HEIDE: AN IDE FOR THE HOMOMORPHIC ENCRYPTION LIBRARY HELIB

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Grant Frame

June 2015

COMMITTEE MEMBERSHIP

TITLE:	HEIDE: An IDE for the Homomorphic Encryption Library HElib
AUTHOR:	Grant Frame
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Assistant Professor Zachary N J Peterson, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Associate Professor John Clements, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Assistant Professor Robert Easton, Ph.D. Department of Mathematics

ABSTRACT

HEIDE: An IDE for the Homomorphic Encryption Library HElib

Grant Frame

Work in the field of Homomorphic Encryption has exploded in the past 5 years, after Craig Gentry proposed the first encryption scheme capable of performing Homomorphic Encryption. Under the scheme one can encrypt data, perform computations on the encrypted result (without needing the original data), and then decrypt the data to get the result as if the computations had been run on the unencrypted data.

Such a scheme has wide reaching implications for cloud computing. Computations on sensitive data, just like regular data, could now be performed in the cloud with the added security that even the cloud service provider couldn't "see" the secure data. With such a benefit one might ask why the encryption scheme is not used currently? It is because, while Craig Gentry's scheme was theoretically sound, it was not quick. As such, recent work has been in finding ways to speed up the scheme. Several improvements in speed have been made and several implementations of those improved schemes have been developed: one being HElib.

As of now HElib is self described as an "assembly language for HE". Our work focused on creating HEIDE, a Homomorphic Encryption IDE, where researchers could write tests at a high-level. This high-level code is then "compiled" into the operations provided by HElib. HElib, like most encryption schemes, can be configured using different setup parameters. These parameters change the run-time and security of the scheme. As such we have also provided an easy way for researchers to simultaneously run their tests using different setup parameters. To support that, timing and memory metrics are provided for each test so that researchers can determine which parameters worked best.

ACKNOWLEDGMENTS

Thanks to:

- My parents, whose support now and during my entire college career has been invaluable.
- Dr. Peterson and Dr. Easton for reading initial drafts of this thesis and providing valuable feedback.
- Shai Halevi for suggesting project ideas related to HElib.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In the past 30 years the world has become digital. Data that was stored previously on paper is being converted to a digital file. Additionally, data is being collected and created by almost everyone with a link to the Internet. However, not all of that data is meant for public consumption. Some data needs to be kept secret. Several cloud services allow one to securely upload private data. This security can guarantee an outsider cannot see your data while it is being sent to the cloud service. In many cases, once the data is uploaded, the cloud service can decrypt the data. It does this because most encryption schemes no longer produce the correct decrypted value once computations have been performed on the data. This is not the case with Homomorphic Encryption schemes.

Homomorphic Encryption was first proposed in the same year RSA was developed [28]. Homomorphic Encryption schemes allow one to perform computations on encrypted data with a guarantee that the decrypted result will match the result, had the computations been performed on the plaintext. This can be done without needing the unencrypted data. Thus, users could encrypt their private data locally; send it to some cloud service along with the computations they would like performed; have the cloud service perform the computations and send back the result; and then decrypt the result with high certainty that no one saw their private data. With such high benefits one might ask why such a scheme is not widely used? The answer is that a viable scheme was not created until 2009.

Prior to 2009, the security community was still uncertain that Homomorphic Encryption could be achieved. Several cryptosystems where found to be partially homomorphic. Such systems could only perform addition of ciphertexts, or only perform multiplication of ciphertexts, but none could support arbitrary computations on ciphertexts. Such a scheme that could perform arbitrary computations on ciphertexts is considered to be a Fully Homomorphic Encryption Scheme (FHE). Craig Gentry, in 2009, was the first to propose a viable FHE scheme [17].

Gentry's scheme was an asymmetric encryption scheme which used ideal lattices. Essentially one generates a secret key and then a number of public keys, each containing "noise". This was done in such a way that it was hard for an attacker to generate the secret key from the public keys. The public keys were then used to encrypt data. From there computations could be performed on the encrypted data. However, the "noise" in the ciphertext grew with each additional computation. At a certain point the secret key could no longer be used to decrypt because the "noise" had grown to large. Such a scheme was called a Somewhat Homomorphic Encryption (SWHE) scheme. In order to make his scheme a FHE scheme Gentry developed a technique called bootstrapping.

Once the "noise" in a ciphertext grows too large it is no longer able to be decrypted properly. Bootstrapping solves this problem. It does so by homomorphically decrypting the ciphertext, performing a single computation on it, and then recrypting under a different public key. After adding bootstrapping Gentry had achieved an FHE scheme. Unfortunately, the bootstrapping procedure was not very fast and had to be performed on every computation after the one where the "noise" grew too large.

Since Gentry's initial scheme several others have been developed [33, 31, 32, 13, 14]. These schemes have increased performance. One scheme even managed to avoid requiring bootstrapping be used. Such a scheme is called Leveled FHE scheme. Essentially if one knows the number of levels within their computation beforehand they can avoid using bootstrapping. This scheme is referred to as the BGV scheme, after its creators, Brakerski, Gentry, and Vaikuntanathan.

Several implementations have been developed for the individual schemes. The primary implementations at present are HElib and FHEW. HElib is a an implementation of the BGV scheme and as of early 2015 supports bootstrapping [5, 25, 23]. FHEW is an implementation which sought to optimize the bootstrapping procedure as uymuch as possible [2, 16]. To do so they used the FFTW (Fastest Fourier Transform in the West) library, which is where they derived their own name, Fastest Homomorphic Encryption in the West (FHEW). For our work, we have solely worked with HElib.

HElib is self described as "assembly language for HE". It is a C++ library that is rather robust even though it is in its infancy. It is not the easiest to use, unfortunately. There is no key generation method and instead key generation takes multiple steps. Several other minor issues also are present with HElib that could be easily solved. As such, we have developed a C++ wrapper class BGV_HE which sits on top of HElib and provides a cleaner interface for users.

We have also developed an Integrated Development Environment (IDE) to be used with HElib. This IDE is called HEIDE (pronounced 'hide'). The IDE is written in Python and allows researchers the ability to easily alter setup parameters to see how runtime and memory usage are affected. In addition, we have developed a Python module, PyHE, which allows users to write Python code that interacts directly with the BGV_HE class. Users write Python code within the IDE and then for each setup configuration the algorithm they define is run. We have also developed a convenient syntax that can be used within HEIDE to write code that makes the code easier to read and shorter overall.

The rest of this paper is formatted as follows. We begin with relevant background information in Chapter 2. We discuss relevant related work in Chapter 3. Then we outline our design and implementation in Chapter 4 and Chapter 5, respectively. Next, we present an evaluation of our implementation in Chapter 6. After that, we discuss some possible future work in Chapter 7. We then finish with some concluding thoughts in Chapter 8.

CHAPTER 2

Background

2.1 Notation

The following notation will be used throughout the paper. Assignment of variable x to variable y will be denoted x $\leftarrow$ y. If A is a set then $x \overset{R}{\leftarrow} A$ means that x was selected from A using the uniform distribution. If A is an algorithm then $x \leftarrow A$ means that x is assigned the output that resulted from running A. For integers x, d we denote the reduction of x modulo d as $[x]_d$. The result of $[x]_d$ is an element of the interval $\left(-\frac{d}{2}, \frac{d}{2}\right]$. If y is a vector then we let y_i denote the i^{th} component of y. Polynomials over an indeterminate X will be (except for the cyclotomic polynomials) denoted by uppercase letters, e.g. $F(X)$. The m^{th} cyclotomic polynomial will be denoted as usual as $\Phi_m(X)$. Elements of finite fields and number fields defined by a polynomial $F(X)$, i.e. elements of $\mathbb{F}_2[X]/F(X)$ and $\mathbb{Q}[X]/F(X)$, can also be represented as polynomials in some fixed root of $F(X)$ in the algebraic closure of the base field. Such polynomials are denoted using lower case Greek letters, with the fixed root also a lower case Greek letter. For instance $\gamma(\theta)$, where $F(\theta) = 0$. If fields F and F' are isomorphic, we denote that as $F \cong F'$.

2.2 Relevant Mathematical Background

In section 2.3.3 the BGV homomorphic encryption scheme, upon which HElib is based, is discussed. Within that section we will mention that the BGV scheme is considered to be SIMD. In order to understand why the scheme is considered SIMD, some mathematical concepts must be first understood. Those concepts are discussed in the following subsections.

2.2.1 Fields

Understanding fields is very important to the construction of a ciphertext within the BGV scheme. In order to understand fields, we first need to define and understand a couple of simpler concepts. We will use these simpler concepts to describe the more complex concept of a field.

The first concept to understand is a group. A *group* $\langle G, * \rangle$ is a set G together with a binary operator $*$ such that the following axioms hold:

1. $*$ is associative. That is, for all $a, b, c \in G$,

$$(a * b) * c = a * (b * c).$$

2. There exists an identity element e for $*$. If e is the identity element for $*$ then, $e \in G$ and for all $x \in G$,

$$e * x = x * e = x.$$

3. Inverses exist. Formally, if $a \in G$ then there exists a element $a' \in G$ such that

$$a * a' = a' * a = e.$$

If the binary operator is commutative, then G is called an *abelian group*. One such group that is abelian is $\mathbb{Z}_n$. While one might think that all groups are abelian, a counter example is the dihedral group of order 6.

Building off the definition of a group we can define a ring. A *ring* $\langle R, +, \cdot \rangle$ is a set R with the two binary operations $+$ and $\cdot$; more commonly referred to as, respectively, *addition* and *multiplication*. The two binary operations are defined on R such that the following axioms hold:

1. $\langle R, + \rangle$ is an abelian group.

2. Multiplication is associative.

3. For all $a, b, c \in R$, the left distributive law,

$$a \cdot (b + c) = (a \cdot b) + (a \cdot b)$$

and right distributive law,

$$(a + b) \cdot c = (a \cdot c) + (b \cdot c)$$

hold.

Multiplication in a ring is traditionally written by omitting the $\cdot$, e.g. $a \cdot b$ is written ab.

The following paragraph will introduce many terms, which will be used to define a field. If in a ring multiplication is commutative, then that ring is called a *commutative ring*. If a ring has a *multiplicative identity* element e, i.e. if $a \in R$, then $ae = ea = a$, then that ring is a *ring with unity*. The multiplicative identity element of a ring R is denoted as 1_R and is called "unity". The additive identity element of a ring is denoted as 0_R. The *multiplicative inverse* of an element a in a ring R with unity $1_R \neq 0_R$, is the element $a^{-1} \in R$ such that $aa^{-1} = a^{-1}a = 1_R$. A couple examples of commutative rings with unity are $\mathbb{Z}$ and $\mathbb{Q}[x]$. The unity element in $\mathbb{Z}$ is 1, while it is $1 + 0x$ in $\mathbb{Q}[x]$.

The last piece of the puzzle in defining a field is a division ring. Let R be a ring with unity $1 \neq 0$. If $u \in R$ has a multiplicative inverse in R, then u is a *unit*. If every nonzero element of R is a unit, the R is a *division ring*. A *field* is a commutative division ring. We mentioned before that $\mathbb{Z}$ was a commutative ring with unity, however it is not a field. This is because for all $a \in \mathbb{Z}$ where $a \neq 1$ or -1, there does not exist a multiplicative inverse for a in $\mathbb{Z}$. $\mathbb{R}$ however is a field because all nonzero elements are have inverses in $\mathbb{R}$.

2.2.2 Algebraic Closure

In order to define the algebraic closure we first must define some preliminary topics. A field E is an *extension field of a field* F if F is a subfield of E, denoted $F \leq E$. An element α of an extension field E of a field F is *algebraic over* F if $f(\alpha) = 0$ for some nonzero $f(x) \in F[x]$. The *algebraic closure of* F *in* E is the set

$$\overline{F}_E = \{\alpha \in E : \alpha \text{ is algebraic over } F\}.$$

The algebraic closure is a subfield of E. The field $\mathbb{C}$, containing the complex numbers, is the algebraic closure of $\mathbb{R}$.

2.2.3 Polynomials

Let R be a ring. A *polynomial* $f(x)$ with coefficients in R is an infinite formal sum

$$\sum_{i=0}^{\infty} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n + \ldots,$$

where $a_i \in R$ and $a_i = 0$ for all but a finite number of the i's. The largest such i where $a_i \neq 0$ is the degree of $f(x)$. An element in R is called a *constant polynomial.* For simplicity, if the degree of $f(x)$ is d then we may write $f(x) = a_0 + a_1 x + \cdots + a_n x^n + \ldots$ as $f(x) = a_0 + a_1 + \cdots + a_d x^d$. Addition and multiplication of polynomials with coefficients in a ring R are defined in the usual way. If

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n + \ldots$$

and

$$g(x) = b_0 + b_1x + \cdots + b_nx^n + \ldots$$

then for polynomial addition, we have

$$f(x) + g(x) = c_0 + c_1x + \cdots + c_nx^n + \ldots,$$

where $c_n = a_n + b_n$. For polynomial multiplication, we have

$$f(x)g(x) = d_0 + d_1x + \cdots + d_nx^n + \ldots,$$

where $d_n = \sum_{i=0}^{n} a_i b_{n-i}$.

Now that we have seen the formal definition of a polynomial, some special polynomials used in the BGV scheme need to be defined. If a polynomial f has degree d and the coefficient on the x^d term of f is 1, then f is called a *monic polynomial.* For a ring R, the set $R[x]$, is the set of all polynomials with coefficients in R. It is important to note that x in this case is called an *indeterminate* and not a variable. A non-constant polynomial $f(x) \in F[x]$ is *irreducible over* F if $f(x)$ cannot be expressed as the product $g(x)h(x)$, where $g(x)$ and $h(x)$ are both polynomials in $F[x]$ of lower degree than $f(x)$. It is important to note here that $f(x)$ may be irreducible over F, but might not be in some larger field containing F. For example, consider $f(x) = x^2+1 \in \mathbb{R}[x]$. This polynomial is irreducible over the real numbers but not over the complex numbers. It factors in $\mathbb{C}$ to become $f(x) = g(x)h(x) = (x + \sqrt{-1})(x - \sqrt{-1})$.

2.2.4 Ideals

An additive subgroup N of a ring R that satisfies the properties

$$aN \subseteq N \text{ and } Nb \subseteq N \text{ for all } a, b \in R$$

is an *ideal*. The additive cosets of N form a ring R/N with the binary operations defined by

$$(a+N) + (b+N) = (a+b) + N$$

and

$$(a+N)(b+N) = ab + N.$$

The ring R/N is called the *factor ring* (or *quotient ring*) *of* R *by* N. One such factor ring is $\mathbb{Z}/n\mathbb{Z} = \{0 + n\mathbb{Z}, \ \ldots, \ (n-1) + n\mathbb{Z}\}$.

Two ideals A and B in commutative ring R are considered coprime if $A + B = R$. Also, if A and B are coprime then $AB = A \cap B$.

Let I be an ideal in a commutative ring with unity R. Let $B_I = \{b_1, \cdots, b_r\}$ be a subset of I. If $I = \langle b_1, \cdots, b_k \rangle$ then B_I is a basis of I.

2.2.5 Homomorphisms and Isomorphisms

For rings R and R' and a map $\phi : R \to R'$, ϕ is a *homomorphism* if for all $a, b \in R$ the following hold:

1. $\phi(a+b) = \phi(a) + \phi(b)$, and

2. $\phi\,(ab) = \phi\,(a)\,\phi\,(b)$.

3. $\phi(0_R) = 0_{R'}$

Usually $\phi(1_R) = 1_{R'}$ as well. If ϕ is one-to-one and onto R' then ϕ is an *isomorphism*, and R and R' are said to be *isomorphic*. One can see that $\mathbb{Z}_n \cong \mathbb{Z}/n\mathbb{Z}$ because for $a \in \mathbb{Z}_n$, $a \longrightarrow a + n\mathbb{Z}$

2.2.6 Chinese Remainder Theorem

We begin this section by presenting the Chinese Remainder Theorem (CRT) with respect to integers so that those unfamiliar with the theorem can get an idea for what is happening. We then introduce the Chinese Remainder Theorem for Commutative Rings.

Let $n_1, \cdots, n_k$ be positive integers that are pairwise coprime, i.e. for $i \neq j$, $gcd\,(n_i, n_j) = 1$. Then for any finite sequence of integers $a_1, \cdots, a_k$, there exists an integer x which solves the system of simultaneous congruences below.

$$\begin{cases} \mathrm{x} \equiv & a_1 \pmod{n_1} \\ & \cdots \\ \mathrm{x} \equiv & a_k \pmod{n_k} \end{cases}$$

For example, consider the small system

$$\begin{cases} \mathrm{x} \equiv & 1 \pmod{2} \\ \mathrm{x} \equiv & 0 \pmod{3} \\ \mathrm{x} \equiv & 4 \pmod{5} \end{cases}$$

In this example the solution is $x = 9 + 30r$, where $r \in \mathbb{Z}$.

One can extend this idea to create the more algebraic theorem, the Chinese Remainder Theorem for Commutative Rings. Let R be a commutative ring and $I_1, \cdots, I_k$ be ideals, see section 2.2.4, that are pairwise coprime (see section 2.2.4 for what it means for ideals to be coprime). Then the CRT states that the product I of these ideals is equal to their intersection and the quotient ring R/I is isomorphic to the product ring $R/I_i \times \cdots \times R/I_k$, via the isomorphism ϕ defined below.

$$\phi : \begin{cases} R/I & \longrightarrow & R/I_1 \times \cdots \times R/I_k \\ (x + I) & \longmapsto & (x + I_1, ..., x + I_k) \end{cases}$$

2.3 Homomorphic Encryption Timeline and Background

The idea of creating a fully homomorphic encryption scheme was first introduced in 1978 by Rivest, Adelman, and Dertouzos [28] shortly after the creation of RSA. RSA is what's known as a partially homomorphic scheme. It only contains the multiplicative homomorphic property. Given an RSA public key $pk = (N, e)$ and ciphertexts $\{c_i \leftarrow p_i^e \bmod N\}$, one can compute $\prod_i c_i = (\prod_i p_i)^e \bmod N$. The LHS product is a ciphertext that is the encryption of the product of the original plaintexts. Naturally Rivest et al. [28] wondered if there was a scheme which was fully homomorphic. A scheme where an efficient algorithm **Eval** could, for a valid public key pk, boolean circuit C, and ciphertexts $\{c_i \leftarrow \textbf{Encrypt}\,(pk, p_i)\}$, output

$$c \leftarrow \textbf{Eval}\,(pk, C, c_i, \cdots, c_n)\,,$$

where it was also the case that

$$c \leftarrow \textbf{Encrypt}\,(pk, C\,(p_i, \cdots, p_n))\,.$$

Such a scheme was not introduced, at least not a viable one, until 2009.

2.3.1 Gentry's Initial Construction

Craig Gentry was the first to propose a viable fully homomorphic scheme [17]. His scheme could perform the addition and multiplication operations on ciphertexts. From this one could construct circuits to perform arbitrary computations. The scheme has four main algorithms (described below): **KeyGen**, **Encrypt**, **Decrypt**, and **Evaluate**. Below we describe Gentry's initial construction abstractly just as he did. This description uses rings and ideals, instead of ideal lattices. For those interested, the more precise construction using ideal lattices can be found in [17].

We begin the description by making some assumptions. Assume that one has a fixed ring R that is set according to a security parameter λ and one also has a fixed basis B_I of an ideal $I \subset R$. In all algorithms assume the plaintext space P is (a subset of) R mod B_I. The notation R mod B_I is used, as in [17], to denote the set of distinguished representatives of $r + I$ over $r \in R$, with respect to the particular basis B_I of I. Also let there be an algorithm $\textbf{IdealGen}(R, B_I)$ that outputs public and secret bases B_j^{pk} and B_j^{sk} of some variable ideal J, where I and J are relatively prime. Finally assume that there exists another algorithm $\mathbf{Samp}\Big(x, B_I, R, B_J^{pk}\Big)$ that samples from the coset $x + I$.

KeyGen(R, B_I):

The input parameters are a ring R and a basis B_I of I. It sets

$$\left(B_J^{pk}, B_J^{sk}\right) \leftarrow \mathbf{IdealGen}\left(R, B_I\right).$$

The public key pk includes R, B_I, B_J^{sk}, and **Samp**. The secret key sk also includes B_J^{sk}. It outputs (pk, sk).

Encrypt(pk, p):

The input parameters are the public key pk and a plaintext $p \in P$. It sets

$$c' \leftarrow \mathbf{Samp}\left(p, B_I, R, B_j^{pk}\right)$$

and then outputs

$$c \leftarrow c' \bmod B_j^{pk}.$$

Decrypt(sk, c):

The input parameters are the secret key sk and a ciphertext c. It outputs

$$p \leftarrow \left(c \bmod B_J^{sk}\right) \bmod B_I.$$

Evaluate$(pk, C, \{c_1, \cdots, c_n\})$:

The input parameters are the public key pk, a permitted circuit C of addition and multiplication gates, and a set of ciphertexts. It calls **Add** and **Mult** in accordance with C's sequence to compute the output ciphertext c.

Add(pk, c_1, c_2). Output $c_1 + c_2 \bmod B_J^{pk}$.

Mult(pk, c_1, c_2). Output $c_1 \times c_2 \bmod B_J^{pk}$.

Gentry showed that his initial abstract scheme was semantically secure using the Ideal Coset Problem (ICP).

Definition 1 *(ICP). Fix R, B_I, algorithm **IdealGen**, and an algorithm **Samp**$_1$, that efficiently samples R. The challenger sets $b \xleftarrow{R} \{0,1\}$ and $\left(B_j^{pk}, B_J^{sk}\right) \leftarrow$ **IdealGen**(R, B_I). If $b = 0$, the challenger sets $r \leftarrow$ **Samp**$_1(R)$ and $t \leftarrow r \bmod B_J^{pk}$. If $b = 1$ the challenger samples t uniformly from $R \bmod B_j^{pk}$. The problem: guess b given $\left(t, B_j^{pk}\right)$.*

An attacker cannot determine the original message because "noise" has been added to the message to make it a secret. As messages are added and multiplied this "noise" grows. Gentry noted that the "noise" might grow too large and stop the decryption algorithm from working properly. Schemes which could only evaluate low degree polynomials, to avoid the "noise" from growing too large, are called *somewhat homomorphic encryption (SWHE)* schemes. Gentry's main contribution was finding a way to "refresh" a "noisy" ciphertext so that further computations could be performed on it.

Gentry called this method of "refreshing" a ciphertext *bootstrapping.* For (an albeit simplified) example, suppose we have a SWHE scheme $\mathcal{E}$, plaintext space $P = \{0, 1\}$, and the circuits $\mathcal{E}$ accepts are boolean. Suppose we have a ciphertext c_1 that encrypts p under pk_1 that needs to be refreshed. Suppose we also have $\overline{sk_1}$, which is the encryption of secret key sk_1, for public key pk_1, under pk_2. Let $\overline{\langle sk_{1j}\rangle}$ be a vector of the bits of $\overline{sk_1}$. Let $D_{\mathcal{E}}$ be the decryption circuit for $\mathcal{E}$. The following algorithm will allow us to "refresh" the ciphertext.

Recrypt$\left(pk_2, D_{\mathcal{E}}, \overline{\langle sk_{1j}\rangle}, c_1\right)$:

$$\text{Set } \overline{c_{1j}} \leftarrow \textbf{Encrypt}\left(pk_2, c_{1j}\right)$$

$$\text{Output } c_2 \leftarrow \textbf{Evaluate}\left(pk_2, D_{\mathcal{E}}, \langle\overline{\langle sk_{1j}\rangle}, \overline{\langle c_{1j}\rangle}\rangle\right)$$

One can see that **Evaluate** takes the bits of sk_1 (encrypted under pk_2) and c_1 and runs the decryption circuit on them. This produces a new ciphertext c_2 which is the encryption of the original message p_1 under pk_2. Running the decryption in this way removed the "noise" in the ciphertext, however **Evaluate** introduced some new "noise". As long as the new "noise" is smaller than the old "noise" then progress has been made. It is also important to note that if we can run **Recrypt** on a circuit which is the decryption circuit augmented with another operation and still have the new "noise" be less than the original, then one can process any circuit. Evaluation of such circuits would take a very long time however because **Recrypt** would have to be used for every computation after the first time the “noise” grew too large. Gentry showed that one could in fact add bootstrapping to the SWHE scheme he had described to create a *fully homomorphic encryption (FHE)* scheme.

In 2010, Smart and Vercaunteren were the first to attempt an implementation of Gentry’s scheme [31]. They implemented a variant of the scheme which used principal-ideal lattices and required the determinant of the lattice to be a prime number. They were only able to implement the SWHE scheme. They could not support large enough parameters so that the decryption circuit could be squashed to one that would allow for bootstrapping. This was because the key generation process was rigorous and as a result they could only generate keys with dimensions < 2048. In order to get a lattice with a prime determinant many lattices had to be generated. After finding one, still more computations needed to be done in order to compute the secret key. They estimated that the squashed decryption procedure would be a several-hundred-degree polynomial and require a lattice of dimension 2^{27}, which was well beyond what they could generate.

Later in 2010, Gentry and Halevi presented another implementation of Gentry’s original scheme [18]. This implementation was the first one to support bootstrapping. They achieved this by dropping the prime determinant requirement. They also found a quicker way of computing the secret key. On top of this they also found many optimizations that squashed the decryption procedure into a much more manageable, degree 15 polynomial.

To evaluate their implementation they used a IBM System x3500 server. It had a 64-bit quad-core Intel Xeon E5450 processor, running at 3 GHz, with 12MB L2 cache and 24GB of RAM. They used Shoup's NTL library [3] and GNU's GMP library [30]. They tested their implementation with lattices of dimension 512, 2048, 8192, and 32768. The public key sizes were, respectively, 17MB, 69MB, 284MB, and 2.25GB. Key generation took 2.5 sec, 41 sec, 8.4 min, and 2.2 hours. The time it took to run bootstrapping was 6 sec, 32 sec, 2.8 min, and 31 min.

2.3.2 A second FHE scheme

In mid 2010 the second FHE encryption scheme was introduced by Dijk, et al. [33]. This scheme provided fully homomorphic encryption over the integers, rather than ideal lattices over a ring. Its main appeal was that it was conceptually simpler to understand then Gentry's initial scheme. Below we present the construction of their SWHE scheme for the plaintext space $P = \{0, 1\}$. The construction of their FHE scheme is slightly different, as there are additional constraints that need to be met in order to allow bootstrapping, but it retains the same basic concepts as the SWHE construction. Below we outline the same four algorithms used by Gentry to build his scheme. The following parameters, in relation to the security parameter λ, are used in the scheme:

1. $\rho = \omega(\log \lambda)$, where ω is the Wright Omega Function.

2. $\eta \geq \rho \cdot \Theta\left(\lambda \log^2 \lambda\right)$, where $\Theta(n)$ means the computational complexity is bounded below and above by n.

3. $\gamma = \omega\left(\eta^2 \log \lambda\right)$

4. $\tau \geq \gamma + \omega(\log \lambda)$

KeyGen(λ)

Take as input the security parameter, λ. The secret key, sk, is an odd η-bit integer $sk \overset{R}{\leftarrow} (2\mathbb{Z}+1) \cap [2^{\eta-1}, 2^{\eta})$. For the η-bit odd positive integer p, the following distribution over the γ-bit integers is used:

$$\mathcal{D}_{\gamma,\rho}(p) = \left\{ \text{choose } q \overset{R}{\leftarrow} \mathbb{Z} \cap \left[0, \frac{2^{\gamma}}{p}\right), r \overset{R}{\leftarrow} \mathbb{Z} \cap (-2^{\rho}, 2^{\rho}) \;:\; \text{output } x = pq + 2r \right\}.$$

To generate the public key sample $x_i \overset{R}{\leftarrow} \mathcal{D}_{\gamma,\rho}(sk)$ for $i = 0, \cdots, \tau$. Relabel so that x_0 is the largest. Restart unless x_0 is odd and $[x_0]_p$ is even. The public key is $pk = \langle x_0, \cdots, x_{\tau} \rangle$.

Encrypt(pk, m)

Take as input the public key, pk, and a message, $m \in P$. Choose a random subset $S \subseteq \{1, \cdots, \tau\}$ and $r \overset{R}{\leftarrow} \left(-2^{2\lambda}, 2^{2\lambda}\right)$. Output

$$c \leftarrow \left[m + 2r + \sum_{i \in S} x_i \right]_{x_0}.$$

Evaluate$(pk, C, c_1, \cdots, c_t)$

Take as input the public key, pk, a circuit, C, and t ciphertexts. Apply the addition and multiplication gates of C to the ciphertexts and return the resulting integer.

Decrypt(sk, c)

Takes as inputs the secret key, sk, and a ciphertext, c. Output

$$m' \leftarrow (c \bmod p) \mod 2.$$

The security of this scheme comes from the approximate GCD problem. Simply, this problem says that it is hard to guess sk from the x_i. This scheme, while conceptually simpler, still was not very efficient. Subsequent work in the field has been in attempting to create an efficient scheme.

2.3.3 Second Generation

The second generation FHE schemes sought to be more efficient than those that had come before. The first systems were slow. In order for an FHE scheme to be deployed on a massive scale it needs to become “usable”. There are three main improvements made in the second generation: a new construction that allowed for SIMD operations on ciphertexts; a new technique called modulus switching which replaced the slow bootstrapping process; and a new scheme which only required polylog overhead.

SIMD Operations

Smart and Vercaunteren introduced a scheme which had smaller key and ciphertext size than previous schemes [31]. They noted that their scheme could support SIMD operations. Their scheme however had a slow key generation process, which Gentry and Halevi addressed in their implementation of Gentry's original scheme [18]. However, Gentry and Halevi didn't include the SIMD style operations mentioned. As such, Smart and Vercaunteren detailed the SIMD operations in full [32].

To understand the SIMD operations they first set up a number of finite fields and homomorphisms. They let $F(X) \in \mathbb{F}_2[X]$ be a monic polynomial of degree N that splits into r distinct irreducible factors of degree $d = \frac{N}{r}$. Thus,

$$F(X) = \prod_{i=1}^{r} F_i(X).$$

This polynomial defines a number field $\mathbb{K} = \mathbb{Q}(\theta) \cong \mathbb{Q}[X]/\langle F\rangle$, where θ is some fixed root in the algebraic closure of $\mathbb{Q}$. Let $A = \mathbb{F}_2[X]/\langle F\rangle$, then by the CRT there exists natural isomorphisms

$$\begin{aligned} A &\cong \mathbb{F}_2[X]/\langle F_1\rangle \times \cdots \times \mathbb{F}_2[X]/\langle F_r\rangle \\ &\cong \mathbb{F}_{2^d} \times \cdots \times \mathbb{F}_{2^d}. \end{aligned}$$

Let θ_i be a fixed root of $F_i(X)$ in the algebraic closure of $\mathbb{F}_2$. Define $\mathbb{L}_i = \mathbb{F}_2[X]/F_i$. Note that all the $\mathbb{L}_i$ are isomorphic under the isomorphism

$$\Delta_{i,j} : \begin{cases} \mathbb{L}_i & \longrightarrow & \mathbb{L}_j \\ \alpha(\phi_i) & \longmapsto & \alpha(\rho_{i,j}(\theta_j)), \end{cases}$$

where $\rho_{i,j}(\theta_j)$ is a fixed root of F_i in $\mathbb{L}_j$, i.e. $F_i(\rho_{i,j}(X)) \equiv 0 \bmod F_j(X)$.

For each divisor n of d, the finite field $\mathbb{K}_n := \mathbb{F}_{2^n}$ is contained in $\mathbb{F}_{2^d}$. Assume a fixed representation for $\mathbb{K}_n$ as $\mathbb{F}_2[X]/K_n(X)$, for some irreducible polynomial $K_n(X) \in \mathbb{F}_2[X]$ of degree n. Let ψ denote a fixed root of $K_n(X)$ in the algebraic closure of $\mathbb{F}_2$. Clearly K_n is contained in each of the $\mathbb{L}_i$, which allows for the homomorphic embeddings given by

$$\Psi_{n,i} : \begin{cases} \mathbb{K}_n & \longrightarrow & \mathbb{L}_i \\ \alpha(\psi) & \longmapsto & \alpha(\sigma_{n,i}(\theta_i)), \end{cases}$$

where $K_n(\sigma_{n,i}(X)) \equiv 0 \bmod F_i(X)$.

Combining the above homomorphism with the Chinese Remainder Theorem we obtain the homomorphic embedding of $l \leq r$ copies of $\mathbb{K}_n$ into A by

$$\Gamma_{n,l} : \begin{cases} \mathbb{K}_n^l & \longrightarrow & A \\ (\kappa_1(\psi), \cdots, \kappa_l(\psi)) & \longmapsto & \sum_{i=1}^{l} \kappa(\sigma_{n,i}(X)) \cdot H_i(X) \cdot G_i(X), \end{cases}$$

where the polynomials $H_i(X)$ and $G_i(X)$ are given by the Chinese Remainder Theorem and defined as

$$H_i(X) \leftarrow F(X)/F_i(X)$$

and

$$G_i(X) \leftarrow 1/H_i(X) \mod F_i(X).$$

Under this construction there are two ways to compute on elements of $\mathbb{K}_n^l$:

1. Compute component wise on the vectors of l elements in $\mathbb{K}_n$.

2. Begin by mapping all inputs to A using $\Gamma_{n,l}$, then perform computations in A, and finally map back to $\mathbb{K}_n^l$ using $\Gamma_{n,l}^{-1}$.

Smart and Vercaunteren provide a concrete example to give one a better idea of what this construction allows for. Let $F(X)$ be the 3485-th cyclotomic polynomial, which means $F(X)$ is of degree $N = \phi(3485) = 2560$. Modulo two, it factors into 64 polynomials, each of degree 40. This means that one can compute in parallel on up to 64 elements of any subfield of $\mathbb{F}_{2^{40}}$. In particular if we let $n = 8$ and $l = 16$, one can perform SIMD operations on 16 elements of $\mathbb{F}_{2^8}$, which is basically an AES state matrix.

Under their construction one creates a plaintext vector which consists of "plaintext slots". Off of this one can create a "packed" ciphertext which encrypts the messages in the "plaintext slots". Then if one adds or multiplies two ciphertexts together, they have essentially component-wise added or multiplied the elements in the "plaintext slots".

Modulus Switching

In all scheme presented above for deep circuits bootstrapping had to be used. However, bootstrapping was slow. Brakerski, Gentry, and Vaikuntanathan developed a scheme, the BGV scheme, which could perform fully homomorphic operations on arbitrary polynomial-size circuits without the use of bootstrapping [13, 12]. Their scheme switched to a new infeasible problem to provide security: the ring learning with error (RLWE) problem.

The RLWE problem was introduced by Lyubaskevsky, Peikert, and Regev [26]. The BGV scheme uses a simplified version of the problem, which is intended for use in a special case [14, 27]. The problem is defined:

Definition 2 *(RLWE). For security parameter λ, let $f(x) = x^d + 1$ where $d = d(\lambda)$ is a power of 2. Let $q = q(\lambda) \geq 2$ be an integer. Let $R = \mathbb{Z}[x]/(f(x))$ and let $R_q = R/qR$. Let $\chi = \chi(\lambda)$ be a distribution over R. The $RLWE_{d,q,\chi}$ problem is to distinguish the following two distributions: In the first distribution, one samples (a_i, b_i) uniformly from R_q^2. In the second distribution, one first draws $s \leftarrow R_q$ uniformly and then samples $(a_i, b_i) \in R_q^2$ by sampling $a_i \in \mathbb{R}_q$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = a_i \cdot s + e_i$. The $RLWE_{d,q,\chi}$ assumption is that $RLWE_{d,q,\chi}$ problem is infeasible.*

It has been shown that the already-proven infeasible shortest vector problem (SVP) over ideal lattices can be reduced to the RLWE problem. Using this infeasible problem allows the BGV scheme to have 2^{λ} security against known attacks.

In addition to switching the hard problem on which the scheme was built they also introduced the process of modulus switching, which was first developed by Brakerski and Vaikuntanathan [15]. This allowed them to not need to use bootstrapping in their scheme, but they noted that bootstrapping could be used as an optimization. We first define modulus switching and then describe how bootstrapping can be used as an optimization.

Definition 3 *(Modulus Switching). Let R be a ring. Let $c \in R^n$ be a ciphertext integer vector (where n is its dimension) which encrypts the message m. Let $sk \in R^n$ be the secret key. Let q be an odd modulus such that $m = \left[[\langle c, sk\rangle]_q\right]_2$. Let p be an odd modulus. Define c' to be the integer vector closest to $(p/q) \cdot c$ such that $c' \equiv c \mod 2$. As long as $\left|[\langle c, sk\rangle]_p\right| < (q/2) - (q/p) \cdot l_1(sk)$, we know*

$$[\langle c, sk\rangle]_p = \left[[\langle c, sk\rangle]_q\right]_2$$

and

$$\left|[\langle c, sk\rangle]_p\right| < (p/q) \cdot \left|[\langle c, sk\rangle]_q\right| + l_1(sk),$$

where $l_1(sk)$ is the l_1-norm of sk.

Using modulus switching one can reduce the "noise" of the ciphertext without knowing the secret key or using bootstrapping. In order to evaluate an L-level circuit one has to first create a modulus chain $q_1, \cdots, q_L$, where $q_i > q_j$ when $i < j$. Brakerski et al. showed that this scheme could evaluate an L-level circuit with $\widetilde{O}(\lambda \cdot L^3)$. However, if one wants to evaluate very deep circuits many moduli must be generated during the setup phase, which can make the scheme inefficient. As such bootstrapping was reintroduced as an optimization. Using bootstrapping one could evaluate arbitrary depth circuits and require the largest modulus to be $\widetilde{O}(\lambda)$ bits in length. Using bootstrapping the per-gate computation becomes $\widetilde{O}(\lambda^2)$. We describe the BGV bootstrapping procedure in more detail in section 2.3.4.

Gentry, Halevi, and Smart developed an implementation of the BGV scheme and reran tests to see how long it would take to perform the homomorphic evaluation of an AES circuit [21]. They detailed two different test cases. First they processed 54 blocks in each evaluation, which took a little over 36 hours in total running time. This resulted in an amortized rate of around 40 minutes per block. The second test processed 720 blocks in each evaluation and took a little over two and a half days. This leads to an amortized rate of around 5 min per block. Thus the BGV implementation was getting closer to becoming "usable".

Polylog Overhead

Building off the SIMD work of Smart and Vercaunteren and the BGV scheme, Gentry, Halevi, and Smart showed that homomorphic evaluation of wide arithmetic circuits could be accomplished with only polylog overhead [20]. Concretely they presented a scheme which for the security parameter λ could evaluate width-$\Omega(\lambda)$ circuits with t gates in time $t \cdot \text{polylog}(\lambda)$. Their scheme used "packed ciphertexts". A main problem with "packed ciphertexts", that they solved was: what if someone wanted to add the i^{th} “plaintext slot” of one vector to the j^{th} “plaintext slot” of another vector? Previously the way to do this was to “unpack” the slots in separate ciphertexts so that the operation could be performed and then “repack” the ciphertext. This approach however would not yield an efficient FHE scheme. They introduced the idea of constructing a permutation network. This would allow them to permute data within the “plaintext slots”, so that one could add the i^{th} “plaintext slot” of one vector with the j^{th} “plaintext slot” of another vector without needing to “unpack” and then “repack” the ciphertexts.

2.3.4 HElib

A few implementations of FHE scheme have been built, but for our work we looked primarily at an implementation of the BGV encryption scheme. The implementation was developed by Shai Halevi and Victor Shoup and is called HElib [5]. HElib is written in C++ and uses the NTL mathematical library [3] as well as the GMP library [30]. HElib is uses the BGV encryption scheme and the Smart-Vercaunteren ciphertext packing techniques. There are several other optimizations and improvements made to decrease run time. Initially bootstrapping was not included in HElib but was added in December of 2014 (and is described below). As of March 2015, HElib also supports multi-threading.

Halevi and Shoup have detailed the numerous algorithms within HElib [24, 25, 23]. In this section we will primarily focus on the parameters involved during the setup stages (as those are the ones that the user of HEIDE will we playing with), and the functions available in HElib to perform the low level operations which we will use after "compiling" the higher level algorithms written in HEIDE.

Shortly after bootstrapping was included in HElib, Gentry, Halevi, and Smart revisited their tests involving the AES circuit [22]. HElib was able to process 180 blocks in about 4 minutes. This lead to an amortized rate of 6 seconds per block.

Setup Parameters

HElib uses the BGV ring-LWE SWHE scheme, described in section 2.3.3. This scheme is defined over a ring $R = \mathbb{Z}[X]/\Phi_m(X)$. A user can provide there own value for m. If they do not, then the HElib provided method `FindM` is used, see Appendix A for function definition. If `FindM` is used an additional parameter, d, must be provided. It corresponds to the degree of the field extension used when trying to find m. It is required that $d \mid \text{ord}(p)$ in $\mathbb{Z}_m^*$ and $\phi(m)/\text{ord}(p) \geq s$, where s is the least number of plaintext slots and $\text{ord}(p)$ is the multiplicative order of p mod m.

The plaintext space in HElib is R_{p^r}, where p is a prime and r is a small positive integer. Both p and r are required parameters.

Recall that the BGV scheme is a (leveled) fully homomorphic scheme. By this we mean that the scheme consists of a sequence of decreasing moduli $q_L \gg \cdots \gg q_0$

which are used for modulus switching in order to decrease the "noise" at each level of the circuit so that bootstrapping is not needed. As such, the user must provide L.

There are also several parameters that have recommended values, but they can be altered. They are c, w, and *security*, which have recommended values 2 or 3, 64, and 128. c represents the number of columns in the key switching matrix, a tool used to switch from one key to another. w represents the Hamming weight of the secret key, which is used when estimating the "noise" in a ciphertext. *security* represents the security parameter and controls how many bits of security the encrypted ciphertext has. Two lists *gens* and *ords* can be given. They correspond to a vector of generators and vector of orders that will be used during key generation.

Low Level Operations

At the moment HElib only provides several low-level functions: set, addition, multiplication, shift, rotate, negate. There are several variations of each function, in case the input data is of two different forms. For our construction we made sure that all input data was of the same form. As such we only use specific variations of the provided functions. A complete list of the operations, along with their function definitions can be found in Appendix A. The algorithm written in HEIDE is "compiled" into these low level operations.

Bootstrapping

Bootstrapping within HElib follows the same basic structure described in [19, 11]. First we will present the structure as described in [19] and then discuss the structure used in HElib, which uses an optimization.

Let level-i ciphertext $ct = (c_0, c_1)$ be the encryption of a plaintext message $m \in R_{p^r}$ with respect to secret key $sk = (1, s)$ such that $[\langle sk, ct\rangle]_{q_i} = [c_0 + s \cdot c_1]_{q_i} \equiv m \bmod p^r$. Since one has to define the number of levels prior to computation it might be the case that the circuit is deeper than the number of levels given. In that case one will have a level-0 ciphertext and need to use bootstrapping as it was originally defined to "recrypt" the ciphertext, ct, to get a new ciphertext, ct'. The new ciphertext is the encryption of the original message with respect to some level-$i > 0$ modulus.

To begin the recryption procedure from [19], where the plaintext space is R_{p^2} we first use modulus switching to compute a new ciphertext, ct'. A specially chosen modulus, $\widetilde{q} = 2^e + 1$ (where e is an integer), is used for this modulus switching procedure. In order to perform the recryption, an encryption of the secret key, sk, with respect to a modulus $Q \gg \widetilde{q}$, is added to the public key to produce $\widetilde{ct}$. This is done is such a way so that $[\langle sk, \widetilde{ct}\rangle]_Q = s \bmod 2^{e+1}$. One then computes

$$\widetilde{ct}' \leftarrow c_1' \cdot \widetilde{ct} + (c_0', 0)$$

which is the encryption of

$$u \leftarrow c_1' \cdot s + c_0' \bmod 2^{e+1}.$$

To decrypt and get the message m back one computes

$$m \leftarrow u\langle e\rangle \oplus u\langle e-1\rangle \oplus u\langle 0\rangle.$$

However we want to be able to still work with an encryption of m so we must perform some operations on $\widetilde{ct}'$ in order to change it into an encryption of m and not of u. This involves first converting $\widetilde{ct}'$ into ciphertexts where the coefficients of u are in the plaintext slots. Then one uses a bit extraction procedure to compute two ciphertexts which are made up of the top and bottom bits of slots. Then, finally, recombine the ciphertexts into a single ciphertext, ct^*, which is the encryption of m.

HElib expands upon this to allow this procedure to be run for arbitrary prime power plaintext spaces. A special recryption key, $\widetilde{sk} = (1, \widetilde{s})$ is used (specifics about $\widetilde{s}$ can be found in Appendix A of [23]). The special secret key is encrypted with respect to a modulus Q and plaintext space mod p^{e+r} where $e > r$. The special modulus is now $\widetilde{q} = p^e + 1$. From this, to decrypt, one computes $u \leftarrow [\langle sk, ct'\rangle]_{p^{e+r}}$ and then $m \leftarrow u\langle r-1, \cdots, 0\rangle_p - \langle e+r-1, \cdots, e\rangle_p \bmod p^r$.

To continue the recryption procedure one adds multiples of $\widetilde{q}$ and p^r to the coefficients of ct' to make them divisible by $p^{e'}$, where $r \leq e' < e$. The resulting ciphertext is denoted $ct'' = (c_0'', c_1'')$. One then computes

$$\widetilde{ct}' \leftarrow (c_1'/p^{e'}) \cdot \widetilde{ct} + (c_0'/p^{e'}, 0)$$

which is the encryption of

$$u' \leftarrow (c_1' \cdot s + c_0')/p^{e'} \bmod p^{e-e'+r}.$$

Again one splits up u' into multiple ciphertexts which have the coefficients of u' in their plaintext slots. One then computes r ciphertexts that contain the digits $e - e' + r - 1$ through $e - e'$ of the integers in the slots. Finally the ciphertexts are combined back to create ciphertext ct^*, which is the encryption of plaintext message m.

CHAPTER 3

Related Work

As homomorphic encryption is a relatively new field there is not an overabundance of related work, but there is some. HElib [5, 25, 23] is not the only implementation of one of the FHE schemes that arose during the second generation. Scarab and FHEW are a couple other open source implementations.

Scarab [9] is part of the hcrypt (`https://www.hcrypt.com`) project, which describes its mission as a “Secret program execution through homomorphic encryption”. Hcrypt also contains projects related to Secure Function Evaluation (SFE), which allows one to obfuscate the function within which data is being computed upon. As a complementary piece Scarab will allow operations to be performed on encrypted data. Thus if a third party is used for computation they will gain no knowledge of the data or the overall computations being done on the data. Scarab is an implementation of a fully homomorphic encryption scheme using large integers. Overall, Scarab follows Gentry's initial construction but, instead of using ideal lattices, uses the approach described by Smart and Vercaunteren. Smart and Vercaunteren's scheme was an integer based approach, and reduced the key and ciphertext sizes.

FHEW [2, 16] is another implementation of an FHE scheme, which sought to create a very time efficient bootstrapping procedure. FHEW uses the FFTW (Fastest Fourier Transform in the West) library, from which it derives its own name, Fastest Homomorphic Encryption in the West (FHEW). FHEW only encrypts a single bit at a time, unlike HElib which can support larger plaintext spaces. FHEW reported that with their scheme one could perform a NAND operation followed by a recryption in less than a second. This is a better recryption time than that of HElib. However, because HElib can support larger plaintext spaces, HElib can achieve a faster overall recryption time than FHEW when they both are given the same large input.

One main issue with the above mentioned schemes is that they are all single threaded. Many of the operations performed within each of the FHE schemes could be performed in parallel, which would greatly speed up performance. One such scheme [29] was written about in early 2014. Kurt Rohloff and David Cousins from Raytheon BBN Technologies discuss their design of an NTRU-like cryptosystem. NTRU is a cyrptosystem that uses lattice based cryptography to perform encryption and decryption. It has been shown to be resistant to attacks that use Shor's algorithm. Rohloff and Cousins system was built to run in a commodity CPU-based computing environment and take advantage of multi-core processors. They ran their implementation on a 64 core server with 2.1GHz Intel Xeon processors and 1TB of RAM in a CentOS environment. However, they noted that at most they only used 10GB of memory and 20 cores during testing. They reported that their implementation provided at least an order of magnitude speedup when compared to the publicly known evaluation times of other FHE schemes.

CHAPTER 4

Design

Homomorphic Encryption saw its first viable scheme in 2009 thanks to Craig Gentry. That scheme consisted of four main algorithms: KeyGen, Encrypt, Decrypt, and Evaluate. All current implementations have built off of that original scheme. For our project we solely worked with the implementation HElib. As of the time of writing, HElib claims to be an "assembly language for HE". As a result, their implementation was not as straightforward as the original scheme. We wanted to design a system that would match, within reason, the original scheme's four algorithms.

Since HElib is written in C++, we sought to write a wrapper class around HElib that would be easy to use while still being useful in most situations. We also sought to create an Integrated Development Environment (IDE) where researchers could easily create tests that used HElib.

The rest of this chapter is organized as follows. We start by describing what features we wanted to incorporate into the C++ wrapper class that we would develop.

Then we discuss the design decisions that went into creating the IDE. Chapter 5 describes in detail how we implemented our design.

4.1 C++ Wrapper Class

The first thing one must do to perform computations on encrypted data using HElib is generate the public and secret keys. Within HElib there is no singular key generation method. First, a user would have to generate an "FHE context", used to create the secret key. since HElib is an implementation of the BGV scheme, one then builds a modulus chain. Next the secret key is built. Then the public key is built. Finally an "encrypted array", used to encrypt plaintext data, is constructed. The average user won't want to do this every time, so a single method must be added to the wrapper class to perform all these steps at once. All the user would need to do is pass in the setup parameters used by these functions and then public and secret keys would be generated for them. The next step is to encrypt data.

Currently HElib has an overloaded `encrypt` method, which can accept plaintext data in varying forms. While useful, a user could easily convert one type of data into another before encryption. So, we decided that having only one `encrypt` method would be the most beneficial for our wrapper class. In HElib when one calls `encrypt` a `Ctxt` object is created and then passed back to the user. Users generally will not be interacting with any of the internal data within the `Ctxt` object and only with the object as a whole. As such, our wrapper class should not pass back the `Ctxt` object, but instead store it internally and pass back a key which they can use to access it. If there is some method within HElib that we have not supported we should allow a user to request a `Ctxt`. After they receive that ciphertext they will most likely alter it. So, we must provide a way for users to update an already stored ciphertext object. As this is a FHE scheme we must allow users to perform computations on ciphertexts.

In order for our wrapper method to be useful, with regards to computing on ciphertexts, we need to support the major operators. Currently HElib supports only the destructive numerical operators and as such so do we. Thus users should be able to add, subtract, and multiply one ciphertext to another. In addition, just like HElib, users should be able to compare one ciphertext to another to see if they are equal.

After all computations are done a user must decrypt all the data. HElib's `decrypt` method returns the plaintext representation of the ciphertext it decrypted. Nothing else can be done and so our class should mimic the already existing method.

HElib currently provides nice timing functionality. Users can measure how long each function was running. They also keep track of how many times the function was called so timing information on a single execution of a function can be seen. This way users can see how functions compare with each other and where the bottlenecks are occurring. Our wrapper method should allow users to call these methods to get timing information as well.

4.2 Integrated Development Environment

In support of our easier to use wrapper method we also wanted to develop an Integrated Development Environment (IDE) where users could easily do research. To make the IDE as easy to use as possible the user should be able to write their tests in a high level language. This high level language will then call the lower level HElib C++ wrapper methods.

There are three parts to conduct a test within an FHE scheme: one must outline the setup parameters to be used, one must outline data is to be encrypted, and finally one must outline the algorithm which computes on the encrypted data. As such, our IDE should separate each of these parts. The IDE should allow users to define each of these parts in some high level language. In addition our IDE should allow users to specify multiple sets of setup parameters and then run the algorithm they constructed for each of those setup parameters. This will allow users to see which setup parameters work and which don't.

In addition the IDE should allow users the ability to see the memory usage related to each of the different runs. This will allow them to see which setup parameters provided the best overall performance. Some users might want to make changes to the underlying functions within HElib and then test them out using the IDE. Thus the IDE needs to allow users to specify which version of HElib to use when running their tests.

CHAPTER 5

Implementation

Homomorphic Encryption is still in its early stages. Researchers are the ones who largely work with it. Currently there are only a couple of implementations, the most well developed of which is HElib. HElib is quite robust for being so new; however, its barrier to entry is high. As such we developed an environment within which new, and experienced, researchers could play around with HElib without necessarily needing as much background knowledge as one would need to use HElib by itself.

Our design considerations and choices made are detailed in more depth in Chapter 4, but here is a quick outline of what we wanted to achieve. Gentry's original homomorphic encryption scheme contained four main algorithms: KeyGen, Encrypt, Decrypt, and Eval. HElib is an implementation of a scheme which evolved out of that original scheme. The scheme that HElib implements is much quicker than the original. However, it lost some ease of use. Our design allowed users to define which parameters to use during key generation, define the data to encrypt, and define an algorithm over the data. Towards this goal we have created a wrapper class around

HElib that has a simpler interface. This class is detailed in more depth in Section 5.1.

For our implementation we wanted users to be able to write their algorithms in a high level language which was easy to use and understand. We choose the high level language Python. We chose Python because of its ease of readability, which will allow researchers to write simple (or possibly complex) algorithms easily. Since HElib is written in C++ we developed a couple Python modules that interact directly with the wrapper class that we created. These Python modules are detailed in Section 5.2.

To further aid researchers we have also developed a Homomorphic Encryption IDE (HEIDE), which uses the wrapper class and allows researchers to easily interact with HElib. We detail the specific tools used and the ways in which the user actually uses HEIDE in Section 5.3. Validation and evaluation testing for our implementation is detailed in Chapter 6. All the code for this project can be found at https://github.com/heide-support/HEIDE.

5.1 Wrapper For HElib

In order to make HElib easier to use we developed a C++ class, BGV_HE, that is a wrapper around HElib. It not as robust as HElib, but allows an easier interface for users. There are `keyGen`, `encrypt`, and `decrypt` methods. `keyGen` takes in setup parameters and then calls a number of HElib functions that setup the public and secret keys. `encrypt` takes in a vector and then calls HElib's `encrypt` method. To ensure that the vector they are trying to encrypt does not have more entries than the number of plaintext slots, a method, `numSlots`, is provided. The resultant ciphertext is then stored in an unordered map object using a string representation of the current time as the key. The key is then passed back to the user. Should a user want to get the HElib ciphertext object from the unordered map so they can apply other HElib methods not supported by BGV_HE to it, the method `retrieve` has been provided. Another function, `replace`, has been provided so an updated ciphertext object can be put back into the unordered map. `decrypt` takes in a key and then calls HElib's `decrypt` on the ciphertext stored in the unordered map at the passed in key. The decrypted vector is then passed back to the user. If the user no longer needs an entry in the unordered map they can call the method `erase`. `erase` takes in a key and then removes the entry in the unordered map at that key.

So far we have only described the ways in which users can encrypt and decrypt data, but the real benefit of FHE is that computations can be done on the encrypted data. As such we have provided wrapper methods around a number of HElib's methods which allow one to compute on ciphertexts. These wrapper methods take in keys and then call the corresponding HElib method with the ciphertext that is stored in the unordered map at the passed in keys. A complete list of the methods we have implemented can be found in Appendix A.

We have also included wrapper methods around the timing methods that HElib has provided. The header file for BGV_HE can be found in B and provides the exact method declarations.

5.2 Python Homomorphic Encryption Modules

As mentioned before we wanted one to be able to write Python code and then have that code run the underlying HElib functions. In order to do this we used Cython [1]. Cython allows one to write Python code that can call C++ code natively. This allowed us to write the Python module PyHE which talks directly with BGV_HE. This module has the exact same methods as BGV_HE. As with HElib, encryption must happen on a plaintext object; computations and decryption happen on ciphertext objects. As such two other modules, PyPtxt and PyCtxt have been developed and represent plaintext and ciphertext objects within PyHE.

5.2.1 PyPtxt

PyPtxt is the plaintext object in PyHE. A PyPtxt object is initialized with a Python list of numbers and a PyHE object. Each element in the list is moded by the modulus provided during setup. The passed in list is broken into multiple smaller lists based on the number of plaintext slots associated with the passed in PyHE object. This means a user need not know the number of plaintext slots beforehand to create and encrypt a plaintext object. Thus one can see how changing the setup parameters for

HElib might affect runtime and memory usage. PyHE expects a PyPtxt object to be passed for encryption. When encrypting a PyPtxt object PyHE calls encrypt on each of the sublists that were created during creation of the plaintext object. PyHE then creates a PyCtxt object and stores the result of each encrypted sublist.One can also perform operations on PyPtxt objects in the same way as they would on ciphertext objects. All resultant values are moded by the modulus provided during setup.

5.2.2 PyCtxt

PyCtxt is the ciphertext object in PyHE. A PyCtxt object is initialized with a PyHE object and the length of the original list used to create the plaintext which has been encrypted into this ciphertext. The PyCtxt object stores the keys passed back from BGV_HE after calling encrypt. When decryption is performed PyHE performs decryption for each of the keys in the PyCtxt object and then returns a Python list which is the same length as the original plaintext associated with this ciphertext. As this is FHE one needs to be able to perform computations with ciphertext objects.

For computations to be performed on ciphertexts users need to be able to add, subtract, multiply, negate, and compare ciphertexts. Python allows one to override the operators within the language so that one can define what it means to, for example, "add" two objects. We have overridden the add ('a + b'), subtract ('a - b'), multiply ('a * b'), negate ('-a'), destructive add ('a += b'), destructive subtract ('a -= b'), destructive multiply ('a *= b'), equal ('a == b'), and not equal ('a != b') operators.

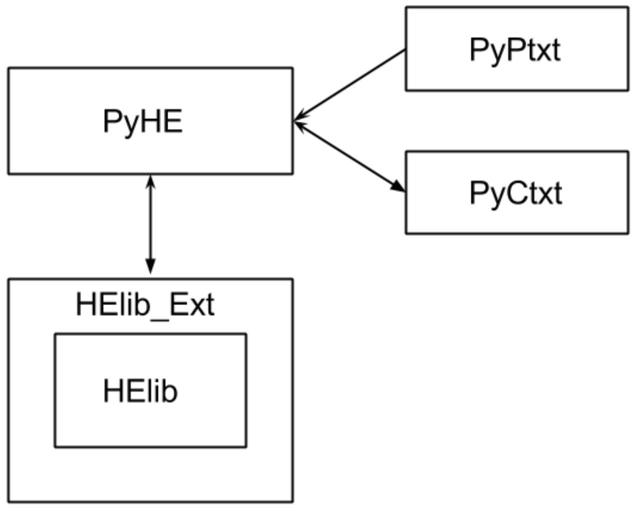


FIGURE 5.1: Code Hierarchy

The non destructive numerical operators return a new PyCtxt object, and the equality operators return boolean values. We have also provided the method `set` which allows one to write `c_new = c_old.set()` and acts as an assignment. The end result of this will be that `c_new` is assigned the value of `c_old`.

Figure 5.1 shows how each of the pieces of code described interact with each other.

5.3 HEIDE

HEIDE was written in Python 3.0 [8] and uses Python's Tkinter package [4]. Tkinter is self described as "Python's de-facto standard GUI package" and is an object oriented layer on top of Tcl/Tk [10]. Tcl (Tool Command Language) is a dynamic programming language used for a variety of applications and Tk is a graphical user interface toolkit used to develop desktop applications.

The rest of this section is outlined as follows. We discuss the settings that are read in by HEIDE upon start-up and then discuss the key windows of HEIDE.

5.3.1 HEIDE Start-up Settings

There are several configuration parameters that can be set for HEIDE. As it is a research tool to be used with HElib, the HElib implementation to be used when performing homomorphic encryption tasks must be provided. This can be the latest released version of HElib or a user modified one.

Researchers will want to test out multiple configurations of the setup parameters used for key generation to see how they affect run-time and memory usage. If each configuration was run only after the previous was complete the researcher's tests might take a long time to complete. Since modern computers allow multiple processes to be run at once, HEIDE will start a new process for each configuration. Depending on one's system the maximum number of processes that should be spawned might differ. Thus the user can define the maximum number of processes that can be started. If the maximum number is less than the number of tests being run, then the maximum number is started. Once they all finish, more are started.

When designing our system we wanted a way for users to get time and memory usage data about each run. HElib already provides timing information. Thus we added to our wrapper class ways for the user to call HElib's timing functions directly so they could obtain timing information. In order to get memory usage we used the Python library psutil v2.2.1 [7]. Psutil allows one to query the operating system and ask for memory usage of a running process. The user can choose if the memory usage data is collected and displayed. If they choose to have it collected then memory usage data is collected every `num_running_processes` * 50ms. This data is then displayed using matplotlib v1.4.3 [6]. Matplotlib is a 2D plotting library written in Python. Each configuration gets its own plot. Matplotlib natively provides users the ability to save plots.

These three initial configuration parameters are stored in the file `HEIDE/heide.config`. Data in the configuration file is stored as a JSON object. On start-up HEIDE will use Python's json decoder to get the parameter values from `HEIDE/heide.config` and then run the makefile in the source folder to make sure that it is up to date and can actually be run. If the implementation fails to build the error encountered by `make` will be displayed to the user and the IDE will not start. If everything compiles correctly then the main IDE window will open.

5.3.2 Main IDE Window

The main IDE window can be seen in Figure 5.2. This window contains the editors where the user will define the setup parameters for key generation, the data to be used in the algorithm, and the algorithm itself. This window also contains a menubar where users can (among other things) create, open, and save HEIDE projects.

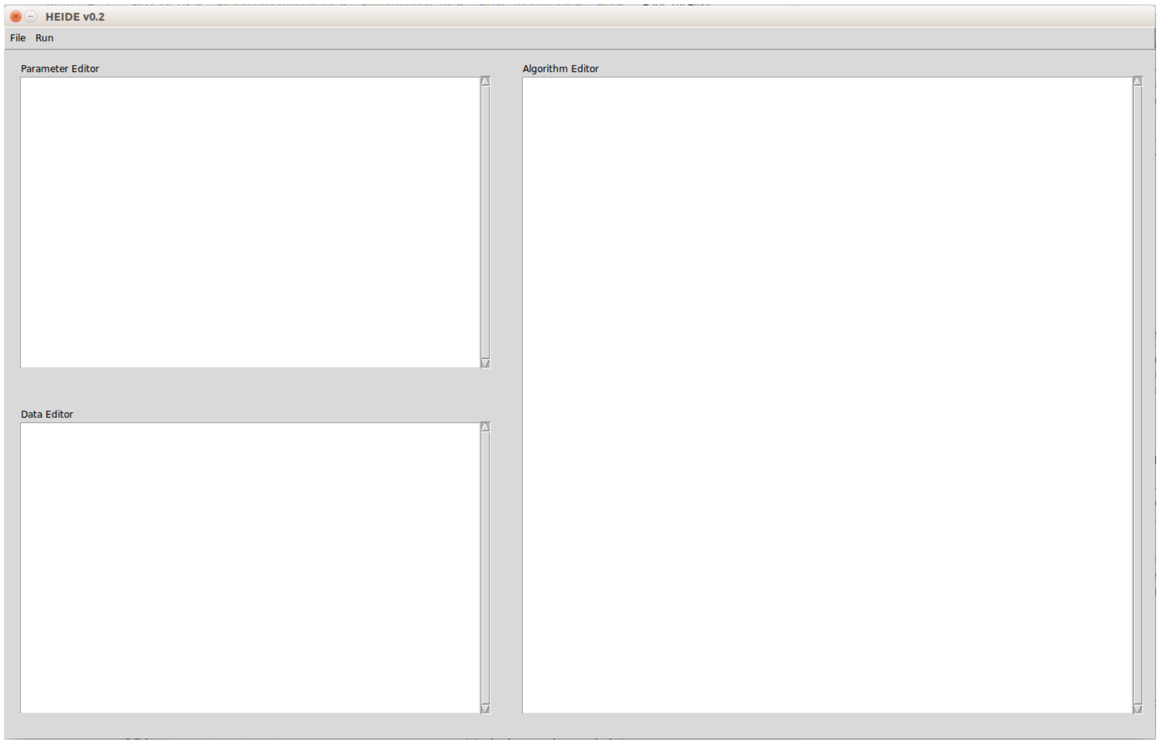


FIGURE 5.2: HEIDE main window.

Menubar Options

The menubar has two drop down menu options: File, and Run. Select options within these two menu options have key bindings and those bindings are detailed in Appendix C. Under File there are several things a user can do relating to a HEIDE project. A user can create a new HEIDE project. When this option is selected the New Project window will be displayed, see Figure 5.3. This window will allow the user to define several settings about this project. The first setting the user will be able to set is the HElib implementation to be used by this project. The second setting is the project name. Next is where the project will be saved. Default project save location is `HEIDE/HEIDE_projects`. Then there are options that allow the user to use existing

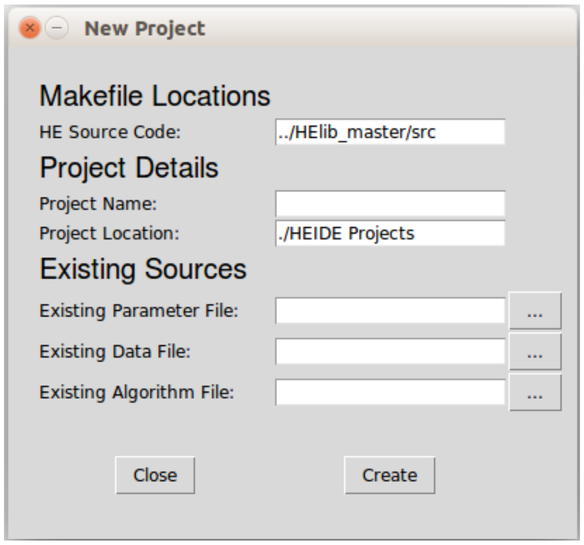


FIGURE 5.3: HEIDE new project window

files as the parameter, data, or algorithm files. If an existing file is provided then the contents of that file will be read and put into the appropriate project file. The original file will not be affected by future changes. When a project is created a file `<proj-name>.heide` is created in the project save location specified by the user. In this is stored the project settings, name, location, and links to files which contain the editor contents. The data is stored in a JSON object. Internally in the IDE, variables are set to keep track of the project's HE implementation. Other variables are also set to keep track of the open project, parameter, data, and algorithm files.

The next options in File are Open, Save, and Save As which can be seen in figures 5.4, 5.5, and 5.6 respectively. They all allow the user to work with a whole project or an individual editor. They all open a tkinter `FileDialog` window which allows the user to choose where to open, or save a file.

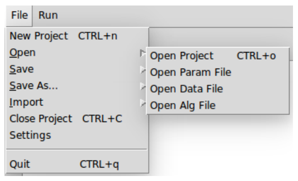


FIGURE 5.4: HEIDE open options

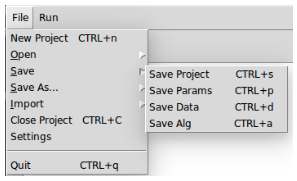


FIGURE 5.5: HEIDE save options

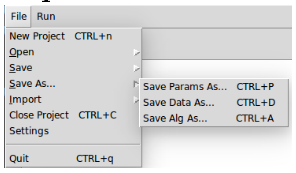
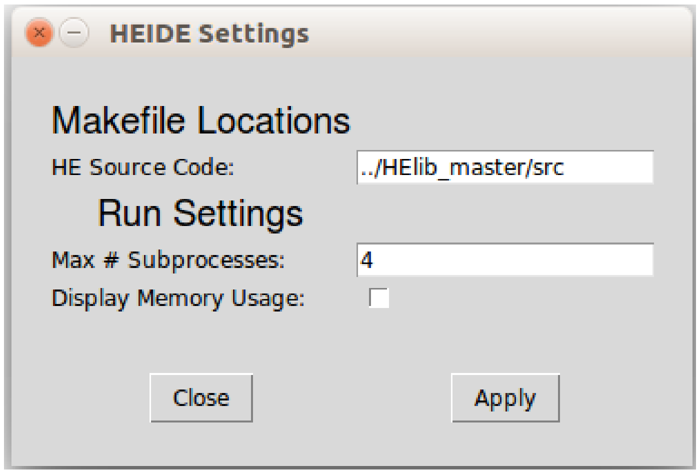


FIGURE 5.6: HEIDE save as options

Next there is an Import option. This allows the user to import an existing file into one of the editors. After that option there is the Close Project option. If selected, the contents of the editors are saved and the project is closed. If the editors don't currently link to files where the content should be saved the user will be asked where to save the contents of each editor.

The next option is Settings. This will open the Settings window, see Figure 5.7, which allows the user to set the IDE's default HE implementation, maximum number of sub-processes to run, and whether or not memory usage information should be collected and stored. When new settings are applied the IDE will run the makefile of the new HE implementation to make sure that the code can compile and is up to date. Lastly there is a Quit option which shuts down the IDE. The currently open project is saved before shutdown in the same manner as if one called Close Project.

In the menubar there is also a Run option. When this is chosen the IDE will open the Console window, see Figure 5.8. A python script is created based off the


HEIDE Settings

Makefile Locations

HE Source Code: ../HElib_master/src

Run Settings

Max # Subprocesses: 4

Display Memory Usage: ☐

FIGURE 5.7: HEIDE settings window

FIGURE 5.8: HEIDE Ouput Console

code in the algorithm editor and takes two arguments: a parameter configuration argument and a data argument. The data argument is created using the contents of the data editor and is the same for every configuration. The parameter configurations are computed based on the code in the parameter editor. The python script is then started and passed the data and the current parameter configuration.

When a subprocess finishes successfully the configuration details and `stdout` are saved internally and when all the subprocesses finish the contents are printed to the console window in order. If the user has chosen to have memory usage displayed then a plot for each of the subprocesses started is displayed. If the user tries to close the output window before all the configurations are run the remaining configurations will not be run and any running configuration will be terminated. A user can then save the output if they would like.

Parameter Editor

The parameter editor window is a Tkinter `ScrolledText` widget. The `ScrolledText` widget allows for basic text editing. By default the contents of the parameter editor window are stored in `<path_to_proj>/<proj_name>.heide_params`. The code within the parameter editor must be written in Python.

The code within the parameter editor should create a list of dictionaries. This list must have the name `RUN_PARAMS`. Each dictionary within the list corresponds to a different configuration to be used during key generation. To add a new dictionary to the `RUN_PARAMS` list one can use the standard Python commands or the ones we developed specifically for HEIDE. If the user types `%+<id>`, that code is converted into `RUN_PARAMS.append(<id>)`. For an error to not occur, `<id>` should be a dictionary consisting of only valid parameters. Optional parameters can be omitted and will be added internally before execution. When a new project is created and no existing parameter file is chosen the parameter editor is filled with a generic dictionary containing all required parameters, see Appendix D for a complete list of configuration parameters. An example, using the HEIDE specific syntax, of what could be in the parameter editor is shown in Listing E.1.

Data Editor

The data editor, like the parameter editor, is a Tkinter `ScrolledText` widget. By defualt the contents of the data editor window are stored in `<path_to_proj>/<proj_name>.heide_data`. The code within the data editor must be written in Python.

The code within the data editor should create a dictionary of lists. This dictionary must have the name `DATA`. Each list corresponds to a plaintext array. This plaintext array is what will be encrypted and computed on. There are three ways in which to add lists to the dictionary. The first is using standard Python commands. The second is by writing `$.<key> = <list>` which is converted into `DATA["<key>"] = <list>`. Using this second way, one must provide a new key every time they want to add a new list to the dictionary. If they would like to do this more pragmatically we have provided a third means for adding lists to the dictionary, `$+<key>`. The command `$+<key> = <list>` is converted into `DATA[<key>] = <list>`. These commands can be used together. Below is an example of what might be written in the data editor. Listing E.2 uses the `$.<key> = <list>` syntax only while E.3 uses the `$+<key> = <list>` syntax only.

Algorithm Editor

The contents of the algorithm editor are what will be run using the current configuration in `RUN_PARAMS` and the data in `DATA`. The code in the algorithm editor must be written in Python. When the user runs the algorithm the contents are read and any conversions that need to take place are made. There are several syntactical additions made specifically for the algorithm editor. They are listed in the Table 5.1. After all conversions are made, HEIDE creates a Python script called `alg_run_file.py` and places it in the `HEIDE/tmp` folder. An example entry in the algorithm editor is shown in Listing E.5.

HEIDE syntax	Python Equivalent syntax
`<var1> := <var2>`	# see Listing E.4 for definition of _set_ `<var1> = _set_(<var1>)`
`&<var>`	`<var> = HE.encrypt(<var>)`
`&$`	`for key in DATA:` `DATA[key] = HE.encrypt(DATA[key])`
`*<var>`	`HE.decrypt(<var>)`
`*$`	`for key in DATA:` `DATA[key] = HE.decrypt(DATA[key])`
`$.<var>`	`DATA["<var>"]`
`$@<ref>`	`DATA[DATA.keys()[<ref>]]`
`$`	`DATA`
`%.<var>`	`RUN_PARAMS["<var>"]`

TABLE 5.1: HEIDE algorithm editor syntax additions and their equivalent Python syntax statements. Note that in all cases ⟨ var ⟩ cannot include any spaces.

The script created is shown in Listing E.4. For each configuration the script is run as a subprocess using Python's built in `subprocess` module. The current `RUN_PARAMS` and `DATA` are converted to their string forms and passed as command line arguments to the script. The script then converts them back to Python objects. Next, the script creates a PyHE object and calls `PyHE.keyGen()` using the passed in `RUN_PARAMS`. From there the contents of the algorithm editor are run.

If an error occurs during the execution of a subprocess, that error is printed to the console window. If an internal error occurs, then it will be appended in `HEIDE/logs/error_log`. For each error while running the date and time the error occurred, `stdout`, and `stderr` is logged. For each error that occurs while running HEIDE, the date, time, a stack-trace, and the error message are logged. Execution of any remaining parameter configurations is abandoned and all currently running processes are terminated.

CHAPTER 6

Evaluation

Our work consists of three main contributions: BGV_HE, PyHE, and HEIDE. Testing was done in two regards. First, validation testing was done. This was done to ensure that computations on encrypted data could still be performed using the BGV_HE class and PyHE module. Those tests are detailed more in Section 6.1. Second, timing tests were done. We timed how long it took to perform key generation, encrypt data, perform a particular computation on said encrypted data, and then decrypt that data. These tests where run using and HElib code only implementation, using a BGV_HE code implementation, and using a PyHE code implementation in HEIDE. Those tests are detailed in Section 6.2. Along with the timing tests we also include the average number of lines of code required to write the test. This was done to verify that our work met its goal of providing a simpler way to perform FHE using HElib.

All our testing was done on a Mid 2014 Macbook Pro laptop. It has a 2.5 GHz Intel Core i7 processor and 16GB 1600 Mhz DDR3 RAM. We were testing using

VMware Fusion on a virtual machine, running 64-bit Ubuntu, which was not limited in processor or RAM usage.

6.1 Validation Testing

Both BGV_HE and PyHE are building off of HElib. As such, we have run tests for both to ensure that computations on encrypted data are in fact still producing the correct results. For BGV_HE that requires testing the following: `addCtxt` (with negation set to false), `addCtxt` (with negation set to true), `multiplyBy`, `multiplyBy2`, `square`, `cube`, `negate`, `equalsTo`, `rotate`, and `shift`. For PyHE that required testing addition ('a + b'), destructive addition ('a += b'), addition by a constant ('a + [int]'), subtraction ('a - b'), destructive subtraction ('a -= b'), subtraction by a constant ('a - [int]'), multiplication ('a * b'), destructive multiplication ('a *= b'), multiplication by a constant ('a * [int]'), negation ('-a'), equality ('a == b'), and non equality ('a != b'). We ran each of these tests using randomly generated plaintext values. We performed the computations first on the generated plaintexts; then encrypted the plaintexts and performed the computations; and then decrypted and compared the result to the plaintext values. We successfully passed all tests. From this we concluded that our implementation were correctly implementing the levelled BGV FHE scheme.

6.2 Timing Tests

All the timing tests we performed were done using HElib's built in timing functions. We timed how long it took to: perform key generation, encrypt a plaintext (or plaintexts) that was exactly `numSlots` in size, perform a single computation on the encrypted ciphertext (or ciphertexts), and then decrypt the result. We tested addition, subtraction, multiplication, squaring a ciphertext, cubing a ciphertext, negation, and equality. For squaring and cubing, because no specific methods are provided in PyHE, we wrote out the equivalent circuit and timed it. As we didn't provide rotation and shifting in PyHE, those tests have not been done. We ran each test five times and averaged the results. For PyHE we ran two separate timing tests. One test only allowed a single process to run at a time and the other allowed 5 to run. HElib's timing functions provide feed back about specific functions. The timing functions keep track of the overall time spent in a particular function as well as the number of times that function was entered. From those two numbers they output the averaged time spend in each function. Some tests call functions that are not called in other tests. So we have selected the functions that were common to all tests. Timing breakdowns per function can be seen in Figure 6.1 through Figure 6.10. The time it took to run all five tests can be seen in Figure 6.11.

From the results one will notice that the BGV_HE implementation and the HElib implementation took almost the exact same amount of time regardless of the

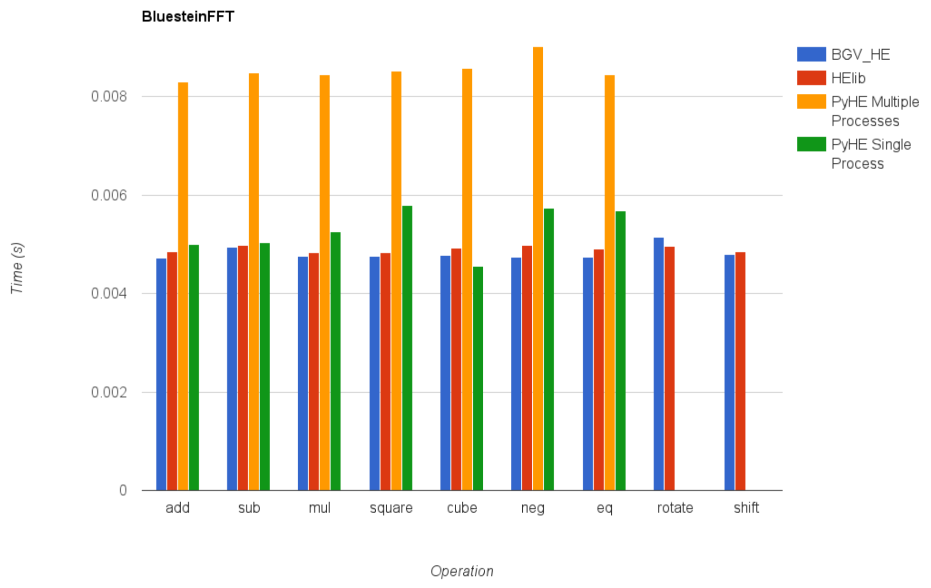


FIGURE 6.1: Average time running `BluesteinFFT`

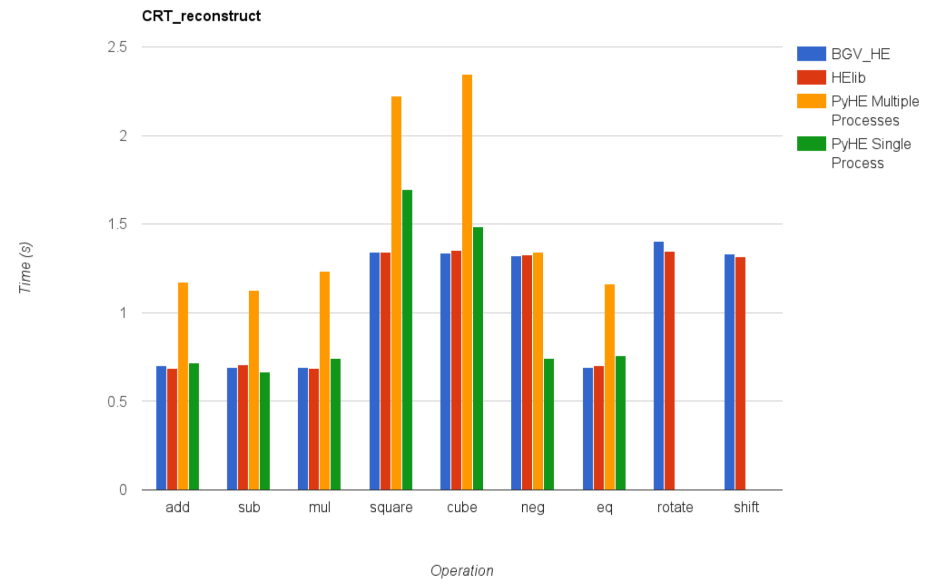


FIGURE 6.2: Average time running `CRT_reconstruct`

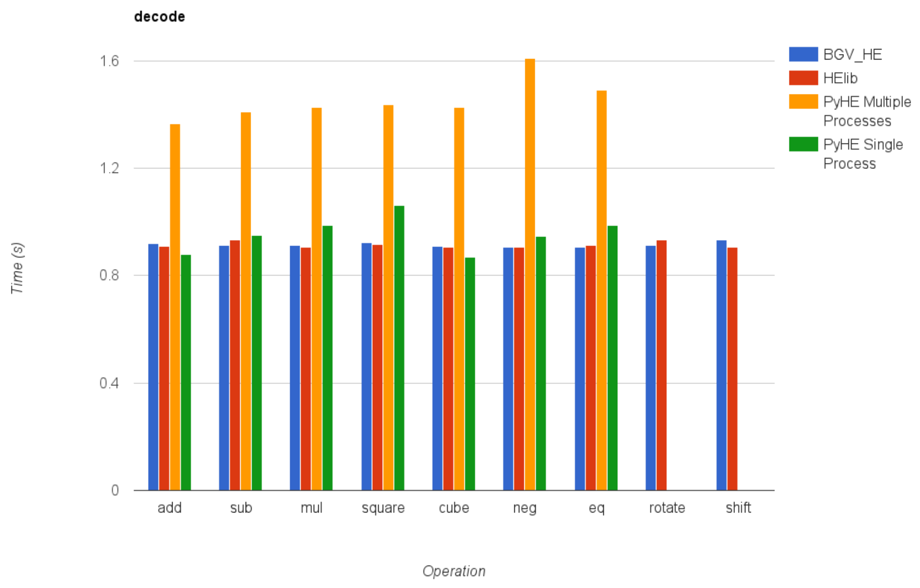


FIGURE 6.3: Average time running `decode`

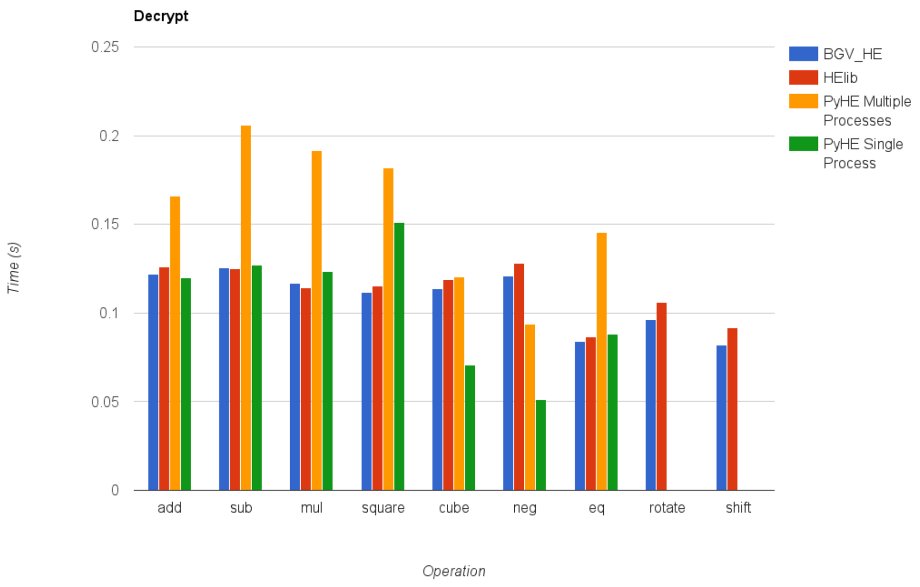


FIGURE 6.4: Average time running `Decrypt`

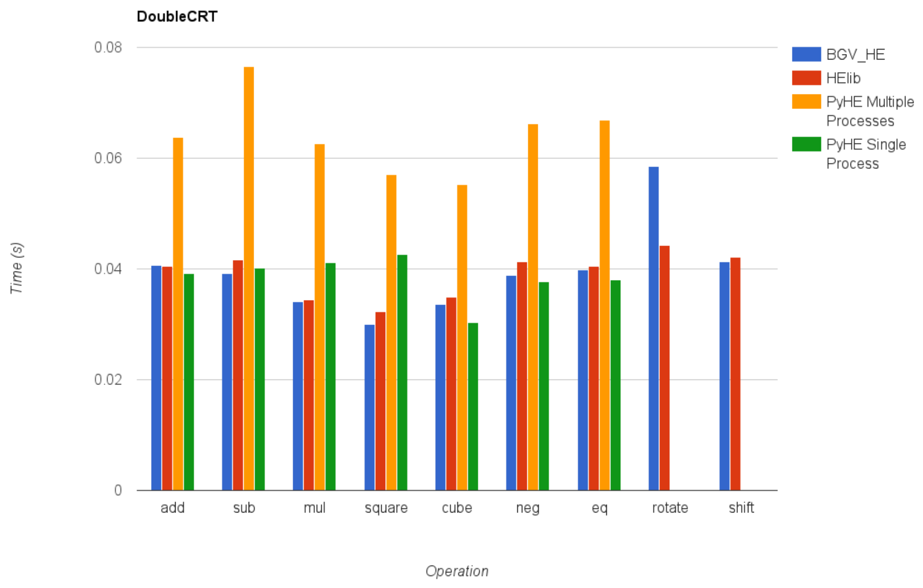


FIGURE 6.5: Average time running `DoubleCRT`

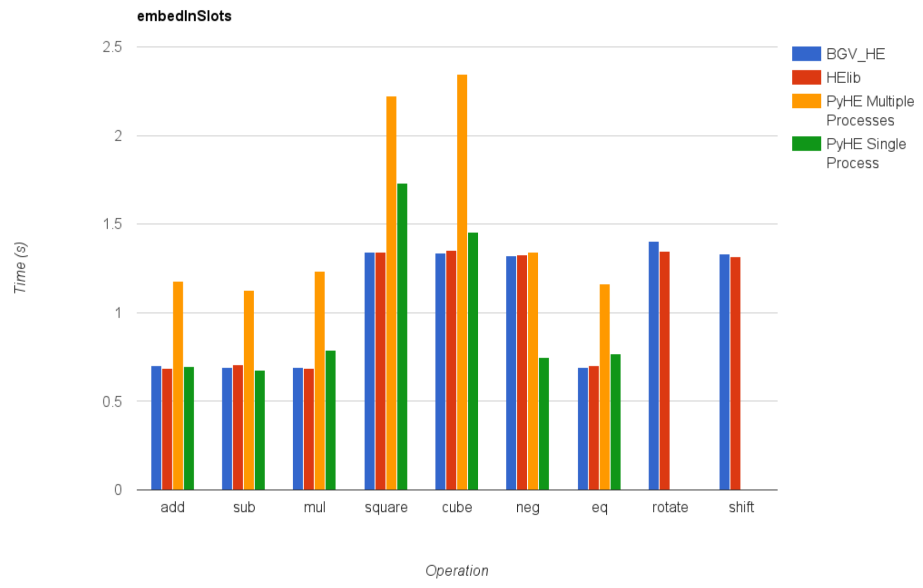


FIGURE 6.6: Average time running `embedInSlots`

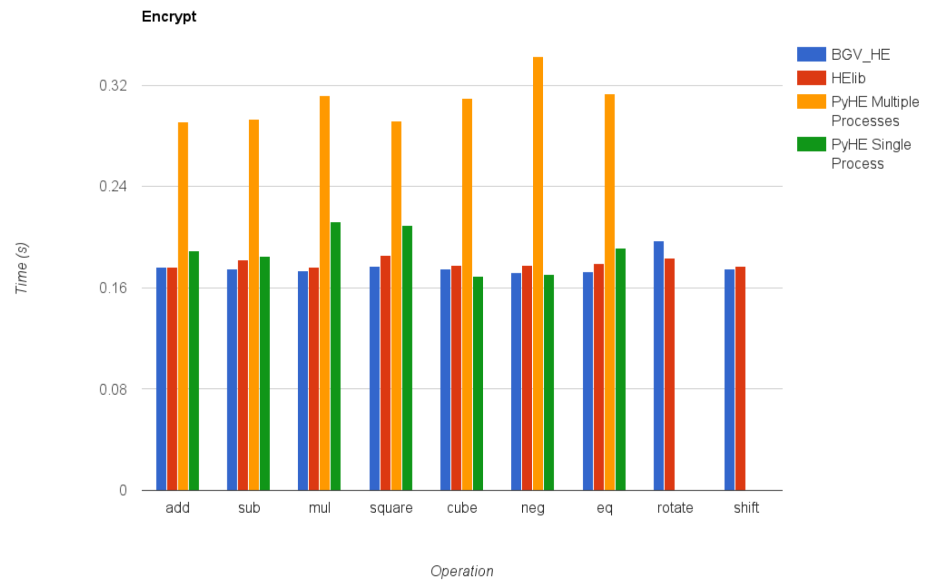


FIGURE 6.7: Average time running `Encrypt`

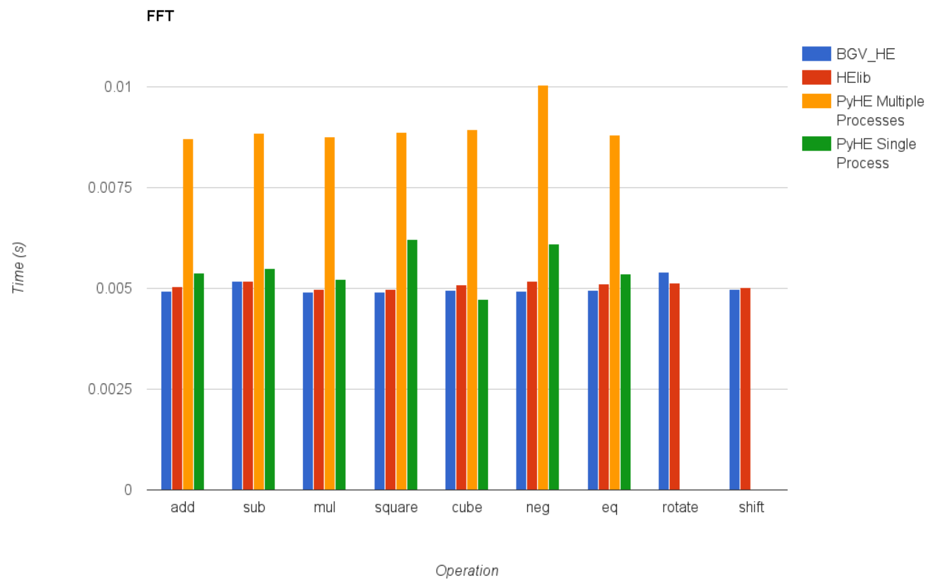


FIGURE 6.8: Average time running `FFT`

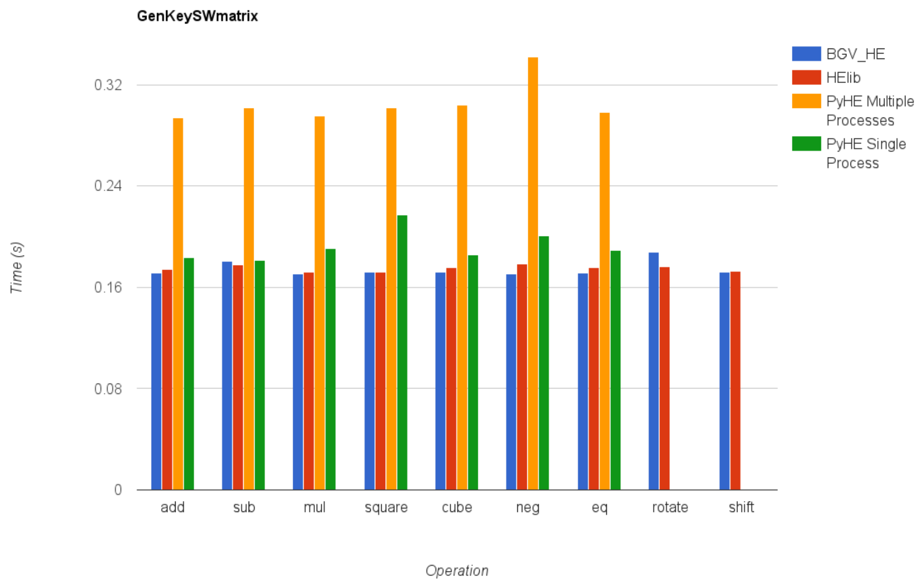


FIGURE 6.9: Average time running `GenKeySWmatrix`

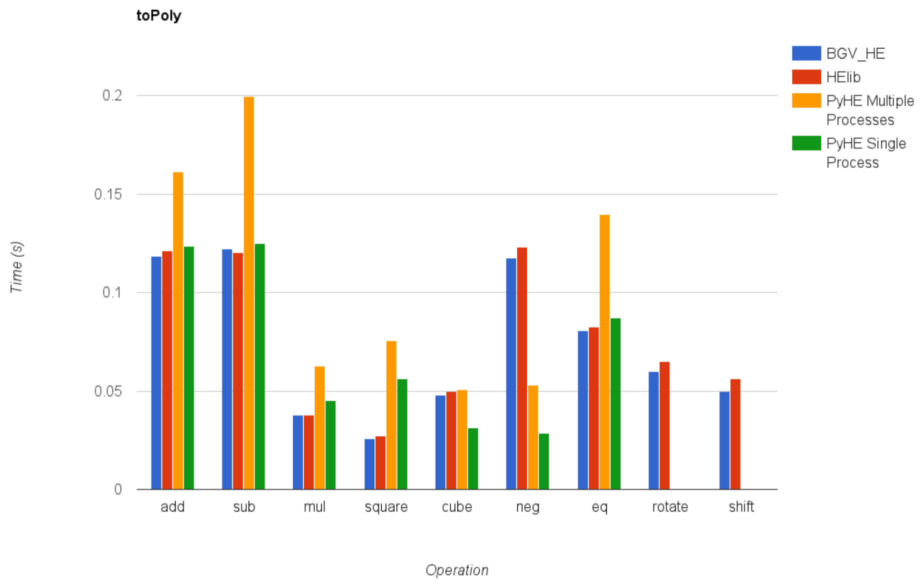


FIGURE 6.10: Average time running `toPoly`

operation. From this we can conclude our implementation is not adding any extra computational time to the operations. When running only a single process, PyHE performs slightly slower than the BGV_HE and HElib implementations. As PyHE is a Python module interacting with a C++ class this is to be expected. Since we built PyHE on top of BGV_HE, it makes sense that it would be slower. When running multiple processes however, it is clear that the individual time spent in the functions is greater. This is to be expected as the CPU can only do so much at one time. On average, with 5 processes running, we saw a slightly less then double runtime when compared to the other tests.

For each of the tests it is also important to note the number of lines of code required to write each test. For the HElib implementation the average was around 35 to 40 lines of code. For the BGV_HE implementation the average was around 25 to 30 lines of code. This decrease was due to the number steps required to perform key generation. Our PyHE tests averaged around 15 to 20 lines of code. This was due to the way in which one can create lists in Python (which is our plaintext). In HEIDE these tests required 10 to 15 lines of code. The various conversions that are done within HEIDE are what decreased the number of lines required here. For each of the test run in HEIDE we also looked at the memory required for the operation. Every operation used about 325MBs. The number of slots in the plaintext vector was around 250, so the plaintext was rather small. The memory consumption is coming from the key generation process.

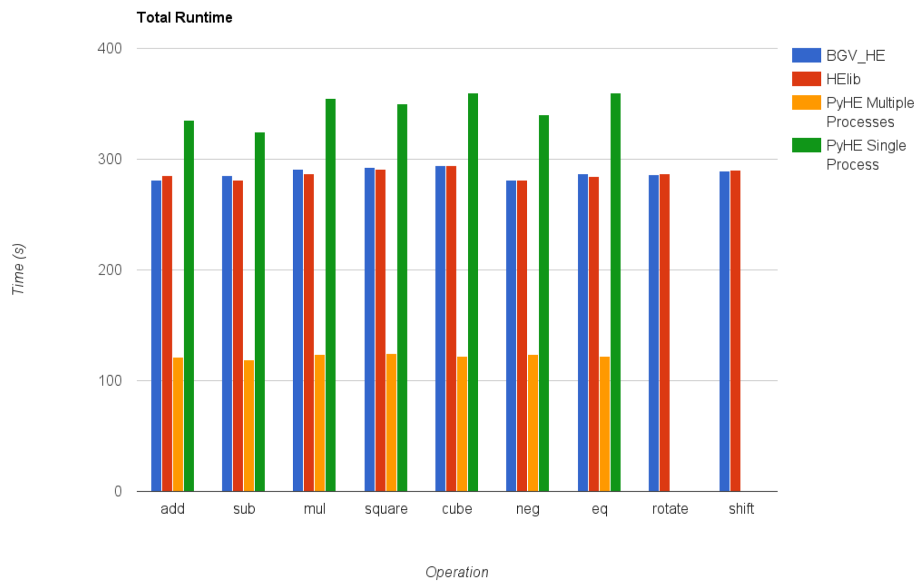


FIGURE 6.11: Total time to run each test

From Figure 6.11 one can see where using HEIDE has its benefits. The runtime required for each operation across the board remained relatively constant for all tests. The single process PyHE instance was the most erratic. It was also the slowest. This, again, is to be expected. However, the runtime for the multiple process PyHE instance was nearly half that of the BGV_HE and HElib implementations.

CHAPTER 7

Future Work

There exist several areas where future work could be done. We'll start with BGV_HE. It would be helpful to provide a ciphertext class that would be used by BGV_HE so that one could simply use, for example '+=', on two ciphtertext objects. Currently this is supported in HElib, but because we are not sending their ciphertext object back, only a string, we cannot perform such operations. This probably wouldn't be too hard to do but it was not attempted due to time constraints. Also in need of support are all the recently added recryption methods. We didn't have enough time to go through and add the recryption methods added to HElib. All this requires of one is to understand how to do recryption using HElib and work out if it can be simplified any. As a possibly crazy idea, one might also look at using a key-value database to store ciphertext objects instead of the unordered map we are currently using. This will really only be useful if the user wants to store a large amount of ciphertexts or have them persist outside of the program they are running.

There is some future work that exists with PyHE. Currently PyHE does not support rotation or shifting. This is because rotation and shifting only happen on each individual ciphertext. A PyCtxt could represent multiple ciphertexts. The easiest solution would be to say each ciphertext within the PyCtxt object is rotated or shifted as currently happens using HElib. One could explore the possibility of performing rotation and shifting as though it were happening on the original list used to create the PyPtxt object which the PyCtxt object is the encryption of.

Lastly we do think there are some useful tools that could be added to HEIDE. A debugger would be useful, but was not implemented. A way for users to link to a git repository so they can push and pull code, would be helpful. Currently HEIDE al lows one to have a single HEIDE project open at once. It might be useful to allow multiple projects to be open at once. From a code writing standpoint we would liked to have put in syntax highlighting and block commenting, but didn't have enough time to do so.

CHAPTER 8

Conclusion

Homomorphic Encryption is highly useful. It allows one to perform computations on a ciphertext without the need of the original plaintext. Since the first viable scheme was proposed by Gentry in 2009 there has been a large amount of research done in the area. The reason Gentry's scheme is not widely used currently is that Gentry's original scheme was slow. As such, the schemes that have been developed in the intermediate time sought to be quicker. The process of bootstrapping was the largest bottleneck in such schemes. However, with the introduction of a levelled fully homomorphic scheme it seemed as though we were one step closer to FHE being widely used.

Several implementations of FHE schemes have been developed. The one we have primarily focused on here is HElib. HElib is an implementation of the levelled BGV Fully Homomorphic Encryption scheme. HElib is a very robust library given its in its infancy. However, it does not have the cleanest user interface. As such, we developed a wrapper class around HElib that would abstract away some of the things required by HElib. We designed this class to reduce the number of lines necessary to write tests that perform homomorphic encryption computations. We also wanted to ensure that the running time of our wrapper class was not far off from that of a pure HElib implementation. From our results we can see that we succeeded in created a wrapper class that wasn't noticeably slower at performing operations on ciphertexts when compared to HElib.

We also sought to create a way for researchers to write homomorphic encryption code in a high level language. PyHE accomplished just that. Researchers can use PyHE to write Python code that then calls the underlying functions found in BGV_HE and HElib's. We additionally added some functionality not found in HElib. One doesn't need to call the destructive version of an operator to perform computations on ciphertexts. From our results we can see that PyHE is slightly slower than a straight HElib implementation. However, not noticeably. Where PyHE does excel is in ease of use. PyHE is much more flexible to use than HElib. One can call encrypt on a list of any length, not just one of size less than or equal to the number of plaintext slots. In other areas as well it is much easier to write tests than if one where strictly using HElib.

In support of PyHE we have also developed an IDE, the Homomorphic Encryption IDE (HEIDE). HEIDE allows users to write Python code and directly use PyHE methods. In addition HEIDE also allows one to write specific commands, that are then converted into the appropriate PyHE method. This makes writing code in HEIDE easy to do and read. HEIDE also allows one to start up multiple processes, so that users can easily play around with the setup parameters. From our tests, one can see that if multiple processes are started the runtime of a single operation is effected quite a bit. The runtime is slightly less than double that of the equivalent BGV_HE and HElib implementations. However, overall runtime is greatly decreased. From our tests one could see that it took only half the time to run the five tests when running multiple processes at once as it did when only a single at a time. In addition we have provided researchers with the ability to see the memory usage of each of their tests.

BIBLIOGRAPHY

[1] Cython: C-extensions for python. http://cython.org.

[2] Fhew github page. https://github.com/lducas/FHEW.

[3] The gnu multiple precision arithmetic library. http://www.shoup.net/ntl/.

[4] Gui programming toolkit for python. https://wiki.python.org/moin/TkInter.

[5] Helib github page. https://github.com/shaih/HElib.

[6] matplotlib homepage. http://matplotlib.org.

[7] psutil homepage. https://pypi.python.org/pypi/psutil.

[8] Python homepage. https://www.python.org.

[9] Scarab library. https://hcrypt.com/scarab-library/.

[10] Tcl developer site. http://www.tcl.tk.

[11] Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In *Advances in Cryptology–CRYPTO 2013*, pages 1–20. Springer, 2013.

[12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology–CRYPTO 2012*, pages 868–886. Springer, 2012.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[14] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[15] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[16] Léo Ducas and Daniele Micciancio. Fhe bootstrapping in less than a second. Technical report, Cryptology ePrint Archive, Report 2014/816, 2014.http://eprint.iacr.org, 2014.

[17] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[18] Craig Gentry and Shai Halevi. Implementing gentrys fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*, pages 129–148. Springer, 2011.

[19] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography–PKC 2012*, pages 1–16. Springer, 2012.

[20] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology–EUROCRYPT 2012*, pages 465–482. Springer, 2012.

[21] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[22] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. Cryptology ePrint Archive, Report 2012/099, 2012. `http://eprint.iacr.org/`.

[23] Shai Halevi and Victor Shoup. Bootstrapping for helib.

[24] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. 2013.

[25] Shai Halevi and Victor Shoup. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014*, pages 554–571. Springer, 2014.

[26] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013.

[27] Oded Regev. The learning with errors problem. *Invited survey in CCC*, 2010.

[28] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[29] Kurt Rohloff and David Bruce Cousins. A scalable implementation of fully homomorphic encryption built on ntru. In Rainer Bhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, volume 8438 of *Lecture Notes in Computer Science*, pages 221–234. Springer Berlin Heidelberg, 2014.

[30] Victor Shoup. Ntl: A library for doing number theory. https://gmplib.org.

[31] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*, pages 420–443. Springer, 2010.

[32] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[33] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.

APPENDIX A

HElib Method Definitions

Listing A.1: FindM Method Definition

```
/**
 * @brief Returns smallest parameter m satisfying various
 * constraints:
 * @param k security parameter
 * @param L number of levels
 * @param c number of columns in key switching matrices
 * @param p characteristic of plaintext space
 * @param d embedding degree (d ==0 or d==1 => no constraint)
 * @param s at least that many plaintext slots
 * @param chosen_m preselected value of m (0 => not
   ↪ preselected)
 * Fails with an error message if no suitable m is found
 * prints an informative message if verbose == true
 **/
long FindM(long k, long L, long c, long p, long d, long s,
           long chosen_m,
           bool verbose=false);
```

Listing A.2: HElib Operation Methods

```
/**
 * Operators
 * Only the "destructive" versions are supported.
 **/
```

```
// Arithmetic operators
AltCRT& operator+=(const AltCRT &other);
AltCRT& operator-=(const AltCRT &other);
AltCRT& operator*=(const AltCRT &other);

// Set operator
AltCRT& operator=(const AltCRT &other);

// Equality operators
AltCRT& operator!=(const AltCRT &other);
AltCRT& operator==(const AltCRT &other);

/**
 * Functions
 **/

// Add/subtract another ciphertxt (depending on the
// negative flag)
void addCtxt(const Ctxt& other, bool negative)

// Higher-level multiply routines
void multiplyBy(const Ctxt& other);
void multiplyBy2(const Ctxt& other1, const Ctxt& other2);
void square();
void cube();

// Negate
void negate();

// Equality function
// a procedural variant with an additional parameter
// performs a "shallow" equality check
```

```
bool equalsTo(const Ctxt& other, bool comparePkeys=true)
    const;

// Manipulation functions
void rotate(Ctxt& ctxt, long k);
void shift(Ctxt& ctxt, long k)
```

APPENDIX B

BGV_HE Method Definitions

Listing B.1: BGV_HE Method Definitions

```
#ifndef BGV_HE_H
#define BGV_HE_H

#include <fstream>
#include <sstream>
#include <cstdlib>
#include <boost/unordered_map.hpp>
#include <boost/lexical_cast.hpp>
#include <sys/time.h>

#include "../HElib/src/FHE.h"
#include "../HElib/src/EncryptedArray.h"
#include "../HElib/src/PAlgebra.h"

class BGV_HE {
public:
    BGV_HE();
    virtual ~BGV_HE();

    /**
     * @brief Performs Key Generation using HElib functions
     * @param p plaintext base
     * @param r lifting
```

```
 * @param L # of levels in modulus chain
 * @param c # of columns in key switching matrix
 * @param w Hamming weight of secret key
 * @param d degree of field extension
 * @param security security parameter
 * @param m (optional parameter) use m'th cyclotomic
    ↪ polynomial
 * @param gens (optional parameter) vector of generators
 * @param ords (optional parameter) vector of orders
 */
void keyGen(long p, long r, long L, long c,
                long w, long d, long security, long m =
                   ↪ -1,
                const vector<long>& gens = vector<long>()
                   ↪ ,
                const vector<long>& ords = vector<long>()
                   ↪ );
/**
 * @brief Calls HElib encrypt function for provided
    ↪ plaintext vector and
 * then stores the ciphertext in the unordered map and
    ↪ returns the key
 * @param ptxt_vect plaintext vector to encrypt
 * @return key where ciphertext stored in unordered map
 */
string encrypt(vector<long> ptxt_vect);
/**
 * @brief Calls HElib decrypt function for ciphertext
    ↪ that is found in
 * unordered map at key
 * @param key the key which corresponds to the ciphertext
    ↪  to decrypt
 * @return the decrypted ciphertext
 */
vector<long> decrypt(string key);

/**
 * @brief Create a new ciphertext and set it equal to the
    ↪  ciphertext
 * stored in unordered map under key
 * @param key ciphertext key in unordered map
 * @return key corresponding to new ciphertext
 */
string set(string key);
/**
 * @brief Add ciphertext at key to ciphertext at
    ↪ other_key and store result
 * back in unordered map at key
 * @param key key in unordered map
 * @param other_key key in unordered map
 * @param negative if True then perform subtraction
 */
void addCtxt(string key, string other_key, bool negative)
   ↪ ;
/**
 * @breif Multiply ciphertext at key by ciphertext at
    ↪ other_key and store
 * result in unordered map at key
 * @param key key in unordered map
 * @param other_key key in unordered map
 */
void multiplyBy(string key, string other_key);
/**
```

```
 * @brief Multiply ciphertext at key by ciphertext at
   ↪ other_key1 and
 * other_key2
 * @param key key in unordered map
 * @param other_key1 key in unordered map
 * @param other_key2 key in unordered map
 */
void multiplyBy2(string key, string other_key1, string
   ↪ other_key2);
/**
 * @brief Square ciphertext at key
 * @param key key in unordered map
 */
void square(string key);
/**
 * @brief Cube ciphertext at key
 * @param key key in unordered map
 */
void cube(string key);
/**
 * @brief Multiply ciphertext at key by -1
 * @param key
 */
void negate(string key);
/**
 * @brief Return true if the ciphertext at key and
   ↪ ciphertext at other_key
 * are equal
 * @param key key in unordered map
 * @param other_key key in unordered map
 * @param comparePkeys if true then pkeys will be
   ↪ compared
 * @return True if ciphertexts are equal
 */
bool equalsTo(string key, string other_key, bool
   ↪ comparePkeys);
/**
 * @brief Rotate ciphertext at key by k spaces
 * @param key key in unordered map
 * @param k number of spaces to rotate by
 */
void rotate(string key, long k);
/**
 * @brief Shift ciphertext at key by k spaces
 * @param key key in unordered map
 * @param k number of spaces to shift by
 */
void shift(string key, long k);

/**
 * @brief Number of plaintext slots
 * @return number of plaintext slots
 */
long numSlots();
/**
 * Replace the ciphertext at key with the new one
    ↪ provided
 * @param key key in unordered map
 * @param new_ctxt new Ctxt object to store in the
    ↪ unordered map
 */
void replace(string key, Ctxt new_ctxt);
/**
```

```
     * @brief Retrieve the ciphertext object from the
        ↪ unordered map
     * @param key key in unordered map
     * @return the ciphertext corresponding to the passed in
        ↪ key
     */
    Ctxt retrieve(string key);
    /**
     * @brief Delete from the unordered map the entry at key
     * @param key key in unordered map
     */
    void erase(string key);

    /**
     * @brief Call HElib timers on method
     */
    void timersOn();
    /**
     * @brief Call HElib timers off method
     */
    void timersOff();
    /**
     * @brief Call HElib timers reset method
     */
    void resetTimers();
    /**
     * @brief Call HElib timers print method
     */
    void printTimers();
private:
    EncryptedArray* ea;
    FHEcontext* context;
    FHESecKey* secretKey;
    const FHEPubKey* publicKey;

    /**
     * Unordered map which stores the ciphertexts
     */
    boost::unordered_map<string, Ctxt> ctxt_unord_map;

    /**
     * @brief Store the ciphertext in the unordered map and
        ↪ return key where
     * it was stored
     * @param ctxt Ciphertext to store in unordered map
     * @return the key used to locate this ciphertext in the
        ↪ unordered map
     */
    string store(Ctxt* ctxt);
};

#endif  /* BGV_HE_H */
```

APPENDIX C

HEIDE Key Bindings

Key Binding	HEIDE Command
Ctrl+n	Create a new project
Ctrl+o	Open project
Ctrl+s	Save project
Ctrl+p	Save parameter editor contents
Ctrl+d	Save data editor contents
Ctrl+a	Save algorithm editor contents
Ctrl+P	Save As parameter editor contents
Ctrl+D	Save As data editor contents
Ctrl+A	Save As algorithm editor contents
Ctrl+C	Close project
Ctrl+q	Quit HEIDE
F1	Run Project

TABLE C.1: HEIDE key bindings

APPENDIX D

Key Generation Parameters

Required Parameters	
p	plaintext base
r	lifting
L	# of levels in modulus chain
c	# of columns in key switching matrix
w	Hamming weight of secret key
d	degree of field extension
security	security parameter

Optional Parameters	
m	use m'th cyclotomic polynomial
gens	vector of generators
ords	vector of orders

TABLE D.1: Key Generation Parameters (more detailed descriptions can be found in Section 2.3.4)

APPENDIX E

Example HEIDE Code

Listing E.1: Example parameter editor entry using HEIDE syntax

```
# parameter dictionary
pd = {   'p' : 65537,
         'r' : 1,
         'L' : 15,
         'w' : 64,
         'd' : 0,
         'security' : 128}

# add current parameter dictionary to RUN_PARAMS
%+pd
```

Listing E.2: Example data editor entry using `$.<key> = <list>` syntax

```
# create two plaintext arrays and add them to DATA
$.p0 = [1, 2, 3, 4]
$.p1 = [5, 6, 7, 8]
```

Listing E.3: Example data editor entry using `$+<key> = <list>` syntax

```
from random import randrange

# create 5 lists of random numbers and add them to DATA
for i in range(5):
        $+i = [randrange(j + 1) for j in range(100)]
```

Listing E.4: Example `run_alg_file.py` file

```
import sys
import ast
sys.path.append('../../PyHE/')
from PyHE import PyHE
from PyPtxt import PyPtxt
from PyCtxt import PyCtxt
_set_ = lambda c: c.set()
def run_heide_alg(RUN_PARAMS, DATA):
    HE = PyHE()
    HE.keyGen(RUN_PARAMS)

    for key in DATA:
        DATA[key] = PyPtxt(DATA[key], HE)

    # algorithm editor content which contains the algorithm
    # to be run
    <alg edit content>

if __name__ == '__main__':
    run_heide_alg(ast.literal_eval(sys.argv[1]),
            ast.literal_eval(sys.argv[2]))
```

Listing E.5: Example Algorithm Editor Entry

```
# encrypt all elements in DATA
&$


# create a ciphertext ctxt_sum and set it equal to
# the first thing in data. Then add up the rest of the
# elements in DATA
ctxt_sum := $@0
for i in range(1, len($)):
        ctxt_sum += $@i

```

```
# decrypt ctxt_sum and print out the plaintext result
print(*ctxt_sum)
```