||_\infty &\leq ||\texttt{Embedding}\left(\Delta^{-1} \cdot e_{\texttt{round}}\right)||_\infty + ||\texttt{Embedding}\left(\Delta^{-1} \cdot e\right)||_\infty \\ &\leq \Delta^{-1} \cdot N \left(||e_{\texttt{round}}||_\infty + ||e||_\infty\right) \\ &\leq \Delta^{-1} \cdot N \left(\frac{1}{2} + ||e||_\infty\right). \end{aligned}$$

In practice, we can use this estimate to choose Δ. For example, if choosing $\Delta = 2^{15}$ gives us a final decoded message with 4 bits of precision, if we wants 6 bits of precision, we can choose $\Delta = 2^{17}$. In certain operations, we must increase the size of the scaling factor if the size of our decoded slots becomes small. For example, in bootstrapping, the size of our slots becomes $\frac{2\pi t}{q}$, so we choose our new scaling factor Δ' such that $e_{\texttt{decode}}$ is much smaller than $\frac{2\pi t}{q}$.

However, note that we must have $||m||_\infty = ||\Delta \cdot \texttt{Embedding}^{-1}(\mathbf{z})||_\infty < \frac{q}{2}$ in order to decrypt correctly. Once Δ gets too close to $\frac{q}{2}$, we will no longer have be able to perform anymore homomorphic operations, since the resulting plaintext norm could become larger than $\frac{q}{2}$. Choosing a larger Δ will result in Δ getting close to q after fewer operations.

Chapter 7

Performance

One of pyFHE's limitations in comparison to C++ libraries [19, 16] is its poor performance. However, this is a tradeoff we decided to make for readability purposes. All experiments were run on a 2.7 GHz Intel Core i5 processor. We give runtimes for one CKKS multiplication (including relinearization) in Table 7.2 and for one CKKS bootstrapping operation in Table 7.3 for various values of N. For multiplication, we used the Parameter Set 1 given in Table 7.1, and for bootstrapping, we used Parameter Set 2. We chose $r \geq 6$, the number of squarings to approximate the exponential function while bootstrapping, such that each component's real and imaginary parts of the resulting message from bootstrapping was within 0.05 of the original message, which was a $(N/2)$-length vector whose elements were of the form $a + bi$ where $(a, b) \leftarrow [0, 1)^2$. Amortized time refers to the total time divided by the number of plaintext slots $\frac{N}{2}$.

We used the RNS representation for $N > 2048$ and did not otherwise, since this gave the optimal performance. Although we need $N \geq 2048$ for any reasonable level of security, we suggest using this library for smaller values of N such as $N = 16$ to obtain reasonable performance. Currently, this library is intended to aid researchers in testing correctness of variations of the schemes rather than to be used in secure applications. It is possible it will be suitable for secure applications in the future after further optimizations have been implemented.

Table 7.1: Parameter Sets

Parameter	$\log_2 \Delta$	$\log_2 q$	$\log_2 Q$
Set 1	30	600	1200
Set 2	30	40	1200

Table 7.2: Runtimes for CKKS Multiplication

$\log_2 N$	Total Time	Amortized Time
4	4.4 ms	0.4 ms
5	12 ms	0.8 ms
6	85 ms	2.6 ms
7	0.19 s	3.1 ms
8	0.69 s	5.4 ms
9	2.8 s	11 ms
10	12 s	23 ms
11	48 s	47 ms
12	100 s	49 ms
13	190 s	46 ms
14	390 s	48 ms

Table 7.3: Runtimes for CKKS Bootstrapping

$\log_2 N$	Total Time	Amortized Time	r
4	0.28 s	35 ms	6
5	1.2 s	77 ms	6
6	6.6 s	0.21 s	6
7	30. s	0.47 s	6
8	170 s	1.4 s	6
9	20 min	4.7 s	7

Chapter 8

Conclusion

We designed the Python library pyFHE v1.0 [17] to provide readable implementations of fully homomorphic encryption schemes that are easy for researchers to modify and test. Currently, pyFHE supports the BFV encryption scheme and the CKKS encryption scheme, and is one of very few libraries [11] that currently includes CKKS bootstrapping. We currently sacrifice performance for readability, but future work includes implementing optimizations [3, 10, 5] to allow the library to be usable in secure applications.

Bibliography

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):79, 2018.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[3] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.

[4] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[5] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[6] 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.

[7] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[8] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[9] Craig Gentry. *A fully homomorphic encryption scheme*, volume 20. Stanford University Stanford, 2009.

[10] 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.

[11] Heaan. https://github.com/snucrypto/HEAAN, March 2020.

[12] Helib. https://github.com/homenc/HElib, July 2020.

[13] Kristin Lauter, Adriana López-Alt, and Michael Naehrig. Private computation on encrypted genomic data. In *International Conference on Cryptology and Information Security in Latin America*, pages 3–27. Springer, 2014.

[14] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[15] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[16] Palisade. https://gitlab.com/palisade/palisade-release, April 2020.

[17] pyfhe. https://github.com/sarojaerabelli/py-fhe, August 2020.

[18] Oded Regev. The learning with errors problem.

[19] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, April 2020. Microsoft Research, Redmond, WA.