5-2020

On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries

Alycia Carey

On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries

An Undergraduate Honors College Thesis
in the
Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR

by

Alycia N. Carey
ancarey@uark.edu

May 9, 2020
University of Arkansas

This thesis is approved.

Thesis Advisor:

Dale R. Thompson

Dale R. Thompson, Ph.D

Thesis Committee:

Matthew Patitz, Ph.D

Brajendra Panda Digitally signed by Brajendra Panda
Date: 2020.04.23 17:14:01 -05'00'

Brajendra Panda, Ph.D

Justin Zhan, Ph.D

Abstract

While fully homomorphic encryption (FHE) is a fairly new realm of cryptography, it has shown to be a promising mode of information protection as it allows arbitrary computations on encrypted data. The development of a practical FHE scheme would enable the development of secure cloud computation over sensitive data, which is a much-needed technology in today's trend of outsourced computation and storage. The first FHE scheme was proposed by Craig Gentry in 2009, and although it was not a practical implementation, his scheme laid the groundwork for many schemes that exist today. One main focus in FHE research is the creation of a library that allows users without much knowledge of the complexities of FHE to use the technology securely. In this paper, we will present the concepts behind FHE, together with the introduction of three open-source FHE libraries, in order to bring better understanding to how the libraries function.

Keywords: *Fully Homomorphic Encryption, Secure Cloud Computing, Secure Computation*

Contents

1 Introduction

In the not so distant future, the majority of storage and computation of data will take place in the cloud. In 2018, 81% of companies with at least 1,000 employees already utilized cloud computing technology in their everyday processes, and this percentage is expected to breach 90% by 2024 [48]. Cloud computing is a promising technology as it offers the benefits of flexibility, improved disaster recovery, and increased collaboration from which organizations can benefit. In spite of these benefits, major security issues can arise when critical data is stored in the cloud. Confidentiality of information in the cloud is not guaranteed, which is an immense hindrance to the adoption of cloud computing technology. But, if suitable encryption is applied to data before storage this problem can be mitigated.

Unfortunately, a new issue arises with this solution. Every time that a computation needs to be performed on the encrypted data in the cloud, the data would first have to be downloaded and decrypted on the client side. Then, after the data is processed, it would have to be re-encrypted and re-uploaded to the cloud. This tedious and time consuming process almost out-weighs the benefits of using cloud storage in the first place.

Homomorphic encryption (HE) is a relatively new realm of cryptography which gives the ability to privately and securely store and compute on data in the cloud without the necessity to decrypt it first. In addition, most existing homomorphic encryption schemes are based on lattice cryptography, specifically the Learning with Errors problem, making them secure against modern and post-quantum cryptography attacks. The development of an efficient homomorphic encryption scheme would not just provide a benefit to one specific sector, but rather it would have a wide breadth of impact in various domains ranging from national security to genomics. In [3], they outline several potential applications for HE, but here we will recount a simple example offered in 1978 by Rivest, Adleman, and Dertouzos [49].

Consider a small loan lending company. Instead of storing their data in-house, which requires expensive equipment, they opt to store their data through a cloud service provider. Loan data contains highly sensitive personal information, and cannot be stored in the clear as anyone could potentially gain access to the cloud platform and view the data.

Now consider that the loan company hires a third-party to run their proprietary software to analyze the loan data to gain insight on how to improve their business, but do not trust the third-party to not sell their data. In addition, since the software is proprietary to the third-party company, they will not simply send it to the loan company to use. One

solution to this situation is to use homomorphic encryption, or *privacy homomorphism* as [49] terms it. Using homomorphic encryption, the loan company could encrypt their data before storing it in the cloud, and allow the third-party to access the cloud in order to download the data for computation. The analysis company would run their software over the encrypted data and then re-upload the encrypted output to the cloud. This would require the analysis company to modify their software to allow it to process computations on encrypted data, but neither party ever has to reveal their secrets in this scheme.

Homomorphic encryption is not without its disadvantages though. Since it is based on lattice cryptography, HE ciphertexts are "noisy" as error is introduced in the encryption process to hide the keys. Even worse, while the noise grows only slightly during homomorphic addition, it grows exponentially during homomorphic multiplication. This noise growth puts a limitation on the amount of computations that can be carried out in an encrypted manner while still being able to decrypt correctly. It also drastically increases run-time and storage requirements which makes it questionable if homomorphic encryption could ever be used effectively as a cryptography scheme.

In his 2009 thesis, *A Fully Homomorphic Encryption Scheme* [24], Craig Gentry not only proved that homomorphic encryption was indeed computationally possible, but constructed the first viable fully homomorphic encryption scheme (FHE) that allowed arbitrary computations on encrypted data. Since his seminal work, research of FHE has grown exponentially, leading to the creation of several FHE libraries that allow even a novice in cryptography to implement the new, powerful technology.

In this thesis we will explore the mechanisms behind fully homomorphic encryption, three important fully homomorphic schemes (BGV, BFV, CKKS), and three recent fully homomorphic encryption libraries (HElib, SEAL, PALISADE). The rest of this thesis is organized in the following manner. Chapter 2 will give the history of FHE and will introduce the foundational work that has been completed in the field. Chapter 3 will explain the preliminary knowledge required for the understanding of the rest of the thesis. In Chapter 4, we will explore three different fully homomorphic encryption schemes as well as an operation called SIMD that can be used to make FHE schemes more efficient. Chapters 5, 6, and 7 will discuss the libraries HElib, Microsoft SEAL, and PALISADE, respectively. The implementation of a simple example in each library will be carried out in Chapter 8. Finally, the concluding remarks will be given in Chapter 9.

2 Related Work

The idea of homomorphic encryption dates back to 1978, one year after the release of RSA. In 1978, Rivest, Adleman, and Dertouzos published [49] which reasoned that homomorphic encryption is a theoretic possibility. For more than 30 years it was unclear whether an efficient solution to homomorphic encryption existed and if efforts should be exerted in trying to find one.

There was no clear vision of how to approach the construction of a feasible homomorphic encryption scheme until 2005, when Regev published the Learning with Errors problem [44]. The introduction of LWE revolutionized the cryptography world, and shed enough light on the HE problem for Craig Gentry to see a path to a viable solution and construct the first plausible fully homomorphic encryption scheme in 2009.

Since the release of Gentry's publication, the development of FHE can be grouped into three different generations corresponding to the approach taken in constructing the FHE scheme.

First-generation FHE: The first generation of FHE development includes Gentry's original scheme which used ideal lattices. Extensive design and implementation work in the years following its release improved upon Gentry's original implementation by many orders of magnitude in run-time performance. A year after the release of [24], Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan published the second FHE scheme that was based on [24], but took a simpler approach by replacing the ideal lattice computations with integer arithmetic [20]. The schemes developed in the first-generation served as a foundation for the second generation's research, but they are not used in production today as they all suffer from very rapid noise growth, which severely limits the amount of homomorphic computations that can be performed.

Second-generation FHE: Many of the homomorphic encryption schemes currently in use today were products of the second generation of FHE development. The second generation started in 2011 with the release of the BGV scheme by Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan [9]. It was followed closely by the NTRU-based scheme LTV by Lopez, Alt, Tromer, and Vaikuntanathan in 2012 [40]. That same year also saw the release of a scale-invariant FHE scheme called BFV by Brakerski, Fan, and Vercauteren [22]. There was then a 4-year gap

until Cheon, Kim, Kim, and Song released CKKS, originally named HEAAN, which allowed homomorphic computations on real numbers [14].

The majority of the schemes in this generation are based on the hard problem of Ring-LWE. In addition, all of the schemes follow Gentry's original blueprint in the sense that they first construct a somewhat homomorphic encryption (SHE) scheme and then convert the SHE scheme to a FHE scheme through the addition of some sort of noise reduction technique such as bootstrapping or modulus switching.

Third-generation FHE: The third generation of FHE development was highlighted by the creation of schemes that avoid the expensive relinearization step required in many second generation schemes to perform multiplication. The main products of this era were the GSW scheme in 2013 by Craig Gentry, Amit Sahai, and Brent Water [29], the FHEW scheme by Ducas and Micciancio in 2014 [21], and the TFHE scheme by Chillotti et al. in 2016 [17]. The schemes of this era greatly improved the efficiency of FHE, and most can bootstrap in less than 0.1 seconds, a task that used to take upwards of hours.

With the creation of the above FHE schemes, several FHE libraries have been published in order to allow even the most novice user to use the new encryption technology. Table 2.1 lists 11 popular FHE libraries as well as their authors, what schemes they support, and what language they are written in.

Table 2.1: Recent Fully Homomorphic Encryption Libraries

Library	Author	Schemes	Language
HElib	Halevi, Shoup	BGV	C++
Microsoft SEAL	Microsoft	BFV, CKKS	C++
PALISADE	NJIT	Lattice crypto library	C++
HEAAN	Cheon, Kim, Kim, Song	CKKS	C++
FHEW	Ducas, Micciancio	FHEW	C++
TFHE	Chillotti et al.	TFHE	C++
FV-NFLlib	CryptoExperts	BFV	C++
NuFHE	NuCypher	GPU based TFHE	Python
Lattigo	EPFL-LDS	BFV, CKKS	Go
$\Lambda \bigcirc \lambda$	Crockett, Peikert	Lattice crypto library	Haskell
cuHE	Dai, Wei, Sunar	DHS	CUDA

In addition, two fully homomorphic frameworks have been published which combine

several of the FHE libraries into one platform. Table 2.2 lists two popular frameworks, their authors, and the libraries they support.

Table 2.2: Recent Fully Homomorphic Encryption Frameworks

Framework	Author	Libraries
E3	MoMA Lab	TFHE, FHEW, HElib, SEAL
SHEEP	Alan Turing Institute	HElib, SEAL, PALISADE, TFHE

With the rapid development of FHE schemes, libraries, and frameworks, it is important that the cryptography community has a standard for how to safely set the security parameters. In order for homomorphic encryption to be adopted in medical, health, and financial sectors, it will have to be standardized and go through an intensive review process by multiple standardization bodies and government agencies. An important part of the standardization process is the agreement on security levels for varying parameter sets. HomomorphicEncryption.org is an open consortium of people from industry, government and academia who have undertaken the task of this standardization [38]. Specifically, they set to uniformize and simplify the FHE library APIs, and provide education to application developers on how to safely implement FHE.

3 Preliminaries

Before delving into the explanation of the fully homomorphic encryption libraries, it will be helpful to have an understanding of the underlying mechanisms of fully homomorphic encryption itself. In this Chapter, we will explore the basics of homomorphic encryption, somewhat homomorphic encryption, bootstrapping, modulus switching, and finally, fully homomorphic encryption.

3.1 HE: Homomorphic Encryption

When data is encrypted under a standard modern day encryption scheme, there are only two operations that can be performed on the data: storage and retrieval. If any computation or analysis needs to be performed on the encrypted data, it must be decrypted first. This requirement comes at a cost of time, resources, and privacy. Homomorphic encryption schemes provide an alternative to regular cryptographic schemes by offering mechanisms to perform operations upon encrypted data without having to decrypt it first.

Homomorphic encryption operates on a circuit level, meaning that the functions used in homomorphic encryption must consist only of binary operations like AND and XOR. For example, addition can be represented as the multiplication of two bits:

$$\text{AND}(b_1, b_2) = b_1 \cdot b_2$$

and XOR can be seen as the addition of two bits modulo 2:

$$\text{XOR}(b_1, b_2) = (b_1 + b_2) \bmod 2$$

This is not an inherent problem as every function that a computer performs can be reduced to some series of bit-wise functions, but as will be shown later, the depth of the circuit, or how many operations are carried out, can greatly affect the efficiency and correctness of the HE scheme. For simplicity, the term function and circuit will be used interchangeably throughout this thesis.

When thinking of a regular public key encryption scheme, it can be helpful to relate it to slamming a door that automatically locks when it is closed. Anyone can close the door, but only the person with the correct key can unlock it and go through [27]. Homomorphic encryption on the other hand can be thought of as a locked glove-box. Gentry provides an attractive example in [23] that we will borrow to explain the concepts throughout this

chapter.

Imagine that Alice owns a jewelry store and has a collection of expensive raw materials that can be transformed into a final product and be sold for a high dollar. Alice does not trust her workers to not walk off with the precious raw materials when she is not looking, so she wishes to find a way to grant her workers the permission to process the materials without having direct access to them. She comes up with an idea to put the materials inside of a transparent glove-box, locking it with her master key, and then giving the box to the workers. The workers can insert their hands into the gloves and manipulate the materials into a finished piece of jewelry without ever having direct access to them. Once assembled, the workers give the box back to Alice who can unlock it and retrieve the final product.

In this example, the placing of the raw materials inside the glove-box and locking it with a key, k, represents the encryption, $\mathbf{Enc}_k$, of some set of plaintext $\{m_1, m_2, \ldots, m_n\}$. The workers' manipulation of the materials with the gloves represents the homomorphism or malleability of the encryption scheme as it allows the raw data to be processed while locked inside the box. The final product inside the box represents the final ciphertext, $c = \mathbf{Enc}_k f(m_1, m_2, \ldots, m_n)$, which is the encryption of the function over the original plaintexts.

In the above example, the property of homomorphism was showed through the manipulation of the materials by the workers, but in a real homomorphic encryption scheme, the encryption and decryption functions can be thought of as homomorphisms between plaintext and ciphertext spaces. Put simply, a homomorphism is the transformation of one set into another that preserves the relationship of the elements in the first set.

Definition 3.1.1 (Homomorphism [11])**.** If $\mathbb{G}$ and $\mathbb{H}$ are groups, a *homomorphism* from $\mathbb{G}$ to $\mathbb{H}$ is a function $f : \mathbb{G} \to \mathbb{H}$ such that $f : (g_1 \odot g_2) = f(g_1) \otimes f(g_2)$ for any elements $g_1, g_2 \in \mathbb{G}$, where $\odot$ denotes the operation in $\mathbb{G}$ and $\otimes$ denotes the operation in $\mathbb{H}$.

While homomorphic encryption is a relatively new concept, homomorphism is not. In fact, the notion of homomorphic encryption, originally called a *privacy homomorphism*, was introduced by Rivest, Adleman, and Dertouzous [49] shortly after the publication of RSA in 1978. RSA is an asymmetric encryption scheme meaning that each user must have a public and private key in order to carry out encryption and decryption.

Let $\{e, n\}$ be some public key. The encryption of message m is given by:

$$E(m) = m^e \bmod n$$

Consider two messages m_1 and m_2. RSA has a multiplicative homomorphic property

as the product of two ciphertexts is equal to the encryption of the product of the two messages.

$$E(m_1) \cdot E(m_2) = m_1^e m_2^e \bmod n = (m_1 m_2)^e \bmod n = E(m_1 \cdot m_2)$$

Since RSA only has multiplicative homomorphism and not addition, it is termed as a partial homomorphic encryption scheme (PHE).

Besides the idea of homomorphism, homomorphic encryption schemes differ from other public-key schemes as they have an additional function called **Eval** . In terms of writing a program, the **Eval** function is where all the encrypted computation actually takes place.

Definition 3.1.2 (Public-key Encryption)**.** A traditional public-key encryption scheme is a 3-tuple of probabilistic, polynomial-time algorithms **KeyGen**, **Enc**, and **Dec**:

> **KeyGen:** Takes as input a security parameter λ and outputs a pair of keys (pk, sk) where pk is the public key and sk is the private key. The plaintext space $\mathcal{P}$ is defined by pk the ciphertext space $\mathcal{C}$ is defined by sk.
>
> **Enc:** Takes as input the public key pk and a message $m \in \mathcal{P}$. It outputs a ciphertext $c \in \mathcal{C}$. This process is denoted as $c \leftarrow \mathbf{Enc}_{pk}(m)$.
>
> **Dec:** Takes as input the private key sk and a ciphertext $c \in \mathcal{C}$. It outputs a message $m \in \mathcal{P}$ if the right combination of keys is used, or $\perp$ if decryption is not successful. This process is denoted as $m := \mathbf{Dec}_{sk}(c)$.

Definition 3.1.3 (Homomorphic Encryption)**.** A homomorphic encryption scheme is a 4-tuple of algorithms **KeyGen**, **Enc**, **Dec**, **Eval**:

> **KeyGen:** Takes as input a security parameter λ and outputs a pair of keys (pk, sk) where pk is the public key and sk is the private key.
>
> **Enc:** Takes as input the public key pk and a message $m \in \{0, 1\}$. It outputs a ciphertext $c \in \mathcal{C}$. This process is denoted as $c \leftarrow \mathbf{Enc}_{pk}(m)$.
>
> **Dec:** Takes as input the private key sk and a ciphertext $c \in \mathcal{C}$. It outputs a message $m \in \mathcal{P}$ if the right combination of keys is used, or $\perp$ if decryption is not successful. This process is denoted as $m := \mathbf{Dec}_{sk}(c)$.
>
> **Eval:** Takes as input the public key pk, n ciphertexts $c_1, c_2, \ldots, c_n \in \mathcal{C}$ and a certain permitted function F. It outputs $F(c_1, c_2, \ldots, c_n) \in \mathcal{C}$. Consider the set of

ciphertexts $c_i = \{c_1, c_2, \ldots, c_n\}$ and their corresponding decrypted messages $m_i = \{m_1, m_2, \ldots, m_n\}$. The evaluation is correct if the following holds:

$$\mathbf{Dec}(\mathbf{Eval}(F, c_i, pk), sk) = F(m_i)$$

or the evaluation of the ciphertexts that are encrypted with pk through the function F decrypts under sk to the computation of the plaintexts themselves through F.

It is important to note that homomorphic encryption schemes do not always have to be asymmetric. In fact, homomorphic encryption in the asymmetric and symmetric setting are essentially the same. Transforming a asymmetric homomorphic encryption scheme to a symmetric one is relatively simple. The main obstacle is that in the public-key setting, the homomorphic evaluation algorithm is also given the encryption key. To overcome this issue, the encryption key can simply be appended to the end of each ciphertext [51]. It is more difficult to go from a symmetric scheme to an asymmetric one.

Theorem 3.1.1. (Symmetric to Asymmetric HE [30]) If $\mathcal{E}$ is a symmetric homomorphic encryption scheme it can be transformed to an asymmetric one by the following process. First, a public key must be chosen and published. This public key can be a collection of ℓ random bits, $\{b_1, \ldots, b_\ell\}$, and the encryption of those bits ℓ, $\{c_1, \ldots, c_\ell\}$.

$$pk = \{(b_1, c_1), \ldots, (b_\ell, c_\ell)\}, c_i = \mathbf{Enc}_{sk}(b_i)$$

The amount of bits ℓ chosen depends on the size of the ciphertext, $\ell \gg |c_i|$. To encrypt a bit σ, first choose another random bit string $\overrightarrow{r}$ such that the inner product between r_i and b_i is equal to σ: $\sum r_i b_i = \sigma$. Then, compute the homomorphic inner product: $c = \sum_{r_i=1} c_i = \mathbf{Enc}(\sum r_i b_i)$. Note that the homomorphic inner product is just the modulo 2 inner product of r_i and b_i.

For the purpose of this thesis, we will be focusing on the asymmetric variant of homomorphic encryption schemes.

3.2 SHE: Somewhat Homomorphic Encryption

Before 2009, all homomorphic encryption schemes were actually only *somewhat* homomorphic [26]. SHE is an extension of PHE which allows an arbitrary number of either additions or multiplications, but only a bounded number of the other. This means that SHE can only handle a limited class of permitted functions or circuits. This can be visualized through the example of Alice's jewelry shop where the glove-boxes are defective

and only allow the workers to manipulate the gloves for a certain amount of time, or a certain number of steps, before they become stiff and inoperable.

One simple example of a SHE scheme built using only modular arithmetic can be seen in [20]. For simplicity, it will be explained as a symmetric encryption scheme, but as noted before, any symmetric encryption scheme can be transformed into an asymmetric one. In [20], they use a security parameter denoted λ, and variables $N = \lambda$, $P = \lambda^2$, and $Q = \lambda^5$.

> **KeyGen:** Takes as input the security parameter λ and generates key p which is a random P-bit odd integer.
>
> **Enc:** Takes as input the key p and a message $m \in \{0, 1\}$. To encrypt the bit m, choose m' to be a random N-bit number such that $m' = m \bmod 2$ and choose q to be a random Q-bit number. It outputs a fresh[1] ciphertext $c = m' + pq$ where m' is the noise of the system that masks the actual message m.
>
> **Dec:** Takes as input the key p and ciphertext c. It outputs $m_f = (c \bmod p) \bmod 2$. Here, $c' = (c \bmod p)$ is in the range $(-p/2, p/2)$ with the condition that p divides $c - c'$ with no remainder.
>
> **Eval:** Takes as input a boolean function f and a set of ciphertexts $S_c = \{c_1, \ldots, c_n\}$. First, f is represented as a circuit C made of XOR and AND gates. Let $C^\dagger$ be a copy of the circuit C, but with the XOR and AND gates replaced by addition and multiplication gates over the integers. Let $f^\dagger$ be the multivariate polynomial that corresponds to the circuit $C^\dagger$. It outputs $c = f^\dagger(c_1, \ldots, c_n)$, or the computation of the ciphertexts through the function $f^\dagger$.

In this scheme, the ciphertexts are near-multiples of p, but not exact multiples. For example, consider $\lambda = 2$. In **KeyGen**, p is a random $P = \lambda^2 = 4$ bit odd integer:

$$p = 1001 = 9$$

Now, choose the message $m = 1$. To encrypt m, first generate

$$m' = 1 \bmod 2 = 1$$

[1] Meaning it only has a small amount of noise, around N-bits.

and q as a $Q = \lambda^5 = 32$ bit integer,

$$q = 11011010110100000011111000111100 = 3671080508$$

The final ciphertext is computed as

$$c = 1 + 9(3671080508) = 33039724573$$

To show that this number is a near-multiple of p consider

$$(c \bmod p) = (33039724573 \bmod 9) = 1$$

This means that c is -1 away from a multiple of p.

Despite there being noise in the system, the correct decryption can still be achieved. Finishing out the example, take

$$(33039724573 \bmod 9) = 1 \bmod 2 = 1$$

which is the original message. This is possible because the noise exists in the same parity as the message, meaning that the message and error are both even or they are both odd.

In addition to being a valid encryption scheme, this scheme is also homomorphic because by adding, subtracting, or multiplying the ciphertext, the underlying messages are actually added, subtracted, or multiplied modulo 2. Despite this being a valid homomorphic scheme, a significant problem exists. As operations are carried out, the noise of the system grows. Eventually it grows to a point where decryption no longer returns the correct result. This means that the scheme can only support *some* functions, namely functions where the accrued noise does not grow over $p/2$.

Since SHE is limited to functions that are not too complicated (e.g. functions that only contain low-degree polynomials), one may wonder if there is any benefit to using SHE over FHE when FHE can compute *any* function. In fact, there are cases in which SHE may be sufficient or even preferred over FHE. In the case where only a simple function needs to be computed, the overhead of SHE is much lower than the overhead of FHE [2].

For example, take the computational times for the Gentry-Halevi's SHE and FHE implementations shown in Table 3.1. Comparing the times for the SHE scheme and the FHE scheme for the different key dimensions, it can be clearly seen that SHE has faster

[2]Here, overhead refers to the ratio of the time required for encrypted computation to the time required for plaintext computation.

computation times than FHE. But, this is at the drawback of only being able to compute some functions, not all. In addition, SHE schemes are not able to handle their own decryption function without significant modifications, meaning that these types of schemes are not inherently bootstrapable. If the SHE scheme can be bootstrapped, it can then be used to construct a FHE scheme. The concept of bootstrapping will be discussed in detail in the next section.

Table 3.1: Comparison of Genty-Halevi's SHE and FHE Schemes [28]

Dimension	SHE KeyGen	FHE KeyGen
2048	1.25 s	40 s
8192	10 s	8 min
32768	95 s	2 hr

3.3 Bootstrapping

Once again consider the defective glove-boxes where the gloves lock after a certain amount of time or number of manipulations. How can Alice circumvent this problem and still be able to produce jewelry? One solution would appear if the glove-boxes have a one-way insertion slot much like that of a mailbox. If this is the case, then one box could be put inside another through the slot. Alice can first give a worker a glove-box, box_1, containing the raw material. In addition, she can give them several other boxes, $\text{box}_2 \ldots \text{box}_n$. Inside each of the additional boxes is the key to the previous box, meaning box_2 contains the key to box_1, box_3 contains the key to box_2 and so on. Note that the final box is locked by a key that only Alice holds.

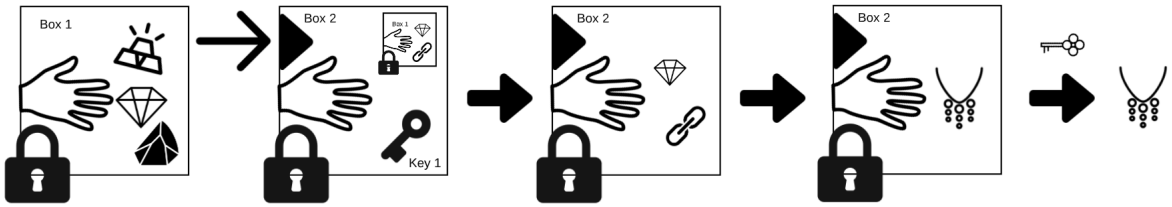

Figure 3.1: Bootstrapping procedure for Alice's jewelry shop.

In the Figure 3.1, box_1 contains the original form of the raw materials. The worker is able to manipulate the materials into a chain and a gemstone before the gloves lock. Then, they put the first box in the second through the one-way slot. Using the key in box_2, the worker unlocks box_1 and retrieves the materials. They then finish construction of the necklace. Alice can then use her key to unlock box_2 and retrieve the final product.

All existing HE schemes have the trait that a small amount of noise is added during encryption. Computing homomorphically on ciphertexts will cause the noise to grow to the point where they become so large that decryption fails [22]. Bootstrapping, originally proposed and termed *recrypt* by Gentry, can be used to lower the noise in a ciphertext to a fixed level that is determined by the complexity of the decryption circuit.

Definition 3.3.1 (Recrypt [23])**.** Let c_1 be the encryption of message bit m with key pk_1 and $\overline{sk_1}$ be a private key encrypted with key pk_2 using the **Enc** function $\mathbf{Enc}(pk_2, sk_{1_j})$ over the bits of sk_1. The function **Recrypt** can then be described as follows:

> **Recrypt:** Takes as input pk_2, the decryption circuit $D = \mathbf{Dec}$, $\overline{sk_1}$, and c_1. First, generate the vector $\overline{c_1}$ using the encrypt function $\mathbf{Enc}(pk_2, c_{1_j})$ which operates over the bits of c_1, i.e. encrypt c_1 element wise with pk_2. It returns the output of $\mathbf{Eval}(pk_2, D, \overline{sk_1}, \overline{c_1})$ which effectively eliminates the pk_1 portion of $\overline{c_1}$ leaving it encrypted only under pk_2.

Bootstrapping refreshes a ciphertext by running the decryption function on it homomorphically. Usually, when a ciphertext is doubly-encrypted (which happens in this scenario $\overline{c_1} = \mathrm{E}_{pk_2}(\mathrm{E}_{pk_1}(m))$) the outside ciphertext is decrypted in order to get to the inside. But in recrypt, the inner ciphertext is actually decrypted homomorphically with the corresponding key, $\overline{c_1} = \mathrm{E}_{pk_2}(\mathrm{D}_{\overline{sk_1}, pk_2}(c_1))$. While this decreases the noise of the system by eliminating the noise from the inner ciphertext, additional noise is added in the evaluate function. But, as long as the new noise added from evaluate is less than the noise eliminated from the inner ciphertext the scheme should continue to function.

Unfortunately, bootstrapping is an extremely costly operation. The complexity of most approaches to bootstrapping is at least the complexity of the decryption times the bit-length of the individual ciphertexts that are used to encrypt the bits of the private key [9]. This is because bootstrapping requires the evaluation of the decryption circuit homomorphically, so each private key bit is replaced by a large ciphertext encrypted representation of that bit.

3.4 Modulus Switching

Because of the large computational overhead of bootstrapping, alternative methods of noise reduction have been proposed. The most notable one is modulus switching, proposed by Brakerski, Gentry, and Vaikuntanathan in [9].

The main idea of modulus switching is to use an evaluator, who knows a bound on the length of the private key sk, but not sk itself, to transform a ciphertext c mod q into a different ciphertext c' mod p without jeopardizing the correctness of the scheme. The

transformation simply scales c by a factor (p/q) and rounds appropriately. If sk is short, and $p \ll q$, then the noise of the system decreases. Modulus switching allows the evaluator to minimize the noise of the system without knowing sk and without bootstrapping, making it a light-weight mode of noise management.

Definition 3.4.1 (Modulus Switching [9])**.** Let p and q be two odd moduli and $\mathbf{c}$ be an integer vector. Define $\mathbf{c}'$ to be the integer vector closest to $(p/q)\cdot\mathbf{c}$ such that $\mathbf{c}' = \mathbf{c} \bmod 2$. Then, for any s with $|[\langle \mathbf{c}, s\rangle]_q| < q/2 - (q/p)\cdot \ell_1(s)$:

$$[\langle \mathbf{c}', s\rangle]_p = [\langle \mathbf{c}, s\rangle]_q \bmod 2 \text{ and } |[\langle \mathbf{c}', s\rangle]_p| < (p/q)\cdot |[\langle \mathbf{c}, s\rangle]_q| + \ell_1(s)$$

where $\ell_1(s)$ is the ℓ_1 norm of s, $\langle a, b\rangle$ represents the dot product of a and b, and $[\cdot]_q$ represents and element modulo q.

It may not be inherently obvious why modulus switching works. The main idea of modulus switching is to scale down the ciphertext $c \in \mathbb{Z}_q$ by a factor β after each multiplication. To do this, modulus switching carefully chooses a gradually decreasing modulus q for each level of multiplication, which allows the noise level to remain small and constant from one level to the next. This results in a new ciphertext $c/\beta \in \mathbb{Z}_{q/\beta}$ and a reduced noise level e/β. By performing this procedure, the absolute magnitude of the noise in the ciphertext decreases. Through modulus switching, k levels of multiplication can be performed before reaching the noise ceiling. This is an exponential improvement compared to the $\log(k)$ levels allowed in regular HE multiplication[3].

3.5 FHE: Fully Homomorphic Encryption

While SHE schemes only allow the computation of *some* functions, fully homomorphic encryption (FHE) schemes enable the computation of *arbitrary* functions on encrypted data. This property makes FHE the most sophisticated homomorphic encryption scheme and the "holy grail" of modern cryptography.

The underlying concept of FHE is rather simple. Given a set of ciphertexts, $\{c_1, \cdots, c_n\}$ that are the encryption of messages $\{m_1, \ldots, m_n\}$, a FHE scheme should allow anyone to output a ciphertext that is the encryption of $f(m_1, \ldots, m_n)$ for any function f, as long as f can be efficiently[4] computed. In other words, if a user has a function f and wants to get the result of the plaintexts through the function, then it is possible to instead compute on the ciphertexts to obtain a result that decrypts to $f(m_1, \ldots, m_n)$. There should be

[3]See [9] for a more detailed explanation of how this exponential improvement is achieved.

[4]In this setting, a function f is efficient if the cost of evaluating f depends polynomially on a security parameter as well as the the complexity of the function itself.

no information leaked about the messages, the computation of the messages through the function f, $f(m_1, \ldots, m_n)$, or any intermediate plaintext values. All components of the scheme should always remain encrypted until the very end when the user with the private key decrypts the final value [24].

A HE scheme is a FHE scheme if it has the algorithms defined in Definition 3.1.3 and also satisfies the property that f can be any arbitrary function (as long as it is efficiently computatble). In addition, if a cryptosystem can encrypt messages $m \in \{0, 1\}$, and can perform addition and multiplication of the data efficiently, then the cryptosystem is a FHE scheme if the following hold:

- **Functionality:** Let S be a set of valid ciphertexts $S = \{c_1, \ldots, c_n\}$ and sk be a private key. For $c_i \in S$, let $C_{\text{add}} = c_1 + c_2 \in S$ and $C_{\text{mult}} = c_1 * c_2 \in S$. Then:
  1. $\text{Dec}(sk, C_{\text{add}}) = \text{Dec}(sk, c_1) + \text{Dec}(sk, c_2)$
  2. $\text{Dec}(sk, C_{\text{mult}}) = \text{Dec}(sk, c_1)$ * $\text{Dec}(sk, c_2)$
- **Efficiency:** For a security parameter λ:
  1. All operations (**KeyGen**, **Enc**, **Dec**, **Add**, **Mult**, **Eval**) only take polynomial time with respect to λ. This is to say that all functions can be computed *compactly.*
  2. All valid ciphertexts of the scheme have polynomial size with respect to λ. This means that the ciphertext size needs to be independent of the function being homomorphically evaluated.

The first practical FHE scheme was proposed by Gentry [24] and was based upon a modified version of his SHE scheme in which the decryption function is *squashed* to make it simple enough for the bootstrapping procedure. To do this, instead of computing the multiplication of two long numbers[5], addition of a small set of numbers is carried out instead. The summation performed corresponds to a low-degree polynomial that can be computed efficiently in a homomorphic fashion. In addition, a "hint" is added to the original public key - namely, a large set with a secret sparse subset that sums to the original private key and relies on the sparse subset sum assumption.

Gentry's original scheme was computed on polynomials, utilizing the hard problem of ring learning with errors (R-LWE). But instead of polynomials, FHE can be computed on integers using the hard problem of approximate greatest common divisor (AGCD) or

[5]In the original decryption scheme, $m = (c \bmod p) \bmod 2$ can be rewritten as $m = \text{LSB}(c)$ XOR $\text{LSB}(\lfloor c/p \rceil)$. The multiplication here refers to c time $1/p$.

even on matrices by using the hardness of normal learning with errors (LWE). LWE is considered a "hard" problem as it cannot be solved efficiently, or concretely that is it cannot currently be solved in polynomial time. No one underlying hard problem is inherently better than another, but rather the underlying hard problem is what provides proof of the security of the scheme. Proof of security is crucial in the design and implementation of any cryptographic system. Hard problems give a proof of security because if a cryptographic system can be reduced to a hard to solve problem, then the cryptosystem itself should be computationally difficult to break. In addition, the FHE libraries currently in use do not use Gentry's original FHE scheme, rather they use more efficient schemes such as BGV, BFV, and CKKS. These schemes and their underlying hard problems will be discussed thoroughly in the next chapter.

4 Underlying Schemes

Underneath every fully homomorphic encryption library lies a fully homomorphic encryption scheme. In the three libraries chosen, the three main FHE schemes used are BGV, BFV, and CKKS. In addition, many of the schemes support SIMD operations that introduce a dramatic decrease in run-time.

4.1 BGV: Brakerski-Gentry-Vaikuntanathan

The BGV scheme was proposed in 2012 in the paper "Fully Homomorphic Encryption without Bootstrapping" by Brakerski, Gentry, and Vaikuntanathan [9]. BGV is a levelled FHE (LFHE) scheme, meaning that the parameters of the scheme depend (polynomially) on the depth of the circuits that the scheme is capable of evaluating.

Definition 4.1.1 (Levelled Fully Homomorphic Encryption [9])**.** We say that a family of homomorphic encryption schemes with a positive integer security parameter L, $\{\mathcal{E}^{(L)} : L \in \mathbb{Z}^+\}$, is levelled fully homomorphic if, $\forall L \in \mathbb{Z}^+$, every $\mathcal{E}$ uses the same decryption circuit, $\mathcal{E}^{(L)}$ compactly evaluates all circuits of depth at most L, and the computational complexity of $\mathcal{E}^{(L)}$'s algorithms are polynomial (the same polynomial for all L) in L, and (in the case of the evaluation algorithm) the size of the circuit.

The depth referred to above is the *multiplicative depth*, which is different than the *multiplicative level* of the scheme. Multiplicative depth is how many sequential multiplications can be performed while multiplicative level is the total amount of multiplications that can be performed on a ciphertext. For example, the multiplicative depth of $x_1 \cdot x_2 + x_3 \cdot x_4$ is 1, not 2, even though two multiplications are carried out. If the equation was changed to $x_1 \cdot x_2 \cdot x_3$, the multiplicative depth would be 2 as two consecutive multiplications are carried out. Multiplicative level cannot exceed the multiplicative depth, otherwise decryption cannot be carried out successfully.

BGV allows the user to choose if they want a RLWE based version of BGV or a plain LWE based one. The main difference between using RLWE or plain LWE is that while they both achieve the same results, LWE does it with worse performance as the run time of a LWE scheme is worse than that of a RLWE one. In addition to using RLWE over LWE, they forgo Gentry's bootstrapping procedure for noise management and instead use modulus switching.

As explained in the previous chapter, the main idea behind modulus switching is that an evaluator, who does not know the private key $\mathbf{s}$ but instead knows a bound on its

length, can transform a ciphertext $\mathbf{c}$ modulo q into a different ciphertext modulo p while preserving correctness. They use modulus switching in their scheme to keep the noise level essentially constant while sacrificing modulus size and gradually sacrificing the remaining homomorphic capacity of the scheme.

In the LWE instantiation, $R = \mathbb{Z}_q^n$ is the ring of dimension n of integers mod q, whereas in RLWE instantiation, $R = \mathbb{Z}[x]_q/(x^d + 1)$ where $d = 2^n$. This is essentially the polynomials of degree less than d with coefficients modulo q.

N1 Analytics offers a simplistic example of polynomial ring arithmetic that we will recount here in order to give a strong foundation for the understanding of the FHE scheme computations [35]. Let $n = 4$, $d = 2^n = 16$, and $q = 24$. Then, plaintexts can take the form:

$$a_{15}x^{15} + a_{14}x^{14} + a_{13}x^{13} + a_{12}x^{12} + a_{11}x11 + a_{10}x^{10} + a_9x^9 +$$

$$a_8x^8 + a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

where $a_i \in \{0, q-1\}$ or 0 to 23. Visually, this can be depicted as in Figure 4.1 where each loop represents one power of x that appears in the polynomial and a dot represents one of the 24 possible values the coefficient can take. We will table this example for now and return back to it in the BFV section.

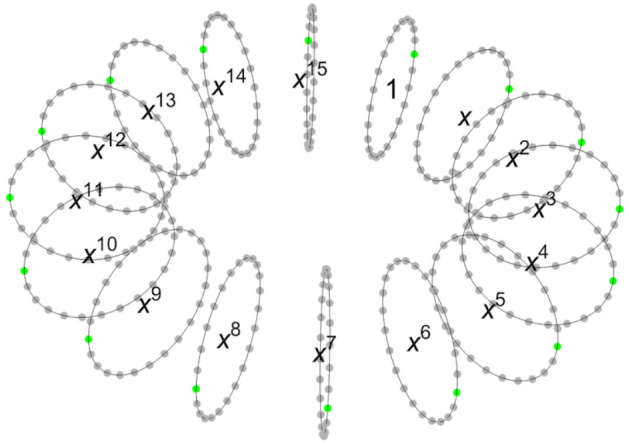

Figure 4.1: Depiction of a polynomial ring with degree 16 and plaintext modulus 24 [35].

A detailed explanation of BGV and how to implement the full scheme can be found in [9], but [2] provides a simplistic overview of how basic BGV functions. Here we will recount the basic version of BGV which provides enough detail for the purpose of this paper.

The basic BGV scheme can be broken down into 7 separate functions:

1. **ParamGen**(λ, μ, b): Takes as input security parameters λ and μ, and a bit $b \in$

$\{0, 1\}$ to set the LWE version (RLWE or LWE). It outputs $params = (q, d, n, \chi)$ where $q = q(\lambda)$ is an odd ciphertext modulus[1], $d = d(\lambda)$ is a power of 2, $n = n(\lambda)$ is the dimension of the system, and χ is a discrete Gaussian distribution used for error sampling. Note that for the RLWE variant $R = \mathbb{Z}[x]/(x^d + 1)$, or the ring with polynomials of degree less than d.

2. **SecKeyGen($params$):** Takes as input the parameters $params$. In the basic scheme the private key **sk** is sampled from the error distribution χ and belongs to the ring R. Namely:
$$\mathbf{sk} = \mathbf{s} = (1, \mathbf{s}'[1], \ldots, \mathbf{s}'[n]) \in R_q^{n+1} \text{ and } \mathbf{s}' \leftarrow \chi^n$$
where $\mathbf{s}'[i]$ is the i-th coefficient of $\mathbf{s}'$.

3. **PubKeyGen($params$):** Takes as input the parameters $params$. **PubKeyGen** first calls **SecKeyGen** to generate **sk**. From **sk** it extract $\mathbf{s}'$. Then, it generates a uniformly random $N \times n$ matrix $\mathbf{A}'$ from the ring $R_q^{N \times n}$ where $N = \lfloor (2n + 1) \log q \rfloor$ and an error term $\mathbf{e} \leftarrow \chi^n$. Set $\mathbf{b} \leftarrow \mathbf{A}'\mathbf{s}' + 2\mathbf{e}$ and $\mathbf{A}$ to be the $n+1$ column-matrix consisting of $\mathbf{b}$ followed by the n columns of $-\mathbf{A}'$. It returns the public key $\mathbf{pk} = \mathbf{A}$.

4. **Enc($params, m,$ pk):** Takes as input the parameters $params$, the message $m \in \{0, 1\}$, and the public key **pk**. First m is mapped into the ring R_q^{n+1} by setting $\mathbf{m} = (m, 0, \ldots, 0) \in R_q^{n+1}$. Then, a value[2] $\mathbf{r}$ is sampled from R_2^N. The final ciphertext output is $\mathbf{c} \leftarrow \mathbf{m} + \mathbf{A}^T\mathbf{r} \in R_q^{n+1}$. Note: while not explicitly shown in this explanation the ciphertext produced is comprised of two elements in R_q^{n+1}. See [2] for an in-depth explanation. The BFV section also touches on this tuple construction.

5. **Dec(sk, c):** Takes as input the private key **sk** (Note: $\mathbf{sk} = \mathbf{s}$) and the ciphertext **c**. **Dec** outputs $m \leftarrow [[\langle \mathbf{c}, \mathbf{s} \rangle]_q]_2$ which is the dot product of **c** with **s** modulo q modulo 2.

6. **EvalAdd($\mathbf{c}_1, \mathbf{c}_2$):** Takes as input two ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$ that are encrypted under the same **sk**. It outputs $\mathbf{c}_3 = \{(c_{1,0} + c_{2,0}), \ldots, (c_{1,n+1} + c_{2,n+1})\}$.

7. **EvalMult($\mathbf{c}_1, \mathbf{c}_2$):** Takes as input two ciphertexts $\mathbf{c}_1 = (c_{1,0}, c_{1,1})$ and $\mathbf{c}_2 = (c_{2,0}, c_{2,1})$ that are encrypted under the same **sk**. It outputs $\mathbf{c}_3 = \{(c_{1,0} * c_{2,0}), c_{1,0} * c_{2,1} + c_{1,1} * c_{2,0}, (c_{1,1} * c_{2,1})\}$.

[1] $x(\lambda)$ means x is polynomial in respect to λ.
[2] $\mathbf{r}$ is a vector of size N with elements modulo 2, i.e. has values of 0 and 1.

In the basic BGV scheme, ciphertexts grow as a result of **EvalMult** as shown in the example above. When performing a d-degree polynomial multiplication on a plaintext in BGV, the resulting ciphertext has $d+1$ ring elements. This problem is mitigated through the relinearization of the ciphertext, which will be explained more in the BFV section.

To turn the explained scheme above into a FHE scheme, after performing the wanted addition and multiplications on the ciphertext, a function named **Refresh** is called. **Refresh** essentially invokes a scaling function that switches the moduli and then switches the key which the resulting ciphertext is encrypted under. By combining **Refresh** with the bootstrapping procedure a FHE scheme can be achieved [50].

BGV offers a nice optimization of the scheme with the concept of batching. The main idea behind batching is to pack multiple plaintexts into each ciphertext so that a function can be homomorphically evaluated on multiple inputs with approximately the same efficiency as homomorphically evaluating it on one. Batching allows the reduction of the per-gate computation time from quasi-linear in the security parameter ($\widetilde{O}(\lambda \cdot L^3$ where L is the level of the system) to poly-logarithmic ($\widetilde{O}(\lambda)$). Batching, also referred to as packing, will be discussed later in detail.

It is important to note that out of the three schemes mentioned in this chapter BGV is the most efficient scheme when performing the same operation on multiple ciphertexts at once due to their batching procedure. But on the opposite side, out of the three, BGV is the most difficult to use and to implement correctly. Additionally, it can only perform computations over integers, not complex integers or real numbers. Despite these downfalls, BGV was chosen as a recommended scheme for HE during the Homomorphic Encryption Standardization Workshop held in 2018.

4.2 BFV: Brakerski-Fan-Vercauteren

In 2012, Fan and Vercauteren [22] modified Brakerski's [8] fully homomorphic encryption scheme based on LWE to work under the security assumption of RLWE. In both [22] and [8], they make use of relinearization, but BFV has a more efficient approach. They also simplify the bootstrapping procedure, by introducing a modulus switching trick that [8] did not have. These improvements will be explained throughout the remainder of this section.

The plaintext space in BFV is $R_t = \mathbb{Z}_t[x]/(x^d + 1)$ where t is the plaintext coefficient modulus and d is the plaintext polynomial modulus. The encryption of a plaintext in BFV generates a ciphertext which is represented by two polynomials from the ring with the same polynomial modulus d, but a different coefficient modulus $q \gg t$, i.e. $R_q = \mathbb{Z}_q[x]/(x^d + 1)$. Similar to BGV, these ciphertexts grow in size when homomorphic

multiplication is carried out. In order to keep the ciphertext from outgrowing what the scheme can support, a relinearization procedure[3] is used to bring a $n+1$ degree polynomial back to degree n. Fully homomorphic computations can be carried out in BFV fashion in the following manner:

Let λ be the security parameter, $q > 1$ be an integer polynomial modulus, d be a degree with $d = 2^n$, t be an integer plaintext coefficient modulus with $1 < t < q$, $\delta = \lfloor q/t \rfloor$, T be a positive integer base that will be used in the relinarization key generation, $\ell = \lfloor \log_T(q)) \rfloor$, R_2 represent the polynomial ring with coefficients modulo 2, i.e., the coefficients of the form {-1, 0, 1}, and χ be the a discrete Gaussian distribution over the integers with a standard deviation σ used for error sampling.

The BFV scheme can be broken down into 7 functions [52, 22, 35]:

1. **PrivateKeyGen(λ):** Takes as input the security parameter λ and randomly samples $\mathbf{s} \leftarrow R_2$. It outputs a private key $\mathbf{sk} = \mathbf{s}$. Referring back to the example from N1 Analytics in the section on BGV, the private key **sk** can take the form:

$$\mathbf{sk} = x^{15} - x^{13} - x^{12} - x^{11} - x^9 + x^8 + x^6 - x^4 + x^2 + x - 1$$

2. **PublicKeyGen(sk):** Takes as input the private key **sk** and sets $\mathbf{s} = \mathbf{sk}$. Sample $\mathbf{a} \leftarrow R_q$ and $\mathbf{e} \leftarrow \chi$. It outputs the public key **pk** as:

$$\mathbf{pk} = ([-\mathbf{a} \cdot \mathbf{s} + \mathbf{e})]_q, \mathbf{a})$$

For example, if $q = 874$ and $t = 7$ then **a** and **e** could equal:

$$\begin{aligned}\mathbf{a} =& 42x^{15} - 256x^{14} - 393x^{13} - 229x^{12} + 447x^{11} - 369x^{10} - 212x^9 + \\ & 52x^7 + 70x^6 - 138x^5 + 322x^4 + 186x^3 - 282x^2 - 60x + 84 \\ \mathbf{e} =& -3x^{15} + x^{14} + x^{13} + 7x^{12} - 6x^{11} - 6x^{10} + x^9 \\ & - x^6 + 3x^5 - 4x^4 + 4x^3 + 4x + 1\end{aligned}$$

The first part of the public key would then be constructed as shown in Figure 4.2

[3] In [22] they offer two different relinearization functions. For the purpose of this paper we will focus on version 1 which minimizes the relinearization error. Version 2 minimizes the time and space requirements of the relinearization function.

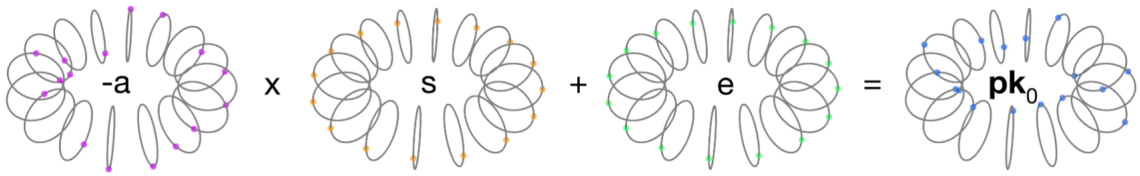

Figure 4.2: Computation of BFV public key [35]

$$\begin{aligned}\mathbf{pk}_0 = &-285x^{15} - 431x^{14} - 32x^{13} + 86x^{12} - 83x^{11} - 142x^{10} - 41x^9 + \\ &430x^8 + 26x^7 - 158x^6 - 281x^5 + 377x^4 + 110x^3 - 234x^2 - 113x + 252\end{aligned}$$

Notice that despite multiplying two polynomials, which usually causes the addition of their exponents, the degree of the plaintext polynomial never exceeds degree 15 due to the reduction of the polynomial by (x^d+1). This extra plus one to the reduction introduces a sign change which helps scramble the result of the multiplication. For instance, consider $2x^{14} \times x^4 = 2x^{18} \bmod (x^{16}+1) = -2x^2$ shown in Figure 4.3 below.

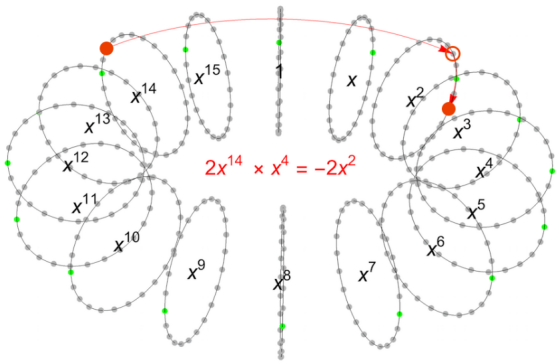

Figure 4.3: Multiplication of polynomials modulo (x^d+1). The green dot indicates the 0 value of the coefficient and the red dot indicates how the term is moved by multiplication [35].

3. **EvaluationKeyGen($\mathbf{sk}, T$):** Takes as input the private key $\mathbf{sk}$ and T. Setting $\mathbf{sk} = \mathbf{s}$, for $i = 0, \ldots, \ell$ sample $\mathbf{a}_i \leftarrow R_q$ and $\mathbf{e}_i \leftarrow \chi$. Output the evaluation key $\mathbf{evk}$ as:

$$\mathbf{evk} = ([-(\mathbf{a}_i \cdot \mathbf{s} + \mathbf{e}_i) + T^i \cdot \mathbf{s}^2]_q, \mathbf{a}_i)$$

for $i = 0, \cdots, \ell$.

4. **Encrypt($\mathbf{pk}, \mathbf{m}$):** Takes as input public key $\mathbf{pk}$ and a message $\mathbf{m} \in R_t$. First, it separates $\mathbf{pk}$ into $\mathbf{pk}[0] = \mathbf{p}_0$ and $\mathbf{pk}[1] = \mathbf{p}_1$. Then, it randomly samples $\mathbf{u}, \mathbf{e}_1, \mathbf{e}_2 \leftarrow \chi$. It computes and returns the ciphertext $\mathbf{ct}$ as:

$$\mathbf{ct} = ([\delta \cdot \mathbf{m} + \mathbf{p}_0\mathbf{u} + \mathbf{e}_1]_q, [\mathbf{p}_1\mathbf{u} + \mathbf{e}_2]_q)$$

5. **Decrypt($\mathbf{sk}, \mathbf{ct}$):** Takes as input private key $\mathbf{sk}$ and ciphertext $\mathbf{ct}$. First, it sets $\mathbf{sk} = \mathbf{s}$, $\mathbf{c}_0 = \mathbf{ct}[0]$, and $\mathbf{c}_1 = \mathbf{ct}[1]$. Then, it computes and outputs the message $\mathbf{m}' \in R_t$ as:

$$\mathbf{m}' = \left[\left\lfloor \frac{t \cdot [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_q}{q} \right\rceil\right]_t$$

6. **Add($\mathbf{ct}_0, \mathbf{ct}_1$):** Takes as input two ciphertexts $\mathbf{ct}_0$ $\mathbf{ct}_1$ and returns the resulting ciphertext $\mathbf{ct}'$ as:

$$\mathbf{ct}' = (\mathbf{ct}_0[0] + \mathbf{ct}_1[0], \mathbf{ct}_0[1] + \mathbf{ct}_1[1])$$

7. **Multiply($\mathbf{ct}_0, \mathbf{ct}_1$):** Takes as input two ciphertexts $\mathbf{ct}_0$ and $\mathbf{ct}_1$ and returns the resulting ciphertext $\mathbf{ct}'$ as:

$$\mathbf{c}_0 = \left[\left\lfloor \frac{t \cdot \mathbf{ct}_0[0] \cdot \mathbf{ct}_1[0]}{q} \right\rceil\right]_q$$

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

$$\mathbf{c}_2 = \left[\left\lfloor \frac{t \cdot \mathbf{ct}_0[1] \cdot \mathbf{ct}_1[1]}{q} \right\rceil\right]_q$$

$$\mathbf{c}'_0 = \mathbf{c}_0 + \sum_{i=0}^{\ell} \mathbf{evk}[i][0]\mathbf{c}_2^{(i)}$$

$$\mathbf{c}'_1 = \mathbf{c}_1 + \sum_{i=0}^{\ell} \mathbf{evk}[i][1]\mathbf{c}_2^{(i)}$$

$$\mathbf{ct}' = (\mathbf{c}'_0, \mathbf{c}'_1)$$

8. **Relinearization:** The main goal of relinearization is to take the size of the ciphertext back to at least degree 2 after a multiplication was carried out. Suppose that some multiplication produces a size 3 ciphertext, i.e., $(\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2)$. To turn this into a size 2 ciphertext $(\mathbf{c}'_0, \mathbf{c}'_1)$ that decrypts to the same result requires the use of the relinearization key created in the function **EvaluationKeyGen**. The natural

approach would be to set:

$$\mathbf{c}'_0 = [\mathbf{c}_0 + \mathbf{evk}[0]\mathbf{c}_2]_q$$

$$\mathbf{c}'_1 = [\mathbf{c}_1 + \mathbf{evk}[1]\mathbf{c}_2]_q$$

Unfortunately, since $\mathbf{c}_2$ can have coefficients up to size q the decryption process will fail. Instead, $\mathbf{c}_2$ will need to changed to a smaller base before being used in the above equations. Specifically:

$$\mathbf{c}_2 = \sum_{i=0}^{\ell} \mathbf{c}_2^i T^i$$

4.3 CKKS: Cheon-Kim-Kim-Song

Both BGV and BFV have a disadvantage in that they can only perform computations over the integers, or put another way, they only support discrete computations such as boolean, integer, or modulo operations. This makes them not very practical for the majority of real world applications since most real-world data belongs to a continuous space such as $\mathbb{R}$ or $\mathbb{C}$. CKKS solves this problem by allowing computation on complex numbers with limited precision by treating the encryption noise as part of the error that occurs naturally during approximate computations [47, 45].

CKKS was proposed by Cheon, Kim, Kim, and Song in the paper "Homorphic Encryption for Arithmetic of Approximate Numbers" released in 2017 [14]. The scheme originally went by the name HEAAN, but to distinguish it from the homomorphic encryption library HEAAN (which is a library that implements CKKS/HEAAN), the name was changed to CKKS after the authors. Since its release in 2017, several improvements have been made to the scheme such as a full residue number system (RNS) variant [16] and an bootstrapping function for the scheme which brings it to be a FHE scheme [12] as CKKS in the original paper was only a levelled HE scheme.

Before delving into the CKKS scheme, it is important to make the distinction between two types of approximate arithmetic operations: floating-point arithmetic and fixed-point arithmetic.

Definition 4.3.1 (Floating-point Representation)**.** In the floating-point number system, a real number is represented by a product of an integer called a significand and a scaling vector which is usually called a scaling factor. For example, $1.011101 = 1011101 * 2^{-6}$ has the significand 1011101 and a scaling factor of 2^{-6}. The intuition behind this representation scheme is that the floating point number is not the exact value we want to store, but it is just the approximate value.

Definition 4.3.2 (Floating-point Arithmetic)**.** In floating-point arithmetic, the significand is assumed to have a binary point to the right of the leftmost bit. In addition, the number of bits in the significand is fixed while the scaling factor can dynamically change during the computation. For example:

$$(101011 * 2^{-5}) * (110111 * 2^{-5}) = 100100111101 * 2^{-10} \approx 100101 * 2^{-4}$$

Definition 4.3.3 (Fixed-point Arithmetic)**.** For fixed-point arithmetic, the scaling vector is fixed, but the bit size of the significand can change. For example:

$$(101011 * 2^{-5}) * (110111 * 2^{-5}) = 100100111101 * 2^{-10} \approx 1001010 * 2^{-5}$$

Fixed point arithmetic is the preferred approximate arithmetic style for homomorphic computations since it is more stable. CKKS supports fixed-point arithmetic but allows for some extra noise from the scheme to be added.

In CKKS, there is a distinction between a message **m** and a plaintext **pk**. The message **m** is a vector of floating-point numbers or complex numbers. It is first transformed into a plaintext by a public encoding map before it is encrypted and computations can be carried out. Specifically, this encoding map is a complex canonical embedding map. The use of this type of map allows the transformation to preserve the precision of the plaintext after encoding and decoding[4]. In similar fashion, to obtain the decrypted message, the returned plaintext from the decryption function will have to be decoded back to a vector of either floating-point or complex numbers. As mentioned, both the encoding and decoding functions are public and are just transformations from $\mathbb{C}^{n/2} \times \mathbb{R}$ to $R = \mathbb{Z}[x]/(x^n + 1)$ for encoding and the opposite for decoding.

The main idea in the CKKS scheme is to treat the noise created during computations as part of the error that naturally occurs during approximate computations. Consider the encryption of significand m that satisfies $\langle c, sk \rangle = m + e \bmod q$ for some small error e. The decryption structure $m' = m + e$ itself is an approximate value of the original message m. If $|e|$ is small enough not to destroy the significand of m, the precision is almost preserved. For example, if $m = 1.23 * 10^4$, $e = -17$ then a possible decryption could be $m' = 12283 \approx m$.

Here we will go over a basic overview of the CKKS scheme with notation taken from [14].

Let $b > 0$ be a base, q_0 be a modulus and $q_\ell = b^\ell \cdot q_0$ for $0 < \ell \leq L$ where L is

[4]See [15] for an in-depth explanation of how canonical embedding works.

the maximum level of the scheme. For a real number $\sigma > 0$, $DG(\sigma^2)$ samples a vector in $\mathbb{Z}^N$ by drawing it coefficients independently from the discrete Gaussian distribution χ with variance σ^2. For a positive integer h, $HWT(h)$ is the set of signed binary vectors in $\{-1, 0, 1\}^N$ whose Hamming weight is exactly h. For a real number $0 \leq p \leq 1$, the distribution $ZO(p)$ draws each entry in the vector from $\{-1, 0, 1\}^N$, with probability $p/2$ for each of -1 and $+1$, and probability $1-p$ of being 0. Note: $\Delta \geq 1$ is the scaling factor. The CKKS scheme can be broken down into 8 functions:

- **KeyGen(λ):** Takes as input the security parameter λ, and generates the parameters of the scheme as $M = M(\lambda, q_L)$ which is a power of two, an integer $h = h(\lambda, q_L)$, an integer $P(\lambda, q_L)$, and a real value $\sigma = \sigma(\lambda, q_L)$.
  It then selects $\mathbf{s} \leftarrow HWT(h), \mathbf{a} \leftarrow R_{q_L}$, **a'** $\leftarrow R_{P \cdot q_L}$, $\mathbf{e} \leftarrow DG(\sigma^2)$, and **e'** $\leftarrow DG(\sigma^2)$. It generates and returns the private key **sk**, public key **pk**, and evaluation key **evk** as:

$$\begin{aligned} \mathbf{sk} &\leftarrow (1, \mathbf{s}) \\ \mathbf{pk} &\leftarrow (\mathbf{b}, \mathbf{a}) \in R_{q_L}^2 \text{ where } \mathbf{b} \leftarrow -\mathbf{as} + \mathbf{e} \bmod q_L \\ \mathbf{evk} &\leftarrow \mathbf{b}', \mathbf{a}') \in R_{P \cdot q_L}^2 \text{ where } \mathbf{b}' \leftarrow -\mathbf{a}'\mathbf{s} + \mathbf{e}' + Ps^2 \bmod P \cdot q_L \end{aligned}$$

- **Ecd($\mathbf{z}, \Delta$):** Takes as input a message $\mathbf{z} \in \mathbb{C}^{N/2}$ which is a vector of Gaussian integers, and returns the corresponding plaintext polynomial $\mathbf{m} \in R$. [5]

- **Dcd($\mathbf{m}; \Delta$):** Takes as input a ciphertext $\mathbf{m} \in R$ and returns the corresponding polynomial in $\mathbb{C}^{N/2}$.

- **Enc($\mathbf{m}, \mathbf{pk}$):** Takes as input a plaintext $\mathbf{m}$ and a public key **pk**. It first samples a vector $\mathbf{v} \leftarrow ZO(0.5)$ and $\mathbf{e}_0, \mathbf{e}_1 \leftarrow DG(\sigma^2)$. It outputs the encrypted message as:

$$\mathbf{c} = \mathbf{v} \cdot \mathbf{pk} + (\mathbf{m} + \mathbf{e}_0, \mathbf{e}_1) \bmod q_L$$

- **Dec($\mathbf{c}, \mathbf{sk}$):** Takes as input a ciphertext $\mathbf{c}$ and a private key **sk**. Taking $\mathbf{c} = (\mathbf{b}, \mathbf{a})$ it outputs a plaintext $\mathbf{m}'$ as:

$$\mathbf{m}' = \mathbf{b} + \mathbf{a} \cdot \mathbf{s}(\bmod q_\ell)$$

- **Add($\mathbf{c}_1, \mathbf{c}_2$):** Takes as input two ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$. It performs the addition of

[5] The technical details of encoding and decoding are beyond the scope of this paper. See [14] for a more detailed description.

the two ciphertexts and returns $\mathbf{c}_{add}$ as:

$$\mathbf{c}_{add} \leftarrow \mathbf{c}_1 + \mathbf{c}_2 \bmod q_\ell$$

- **Mult($\mathbf{c}_1, \mathbf{c}_2$, evk):** Takes as input two ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$ and represents them as $\mathbf{c}_1 = (\mathbf{b}_1, \mathbf{a}_1), \mathbf{c}_2 = (\mathbf{b}_2, \mathbf{a}_2)$. Let $\mathbf{d} = (\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2) = (\mathbf{b}_1\mathbf{b}_2, \mathbf{a}_1\mathbf{b}_2 + \mathbf{a}_2\mathbf{b}_1, \mathbf{a}_1\mathbf{a}_2) \bmod q_\ell$ represent the multiplication of the two ciphertexts. Output the multiplication and relinearization of $\mathbf{d}$ as:

$$\mathbf{c}_{mult} \leftarrow (\mathbf{d}_0, \mathbf{d}_1) + \lfloor P^{-1} \cdot RS(\mathbf{d}_2) \cdot \mathbf{evk} \rceil \bmod q_\ell$$

  where $\lfloor \cdot \rceil$ stands for rounding to the nearest integer.

- **RS$_{\ell \leftarrow \ell'}$($\mathbf{c}$):** Takes as input a ciphertext $\mathbf{c}$ and performs the change of basis from ℓ to ℓ' to be used in the relinearization procedure after multiplication. Namely:

$$\mathbf{c}' \leftarrow \left\lfloor \frac{q'_\ell}{q_\ell} \mathbf{c} \right\rceil \in \bmod q'_\ell$$

4.4 SIMD: Single Instruction Multiple Data

When performing the same operations on multiple ciphertext, the Single Instruction Multiple Data (SIMD) method should be utilized in order to maximize the efficiency of the scheme via parallel processing. SIMD allows the encryption of not just a single message per ciphertext, but rather a vector of messages in a single ciphertexts. Then, additions and multiplications of ciphertexts are carried out component wise. Using fully homomorphic SIMD operations enables a more efficient use of both space and computational resources [53].

All of the libraries discussed in this thesis use some sort of plaintext packing, also denoted batching, in their implementation of the FHE schemes to mimic SIMD operations and achieve more efficient computations.

5 HElib

Homomorphic Encryption Library (HElib)[1] is an open-source software library released in 2013 by Halevi and Shoup under the Apache License v2.0. It implements the RLWE version of the BGV scheme and many optimizations to make the homomorphic evaluations run more efficiently. Their optimizations include the implementation Smart-Vercauteren [53] ciphertext packing techniques, relinearization, and bootstrapping. It is written in C++ and makes use of the NTL mathematical library for polynomial arithmetic and multi-threading optimizations. In addition, HElib has native implementations for Linux and MacOS systems.

One way to view HElib is as implementing an assembly language which is executed on a hardware platform constructed by the underlying FHE scheme [55]. The hardware platform (the FHE scheme) defines the operation that can be applied homomorphically as well as the cost of the operations. Like assembly language, HElib is fairly low-level as it only carries computations such as set, add, multiply, and shift. At this time it is mostly meant for researchers working on HE rather than production implementations.

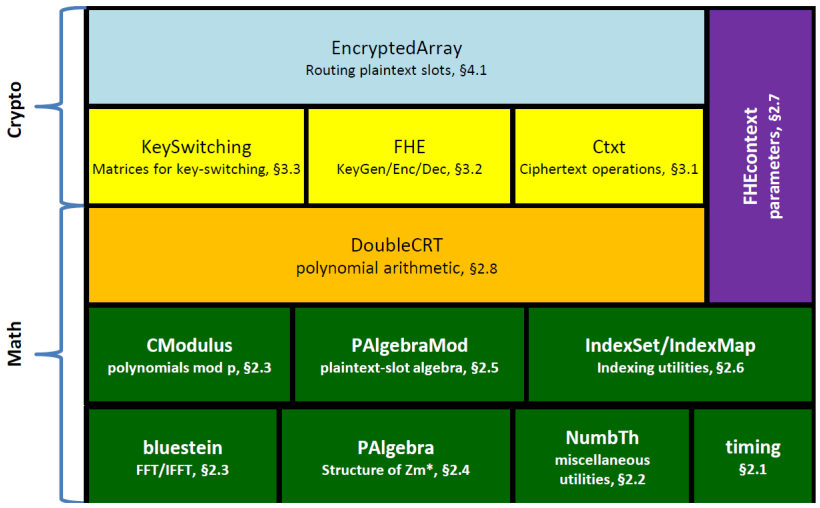

Figure 5.1: The layers of HElib. Taken from [32] page 4.

The HElib library can be broken down into 2 main layers, namely the math layer and the cryptography layer as shown in Figure 5.1. These 2 layers can then each be broken down into 2 more layer. The bottom layer belongs to the math layer and contains modules for implementing mathematical structures and various other utilities. The second layer

[1] http://homenc.github.io/HElib/

also belongs to the math layer and implements the function for computing the Double-Chinese Remainder Theorem (Double-CRT) representation of polynomials, which is a vital function as HElib only operates over polynomials in Double-CRT representation. This is different from other FHE libraries which deal with the coefficient representation of plaintexts. Double-CRT has the benefit of allowing additions and multiplications to take place in linear time, but at the cost of an expensive conversion between coefficient and Double-CRT representations as well as the requirement to perform a key switching function after every multiplication.

The third layer belongs to the crypto layer and implements the cryptosystem functions such as key generation, encryption, decryption, and evaluation. Finally, the top layer provides interfaces for using the cryptosystem to operate on arrays of plaintext values. For more information on specifically which functions each layer provides, see [32].

It is important to note that rather than working with integers in HElib, each ciphertext encrypts a vector $\mathbf{v} \in F^n$, where F can be any finite field that the user chooses. The length of $\mathbf{v}$ is not chosen by the user, rather it is determined by other parameters in the system. Typically, the vectors lie in the range of length $n \in [100, 1000]$. Operating with encrypted vectors makes HElib very synonymous to a SIMD architecture environment.

As noted before, the costs of each operation in HElib are predefined depending on the parameters set. But, generally the costs follow the pattern of addition being the cheapest operation and the multiplication of two vectors being the most expensive as summarized in Table 5.1.

Table 5.1: HElib operations and their cost which is measured in time and noise.

Operation	Time Cost	Noise Cost
Constant Addition	cheap	cheap
Addition	cheap	cheap
Constant Mult.	cheap	moderate[2]
Multiplication	expensive	expensive
Rotation	expensive	cheap

Since mid-2018, HElib has been extensively revised to improve its reliability, robustness, and performance. This includes the introduction of many new algorithms [33], more robust parameter derivations to limit the necessity of recryption, as well as a significantly improved bootstrapping procedure [34].

[2]This is even for multiplying by a constant factor of 0 or 1 which is basically a free operation in other FHE libraries' implementations.

5.1 Implementation

In order to install and use HElib 1.0.0, several external libraries will need to be installed. Specifically: CMake version 3.5.1 or greater, Make, g++ version 5.4.0 or greater, pthreads, git, patchelf, and m4. HElib has two main external dependencies, NTL version 11.0.0 or greater and GMP version 6.0.0 or greater.

The HElib documentation offers two modes of installation. The first mode is a package build that bundles HElib and its dependencies in a directory where they can be moved around freely on the system to wherever the user wishes. In addition, NTL and GMP are automatically downloaded and installed in this process. The second mode of installation requires the user to build the libraries themselves. For simplicity, we will install HElib using the package mode of installation. Here, we will recount installing HElib on Ubuntu 18.04 but other modes of installation can be found on their github.

1. Install the pre-requisites : CMake, Make, g++, pthreads, git, patchelf, and m4

   ```
   $ sudo apt-get install build-essential cmake make
      libpthread-stubs0-dev git patchelf m4
   ```

2. Clone the HElib repository from Github

   ```
   $ git clone https://github.com/homenc/HElib.git
   ```

3. Create a build directory as a sibling of `src`

   ```
   $ cd HElib
   $ mkdir build
   $ cd build
   ```

4. Run the CMake configuration, specifying that the mode of installation should be package build. Other build options can be specified here as well, for example, testing can be enabled and is done so in this example.

   ```
   $ cmake -DPACKAGE_BUILD=ON -DENABLE_TEST=ON ..
   ```

5. Compile the install

   ```
   $ make -j
   ```

   Occasionally, the HElib installer will not be able to download GMP and NTL. If this is the case, download the required version of each and place the .tar.bz2 and .tar.gz

files in their needed directories `/HElib/build/dependencies/Download/gmp_fetched` and `/HElib/build/dependencies/Download/ntl_fetched` then run `$ make -j` again.

6. Test the install compilation

```
$ make -j test
```

   Note: It will look like not much is happening as there is no progress bar while the tests run, but just be patient. Eventually the tests will finish. If all the tests are successful, the final output will look as in Figure 5.2.

7. Run the install

```
$ sudo make install
```

HElib comes with two examples that the user can reference when learning how to use the library. The first is `BGV_general_example` and the second is `binaryArith_example`. Here, we will show how to compile and run `BGV_general_example`.

1. Go to the directory of the wanted example

```
$ cd ~/Helib/examples/BGV_general_examples
```

2. Run the CMake configuration, build the executable, and run the example

```
$ cmake .
$ make
$ ./BGV_general_example
```

The results of the `BGV_general_example` should resemble those of Figure 5.3

The easiest way for a user to build their own programs with HElib is to use CMake. In the CMake file for the project, add the line:

```
find_package(helib)
```

Then, when running CMake, use the option:

```
-Dhelib_DIR=/usr/local/helib_pack/share/cmake/helib
```

See Figure 5.4 for an example CMake file.

```
(base) ancarey@445-7920-02:~/HElib/build$ make -j test
Running tests...
Test project /home/ancarey/HElib/build
    Start 1: helib_check
1/3 Test #1: helib_check ......................   Passed  338.94 sec
    Start 2: gmp_check
2/3 Test #2: gmp_check ........................   Passed   14.52 sec
    Start 3: ntl_check
3/3 Test #3: ntl_check ........................   Passed  566.55 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) = 920.03 sec
```

Figure 5.2: Output of HElib test script.

```
(base) ancarey@445-7920-02:~/HElib/examples/BGV_general_example$ ./BGV_general_example
Initialising context object...
Building modulus chain...
m = 32109, p = 4999, phi(m) = 16560
  ord(p)=690
  normBnd=2.32723
  polyNormBnd=58.2464
  factors=[3 7 11 139]
  generator 320 has order (== Z_m^*) of 6
  generator 3893 has order (== Z_m^*) of 2
  generator 14596 has order (== Z_m^*) of 2
  T = [1 14596 3893 21407 320 14915 25618 11023 6073 20668 9965 27479 16820 31415 10009 27523 20197 2683 24089 9494 9131 23726 23
20 19834 ]

Security: 127.626
Creating secret key...
Generating key-switching matrices...
Number of slots: 24
Initial Ptxt: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
Decrypted Ptxt: [0 2 8 18 32 50 72 98 128 162 200 242 288 338 392 450 512 578 648 722 800 882 968 1058]
```

Figure 5.3: HElib BGV Example output.

```
cmake_minimum_required(VERSION 3.12)

#set project name
project(HElibBGV)

find_package(helib)


#add the executable
add_executable(HElibBGV HElibBGV.cpp)
target_link_libraries(<any required libraries>)
```

Figure 5.4: HElib example CMake file.

6 Microsoft SEAL

Microsoft's Simple Encrypted Arithmetic Library (SEAL) is an open-source FHE library that aims for making homomorphic encryption easy to use and available for everyone [52]. The library is designed with cloud computing cases in mind, wanting to give users an easy way to interact with encrypted data stored on cloud services without having to download and decrypt it first. Despite FHE schemes being able to be symmetric or asymmetric, SEAL only offers the asymmetric variants.

SEAL is already deployed by several companies, meaning that SEAL can be seen as a production-quality open-source FHE library. For example, Intel integrates SEAL into their neural network compiler nGraph, allowing artificial intelligence models to directly process encrypted data.

Development of SEAL started in 2015 and it was first released under the MIT license to the public on github[1] in 2018. Currently it is on version 3.4.0, but version 3.5.0 is set to release mid-April 2020. It was developed in C++17, although it does come with some C# components namely the .NET standard wrapper which allows for cross-platform implementation. SEAL can be deployed on Windows, Linux, MacOS, and Android, making it a highly versatile library.

SEAL currently implements two homomorphic encryption schemes, BFV and CKKS. The only real similarity between SEAL's BFV implementation and the textbook implementation is that the plaintext space is still R_t. SEAL's ciphertext space is tuples of R_q of at least length 2, which is different than textbook BFV which has a ciphertext space of $R_q \times R_q$. This change allows arbitrarily sized ciphertexts, but at the cost of losing the compactness property of homomorphic encryption [13] and that addition and multiplication functions have to now support arbitrary sized ciphertexts. A benefit to using the arbitrary sized ciphertexts is that there is no need for relinearization before decrypting the ciphertext back to the wanted message.

Although SEAL's end API is not too difficult to use, they give the warning that there is a steep learning curve and that the user should understand many homomorphic encryption specific concepts before using the library. This is because the performance differences can be severe (up to 100,000 times slower) between a simple implementation, versus a highly optimized implementation that was created by someone who is experienced with FHE concepts. If the user tries to simply re-use or adapt code from the examples, they will

[1] GitHub.com/Microsoft/Seal

most likely produce code that is vulnerable, malfunctioning, or extremely slow.

Luckily, SEAL provides extensively commented code and examples to help users who are new to the concepts of homomorphic encryption to learn what each parameter does, and how changing the parameters affects the overall security. All of the examples can be found in `/SEAL/native/examples` and run by typing `./sealexamples` in the `/SEAL/native/bin` folder.

They offer six main examples, namely:

1. BFV Basics : `1_bfv_basics.cpp`
   In this example, they show how to evaluate a simple polynomial of encrypted integers through using the BFV mode of operation. For new users, this is the best place to start as they give detailed explanations of the parameters used in SEAL.

2. Encoders : `2_encoders.cpp`
   The encoders example introduces the user to three different encoders at their disposal: `IntegerEncoder`, `BatchEncoder`, and `CKKSEncoder`. Note that the first two encoding schemes are only available in BFV mode and the last in CKKS mode. An encoder takes messages from the user and encodes them into a plaintext polynomial that can be used by SEAL.

3. Levels : `3_levels.cpp`
   In the levels example, the concept of levels in the BFV and CKKS scheme are discussed. In SEAL, a set of encryption parameters is uniquely identified by a 256-bit hash of the parameters which allows for easy access to the parameters. But, as soon as any of the parameters change, the hash will as well. To overcome this issue, SEAL creates a chain of encryption parameters derived from the original parameter set. Creating this chain allows for easy access to all of the parameter sets, which in turn enables modulus switching (i.e., changing the ciphertext parameters down in the chain) to be performed.

4. CKKS Basics : `4_ckks_basics.cpp`
   This example explains the basics of how to implement CKKS in SEAL. Namely, it introduces the re-scaling function. Re-scaling is used after multiplications in CKKS to reduce the size of the polynomial and stabilize how the polynomial expands. In order to perform an operation involving two or more different ciphertexts all of the ciphertexts must be encrypted under the same parameter set. To achieve this, the modulus switching procedure explained in `3_levels.cpp` is used. In the example

they show how to compute the polynomial function $\pi x^3 + 0.4x + 1$ for several floating point values x.

5. Rotation : `5_rotation.cpp`
   The rotation example shows how to cyclically rotate encrypted vectors in BFV as well as CKKS.

6. Performance : `6_performance.cpp`
   Rather than an example, the performance script offers the times for all operations in SEAL based off of user selected parameters. Tests can be run for CKKS and BFV with both default SEAL set degrees and user specified degrees.

6.1 Implementation

SEAL can be installed on Windows, Linux, MacOS, as well as Android. The following instructions will detail the global installation of SEAL on an Ubuntu 18.04 systems. Other operating systems, as well as a local installation instructions, can be found on SEAL's github repository.

In order to install SEAL version 3.4, only CMake version 3.12 or higher and g++ version 6.0 or higher needs to be installed. SEAL has no external dependencies that must be installed for it to function.

1. Install the pre-requisites: CMake, g++

```
$ sudo apt-get install build-essential
$ sudo apt-get install cmake
```

2. Clone the SEAL github repository to the local machine. For this tutorial, SEAL was downloaded to a file in the home directory.

```
$ mkdir SEAL
$ cd SEAL
$ git clone https://github.com/microsoft/SEAL.git --
   recurse-submodules
```

3. Build SEAL

```
$ cd native/src
$ cmake .
$ make
$ cd ../..
```

4. Build SEAL examples

```
$ cd native/examples
$ cmake .
$ make
$ cd ../..
```

5. Build unit tests

```
$ cd native/tests
$ cmake .
$ make
$ cd ../..
```

6. Install SEAL

```
$ cd native/src
$ cmake .
$ make
$ sudo make install
$ cd ../..
```

Before SEAL is used, it is important to test that all of the modules were installed and configured correctly. To do this, run the tests found in `/SEAL/native/bin` with the command `./sealtest`. If the installation was carried out correctly, an output similar to Figure 6.1 will be shown.

SEAL also makes it simple for users to run their own programs through using CMake. After creating the program, the user writes the CMake file in the same directory. The contexts of the CMake file can be seen in Figure 6.2.

If the installation was done globally, CMake can be run with `$ cmake .`, but if SEAL was installed to not the default location (`/usr/local/`), CMake must be run with:

```
$ cmake .  -DCMAKE_PREFIX_PATH=<your path>
```

```
[----------] 23 tests from UIntCore
[ RUN      ] UIntCore.AllocateUInt
[       OK ] UIntCore.AllocateUInt (0 ms)
[ RUN      ] UIntCore.SetZeroUInt
[       OK ] UIntCore.SetZeroUInt (0 ms)
[ RUN      ] UIntCore.AllocateZeroUInt
[       OK ] UIntCore.AllocateZeroUInt (0 ms)
[ RUN      ] UIntCore.SetUInt
[       OK ] UIntCore.SetUInt (0 ms)
[ RUN      ] UIntCore.SetUIntUInt
[       OK ] UIntCore.SetUIntUInt (0 ms)
[ RUN      ] UIntCore.SetUIntUInt2
[       OK ] UIntCore.SetUIntUInt2 (0 ms)
[ RUN      ] UIntCore.IsZeroUInt
[       OK ] UIntCore.IsZeroUInt (0 ms)
[ RUN      ] UIntCore.IsEqualUInt
[       OK ] UIntCore.IsEqualUInt (0 ms)
[ RUN      ] UIntCore.IsBitSetUInt
[       OK ] UIntCore.IsBitSetUInt (0 ms)
[ RUN      ] UIntCore.IsHighBitSetUInt
[       OK ] UIntCore.IsHighBitSetUInt (0 ms)
[ RUN      ] UIntCore.SetBitUInt
[       OK ] UIntCore.SetBitUInt (0 ms)
[ RUN      ] UIntCore.GetSignificantBitCountUInt
[       OK ] UIntCore.GetSignificantBitCountUInt (0 ms)
[ RUN      ] UIntCore.GetSignificantUInt64CountUInt
[       OK ] UIntCore.GetSignificantUInt64CountUInt (0 ms)
[ RUN      ] UIntCore.GetNonzeroUInt64CountUInt
[       OK ] UIntCore.GetNonzeroUInt64CountUInt (0 ms)
[ RUN      ] UIntCore.GetPowerOfTwoUInt
[       OK ] UIntCore.GetPowerOfTwoUInt (0 ms)
[ RUN      ] UIntCore.GetPowerOfTwoMinusOneUInt
[       OK ] UIntCore.GetPowerOfTwoMinusOneUInt (0 ms)
[ RUN      ] UIntCore.FilterHighBitsUInt
[       OK ] UIntCore.FilterHighBitsUInt (0 ms)
[ RUN      ] UIntCore.CompareUIntUInt
[       OK ] UIntCore.CompareUIntUInt (0 ms)
[ RUN      ] UIntCore.GetPowerOfTwo
[       OK ] UIntCore.GetPowerOfTwo (0 ms)
[ RUN      ] UIntCore.GetPowerOfTwoMinusOne
[       OK ] UIntCore.GetPowerOfTwoMinusOne (0 ms)
[ RUN      ] UIntCore.DuplicateUIntIfNeeded
[       OK ] UIntCore.DuplicateUIntIfNeeded (0 ms)
[ RUN      ] UIntCore.HammingWeight
[       OK ] UIntCore.HammingWeight (0 ms)
[ RUN      ] UIntCore.HammingWeightSplit
[       OK ] UIntCore.HammingWeightSplit (0 ms)
[----------] 23 tests from UIntCore (1 ms total)

[----------] Global test environment tear-down
[==========] 239 tests from 39 test suites ran. (961 ms total)
[  PASSED  ] 239 tests.
```

Figure 6.1: Output of SEAL test script.

```
cmake_minimum_required(VERSION 3.12)

#set project name
project(SEALBFV)

#project(SEAL VERSION 3.4.5 LANGUAGES CXX C)
find_package(SEAL 3.4 REQUIRED)


#add the executable
add_executable(SEALBFV SealBFV.cpp)
target_link_libraries(SEALBFV SEAL::seal)
```

Figure 6.2: Example SEAL CMake file.

7 PALISADE

PALISADE was released in 2017 and is currently supported by a team at the New Jersey Institute of Technology along with the backing of several partners and collaborators in academia (MIT, UCSD, WPI, ...) and industry (Raytheon, IBM Research, Galois, ...). It is released under the BSD 2 clause and has cross platform support for Windows, Linux, MacOS, and Android environments. It is written in C++ and the objects that are created by and manipulated within PALISADE are instances of C++ classes.

PALISADE is unique from the other FHE libraries discussed in this thesis in the sense that it is actually a lattice cryptography toolkit. In addition to being able to perform FHE computations, it provides implementations for the building blocks of lattice cryptography capabilities along with end-to-end implementations of advance lattice cryptography protocols for public-key encryption, proxy re-encryption, program obfuscation and more. They also provide an experimental platform for research and development as well as an implementation ready platform of known protocols that can directly be integrated into applications. Major contributions to secure computing have already been made using PALISADE. The main one being that it was used as the library for a winning Genome-Wide Association Studies (GWAS) solution at the iDASH Secure Genome Analysis Competition in 2018 [1].

Here, we will recount an overview of the PALISADE construction, paying specific mind to how the scheme handles FHE computations. PALISADE offers several HE schemes to choose from. Namely: BGV, 3 variants of BFV, CKKS, and Stehle-Steinfeld (StSt)[2]. In addition, they offer other HE related protocols such as proxy re-encryption (PRE), SHE, levelled SHE, and multiparty homomorphic encryption.

Knowing that debugging FHE applications can be a slow process due to the computations on encrypted data being significantly slower and more compute-intensive than computing on plaintext data, PALISADE provides a `Null` scheme for fast error checking. `Null` supports the same API as BFV, BGV, and StSt implementations, but does not encrypt the data and performs all operations on unencrypted plaintexts. It serves as a light-duty no-security equivalent of the encrypted computing protocols so that developers can test the correctness of their PALISADE program efficiently.

[1] http://www.humangenomeprivacy.org/2018/

[2] Version 1.7 of PALISADE also supports the FHEW scheme.

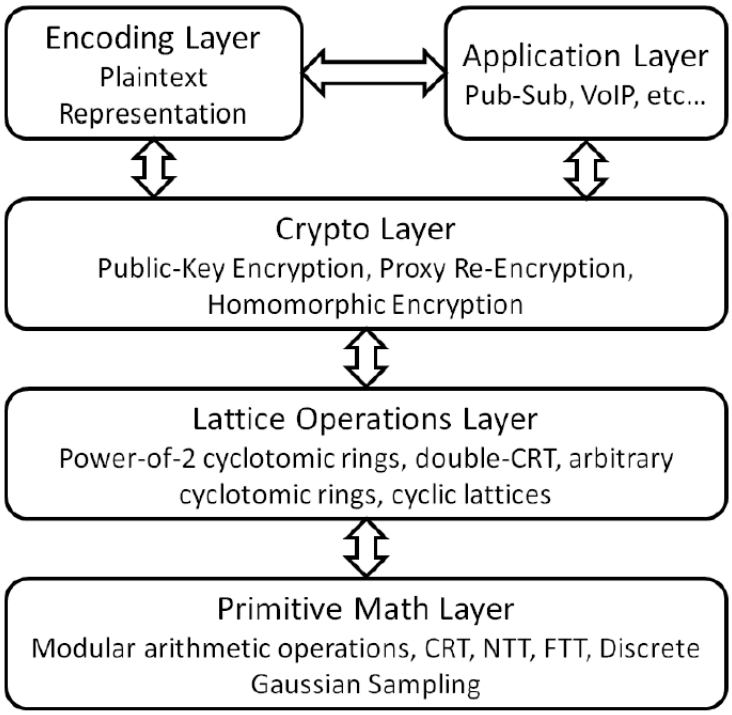

Figure 7.1: The layers of PALISADE. Taken from [43] page 10.

Like HElib, PALISADE also has a layered design as shown in Figure 7.1. Each level of PALISADE offers a set of services to the layer above it, and makes use of services in the layer below. The layers of PALISADE are as follows:

1. **Application:** All program that use the PALISADE library functions can be found in this layer. This layer makes calls to functions found in the crypto layer as well as functions in the encoding layer. When developing applications, the Application layer serves as an entry point to the rest of the PALISADE library.

2. **Encoding:** The encoding layer contains all the functions necessary to encode a plaintext message into a plaintext object that is usable by PALISADE as well as the corresponding decoding functions.

3. **Crypto:** All of the classes corresponding to lattice cryptography functions are found at this layer.

4. **Lattice Operations:** This layer provides support for lattice constructions such as rings. The Double-CRT representation of rings is also implemented in this layer. Operations are performed on lattices by decomposing the operations into primitive arithmetic operations represented as integers, vectors, and matrices. The primitive math layer is utilized to perform these operations.

5. **Primitive Math:** All low-level mathematical operations can be found at this layers. The primitive math layer provides support for basic modular arithmetic, efficient Number Theoretic Transform (NTT) computations, Fermat-Theoretic Transform (FTT) functions, and discrete Gaussian samplers among other functions.

The official PALISADE documentation [43] provides an easy to read and in-depth look at all of the capabilities of PALISADE along with several sample implementations that a user can reference when building their projects. In addition, they provide implementation instructions on their git repository [3] that we will detail next.

7.1 Implementation

In order to install PALISADE version 1.7, a few external libraries will need to be installed, namely: a C++ compiler with OpenMP library support, CMake, Make, and autoconf. By default, PALISADE does not have any external dependencies but the user is given the option to add GMP/NTL and tcmalloc third-party libraries if they wish.

Here, we will provide the instruction to install PALISADE on a Linux system, specifically Ubuntu 18.04. Other OS installation instructions can be found on their gitlab wiki[4].

1. Install pre-requisites: g++, CMake, Make, and autoconf

```
$ sudo apt-get install build-essential
$ sudo apt-get install cmake
$ sudo apt-get install autoconf
```

2. Clone the PALISADE git repository to the local machine

```
$ git clone https://gitlab.com/palisade/palisade-
   release.git
```

3. Change directories to the file where the cloned PALISADE repository is and download the sub-modules

```
$ cd PALISADE
$ git submodule sync --recursive
$ git submodule update --init --recursive
```

[3] https://gitlab.com/palisade/palisade-release/-/tree/master
[4] https://gitlab.com/palisade/palisade-release/-/wikis/home

4. Create a build directory where the binaries will be built

```
$ mkdir build
$ cd build
$ cmake ..
```

5. Install external dependencies such as GMP/NTL or tcmalloc if desired

6. Build PALISADE

```
$ make -j(number of processes)
```

   Note: the `-j` command specifies the number of jobs to run simultaneously. We used `-j16` in our installation.

7. Install PALISADE

```
$ sudo make install
```

Before using PALISADE for the first time, it is important to test and clean the build to make sure everything is functioning correctly. In the `build` folder:

1. Run unit tests to make sure all capabilities operate as expected

```
$ make testall
```

   This step will take a couple minutes to run. If all goes well, the output in Figure 7.2 will show.

2. Run the sample example

```
$ cd bin/demo/pke
$ ./demo-simple-example
```

   The output should be very similar to Figure 7.3

PALISADE makes it easy to build C++ projects by providing a sample CMake file that can be copied into the working project directory. For example, to build a project called `TestProject` perform the following:

1. Build and install PALISADE

2. Create the C++ project anywhere on your system. For this example, we have copied the code from the CKKS example demo-simple-real-numbers.cpp into a folder called `SampleProjectBuild` that was on the desktop.

3. Copy CMakeLists.User.txt from the root directory of the git repository to the folder of the project.

4. Rename CMakeLists.User.txt to CMakeLists.txt.

5. Update CMakeLists.txt to specify the desired name of the executable and the source code files. See Figure 7.4 for an example.

6. Run the CMake and Make commands to build the executable. Note: the directory of PALISADE is `/usr/local` by default, but if upon installation the directory was specified to be something different, the path to that directory would go where `/usr/local` is here.

```
$ cmake -DPALISADE_DIR=‘/usr/local‘ ..
$ make
```

7. Run the final program

```
$ ./SampleProjectBuild
```

For reference, the output of the demo-simple-real-numbers.cpp example is shown in Figure 7.5.

```
ancarey@445-7920-02:~/PALISADE/build$ make testall
-- demoData folder already exists
[  0%] Built target third-party
[ 43%] Built target coreobj
[ 45%] Built target PALISADEcore
[ 54%] Built target pkeobj
[ 54%] Built target PALISADEpke
[ 56%] Built target binfheobj
[ 56%] Built target PALISADEbinfhe
[ 58%] Built target binfhe_tests
[ 71%] Built target core_tests
[ 90%] Built target pke_tests
[ 92%] Built target abeobj
[ 94%] Built target PALISADEabe
[ 96%] Built target abe_tests
[ 98%] Built target sigobj
[ 98%] Built target PALISADEsignature
[100%] Built target signature_tests
Scanning dependencies of target testall
core:
Testing Backends: 2 Native
****** PALISADE Version 1.7.4
****** Date 2020-01-25T17:01:33
****** End 171 cases 171 passed 0 failed
pke:
Testing Backends: 2 Native
****** PALISADE Version 1.7.4
****** Date 2020-01-25T17:01:45
****** End 471 cases 471 passed 0 failed
abe:
Testing Backends: 2 Native
****** PALISADE Version 1.7.4
****** Date 2020-01-25T17:03:37
****** End 25 cases 25 passed 0 failed
signature:
Testing Backends: 2 Native
****** PALISADE Version 1.7.4
****** Date 2020-01-25T17:04:07
****** End 6 cases 6 passed 0 failed
binfhe:
Testing Backends: 2 Native
****** PALISADE Version 1.7.4
****** Date 2020-01-25T17:04:08
****** End 9 cases 9 passed 0 failed
[100%] Built target testall
```

Figure 7.2: Sample output from running the PALISADE unit tests

```
ancarey@445-7920-02:~/PALISADE/build/bin/demo/pke$ ./demo-simple-example
Plaintext #1: ( 1 2 3 4 5 6 7 8 9 10 11 12 ... )
Plaintext #2: ( 3 2 1 4 5 6 7 8 9 10 11 12 ... )
Plaintext #3: ( 1 2 5 2 5 6 7 8 9 10 11 12 ... )

Results of homomorphic computations
#1 + #2 + #3: ( 5 6 9 10 15 18 21 24 27 30 33 36 ... )
#1 * #2 * #3: ( 3 8 15 32 125 216 343 512 729 1000 1331 1728 ... )
Left rotation of #1 by 1: ( 2 3 4 5 6 7 8 9 10 11 12 ... )
Left rotation of #1 by 2: ( 3 4 5 6 7 8 9 10 11 12 ... )
Right rotation of #1 by 1: ( 0 1 2 3 4 5 6 7 8 9 10 11 ... )
Right rotation of #1 by 2: ( 0 0 1 2 3 4 5 6 7 8 9 10 ... )
```

Figure 7.3: Output of running the example of PALISADE in `bin/demo/pke`

```
### ADD YOUR EXECUTABLE(s) HERE
### add_executable( EXECUTABLE-NAME SOURCES )
###
### EXAMPLE:
### add_executable( test demo-simple-example.cpp )
add_executable( SampleProjectBuild  demo-simple-real-numbers.cpp)
```

Figure 7.4: Updated CMakeLists.txt file.

```
(base) ancarey@445-7920-02:~/Desktop/TestProject/build$ ./SampleProjectBuild
CKKS scheme is using ring dimension 16384

Input x1: ( 0.25 0.5 0.75 1 2 3 4 5 ... )
Input x2: ( 5 4 3 2 1 0.75 0.5 0.25 ... )

Results of homomorphic computations:
x1 + x2 = ( 5.25 4.5 3.75 3 3 3.75 4.5 5.25 ... )
x1 - x2 = ( -4.75 -3.5 -2.25 -1 1 2.25 3.5 4.75 ... )
4 * x1 = ( 1 2 3 4 8 12 16 20 ... )
x1 * x2 = ( 1.25 2 2.25 2 2 2.25 2 1.25 ... )

In rotations, very small outputs (~10^-10 here) correspond to 0's:
x1 rotate by 1 = ( 0.5 0.75 1 2 3 4 5 1.7434774e-11 ... )
x1 rotate by -2 = ( -9.121587e-11 -4.3297762e-10 0.25 0.5 0.75 1 2 3 ... )
```

Figure 7.5: Output from executing demo-sample-real-numbers.cpp

8 Implementation of Example Programs

In order to show the comparison between the different libraries, we will implement the same example program in each library for the schemes it supports. The test will be the homomorphic evaluation of a final velocity (V_f) given an acceleration (a), time (t), and initial velocity (V_i). Namely, we will homomorphically compute $V_f = V_i + at$. While the example is fairly simplistic, it helps to provide insight not only on how the libraries work, but also how the libraries' run times compare. When possible, we will use batching/packing or any other optimizations available. The target security level for all of the tests is 128 bits, meaning that it should take an attacker $O(2^{128})$ operations to break the scheme. We compare the schemes based on timing, testing the times for parameter generation, key generation, encryption, evaluation, and decryption. Tables 8.1 and 8.2 show the average times for each library and the standard deviations, respectively. All tests were performed on a Dell Precision 7920 with an Intel Xeon processor and 256 GB ram running Ubuntu 18.04 LTS. All programs are written in C++ and can be found on `https://github.com/ancarey/OpenSourceFHE`.

8.1 BGV Test

The BGV scheme is available in the HElib and PALISADE libraries. Unfortunately, since parameter selection in BGV is rather tedious, the tests between HElib and PALISADE are slightly different. For HElib, we were able to get 2760 elements in each vector and performed the calculations of the final velocities for each one. On the other hand, for PALISADE we could only generate vectors with 8 elements due to the complexity in choosing the parameters. In PALISADE for the BGV setting, several additional parameters have to be chosen over the other schemes it implements. The main variables being the big ciphertext modulus, the root of unity used in the ciphertext, and the the big root of unity which is the modulus used for the bit packing operations. There are no helper functions to generate these values, so they have to be hand chosen and fine tuned in order to work correctly. PALISADE is currently in the process of streamlining and improving the process of parameter selection so in the future, the PALISADE implementation could be fine tuned to allow vectors of length 2760. Regardless, based on the other PALISADE experiments, it is safe to assume that the PALISADE implementation of BGV would run faster than HElib's. In addition, we ran the HElib scheme with vectors of length 8 for direct comparison. Figures 8.1, 8.2, and 8.3 show the outputs of the BGV test for

PALISADE, HElib with length 8, and HElib with length 2760, respectively.

8.2 BFV Test

The BFV scheme is available in the SEAL and PALISADE libraries. The time, initial velocity, and acceleration vectors were 2760 elements long and all of the elements were generated randomly. Operations were performed element wise, starting first with the multiplication of the acceleration and time vectors followed by the addition of the initial velocity vector to calculate the final velocity. While the times for parameter generation were fairly close for the two libraries, the times for the remaining sections were vastly different with SEAL always being the fastest. See Figures 8.4 and 8.5 for the outputs of the experiments.

8.3 CKKS Test

The CKKS scheme is available in the SEAL and PALISADE libraries. The time, initial velocity, and acceleration vectors were 2760 elements long and all of the elements were generated randomly. Operations were performed element wise, starting first with the multiplication of the acceleration and time vectors as in the BFV tests. PALISADE was able to do the addition of the initial velocity and `a · t` with no steps in between, but SEAL required a relinearization and re-scaling step between the two. Once again, the times for the two schemes were comparable for the parameter generation, but for all the other parts, SEAL was significantly faster. Figures 8.6 and 8.7 show the outputs of these tests.

```
(base) ancarey@445-7920-02:~/Desktop/PalisadeBGV/build$ ./PalisadeBGV
Initial Velocity
        [ 1 2 3 4 5 6 7 8 ]
Times
        [ 10 14 24 23 18 9 13 7 ]
Acceleration
        [ 1 2 3 2 1 2 1 2 ]
Final Velocity
        ( 11 30 75 50 23 24 20 22 ... )
Times:
Parameter Generation  : 0.001141
Key Generation        : 0.037275
Encryption            : 0.007
Evaluation (v_i + at) : 0.004337
Decryption            : 0.000759
```

Figure 8.1: Final Velocity Calculator implemented with BGV on PALISADE

```
(base) ancarey@445-7920-02:~/Desktop/HElibBGV$ ./HElibBGV
Security: 127.626
Starting the velocity caluculator with 8 instances.

Acceleration:

    [  2,  6, 22, 20, 16, 13,  6,  9 ]

Initial Velocity:

    [ 25, 16,  9, 41,  3, 21, 21, 47 ]

Time:

    [ 22, 14, 26, 22, 28,  7, 18, 29 ]

Final Velocity:

    [ 69,100,581,481,451,112,129,308 ]

Times:
Parameter Generation  : 12.83
Key Generation        : 0.585766
Encryption            : 0.0405823
Evaluation (v_i + at) : 0.0337013
Decryption            : 0.0427523
```

Figure 8.2: BGV on HElib with vectors of length 8

```
(base) ancarey@445-7920-02:~/Desktop/HElibBGV$ ./HElibBGV
Security: 123.892
Number of slots: 2760
Starting the velocity caluculator with 2760 instances.

Acceleration:

    [ 17, 13, 14,  8,  5,  8, 17, 21, 20,  4, 11, 20, 10, 20,  9, 17,  9, 11,  5, 15, ...,  3, 18 ]

Initial Velocity:

    [ 19, 10, 44,  1, 34, 49, 30, 30,  6, 43, 10,  1, 40,  0,  5, 15, 32,  5, 28, 32, ..., 40, 49 ]

Time:

    [ 23, 13, 18, 28,  8, 10, 10, 19,  9,  1, 22, 23,  7, 20, 15, 24, 16, 16, 24, 17, ...,  0, 29 ]

Final Velocity:

    [410,179,296,225, 74,129,200,429,186, 47,252,461,110,400,140,423,176,181,148,287, ..., 40,571 ]

Times:
Parameter Generation  : 11.9937
Key Generation        : 3.53214
Encryption            : 0.189126
Evaluation (v_i + at) : 0.098851
Decryption            : 0.800877
```

Figure 8.3: Final Velocity Calculator implemented with BGV on HElib

```
(base) ancarey@445-7920-02:~/Desktop/SEALBFV$ ./SEALBFV
Starting the velocity caluculator with 2760 instances.

Acceleration:

    [  8, 15, 11, 21, 15,  1, 22, 18,  7, 23,  4,  8, 18, 17, 21, 12, 15,  1,  6, 20, ...,  0,  0 ]
    [  7,  7,  5, 19,  9, 15,  1, 12,  3, 18, 20,  0,  1, 12,  6, 20, 23,  7, 15,  0, ...,  0,  0 ]

Initial Velocity:

    [ 36, 43, 42, 12,  9, 40, 36, 17, 30, 17,  2, 19,  6, 29, 19, 48, 20, 41, 23, 46, ...,  0,  0 ]
    [ 19, 24,  7, 31,  4, 15,  7, 28, 10,  4, 11, 19, 26, 45,  6, 49,  0, 14, 42, 12, ...,  0,  0 ]

Time:

    [ 27, 25,  9,  7, 23,  6, 11,  9,  2, 25, 12, 27,  1,  3, 14, 14, 23, 20,  2,  1, ...,  0,  0 ]
    [ 23,  5,  4,  9, 19, 21, 28, 26,  5, 11, 21, 15,  4, 25,  0, 27,  6, 15, 27,  3, ...,  0,  0 ]

 Final Velocity:

    [252,418,141,159,354, 46,278,179, 44,592, 50,235, 24, 80,313,216,365, 61, 35, 66, ...,  0,  0 ]
    [180, 59, 27,202,175,330, 35,340, 25,202,431, 19, 30,345,  6,589,138,119,447, 12, ...,  0,  0 ]

Times:
Parameter Generation  : 0.051157
Key Generation        : 0.034238
Encryption            : 0.020687
Evaluation (v_i + at) : 0.022442
Decryption            : 0.003062
```

Figure 8.4: Final Velocity Calculator implemented with BFV on SEAL

```
(base) ancarey@445-7920-02:~/Desktop/PalisadeBFV/build$ ./PalisadeBFV
Starting the velocity caluculator with 2760 instances.

Acceleration:

    [ 19, 19, 14, 18, 24,  2, 23, 15,  9, 23,  5,  6, 23,  6, 11,  7,  2, 15, 15, 13, ...,  8, 23 ]

Initial Velocity:

    [ 17, 41, 38, 49, 14, 38, 41, 24,  0, 42, 11, 32, 22, 17, 42, 46, 21, 13, 23, 21, ...,  1, 16 ]

Time:

    [ 14,  5,  3, 24,  4,  5,  1,  7,  3, 22, 19,  1, 11, 20,  6, 16,  8, 12,  2, 16, ..., 23,  8 ]

 Final Velocity:

    [283,136, 80,481,110, 48, 64,129, 27,548,106, 38,275,137,108,158, 37,193, 53,229, ...,185,200 ]

Times:
Parameter Generation  : 0.04588
Key Generation        : 3.48488
Encryption            : 2.06245
Evaluation (v_i + at) : 1.10516
Decryption            : 0.275386
```

Figure 8.5: Final Velocity Calculator implemented with BFV on PALISADE

```
(base) ancarey@445-7920-02:~/Desktop/SEALCkks$ ./SEALCkks
Number of slots: 4096
Starting the velocity caluculator with 2760 instances.

Acceleration:

    [ 21.0047, 19.9610, 8.3806, 13.8492, 9.1196, 22.9049, 3.5401, 6.0722, 3.9170, 2.7202, ..., 11.1652, 14.6
291, 5.3964, 16.8692, 13.6493, 19.1860, 2.1372, 22.1264, 1.3526, 14.2939 ]

Initial Velocity:

    [ 19.7191, 45.5824, 38.4115, 23.8699, 25.6700, 31.7856, 30.3484, 6.8616, 20.0472, 49.9462, ..., 36.9784,
 47.9529, 25.2830, 26.7821, 18.1890, 39.5543, 36.6676, 17.7146, 23.9983, 23.5359 ]

Time:

    [ 23.4930, 5.9265, 8.3332, 18.8661, 28.5669, 21.5189, 0.4890, 24.1253, 3.8937, 6.5477, ..., 11.4578, 6.3
623, 11.1342, 8.5033, 10.3838, 25.9517, 16.5494, 20.7666, 9.9606, 14.1221 ]

 Final Velocity:

    [ 513.1820, 163.8821, 108.2488, 285.1516, 286.1890, 524.6736, 32.0796, 153.3545, 35.2988, 67.7574, ...,
164.9073, 141.0280, 85.3672, 170.2266, 159.9204, 537.4635, 72.0378, 477.2056, 37.4706, 225.3966 ]

Times:
Parameter Generation  : 0.00096
Key Generation        : 0.051048
Encryption            : 0.029484
Evaluation (v_i + at) : 0.006612
Decryption            : 0.000142
```

Figure 8.6: Final Velocity Calculator implemented with CKKS on SEAL

```
(base) ancarey@445-7920-02:~/Desktop/PalisadeCKKS/build$ ./PalisadeCKKS
Starting the velocity caluculator with 2760 instances.

Acceleration:

    [21.0047,19.961,8.38057,13.8492,9.11961,22.9049,3.54006,6.07217,3.91698,2.72022,12.8233,7.40079,12.3396,
19.2839,10.0057,8.81146,1.74388,2.1514,22.2558,0.500576, ...,1.35256,14.2939 ]

Initial Velocity:

    [19.7191,45.5824,38.4115,23.8699,25.67,31.7856,30.3484,6.86158,20.0472,49.9462,41.9556,31.8776,48.6388,2
6.3372,44.5765,40.3862,47.4664,9.61069,17.4446,22.8851, ...,23.9983,23.5359 ]

Time:

    [23.493,5.92654,8.33324,18.8661,28.5669,21.5189,0.489017,24.1253,3.89371,6.54771,18.3792,15.7286,8.7755,
23.0974,8.49944,27.5708,15.7799,19.8968,1.92514,1.89288, ...,9.96055,14.1221 ]

 Final Velocity:

    [513.182,163.882,108.249,285.152,286.189,524.674,32.0796,153.354,35.2988,67.7574,277.638,148.282,156.925
,471.746,129.619,283.325,74.9846,52.4166,60.2902,23.8326, ...,37.4706,225.397 ]

Times:
Parameter Generation  : 0.042857
Key Generation        : 5.21859
Encryption            : 2.61714
Evaluation (v_i + at) : 1.10738
Decryption            : 0.930524
```

Figure 8.7: Final Velocity Calculator implemented with CKKS on PALISADE

Table 8.1: Average time (in seconds) of the final velocity calculations in each of the libraries for BGV, BFV, and CKKS.

Library	BGV (8)						BFV						CKKS					
	Param	Key	Enc	Eval	Dec	Total	Param	Key	Enc	Eval	Dec	Total	Param	Key	Enc	Eval	Dec	Total
HElib	12.2039	3.5480	0.0227	0.0127	0.1028	15.8901	-	-	-	-	-	-	-	-	-	-	-	-
SEAL	-	-	-	-	-	-	0.0503	0.0287	0.0186	0.0211	0.0029	0.1216	0.0010	0.0523	0.0298	0.0067	0.0001	0.0900
PALISADE	0.0011	0.0382	0.0069	0.0043	0.0008	0.0512	0.0467	3.4591	2.0719	1.2164	0.2279	7.0221	0.0428	4.6342	2.5788	1.2508	1.0051	9.5116

Table 8.2: Standard deviation (in seconds) of the final velocity calculations in each of the libraries for BGV, BFV, and CKKS.

Library	BGV (8)						BFV						CKKS					
	Param	Key	Enc	Eval	Dec	Total	Param	Key	Enc	Eval	Dec	Total	Param	Key	Enc	Eval	Dec	Total
HElib	0.1856	0.0884	0.0011	0.0005	0.0023	0.2217	-	-	-	-	-	-	-	-	-	-	-	-
SEAL	-	-	-	-	-	-	0.0013	0.0017	0.0008	0.0005	0.0001	0.0041	0.0006	0.0017	0.0009	0.0002	0.0000	0.0020
PALISADE	0.0001	0.0017	0.0004	0.0003	0.0001	0.0017	0.0017	0.2717	0.0885	0.1527	0.1154	0.3365	0.0016	0.3374	0.2204	0.2162	0.0929	0.4254

9 Conclusion

Fully homomorphic encryption, while still in its development phases, has seen significant growth over the past few years. With the hard work of companies and private researchers, FHE libraries are being developed that enable users without much background knowledge in FHE to be able to reap the powerful benefits it provides when it comes to online secure storage and computation. This work has shown the concepts behind homomorphic encryption, how fully homomorphic schemes are constructed, and the main aspects of three different open-source FHE and lattice libraries. In addition, we implemented the same program, the calculation of a final velocity, for each scheme in each library to show how each of the libraries compares. In the future, we hope to use the knowledge discussed here as a springboard into a graduate thesis on the improvement of current fully homomorphic systems, hoping to one day get FHE as a normalized mode of encryption.

References

[1] M. Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018.

[2] M. Albrecht et al. *Homomorphic Encryption Standard*. `https://projects.csail.mit.edu/HEWorkshop/HomomorphicEncryptionStandard2018.pdf`. 2018.

[3] D. Archer et al. *Applications of Homomorphic Encryption*. Tech. rep. Redmond WA, USA: HomomorphicEncryption.org, July 2017.

[4] F. Armknecht et al. *A Guide to Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2015/1192. `https://eprint.iacr.org/2015/1192`. 2015.

[5] A. A. Badawi et al. *Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Report 2018/589. `https://eprint.iacr.org/2018/589`. 2018.

[6] M. Belland et al. *Somewhat Homomorphic Encryption*. `https://courses.csail.mit.edu/6.857/2017/project/22.pdf`. 2017.

[7] M. Blatt et al. *Optimized Homomorphic Encryption Solution for Secure Genome-Wide Association Studies*. Cryptology ePrint Archive, Report 2019/223. `https://eprint.iacr.org/2019/223`. 2019.

[8] Z. Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP*. Cryptology ePrint Archive, Report 2012/078. `https://eprint.iacr.org/2012/078`. 2012.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. `https://eprint.iacr.org/2011/277`. 2011.

[10] M. Brenner et al. *A Standard API for RLWE-based Homomorphic Encryption*. Tech. rep. Redmond WA, USA: HomomorphicEncryption.org, July 2017.

[11] A. Chatterjee. *Fully homomorphic encryption in real world applications*. 2019.

[12] H. Chen, I. Chillotti, and Y. Song. *Improved Bootstrapping for Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Report 2018/1043. `https://eprint.iacr.org/2018/1043`. 2018.

[13] H. Chen, K. Laine, and R. Player. *Simple Encrypted Arithmetic Library - SEAL v2.1*. Cryptology ePrint Archive, Report 2017/224. `https://eprint.iacr.org/2017/224`. 2017.

[14] J. H. Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Report 2016/421. `https://eprint.iacr.org/2016/421`. 2016.

[15] J. H. Cheon et al. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Report 2016/421. `https://eprint.iacr.org/2016/421`. 2016.

[16] J. H. Cheon et al. *A Full RNS Variant of Approximate Homomorphic Encryption*. Cryptology ePrint Archive, Report 2018/931. https://eprint.iacr.org/2018/931. 2018.

[17] I. Chillotti et al. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*. Ed. by J. H. Cheon and T. Takagi. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 3–33. DOI: 10.1007/978-3-662-53887-6_1. URL: https://doi.org/10.1007/978-3-662-53887-6%5C_1.

[18] I. Chillotti et al. *Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds*. Cryptology ePrint Archive, Report 2016/870. https://eprint.iacr.org/2016/870. 2016.

[19] A. Costache and N. P. Smart. *Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?* Cryptology ePrint Archive, Report 2015/889. https://eprint.iacr.org/2015/889. 2015.

[20] M. van Dijk et al. *Fully Homomorphic Encryption over the Integers*. Cryptology ePrint Archive, Report 2009/616. https://eprint.iacr.org/2009/616. 2009.

[21] L. Ducas and D. Micciancio. *FHEW: Bootstrapping Homomorphic Encryption in less than a second*. Cryptology ePrint Archive, Report 2014/816. https://eprint.iacr.org/2014/816. 2014.

[22] J. Fan and F. Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144. 2012.

[23] C. Gentry. *Computing Arbitrary Functions of Encrypted Data*. https://crypto.stanford.edu/craig/easy-fhe.pdf. 2008.

[24] C. Gentry. "A Fully Homomorphic Encryption Scheme". PhD thesis. Stanford, CA, USA, 2009. ISBN: 9781109444506.

[25] C. Gentry. "Fully homomorphic encryption using ideal lattices". In: *In Proc. STOC*. 2009, pp. 169–178.

[26] C. Gentry. *Winter School on Cryptography: Fully Homomorphic Encryption*. https://www.youtube.com/watch?v=Y1TxCiOuoYY. 2012.

[27] C. Gentry. *How would you explain homomorphic encryption?* https://www.youtube.com/watch?v=pXb39wj5ShI. 2018.

[28] C. Gentry and S. Halevi. *Implementing Gentry's Fully-Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Report 2010/520. https://eprint.iacr.org/2010/520. 2010.

[29] C. Gentry, A. Sahai, and B. Waters. *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based*. Cryptology ePrint Archive, Report 2013/340. https://eprint.iacr.org/2013/340. 2013.

[30] S. Halevi. *Homomorphic Encryption (Part I): SWHE*. https://www.youtube.com/watch?v=xlcb_G1_rzk. 2015.

[31] S. Halevi. *Homomorphic Encryption*. https://shaih.github.io/pubs/he-chapter.pdf. 2017.

[32] S. Halevi and V. Shoup. "Design and Implementation of a Homomorphic-Encryption Library". In: 2013.

[33] S. Halevi and V. Shoup. *Algorithms in HElib*. Cryptology ePrint Archive, Report 2014/106. https://eprint.iacr.org/2014/106. 2014.

[34] S. Halevi and V. Shoup. *Bootstrapping for HElib*. Cryptology ePrint Archive, Report 2014/873. https://eprint.iacr.org/2014/873. 2014.

[35] S. Hardy. *A Homomorphic Encryption Illustrated Primer*. 2018. URL: https://blog.n1analytics.com/homomorphic-encryption-illustrated-primer/ (visited on 05/09/2020).

[36] *HElib (release 1.0.0)*. https://github.com/homenc/HElib. Jan. 2020.

[37] HomomoephicEncryption.org. *Building Applications with Homomorphic Encryption*. http://homomorphicencryption.org/wp-content/uploads/2018/10/CCS-HE-Tutorial-Slides.pdf. 2018.

[38] HomomorphicEncryption.org. *Homomorphic Encryption Standardization*. 2020. URL: https://homomorphicencryption.org/introduction/ (visited on 05/09/2020).

[39] N. Koblitz et al. *Algebraic Aspects of Cryptography*. Berlin, Heidelberg: Springer-Verlag, 1998. ISBN: 3540634460.

[40] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan. *On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2013/094. https://eprint.iacr.org/2013/094. 2013.

[41] *PALISADE Lattice Cryptography Library (release 1.7.4)*. https://palisade-crypto.org/. Jan. 2020.

[42] P. V. Parmar et al. "Survey of Various Homomorphic Encryption algorithms and Schemes". In: 2014.

[43] Y. Polyakov et al. *PALISADE Lattice Cryptography Library User Manual (v1.7.c)*. https://gitlab.com/palisade/palisade-development/blob/release-v1.7.c/doc/palisade_manual.pdf. 2016.

[44] O. Regev. *The Learning with Errors Problem*. https://cims.nyu.edu/~regev/papers/lwesurvey.pdf. 2017.

[45] M. Research. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. https://www.youtube.com/watch?v=brAXXghiqM0. 2017.

[46] M. Research. *Intro to Homomorphic Encryption*. https://www.youtube.com/watch?v=SEBdYXxijSo. 2019.

[47] M. Research. *Introduction to CKKS (Approximate Homomorphic Encryption)*. https://www.youtube.com/watch?v=iQlgeL64vfo. 2020.

[48] RightScale. *RightScale 2018 State of the Cloud Report*. https://www.suse.com/media/report/rightscale_2018_state_of_the_cloud_report.pdf. 2018.

[49] R. L. Rivest, L. Adleman, and M. L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation, Academia Press* (1978), pp. 169–179.

[50] V. F. Rocha and J. López. *An Overview on Homomorphic Encryption Algorithms*. https://www.ic.unicamp.br/~reltech/PFG/2018/PFG-18-28.pdf. 2018.

[51] R. Rothblum. "Homomorphic Encryption: From Private-Key to Public-Key". In: *Theory of Cryptography*. Ed. by Y. Ishai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 219–234. ISBN: 978-3-642-19571-6.

[52] *Microsoft SEAL (release 3.4)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Oct. 2019.

[53] N. Smart and F. Vercauteren. *Fully Homomorphic SIMD Operations*. Cryptology ePrint Archive, Report 2011/133. https://eprint.iacr.org/2011/133. 2011.

[54] P. Thaine. *Homomorphic Encryption for Beginners: A Practical Guide (Part 1)*. 2018. URL: https://medium.com/privacy-preserving-natural-language-processing/homomorphic-encryption-for-beginners-a-practical-guide-part-1-b8f26d03a98a (visited on 05/09/2020).

[55] TheIACR. *Algorithms in HElib*. https://www.youtube.com/watch?v=afmbgkViuqw. 2014.

A Lattices

An n-dimensional lattice $\mathcal{L}$ is the set of all linear combinations of n linearly independent vectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$.

$$\mathcal{L} = \{a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_n\mathbf{v}_n | a_i \in \mathbb{Z}\}$$

In other words, lattices are mathematical structures that consist of points in an n-dimensional space, with some periodic structure [serious crypto]. For example, consider a lattice with $n = 2$:

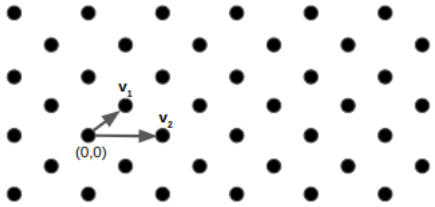

Figure 1: An example of a 2 dimensional lattice

The linearly independent vectors $\mathbf{v}_1, ..., \mathbf{v}_n$ that make up the lattice are called the basis $\mathbf{B}$ of the lattice. Generally the basis vectors are organized into an $n \times n$ matrix where each vector becomes a column:

$$\mathbf{B} = \begin{pmatrix} \mathbf{v}_{1_0} & \mathbf{v}_{2_0} & \cdots & \mathbf{v}_{n_0} \\ \mathbf{v}_{1_1} & \mathbf{v}_{2_1} & \cdots & \mathbf{v}_{n_1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_{1_{n-1}} & \mathbf{v}_{2_{n-1}} & \cdots & \mathbf{v}_{n_{n-1}} \end{pmatrix}$$

The basis in matrix form can be used to represent the lattice in the following way:

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) = \sum_{i=1}^{n} a_i\mathbf{v}_i : a_i \text{ are integers}$$

Manipulating these matrices is the core of lattice-based cryptography.

A.1 Ideal Lattices

An ideal is a subset I or a ring R that is:

- Closed under addition: $\forall i_1, i_2 \in I$, $i_i + i_2 \in I$
- Closed under multiplication with R: $\forall i \in I$, $\forall r \in R$, $i \cdot r \in I$

An ideal lattice is simply a lattice with some additional algebraic structure. The main difference between a lattice and an ideal lattice is that while normal lattices are groups, ideal lattices are ideals.

B LWE : Learning with Errors

In 2005, Regev [regev 2005] introduced the Learning with Errors (LWE) problem and showed that solving LWE for the average case is as hard as solving several standard lattice problems in the worst case. This allows cryptographic constructions that are based on LWE to be secure under the idea that worst-case lattice problems are hard.

The main idea behind the LWE problem is trying to find a secret $\mathbf{s}$ given some set of "noisy" linear equations of the form $\mathbf{b} = \mathbf{as} + e$. If the error term e did not exist, finding $\mathbf{s}$ would be as simple as performing Gaussian elimination. The introduction of the error into the equation makes the problem significantly more difficult.

The error generally comes a Gaussian distribution that is rounded to the nearest integer and then reduced modulo q, where q can be an integer or a polynomial depending on if plain LWE or ring LWE is being being implemented.

Definition B.1 (LWE Distribution)**.** For positive integers n and q, an error distribution χ taken as a discrete Gaussian distribution, and a vector $\mathbf{s}$ that is taken as an n-dimensional integer vector modulo q (i.e. $s \in \mathbb{Z}_q^n$), the LWE distribution $A_{s,\chi}$ generates a sample $(\mathbf{a}, \mathbf{b})$ by choosing $\mathbf{a} \in \mathbb{Z}_q^n$ uniformly at random, choosing $e \leftarrow \chi$, and computing $\mathbf{b} = \langle \mathbf{sa} \rangle + e$.

B.1 RLWE: Ring Learning with Errors

Ring Learning with Errors (RLWE) was introduced by Lyubaskevsky, Peikert, and Regev in 2013 [lattice LWE ring]. It is simply a ring based version on the LWE problem explained above.

Definition B.2 (RLWE (lattice lwe ring))**.** For a security parameter λ, let $f(x)$ be a cyclotomic polynomial $\Phi_m(x)$ with $\deg(f) = \rho(m)$ depending on λ and set $R = \mathbb{Z}[x]/(f(x))$. Let $q = q(\lambda) \geq 2$ be an integer. For a random element $\mathbf{s} \in R_q$ and a distribution $\chi = \chi(\lambda)$ over R, denote with $A_{\mathbf{s},\chi}^{(q)}$ the distribution obtained by choosing a uniformly random element $\mathbf{a} \leftarrow R_q$ and a noise term $\mathbf{e} \leftarrow \chi$ and outputting $(\mathbf{a}, [\mathbf{a} \cdot \mathbf{s} + \mathbf{e}]_q)$.