University of Bergen
Department of Informatics

TRUST NO ONE;
Homomorphic Encryption and its Applications

Author: Knut Mathias Gaard Storvestre
Supervisor: Chunlei Li

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

February, 2023

Abstract

In my master thesis, I describe cryptographic processes. Moreover, I suggest to implement the homomorphic encryption scheme for approximate arithmetic in Java. In addition, I provide a graphical user interface (GUI). I develop a more user-friendly and low-threshold alternative than what is commonly used today. I contribute to existing literature, as well as practice, by implementing a library based on the cryptographic scheme, CKKS, in Java. This scheme protects digital communication against the power of quantum computers. The source code is available at https://github.com/KnutStorvestre/CKKS.

Keywords: Homomorphic Encryption, Approximate arithmetic, Implementation of CKKS scheme, Java

Acknowledgements

I would like to thank Chunlei Li, my thesis advisor. Completing my master's thesis without him would have been much harder. Thanks also to valuable help from Anya Helene Bagge, my supervisor for the bachelor thesis. This laid the foundations for my master's thesis. I would also like to thank my mom, dad, sister, and brother for encouraging me throughout all my ups and downs on working with my thesis.

`Knut Storvestre`
Wednesday 1st February, 2023

Motivation

My thesis addresses a question of topical interest: homomorphic encryption. I have explored this forward-looking and important area of research and especially the CKKS scheme. It was developed by the scientists Cheon, Kim, Kim and Song, thus the name CKKS. CKKS proposes a cryptographic scheme that enables approximate computations on encrypted data. My motivation has been to move beyond describing encryption. I want to contribute by implementing the scheme, not in C++ as is common now, but in an easily understandable language. Choosing to implement the scheme in a low threshold language, Java, lowers the bar and enable more people to use CKKS and homomorphic encryption.

Contents

List of Figures

Chapter 1

Introduction

Fully Homomorphic Encryption (FHE) has become increasingly popular in the last years due to the reliance on cloud computing. Cloud computing is when a professional provider offers computer system resources, in particular data storage and computing power. Examples of providers include Amazon, Microsoft and Google. They have distributed data centers allowing customers to share facilities, thus reducing their capital expenses. This explains cloud computing's popularity. The popularity of 3rd party solutions entails an increasing concern about privacy. FHE is important because it allows you to use cloud computing without compromising your privacy.

The area of research within cryptography is dynamic, and new developments come all the time. One of the latest developments in FHE is a public key crypto system, the CKKS scheme. This scheme is named after the scientists Cheon, Kim, Kim, and Song who introduced the CKKS scheme in 2016 [5]. A scheme is defined as a large-scale systematic plan, or arrangement, for attaining a particular object or putting a particular idea into effect.

This thesis contributes to research by implementing the cryptographic scheme, CKKS, in a novel language, Java. Originally CKKS was implemented in C++, in an open source library called Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN) [2]. This library was released in 2016, and is still under improvement. A library in computer science is a collection of code that can be added to another code to provide new functionality. C++ has an advantage of speed. However, a disadvantage of C++ is that it is considered as an advanced language for advanced users. In my master thesis, I initially programmed the scheme in Python. Python is an easy-to-use programming

language but, is significantly slower compared to C++. Later, when I discovered the advantages of Java, this became a dominant alternative. Subsequently, the scheme was re-programmed in Java. Java has several benefits. Firstly, it is easy to understand. Secondly, it is extensively used on the backend. The backend is where the processes and operations are taking place. This is "behind the scenes" where the cryptographic data is processed. Lastly, Java is about three times faster than a commonly used language, Python. In sum, applying Java simplifies the code and speed up the processing time. However, despite benefits, the CKKS scheme has not yet been implemented in Java. In this thesis I contribute to do so. In addition, I have developed a Graphical User Interphase (GUI). This greatly improves the user-friendliness of the scheme and lowers the threshold for the user. The low threshold opens this cryptographic scheme to users with limited knowledge of Java.

To conclude, I contribute to research by applying a novel approach of how we can defend ourselves in the coming era of quantum computers. In this master thesis I first describe different types of encryption. Second, I discuss preliminaries. Third, I describe the CKKS scheme. Lastly, I describe and discuss implementation and end with discussion and conclusion.

Chapter 2

Types of encryption

First of all, I would like to introduce some central concepts in this thesis. Cryptography comes from ancient Greek and means "secret text". The field of cryptography is based on some core criteria: data confidentiality, data integrity, authentication and non-reputation. This entails that cryptographic algorithms should follow these principles or be set up an in an environment where they are not vulnerable to exploitation.

- *Confidentiality*: Keeps the data secret from attackers
- *Integrity*: Detects unauthorized changes to ciphertext
- *Authentication*: Allows the receiver to know that the message comes from a trusted user
- *Non-reputation*: Secures that the authorship of a message in non-disputable.

A common misconception is that we should only use cryptographic algorithms that are completely impenetrable. Modern ciphers are designed to achieve sounding computational security, indicating that it is intractable to develop an attack against the ciphers with complexity less than the brute-force attack, which exhausts all possible keys for the target ciphers. Almost all cryptographic algorithms in use today are in theory vulnerable to brute force attacks. Creating a cipher that is not vulnerable to brute force attacks is generally considered to be an impossible task, except for the One Time Pad (OTP). However, it has a significant shortcoming which I will discuss in Subsection 2.1.1. The thinking behind many cryptographic algorithms is not that it should be impenetrable. Even tough many of the ciphers that are commonly used today are vulnerable in theory, they are generally secure in practice.

2.1 Symmetric encryption(SE)

The pioneer case of cryptography is symmetric cryptography. Symmetric encryption is when you use the same cryptographic key to encrypt and decrypt a message. Symmetric encryption can be categorised into stream- and block-ciphers.

2.1.1 Stream Ciphers

Stream ciphers encrypt data in a continuous stream while the block ciphers encrypts data block by block. Stream ciphers are generally faster than block ciphers in hardware implementations, but also generally less complex and secure.

An early recorded example of symmetric encryption is the Caesar cipher [17]. This stream cipher encrypts by moving each letter of the message a fixed number of positions forward in the alphabet. The receiver must move each of the letters the same number of positions back in the alphabet to decrypt. The number of positions that the user has to move each letter back in the alphabet is known as the private key.

Example. If you want to encrypt the message "attack at morning" it will become "nggnpx ng zbeavat" if you use the private key of 13.

Today's stream ciphers typically use bits instead of letters. The stream ciphers operates on two bit streams: One stream for the data we want to encrypt called the **plaintext stream** and, a stream for the key we encrypt the data with called the **key stream**.

$$\text{plaintext stream } X = x_1, x_2, \ldots, x_n$$
$$\text{key stream } K = k_1, k_2, \ldots, k_n$$

Definition 2.1.1 (Stream cipher encryption and decryption) *The plaintext, the ciphertext and the key stream consists of individual bits, i.e.,* $x_i, y_i, k_i \in \{0, 1\}$ *[24].*

$$y_i = encryption_{k_i}(x_i) \equiv x_i + k_i \pmod 2$$
$$x_i = encryption_{k_i}(y_i) \equiv y_i + k_i \pmod 2$$

We add the secret key bit to the encryption, and subtract the secret key bit from the decryption. This is possible since we are using (mod 2) which means that adding $2k_i \pmod 2 \equiv 0$ where subtraction and addition is identical.

We differentiate the different stream ciphers from the way they generate the key streams.

One-Time Pad(OTP)
OTP is the only stream cipher that satisfies the confidentiality criteria and keeps the data secret from attackers. OTP is unconditionally secure. It can not be broken even tough the attacker has unlimited computational resources and time. This stream cipher achieves this by generating a key stream that is completely random and where each of the key stream bits is only used once. One major drawback is that the key has to have the same length as the message. If you want to encrypt a film of 2 gigabits you have to use a cryptographic key of 2 gigabits. OTP is rarely used today because of its extensive use of memory.

Linear Feedback Shift Registers(LFSR)
LFSRs uses a significantly less memory than OTP to encrypt messages. The LFSRs are known to be simple in structure, but able to produce a complicated output. LFSRs have a huge variety of different use cases from encryption in mobile telephone networks to sudo random number generator.

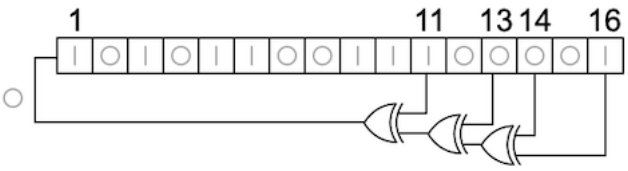

Figure 2.1: Simple LFSR

The LFSR generates a key stream bit by pushing each bit one space to the right. The rightmost bit will be removed and added to the key stream. The bit will also be XORed with multiple other bits in the LFSR and then put in the first slot. 2.1 The rightmost bit which is the bit at slot 16 will become the first bit of the key stream. This bit will also be xor'ed with the bit at slot 14, 13 and 11 before being put in slot 1. If both the sender and the receiver share the initial state of the LFSR the sender and the receiver is able to generate the same key stream.

An example of an LFSR cipher used in practice is the A5/1 cipher. This cipher was a part of the Global System for Mobile communications (GSM). This cellular telephone standard

describes the protocols used in 2G and was developed by European Telecommunications Standards Institute(ETSI). 2G was first commercially launched in December 1991 in Finland [14].

2.1.2 Block Ciphers

Block ciphers encrypts multiple bits at once.

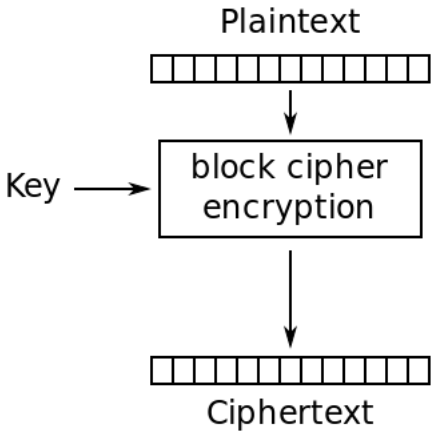

Figure 2.2: Block cipher example

There are many block cipher encryption algorithms. I will cover AES.

AES (Advanced Encryption Standard)
AES is a block cipher designed by the Belgian cryptographers Vincent Rijmen, and Joan Daemen [7] in 1998.

The AES algorithm supports three different choices of key bits:

- AES-128
- AES-192
- AES-256

Each of the numbers represents the bit length of the encryption key. The U.S. government uses AES-192 or AES-256 to store top secret information. AES-192 and AES-256 are therefore widely assumed that to be secure. AES operates on blocks of plaintext data that is 128 bits long. If the block is smaller than 128 bits the remaining bits will be filled

with padding. AES is widely used in a variety of security protocols including Transport Layer Security (TLS)/Secure Sockets Layer (SSL), Secure Shell (SSH) and many Wi-Fi encryption standards networks.

Mode of operation

Mode of operation is an algorithm that describes how to repeatedly apply block cipher algorithms such as AES to transform a message into ciphertext, while at the same time use the same cryptographic key. Different modes of operation can be used to improve the security of a cipher. It becomes more difficult for an attacker to guess the key used to encrypt the message. Even tough we use AES, the encrypted data is not guaranteed to be secure if we use an unsecure mode of operations. Some common modes of operation include electronic codebook (ECB) mode and cipher block chaining (CBC).

Electronic codebook (ECB) mode

This is the simplest mode of operation. ECB is when you only use the same key and the same block cipher encryption each time you encrypt a message.

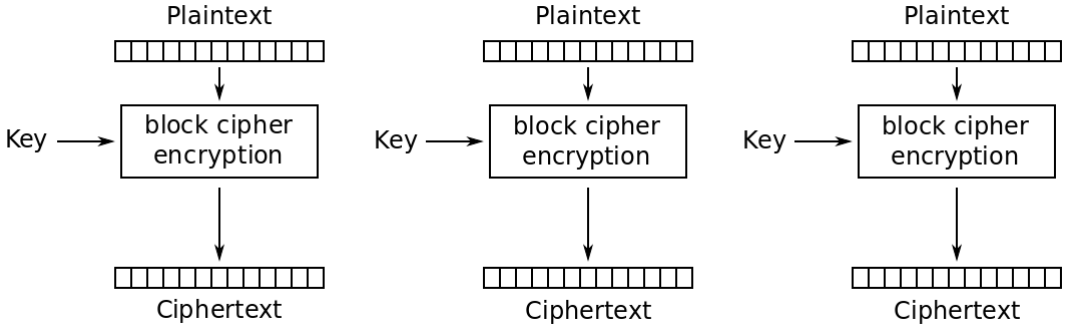

Figure 2.3: Electronic Codebook mode

Even tough we use AES-256 as a block cipher encryption, the encrypted data can still in some cases be recovered by an attacker. An example of this is encrypting a penguin image using ECB. This example is illustrated in Figure 2.4.

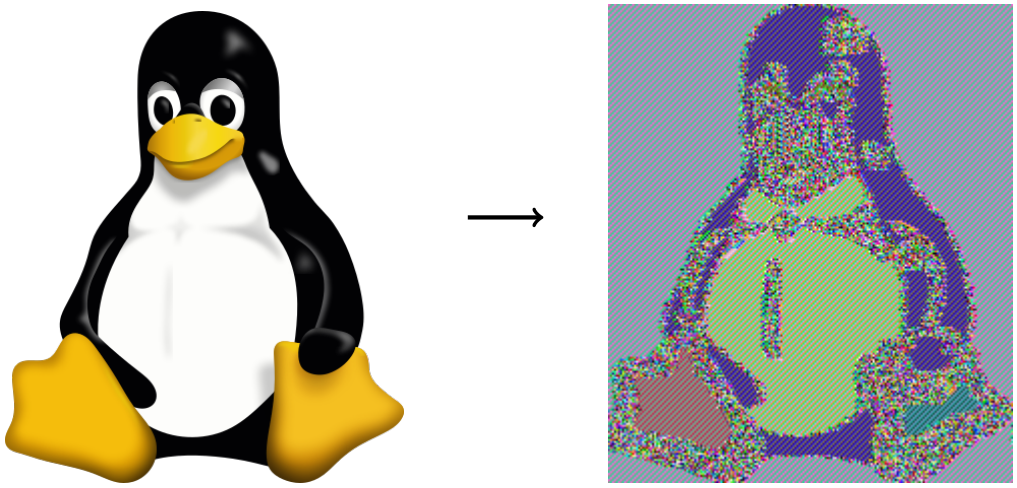

Figure 2.4: Encrypted penguin with Electronic Codebook mode

You can still recognise that the pattern is a penguin because the same plaintext is always mapped to the same ciphertext. The attacker in this case does not have to do anything since the human eye can recognise the information.

Cipher block chaining (CBC) mode

ECB tries to mitigate this problem by XORing the first plaintext block to be encrypted with an Initialization Vector (IV). This vector has the same length as the plaintext block. It contains randomly generated bits. All the next blocks will be XORed with the ciphertext of the previous block. The CBC algorithm is visualized in Figure 2.5.

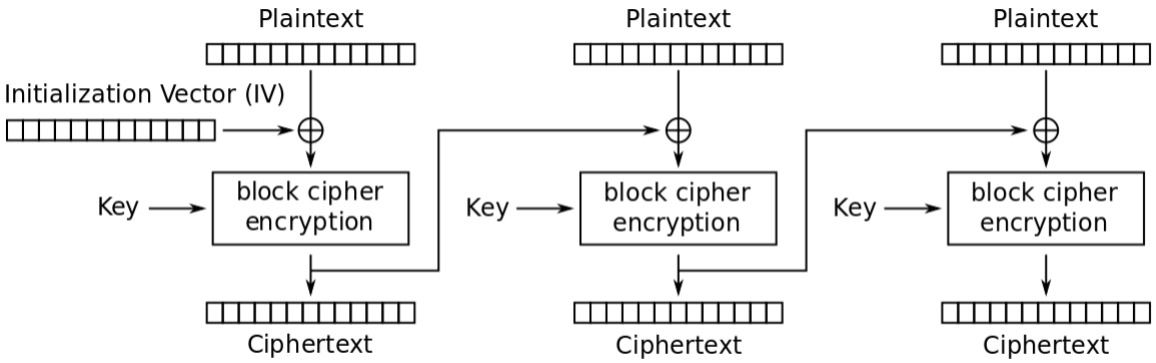

Figure 2.5: Cipher Block Chaining mode

As you can see in Figure 2.6 CBC does not have the pattern of a penguin.

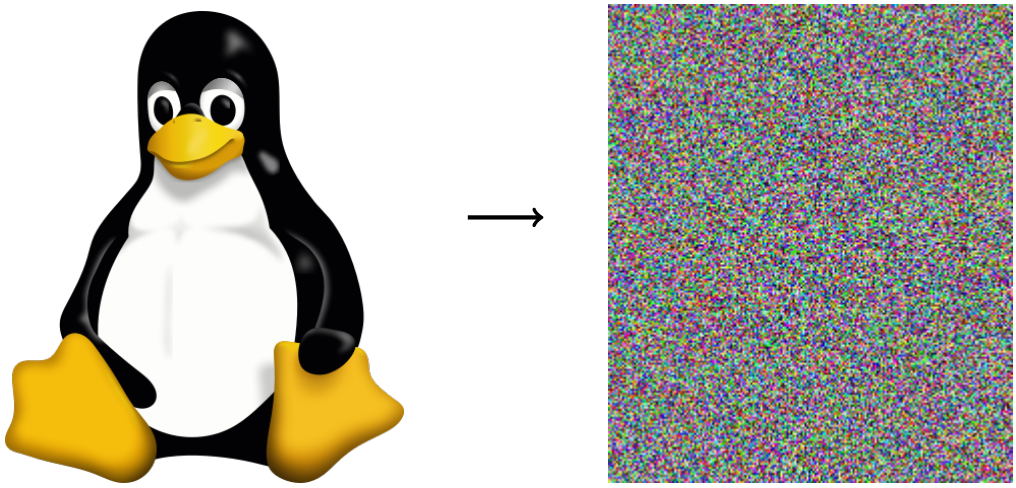

Figure 2.6: Encrypted penguin with Cipher Block Chaining mode

CBC can be vulnerable to the same pattern recognition if the attacker uses brute force to test all different types of IV's. The security of CBC is therefore reliant on the length of IV. CBC is also more vulnerable to noise interference. If one bit is flipped before the decryption all the other blocks that are chained after the block will be affected. There are other more advanced modes of operations like Counter mode which is able to generate a different output given the same plaintext without chaining. Even tough CBC has some shortcomings, they are by far outweighed by its advantages. CBC is used today in the Transport Layer Security (TLS) protocol which provides privacy while communicating over the Internet.

Many of the different modes of operation are used in conjunction with each other to complement each other's weaknesses in many of the modern applications.

2.2 Asymmetric encryption

A big problem when it comes to symmetric encryption is the difficulties with exchanging the private key in a secure way. Asymmetric encryption solves this problem by using two keys, a public key used for encryption and a private key used for decryption. You can therefore publish the public key. If anyone wants to send you a secret message, they can safely do so by encrypting their message with the public key before sending it.

There are many cases where asymmetric and symmetric encryption work hand in hand. We can use asymmetric cryptography to transfer symmetric keys. Asymmetric encryption was first developed in 1973 by the English mathematician Clifford Cocks at the Government Communications Headquarters (GCHQ). The system was secret and classified until 1997.

2.2.1 RSA encryption

In the meantime, the computer scientists Rivest, Shamir, and Adleman developed the "RSA algorithm" to perform public key cryptography. It was first described in 1977 and later published in the paper "A method for obtaining digital signatures and public-key cryptosystems" in 1978 [26]. RSA is based on the hardness of factorizing big integers. The RSA algorithm is described below.

RSA key generation:

1. Choose two prime numbers p and q
2. Calculate $n = p \cdot q$ and $\phi(n) = (p-1)(q-1)$
3. Choose the discrete logarithm integer e that is $1 < e < z$
4. Calculate $d = e^{-1} \pmod{z}$
5. You now have two keys.
    - **public key**: (n, e)
    - **private key**: (n, d)

RSA encryption:

$$\mathrm{c} = m^e \pmod{n}$$

RSA decryption:

$$\mathrm{m} = c^d \pmod{n}$$

However, the integer factorization problem might be vulnerable to quantum computers. It is therefore important to find new mathematical problems that can replace it. I will return to this topic later.

2.2.2 Diffie-Hellman key exchange

Asymmetric cryptography is also known as public-key cryptography. Most modern crypto systems are based on public key cryptography. It was first presented by Hellman, Merkle, and Diffie at Stanford University in 1976 [27] in a paper called "New Directions in cryptography" [9] where they presented the Diffie-Hellman key exchange.

This key-exchange scheme allows two parties, that have no prior knowledge of each other, to establish a shared secret key over an insecure channel. This key exchange is based on a hard mathematical problem called the "discrete logarithm problem". This problem is also a part of the NP-Complete group. An example of this can be that two persons, Alice and Bob, want to share a common secret.

1. Alice and Bob agree on two integer values a modulus value **p** and a generator **g**. Note that **g** is a generator of $\mathbb{Z}_p^*$ if for every $a \in \mathbb{Z}_p^*$ we have $g^k \equiv a \pmod{p}$ for some k.
2. Alice chooses a secret integer **a** and sends Bob $\mathbf{A} \equiv g^a \pmod{p}$. The secret can also be referred to as the discrete logarithm of **A** with respect to the base **g**.
3. Bob chooses a secret integer **b** and sends Bob $\mathbf{B} \equiv g^a \pmod{p}$.
4. Alice computes the secret $\mathbf{s} \equiv A^b \pmod{p}$.

We know that Alice and Bob has the same secret because

$$A^b \pmod{p} \equiv g^{ab} \pmod{p} \equiv g^{ba} \pmod{p} \equiv B^a \pmod{p}$$

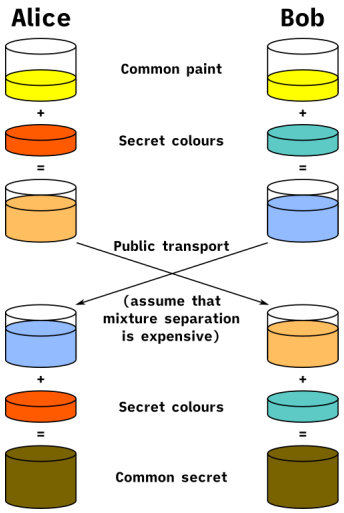

Figure 2.7: Overview of Diffie-Hellman key exchange [28]

In Figure 2.7 provides an illustration of how the Diffie-Hellman key exchange works. The common paint represents the variables **g** and **p**. The secret colors represents the variables **a** for Alice and **b** for Bob. Even tough the color generated from the mixture of the common and secret paint is public knowledge, it is hard for the attacker to separate the secret colors from the common paint. The hardness of this is known as the "Discrete-Logarithm Problem", which is a part of the NP-Complete group. The NP-Complete group will be discussed in more detail in Section 3.2.

2.3 Homomorphic encryption(HE)

The notion of homomorphic encryption has a close connection with homomorphism in algebra. In algebra, a mapping φ from one algebraic structure $\mathcal{A}$ to another one $\mathcal{B}$ is called homomorphic if φ preserves the operation(s) from $\mathcal{A}$ to $\mathcal{B}$. For example, if $\mathcal{A}$ supports an operation $\circ$ and $\mathcal{B}$ supports $\odot$, then $\varphi(x \circ y) = \varphi(x) \odot \varphi(y)$. Homomorphic encryption schemes have the addition feature of supporting homomorphic addition and/or multiplication from the plaintext space to ciphertext space.

Homomorphic encryption allows a third party to perform computations on encrypted data. The encrypted result of the data will be available to whoever has the decryption key for the original data. However, the third party cannot decrypt it.

Analogously, if we consider traditional encryption as locking your jewellery in a box, then homomorphic encryption can be considered as locking your jewellery in a glove box that only you have the key for, where the glove box can be in a jewellery store where the goldsmith can perform changes on the jewellery, but still he cannot steal it.

When the idea of homomorphic encryption was first presented by Rivest et al. in 1978 [25], they discovered that RSA had a multiplicative homomorphism. To explain this idea we can imagine two values **x** and **y**. We encrypt the **x** and **y** with value **e**. Then, encrypt each of the values by putting **e** as the exponent as you can recall from the Subsection 2.2.1.

$$encrypt(x) \equiv x^e \pmod{n}$$
$$encrypt(y) \equiv y^e \pmod{n}$$

The multiplicative homomorphism is that you can multiply each of the encrypted values and get their product when you decrypt them.

$$encrypt(x) \cdot encrypt(y) \pmod{n} \equiv (xy)^e \pmod{n} \equiv encrypt(xy)$$

Designing a crypto system that has both additive and multiplicative homomorphism is much harder. Solving this problem was for long considered the holy grail of cryptography. A breakthrough occurred 30 years after the idea first was presented Craig Gentry published the first ever fully homomorphic encryption scheme in his PhD thesis in 2009 [10].

Craig Gentry and Shai Halevi later implemented the scheme and published their work in a proceedings in 2011 [11]. He showed how the scheme's security and performance work in practice. They made a rough estimate, and found that the security parameter λ, which is the same as the dimensions of the lattice, should be at least 2^{13} to 2^{15} to be considered secure. A lattice is defined in Section 3.3. They ran the implementation on a powerful IBM System x3500 server, featuring a 64-bit quad-core Intel Xeon E5450 processor, running at 3GHz, with 12MB L2 cache and 24GB of RAM. The implementation used 2.2 hours to generate the cryptographic keys and 31 minutes to recrypt, which is an operation to reduce the noise by treating decryption as an evaluation process. Gentry admitted that the scheme was atrociously slow. This scheme would not be usable in a long time, even with the help of Moore's law. Since then, the speed of the schemes for FHE has increased around 8 times each year, but performance is still a major obstacle.

2.4 Types of homomorphic encryption

There are multiple levels of homomorphic encryption. The levels are separated by the types and amount of operations you can perform on the encrypted data.

2.4.1 Partially homomorphic encryption

This only works for one type of abstract algebraic operation: Either addition **encrypt(a+ b)** or multiplication **encrypt(a · b)** on encrypted data. RSA [26] is an example of a somewhat homomorphic encryption.

2.4.2 Leveled fully homomorphic encryption

This allows both types of algebraic operations: Addition and multiplication on encrypted data, although it allows only for a limited number of operations. The advantage of using leveled homomorphic encryption is that it is fast when you know how many operations you are going to perform. By performing mathematical operations the noise increases. The noise slowly corrupts the data making it less and less accurate, until the data is unusable.

2.4.3 Fully homomorphic encryption (FHE)

FHE allows for an unlimited amount of arithmetic operations on encrypted data. It is the most powerful form of homomorphic encryption. CKKS is not FHE, but approximate FHE. This will be explained in chapter 4. A problem in these encryption processes is noise. Noise can be removed with demanding and expensive methods like bootstrapping.

2.4.4 Areas of Applications

FHE is an increasingly good method for computation on sensitive genomic data. FHE can be applied to the evaluation of various algorithms like machine learning on encrypted financial, medical, or genomic data [15][20][16][21]

Chapter 3

Preliminaries

A good crypto system should be easy to construct and implement, but hard to crack. This can be compared to building a chest for keeping secrets. It is a delicate tradeoff between how expensive the chest is to build and its security level against intruders. The era of quantum computers has not yet arrived. However, quantum algorithms capable of solving number-theoretic hard problems, such as integer factorization and discrete logarithm, in polynomial complexity have already been developed. This entails that the widely used RSA cryptosystem, "DH key exchange" scheme is vulnerable when quantum computers are brought from theory to reality. To prevent successful attacks, we must therefore develop new and resilient mathematical problems.

3.1 Basic notation

We define $\lfloor \cdot \rceil$ as rounding to the closest integer, if the value is exactly between two integers we round upwards. We use $\mathbb{Z}_q$ to define a set of all integers modulo q. We use $\langle \cdot, \cdot \rangle$ to denote the dot product of two vectors. We use $\Omega(\cdot)$ as a symbol for the average case time complexity that the fastest known algorithm use to solve a problem. We use $\mathcal{O}(\cdot)$ as the worst case time complexity an algorithm uses give all possible inputs.

3.2 Problems complexity

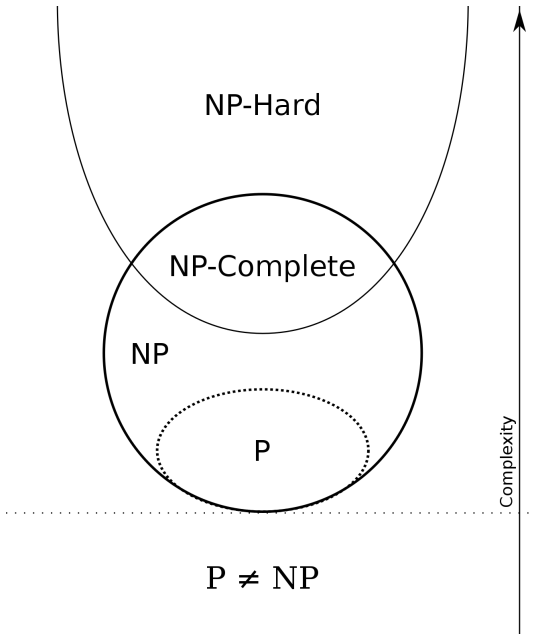

Figure 3.1: Classifications of problems

This image shows the different classifications of problems ranging from the least complex problems at bottom to the most complex problems at the top. This image is based on the popular assumption that P≠NP.

P

P is a set of problems that are proven to be solvable in polynomial time.

NP

NP stands for non-deterministic polynomial time. Solutions in NP can be verified in polynomial time. However, we do not necessary know if we can find a solution in polynomial time.

NP-Complete

If a problem in NP is not solvable in polynomial time, then it is probably part of NP-Complete. All these problems can be reduced to each other in polynomial time. This means that if you are able to solve one of them in polynomial time, you can solve all of them in polynomial time.

NP-Hard
The class of problems π such that all problems in NP can be reduced to π. While π may not necessarily be in NP itself, it is the class of NP-hard problems.

Both of the following conditions have to be true for a problem to be NP-Complete.

1. NP-Hard π: all of NP can be reduced to π.
2. $\pi \in$NP: Solutions to π can be verified in polynomial time.

In my thesis, when I am discussing NP-Hard, I will exclude NP-Complete problems for simplicity.

Cryptosystems
If you base your cryptosystem on a very complex problem, it becomes resilient against attacks. By properly using problems that are NP-Hard we could design a resilient crypto system. When it comes to cryptography, it is important that the problem we use is Ω(NP-Hard).

3.3 Lattice Theory

A lattice is a simply a vector space over $\mathbb{R}$. Each of the points in the lattice is a linear combination over $\mathbb{R}$ of the basis vectors.

Definition 3.3.1 (Lattice) *Given $\boldsymbol{n}$ linear independent vectors $B = (b_1, b_2, ..., b_n) \in \mathbb{R}^n$. The lattice generated by the vectors is defined as*

$$\mathcal{L}(B) = \mathcal{L}(\vec{b_1}, \vec{b_2}, ..., \vec{b_n}) = \{\Sigma x_i b_i | x_i \in \mathbb{Z}\}.$$

Here we refer to B as the basis of the lattice. We can take the scaled sum of the vectors and create other vectors that are also in the lattice.

3.4 Shortest Vector Problem(SVP)

Lattice contains many problems that are NP-hard. The Shortest Vector Problem (SVP) which is defined in 3.4.1 is proven to be NP-hard by Daniele Micciancio [22]. Here we use the Euclidean norm which is defined in 3.4.2.

Definition 3.4.1 (Shortest Vector Problem) *Given a basis* $B = \{\vec{b_1}, \vec{b_2}, ..., \vec{b_n}\} \in \mathbb{Z}^{m \times n}$ *, the shortest vector problem is to find a vector* $\vec{v}$ *satisfying*

$$\|\vec{v}\| = \min_{\vec{u} \in L(B) \setminus 0} \|\vec{u}\| = \lambda_1(L(B)).$$

Definition 3.4.2 (Euclidean Norm) *On the n-dimensional Euclidean space* $\mathbb{R}^n$*, in intuitive notion of the length of the vector* $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ *is captured by the formula.*

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$

Example. The Shortest Vector Problem might be represented as this. For simplicity, we will be basing our lattices over integers.

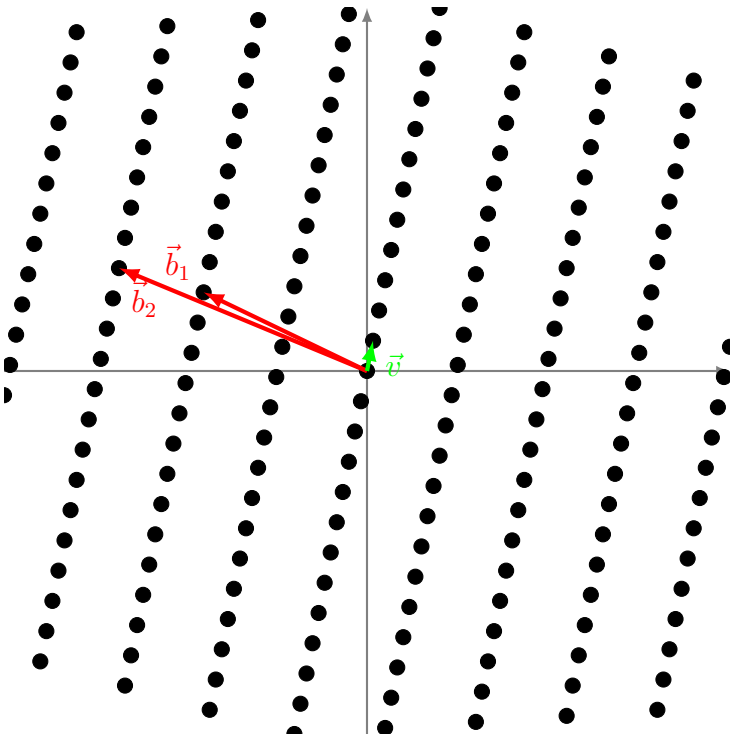

Figure 3.2: Example 1 of SVP

In Figure 3.2 we illustrate a lattice $\mathcal{L} \subset \mathbb{R}^2$ with the basis of $\mathbf{B} = \{[-27, 13], [-41, 17]\}$. In this case, the shortest vector $\vec{v}$ is [1,5] because:

$$3 \cdot [-27, 13] - 2 \cdot [-41, 17] = [1, 5]$$

This might look like an easy problem to solve, given that you only have two dimensions. However, the problem's difficulty rapidly increases as the number of dimensions increases. The fastest discovered algorithm for solving this problem is the Lenstra–Lenstra–Lovász lattice basis reduction algorithm [18] which has a time complexity of the non-deterministic polynomial $\Omega(2^n)$ where n represents the dimensions of the lattice. SVP has also been proven by to be Ω(NP-Hard) by Miklós Ajtai in his proceedings in 1998 [3]. As n grows, this becomes too time-consuming for existing computers to solve, and there has not yet been discovered any post-quantum algorithms that can solve this.

The SVP problem is hard to solve given a bad basis, and is easy if you have a good basis. A good basis is when the vectors are reasonably orthogonal to one another. SVP on a good basis is just the shortest basis vector. This is illustrated in Figure 3.3 where $\vec{b_1}$ is the shortest vector.

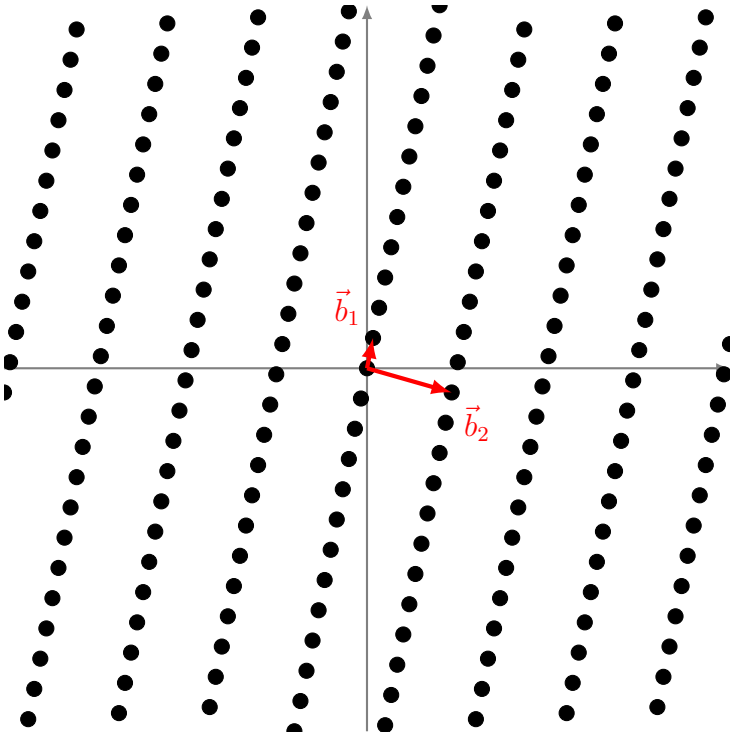

Figure 3.3: Example 2 of SVP

3.5 Closest Vector Problem(CVP)

CVP is closely related to SVP. Given a lattice $\mathcal{L}$ and a target point x, CVP asks us to find the closest lattice point to the target [23]. A mathematical definition has been given in Definition 3.5.1. Goldreich et al. showed that any hardness of SVP implies the same hardness for CVP [12]. Solving CVP has a Ω(NP-hard) time complexity if you have a bad basis. On the other hand if you have a good basis CVP can be solved in polynomial time using Babais algorithm [4].

Definition 3.5.1 (Search Closest Vector Problem) *For any approximation parameter $\gamma = \gamma(n) \geq 1$, the search problem CVP_γ is defined as follows.*

The input is a basis for a lattice $B \subset \mathbb{R}^n$ and vector $t \in \mathbb{R}^n$, the target. The goal is to output a vector $y \in \mathcal{L}$ satisfying

$$\|t - y\| \leq \gamma \cdot dist(t, \mathcal{L})$$

In general, if the basis vectors are close to orthogonal on each other, CVP can easily be solved using Babai's Closest Vertex Algorithm 1. László Babai was able to show that you could solve CVP with polynomial time complexity.

Theorem 1 (Babai's Closest Vertex Algorithm) *Let $\mathcal{L} \subset \mathbb{R}^n$ be a lattice with basis $v_1, v_2, \ldots, v_n$ and let $w \in \mathbb{R}^n$be an arbitrary vector. If the vectors in the basis are sufficiently orthogonal to one another, then the following algorithm solves CVP [13].*

$$\text{Write } w = t_1v_1 + t_2v_2 + \cdots + t_nv_n \text{ with } t_1, \ldots, t_n \in \mathbb{R}.$$
$$\text{Set } a_i = \lfloor t_i \rceil = 1, 2, \ldots, n.$$
$$\text{Return the vector } v = a_1v_1 + a_2v_2 + \cdots a_nv_n.$$

3.5.1 Solving CVP with a good basis

Suppose we want to solve CVP with a good basis as illustrated in Figure 3.4.

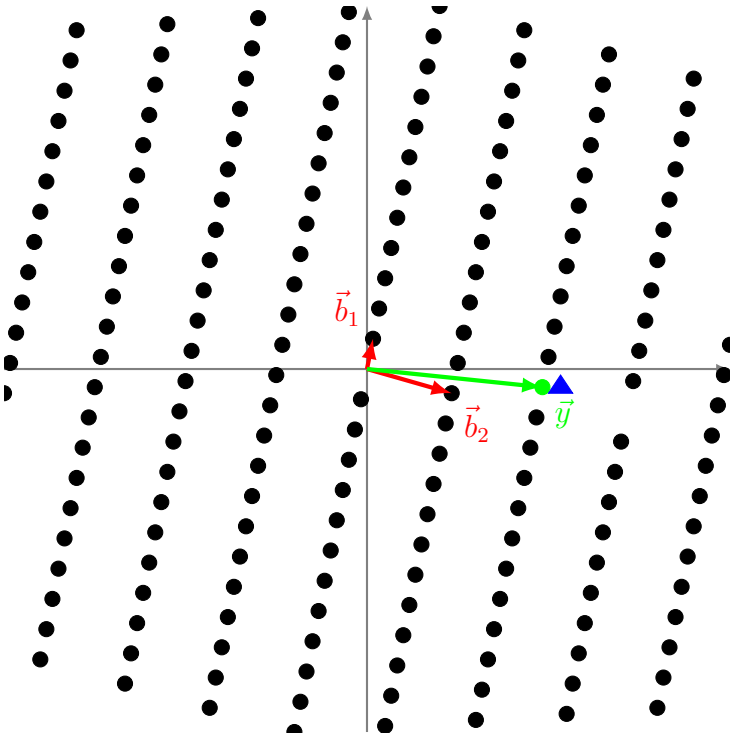

Figure 3.4: Example of CVP with good basis

Problem: To solve this CVP we must find the closest vector to the target point [31,-2]. This point is marked as a blue triangle.

Solution: Since the basis vectors are reasonably orthogonal we can solve CVP by using Babai's algorithm 1, and we write the equation:

$$[31, -2] = a_1 \cdot [1, 5] + a_2 \cdot [14, -4]$$

This corresponds to the following systems:

$$1a_1 + 14a_2 = 31$$
$$5a_1 - 4a_2 = -2$$

The systems have the following solutions

$$a_1 = \frac{48}{37} \approx 1$$
$$a_2 = \frac{157}{74} \approx 2$$

The solution is therefore

$$1 \cdot [1, 5] + 2 \cdot [14, -4] = [29, -3]$$

[29,-3] is the closest lattice point and is marked as a green dot in Figure 3.4.

3.5.2 Solving CVP with bad basis

When we are to solve CVP for the same target point [31,-2] and we are given the bad basis form Figure 3.2. We write the equation:

$$[31, -2] = a_1 \cdot [-27, 13] + a_2 \cdot [-41, 17]$$

This corresponds to the following systems:

$$-27a_1 - 41a_2 = 31$$
$$13a_1 - 17a_2 = -2$$

The systems have the following solutions:

$$a_1 = \frac{445}{74} \approx 6$$
$$a_2 = -\frac{349}{74} \approx -5$$

Using Babais algorithm to solve CVP with a bad basis gives us

$$6 \cdot [-27, 13] - 5 \cdot [-41, 17] = [43, -7]$$

This is not the closest lattice point. We can see that the lattice point with the good basis marked with a green dot in Figure 3.4 is significantly closer than the point we got from the bad basis (marked as a red dot) in Figure 3.5.

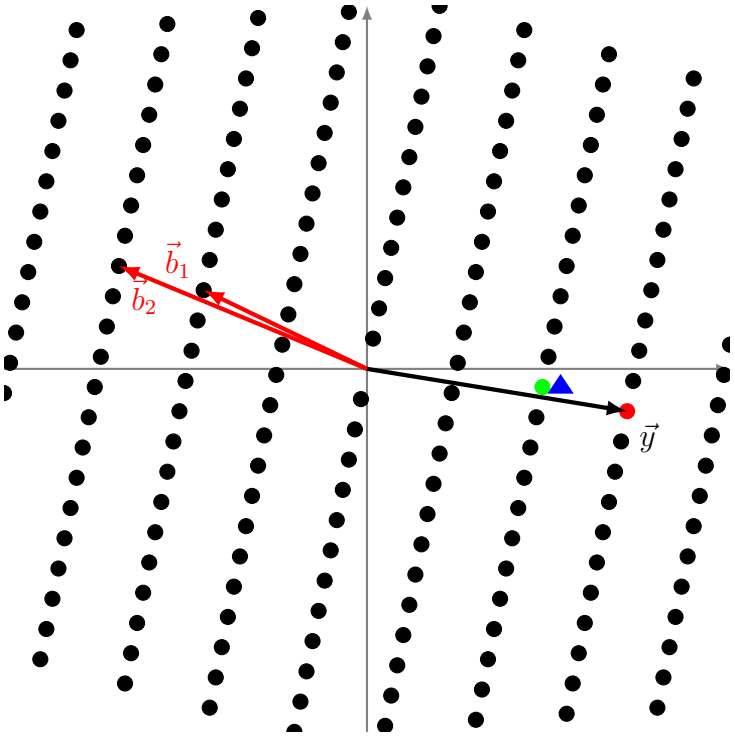

Figure 3.5: Example of CVP with good basis

3.5.3 Crypto system based on CVP

As shown in the previous subsection solving the same CVP problem with a bad basis is hard while solving it with a good basis is easy. We can use this idea to create a crypto system where we use the bad basis as a public key and the good basis as a private key.

Example of a CVP crypto system
Alice wants to send Bob a message consisting of n secret numbers.

1. Bob creates a lattice $\mathcal{L} \subset \mathbb{R}^n$ with a good basis.
2. Bob then reduces the good basis to a bad basis.
3. Bob sends the bad basis to Alice with each of the vectors enumerated $\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n$.
4. Alice encrypts her numbers by multiplying them with the bad basis which generates a pont x in the lattice.

5. Alice then adds a small error to x lets call this new point z.
6. Alice then sends z to Bob
7. Bob can use his good basis to quickly solve the CVP problem by finding z closest lattice point x.
8. Bob can then multiply x with the inverse of the bad basis and get Alice's secret numbers.

The reason Alice sends z instead of x is because the attacker can easily find Alice's secret numbers as shown in step 8.

Lattice-based cryptosystems today essentially use the above idea, but they are more elaborate.

3.6 Learning With Errors (LWE)

The LWE problem was first introduced by Oden Regev in 2005 [1]. LWE is based on solving a linear system of equations with errors. There are two types of LWE problems: Search-version LWE and decisional-version LWE.

3.6.1 Search LWE

Search LWE asks us to recover a secret vector $s \in \mathbb{Z}_q^n$, when we are given m samples of

$$(a_i, \langle a_i, s \rangle + e_i),$$

where $a_i \in \mathbb{Z}_q^n$ is sampled from a uniform distribution and e_i is sampled from a discrete Gaussian distribution χ.

Example 1 (Search LWE) *This problem asks us to recover the secret $s \in \mathbb{Z}_{21}^5$ with the following system of equations. Each of the equations has an error between $e \pm 1$.*

$$
\begin{gathered}
6s_1 + 3s_2 + 13s_3 + 7s_4 + 6s_5 \approx 3 \pmod{21} \\
7s_1 + 13s_3 + 7s_4 + 6s_5 \approx 17 \pmod{21} \\
2s_1 + 15s_2 + 13s_3 + 2s_4 + 15s_5 \approx 9 \pmod{21} \\
20s_1 + 5s_2 + 12s_4 + 5s_5 \approx 20 \pmod{21} \\
8s_1 + 9s_2 + 17s_3 + 14s_4 + 18s_5 \approx 15 \pmod{21} \\
\vdots \\
2s_1 + 9s_2 + 10s_3 + 15s_4 + 11s_5 \approx 12 \pmod{21}
\end{gathered}
$$

In this problem $s = (19, 9, 2, 5, 16)$. If it was not for the error this problem could easily be solved in polynomial time using Gaussian elimination. But in LWE the Gaussian elimination will amplify the error to such an extent that we would be unable to recover the information in s. This makes the problem significantly harder.

3.6.2 Decisional LWE

The Decisional LWE problem gives an input of samples (a, v) where $a \in \mathbb{Z}_q^n$ is chosen from a uniform probability distribution. The difference between search and decisional LWE is that we must determine with some non-negligible error probability weather v is chosen uniformly from $\mathbb{Z}_q^n$ or if v is chosen to be $(a_i, \langle a_i, s\rangle + e_i)$ where $s \in \mathbb{Z}_q^n$ is uniformly chosen and $e_i \in \mathbb{Z}_q^n$ is chosen from χ^n.

3.7 Ring Learning With Errors (RLWE)

The RLWE problem was first introduced by Vadim Lyubashevsky, Chris Peikert, and Oded Regev in 2013 [19]. RLWE is a variant of LWE, but instead of using vectors $\mathbb{Z}_q^n$ it works with polynomials over rings $\mathbb{Z}_q[X]/(X^n + 1)$. RLWE can be defined similarly to LWE $(a_i, \langle a_i, s\rangle + e_i)$, but the variables are polynomials drawn from $\mathbb{Z}_q[X]/(X^n + 1)$. Each of the variables uses the same probability distributions as in LWE.

RLWE has multiple advantages over LWE when it comes to homomorphic encryption. One is that multiplication between polynomials can be done a lot quicker by using the

Fast Fourier Transform(FFT) algorithm. FFT has a time complexity of $\mathcal{O}(n\ log(n))$. This is significantly faster than matrix multiplication which has a time complexity of $\mathcal{O}(n^2)$.

3.7.1 RLWE crypto system

By applying RLWE we can develop a crypto system: Let the public key be $pk = (a, -as + e) \in (\mathbb{Z}_q[X]/(X^n + 1))^2$, the secret key be $s \in \mathbb{Z}_q[X]/(X^n + 1)$ and the plaintext be $m \in \mathbb{Z}_q[X]/(X^n + 1)$.

Encryption:
ciphertext $= (0, m) + pk = (a, m - as + e) = (c_0, c_1)$

Decryption:
m' $= c_0 + c_1 \cdot s = m - a \cdot s + e + a \cdot s = m + e \approx$ m

Chapter 4

CKKS Scheme

In this chapter I describe the different stages of the CKKS scheme. CKKS allows you to perform approximate fully homomorphic operations on complex floating point numbers at a high speed. CKKS was introduced as an approximate encryption scheme in 2016 by Cheon et al. [5]. "Approximate" means that you will not get the exact message you originally encrypted, but a number close to it, depending on the encryption parameters. The CKKS scheme only encrypts numbers. Numbers can represent everything from letters and symbols to entire computer programs. The authors of CKKS argue that all real world data will have some error. Trying to get the exact number you encrypted is irrelevant. CKKS is a public key crypto system. This implies that you have a public key for encryption and a private key for decryption. The secret key should be kept private.

In this section I give a brief introduction on the mathematical background of CKKS. This includes the message, the encoding and the encryption part.

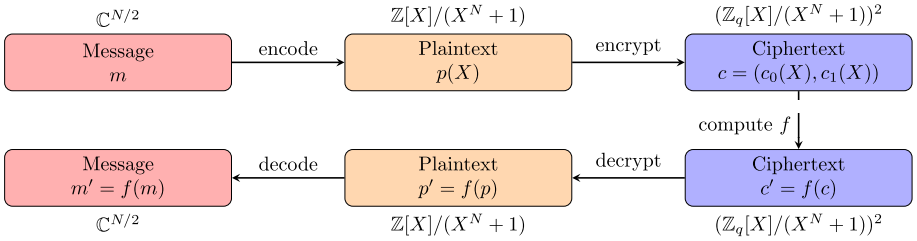

Figure 4.1: Overview of CKKS stages[8]

4.1 Message

The CKKS scheme supports approximate arithmetics over complex numbers. The input message of the crypto system must be vectors from the space of $\mathbb{C}^{\frac{n}{2}}$, where n is some power-of-two integer. This vector is defined as $\vec{z} = (z_1, z_2, ..., z_{\frac{n}{2}}) \in \mathbb{C}^{\frac{n}{2}}$.

4.2 Encoding

Encoding is the process of changing the data, or the message, into a new format called plaintext. This entails that it can be interpreted and encrypted by the CKKS scheme. In this Subsection I follow the encoding technique for packing messages from [5].

In CKKS we are mapping the message $\vec{z} \in \mathbb{C}^{\frac{n}{2}}$ into a cyclotomic polynomial ring with integer coefficients $m(X) = \mathbb{Z}[X]/(X^n + 1)$, where n is the degree modulus of the polynomial. For simplicity and security n will be a power of 2. The encoding algorithm receives two parameters: the message and the scaling factor $\Delta > 1$. Having a large Δ gives us more accuracy but reduces the number of homomorphic operations we can perform.

The values generated from the operations:

$$\mathbb{C}^{\frac{n}{2}} \xrightarrow{\pi^{-1}} \mathbb{H} \xrightarrow{\Delta \cdot \mathbb{H}} \mathbb{V} \xrightarrow{\lfloor \cdot \rceil_{\sigma(\mathbb{Z}[X]/(X^n+1))}} \sigma(\mathbb{Z}[X]/(X^n+1)) \xrightarrow{\sigma^{-1}} \mathbb{Z}[X]/(X^n+1)$$

The operations performed on vector $\vec{z}$:

$$\vec{z} \to \pi^{-1}(\vec{z}) \to \Delta \cdot \pi^{-1}(\vec{z}) \to \lfloor \Delta \cdot \pi^{-1}(\vec{z}) \rceil_{\sigma(\mathbb{Z}[X]/(X^n+1)} \to \mathbb{Z}[X]/(X^n+1)$$

The easiest way to understand CKKS encoding, is by explaining each of the steps individually.

4.2.1 Embedding

The inverse embedding operation defined as σ^{-1} is the last step of the encoding process. We will start with this step because it will give us a better understanding of the steps leading up to it and the properties to embedding. The embedding works as a map between polynomials and vectors, which is an isometric ring homomorphism.

In this example we will use inverse embedding to map vector $\vec{z} \in \mathbb{C}^n$ to a polynomial $m(X) \in \mathbb{C}[X]/(X^n + 1)$ and use embedding to map vice versa.

The **embedding^{-1}** is the map from:

$$\mathbb{C}^n \xrightarrow{\sigma^{-1}} \mathbb{C}[X]/(X^n + 1)$$

The **embedding** is the map from:

$$\mathbb{C}[X]/(X^n + 1) \xrightarrow{\sigma} \mathbb{C}^n$$

Embedding is defined as an isomorphism. This means that it is a bijective homomorphism meaning that every vector maps to a unique polynomial and every polynomial maps to a unique vector.

Embedding
The **embedding** is easier to understand than the **embedding^{-1}**. I will therefore start with the **embedding**.

The **embedding** is performed by evaluating the encoded polynomial on specific values. These values are the roots of unity for cyclotomic polynomial $\Phi_{2n}(X) = X^n + 1$. We define these roots as $(\zeta, \zeta^3, \ldots, \zeta^{2n-1})$ where $\zeta = e^{\frac{2i\pi}{2n}}$.

The embedding process can be described like this.

$$\sigma(m) = (m(\zeta), m(\zeta^3), \ldots, m(\zeta^{2n-1})) \in \mathbb{C}^n = [z_1, z_2, \ldots, z_n] = \vec{z}$$

Inverse embedding
Finding the polynomial that when evaluated on the roots of unity maps to the embedded

vector $\vec{z}$ is harder. The polynomial must satisfy $m(X) = \Sigma_{i=0}^{n-1}\alpha_i X^i \in \mathbb{C}[X]/(X^n+1)$, given the vector $\vec{z}$, where α is the coefficients of the polynomial that we need to find.

To find the polynomial coefficients α to solve the following system.

$$\Sigma_{j=0}^{n-1}\alpha_j(\zeta^{2i-1})^j = z_i, i = 1, \ldots, n$$

This can be solved by using the values as a system of linear equations. $A\alpha = \vec{z}$ where A is the Vandermonde matrix of $\zeta_{i=1,\ldots,n}^{2i-1}$.

$$A = \begin{bmatrix} (\zeta)^0 & (\zeta)^1 & (\zeta)^2 & \ldots & (\zeta)^n \\ (\zeta^3)^0 & (\zeta^3)^1 & (\zeta^3)^2 & \ldots & (\zeta^3)^n \\ (\zeta^5)^0 & (\zeta^5)^1 & (\zeta^5)^2 & \ldots & (\zeta^5)^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\zeta^{2n-1})^0 & (\zeta^{2n-1})^1 & (\zeta^{2n-1})^2 & \ldots & (\zeta^{2n-1})^n \end{bmatrix}$$

We get the polynomial by solving the following equation

$$\alpha = A^{-1}\vec{z}$$

Example

1) Inverse embedding

Suppose you want to map the vector $\vec{z} = [4.44+2i, 4, 2+i, 2i]$ to a cyclotomic polynomial $\mathbb{C}[X]/(X^4+1)$.

We can describe this as

$$\vec{z} \in \mathbb{C}^4 \xrightarrow{\sigma^{-1}} \mathbb{C}[X]/(X^4+1)$$

We start by calculating the Vandermonde matrix A where $\zeta = e^{\frac{2i\pi}{8}} \approx 0.7071 + 0.7071i$

$$\begin{bmatrix} (\zeta)^0 & (\zeta)^1 & (\zeta)^2 & (\zeta)^3 \\ (\zeta^3)^0 & (\zeta^3)^1 & (\zeta^3)^2 & (\zeta^3)^3 \\ (\zeta^5)^0 & (\zeta^5)^1 & (\zeta^5)^2 & (\zeta^5)^3 \\ (\zeta^7)^0 & (\zeta^7)^1 & (\zeta^7)^2 & (\zeta^7)^3 \end{bmatrix} = \begin{bmatrix} 1 & (0.7071+0.7071i) & i & (-0.7071+0.7071i) \\ 1 & (-0.7071+0.7071i) & -i & (0.7071+0.7071i) \\ 1 & (-0.7071-0.7071i) & i & (0.7071-0.7071i) \\ 1 & (0.7071-0.7071i) & -i & (-0.7071-0.7071i) \end{bmatrix}$$

We now have the following system $A\alpha = \vec{z}$.

$$\begin{bmatrix} 1 & (0.7071+0.7071i) & i & (-0.7071+0.7071i) \\ 1 & (-0.7071+0.7071i) & -i & (0.7071+0.7071i) \\ 1 & (-0.7071-0.7071i) & i & (0.7071-0.7071i) \\ 1 & (0.7071-0.7071i) & -i & (-0.7071-0.7071i) \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} (4.44+2i) \\ 4 \\ 2+i \\ 2i \end{bmatrix}$$

By solving $\alpha = A^{-1}\vec{z}$ we get.

$$\alpha_1 = 2.61 + 1.25i$$
$$\alpha_2 = -0.4525 - 0.6081i$$
$$\alpha_3 = 0.25 - 0.6081i$$
$$\alpha_4 = 0.0990 - 1.6688i$$

This gives us the polynomial

$$m(X) = (2.61+1.25i) + (-0.4525-0.6081i)x + (0.25-0.6081i)x^2 + (0.0990-1.6688i)x^3$$

2) Additive homomorphism

To show the embeddings additive homomorphism we create a new polynomial $m'(X)$.

$$m'(X) = m(X)+m(X) = (5.22+2.5i)+(-0.9051-1.2162i)x+(0.5-1.22i)x^2+(0.1980-3.3375i)x^3$$

3) Embedding

We now map the polynomial back to a vector $\vec{z}'$.

$$\mathbb{C}[X]/(X^4+1) \xrightarrow{\sigma} \vec{z}' \in \mathbb{C}^4$$

This is done by solving $A\alpha = \vec{z}'$ where α is $m'(X)$ coefficients .

$$\begin{bmatrix} 1 & (0.7071+0.7071i) & i & (-0.7071+0.7071i) \\ 1 & (-0.7071+0.7071i) & -i & (0.7071+0.7071i) \\ 1 & (-0.7071-0.7071i) & i & (0.7071-0.7071i) \\ 1 & (0.7071-0.7071i) & -i & (-0.7071-0.7071i) \end{bmatrix} \cdot \begin{bmatrix} (5.22+2.5i) \\ (-0.9051-1.2162i) \\ (0.5-1.22i) \\ (0.1980-3.3375i) \end{bmatrix} = \begin{bmatrix} (8.88+4i) \\ 8 \\ 4+2i \\ 4i \end{bmatrix}$$

Here we can see that $\vec{z}' = \vec{z} + \vec{z}$. This shows the additive homomorphism from step 2.

4.2.2 Inverse Natural Projection

In 4.2.1 we mapped a vector $\vec{z} \in \mathbb{C}^n$ to $\mathbb{C}[X]/(X^n+1)$. CKKS maps $\vec{z} \in \mathbb{C}^{\frac{n}{2}}$ to the cyclotomic polynomial $\mathbb{Z}[X]/(X^n+1)$. We therefore have to modify the values of $\vec{z} \in \mathbb{C}^{\frac{n}{2}}$ to be embedded in $\mathbb{Z}[X]/(X^n+1)$. One of these steps is expanding $\vec{z} \in \mathbb{C}^{\frac{n}{2}}$ by using inverse natural projection π^{-1}.

$$\mathbb{C}^{\frac{n}{2}} \xrightarrow{\pi^{-1}} \mathbb{H}$$

such that

$$[z_1, \ldots, z_{\frac{n}{2}}] \xrightarrow{\pi^{-1}} [z_1, \ldots, z_{\frac{n}{2}}, \overline{z}_{\frac{n}{2}+1}, \ldots, \overline{z}_n]$$

where $\overline{z}_{n-i}$ is the conjugate of z_i

4.2.3 Scaling

This step is simple. We multiply with the scaling factor Δ. The size of the scaling factor determines the accuracy of the encoding.

$$\mathbb{H} \xrightarrow{\Delta \cdot \mathbb{H}} \mathbb{V}$$

4.2.4 Projection to the lattice

We will now project the vector $\vec{z} \in \mathbb{V}$ on to the lattice of $\sigma(\mathbb{Z}[X]/(X^n+1))$. We compute the coordinates of z with repect to the orthogonal lattice basis. The orthogonal lattice basis is the good basis as discussed in Section 3.5.

$$\mathbb{V} \xrightarrow{\lfloor \cdot \rceil_{\sigma(\mathbb{Z}[X]/(X^n+1))}} \sigma(\mathbb{Z}[X]/(X^n+1))$$

$$\vec{z} \in \mathbb{V} = [z_1, z_2, \ldots, z_n] \rightarrow a = (a_1, a_2, \ldots, a_n) = \left(\frac{(\vec{z}, V_1)}{(V_1, V_1)}, \frac{(\vec{z}, V_2)}{(V_2, V_2)}, \ldots, \frac{(\vec{z}, V_n)}{(V_n, V_n)} \right)$$

V_i represents the i-th column of the Vandermonde matrix A defined in Subsection 4.2.1.

We then round each of the coordinate values to its closest integer and multiply it with the Vandermonde matrix.

$$a \rightarrow \lfloor a \rceil \cdot V$$

4.2.5 Decoding

The decoding is just the inverse of the encoding.

$$\vec{z} = \pi \circ \sigma(\Delta^{-1} \cdot m)$$

4.2.6 Encoding and decoding example

$\vec{z}$ = [2.25+3.4i, 5+9.1i], $\Delta = 64$

Encoding

1) Inverse Natural Projection

$$[2.25 + 3.4i, 5 + 9.1i] \xrightarrow{\pi^{-1}} [2.25 + 3.4i, 5 + 9.1i, 5 - 9.1i, 2.25 - 3.4i]$$

2) Scaling

$$[2.25+3.4i, 5+9.1i, 5-9.1i, 2.25-3.4i] \xrightarrow{\mathbb{H}\cdot\Delta} [144+217.6i, 320+582.4i, 320-582.4i, 144-217.6i]$$

3) Projection

$$\vec{z} = [144 + 217.6i, 320 + 582.4i, 320 - 582.4i, 144 - 217.6i]$$

$$V = \begin{bmatrix} 1 & (0.7071 + 0.7071i) & i & (-0.7071 + 0.7071i) \\ 1 & (-0.7071 + 0.7071i) & -i & (0.7071 + 0.7071i) \\ 1 & (-0.7071 - 0.7071i) & i & (0.7071 - 0.7071i) \\ 1 & (0.7071 - 0.7071i) & -i & (-0.7071 - 0.7071i) \end{bmatrix}$$

$$\left[\frac{(\vec{z}, V_1)}{(V_1, V_1)}, \frac{(\vec{z}, V_2)}{(V_2, V_2)}, \frac{(\vec{z}, V_3)}{(V_3, V_3)}, \frac{(\vec{z}, V_4)}{(V_4, V_4)}\right] = [232, 220.61731573, -182.4, 345.06810922]$$

$$\approx [232, 221, -182, 345]$$

$$[232, 221, -182, 345] \cdot V = [144.319+218.222i, 319.681+582.222i, 319.681-582.222i, 144.319-218.222i]$$

4) Inverse Embedding

We now have $A\alpha = \vec{z}$

$$\begin{bmatrix} 1 & (0.7071+0.7071i) & i & (-0.7071+0.7071i) \\ 1 & (-0.7071+0.7071i) & -i & (0.7071+0.7071i) \\ 1 & (-0.7071-0.7071i) & i & (0.7071-0.7071i) \\ 1 & (0.7071-0.7071i) & -i & (-0.7071-0.7071i) \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} (144.319+218.222i) \\ 319.681+582.222i \\ 319.681-582.222i \\ 144.319-218.222i \end{bmatrix}$$

By solving $\alpha = A^{-1}\vec{z}$ we get.

$$\alpha_1 = 232$$
$$\alpha_2 = 221$$
$$\alpha_3 = -182$$
$$\alpha_4 = 345$$

We have now completed the encoding, and we now have the polynomial:

$$m(x) = 232 + 221x^1 - 182.0x^2 + 345.0x^3$$

.

Decoding

1) Inverse Scaling

$$(232 + 221x^1 - 182.0x^2 + 345.0x^3) \cdot \Delta^{-1} = 3.625 + 3.453125x - 2.84375x^2 + 5.390625x^3$$

2) Embedding

$p(x) = 3.625 + 3.453125x - 2.84375x^2 + 5.390625x^3$

$$[p(\zeta^1), p(\zeta^3), p(\zeta^5), p(\zeta^7)] = [2.255 + 3.410i, 4.995 + 9.097i, 4.995 - 9.097i, 2.255 - 3.410i]$$

3) Natural Projection

$$[2.255+3.410i, 4.995+9.097i, 4.995-9.097i, 2.255-3.410i] \xrightarrow{\pi} [2.255+3.410i, 4.995+9.097i]$$

4.3 Leveled Homomorphic Encryption

This chapter covers encryption and the homomorphic operations that can be performed on encrypted data, also referred to as ciphertext.

4.3.1 Key Generation

We will also introduce a new sampling distribution $\mathcal{HWT}(h)$ which contains a set of vectors in $\{-1, 0, 1\}^n$ with a Hamming weight of h. This means that each of the vectors has exactly h non-zero values. For simplicity, I will write $\mathcal{R}$ instead of $\mathbb{Z}[X]/(X^n+1)$ and $\mathcal{R}_Q$ instead of $\mathbb{Z}_Q[X]/(X^n+1)$. The notation $a \leftarrow \mathcal{R}_Q$ means that a is a sample from a uniform probability distribution of the set $\mathcal{R}_Q$. χ is a discrete Gaussian distribution.

Choose values for n, q, Q, Δ, h, where q, Q, Δ and n have to be powers-of-two. To avoid errors under the rescaling procedure $\Delta < q < Q$.

Secret key
The secret key is generated as a sample from $s \leftarrow \mathcal{HWT}(h)$.

$$sk = (1, s) \in \mathcal{R}_Q^2$$

Public key
The CKKS encryption uses variables from the RLWE problem as we discussed in 3.7 as the public key pk. The variables are generated as $a \leftarrow \mathcal{R}_Q$ and $e \leftarrow \chi$.

$$pk = (a, -as + e) \in \mathcal{R}_Q^2$$

We use $-a$ in the second polynomial because it makes the decryption slightly easier 4.3.3.

Switching key
We generate the switching key by inputting two values sk and $s' \in \mathcal{R}_Q$. We sample $a \leftarrow \mathcal{R}_Q$ and $e \leftarrow \chi$.

$$swk = (a, -as + e + Q \cdot s') \in \mathcal{R}_{Q^2}^2$$

Relinearization key
The relinearization key is generated with the help of the switching key generation method. The variable s is the same as in sk.

$$rlk = SwitchingKeyGenerator(sk, s^2) \in \mathcal{R}_{Q^2}^2$$

4.3.2 Encryption

We will encrypt the message $m \in \mathcal{R}$ by using pk into a ciphertext ct. For encryption, we will introduce a new probability distribution $\mathcal{ZO}(p)$. For real $0 \leq p \leq 1$, the distribution $\mathcal{ZO}(p)$ draws vectors form $\{-1, 0, 1\}^n$, where each value in the drawn vector has a probability of $1 - p$ for being a zero and $p/2$ for each of -1 and $+1$.

Encryption
Let $v \leftarrow \mathcal{ZO}(0, 5)$ be an ephemeral value, $e_0, e_1 \leftarrow \chi$ and the public key $pk = (a, b)$.

$$ct = v \cdot pk + (m + e_0, e_1) = (v \cdot a + e_0 + m, v \cdot b + e_1) \in \mathcal{R}_q^2$$

4.3.3 Decryption

We decrypt $ct = (c_0, c_1)$ by using the secret key $sk = \langle 1, s \rangle$.

$$m' = c_0 + c_1 \cdot s = m - a \cdot s + e + a \cdot s \pmod{q} = m + e \approx m$$

4.3.4 Evaluation

The part where we preform homomorphic operations on encrypted data. We can not perform operations ciphertext that has different modulo q. The ciphertexts we will do operations on are $ct = (a, b)$ and $ct' = (a', b')$.

Addition(ct, ct'):
When perform addition, the ciphertexts must have the same scaling factor.

$$ct_{add} = (a + a' \pmod{q}, b + b' \pmod{q})$$

Multiplication(ct, ct', rlk):
We will use Δ and Δ' as the scaling factor for ct and ct'.

$$c_0 = a \cdot a' \pmod{q}$$
$$c_1 = a \cdot b' + b \cdot a' \pmod{q}$$
$$c_2 = b \cdot b' \pmod{q}$$

We will use $rlk = (r_0, r_1)$ to reduce the number of polynomials to two.

$$c_0' = ct + \lfloor Q^{-1} \cdot c_2 \cdot r_0 \rceil \pmod{q}$$
$$c_1' = ct' + \lfloor Q^{-1} \cdot c_2 \cdot r_1 \rceil \pmod{q}$$

The resulting ciphertext $ct_{mult} = (c_0', c_1')$ will have the scaling factor $\Delta_{mult} = \Delta \cdot \Delta'$.

Chapter 5

Specifications for implementation

I have implemented the CKKS scheme with a Graphical User Interface(GUI) and a code-based interface. The GUI is made to give the user an easy and intuitive introduction to the CKKS scheme with no programming or mathematical experience. One example is that it renders error messages to help the user write valid input. The code-based interface gives the user more flexibility and room to experiment. The downside is that it does not have the same restrictions as the GUI, which makes it easier to do mistakes and generate invalid output or crash the program. I have implemented the scheme using the Java version 15.

5.1 Architecture

This library is divided into 7 parts: Parameters, KeyGenerator, Encoder, Encryptor, Evaluator, Decryptor and GUI I will discuss each of them in the following Subsections.

The user sets the parameters: Polynomial degree n, small modulus q, Big modulo Q, scaling factor Δ, prime number bit size θ and Miller-Rabin iterations β.

5.1.1 Parameters

The purpose of parameters is to store the values necessary for encoding, encryption and key generation. This is an efficient way to sore information from operation to operation.

Parameters creates and instance of the ChineseRemainderTheorem class (CRT). CRT generates $2+\log_2(n)+\frac{4\cdot\log_2(Q)}{\theta}$ primes, where each of the primes p satisfies $p \equiv 1 \pmod{2n}$. Each of the primes is generated with the Miller-Rabin primality test, which starts by testing if $10^{\theta-1}+1$ is prime and for each iteration adds an extra $2n$ to the initial value until all the necessary primes has been created. I add $2n$ such that $1 \equiv p \pmod{2n}$.

CRT then creates an instance of the Number Theoretical Transform(NTT) for each of the primes. Each of the NTTs then finds the primitive element (ψ) to its prime (ι) which is Defined in 5.1.1. NTT then finds the root of unity $\kappa = \frac{\psi^{\iota-1}}{2n} \pmod{\iota}$ and the inverse root of unity $\rho = \kappa^{\iota-2} \pmod{\iota}$. NTT then calculates the roots of unity $[\kappa^0, \kappa^1, \kappa^2, \ldots, \kappa^{n-1}]$ then the roots of unity inverse $[\rho^0, \rho^1, \rho^2, \ldots, \rho^{n-1}]$.

Definition 5.1.1 (Primitive element(ψ))

$$\mathbb{Z}_q^* = \{a \in \mathbb{N} : 1 \leq a < n, gcd(a, n) = 1\}$$

A primitive root $\pmod{q}$ *is an element* $g \in \mathbb{Z}_q^*$ *whose powers generate all of* $\mathbb{Z}_n^*$. *That is, every element* $b \in \mathbb{Z}_n$ *can be written as* $g^x \pmod{q}$ *for some integer* x.

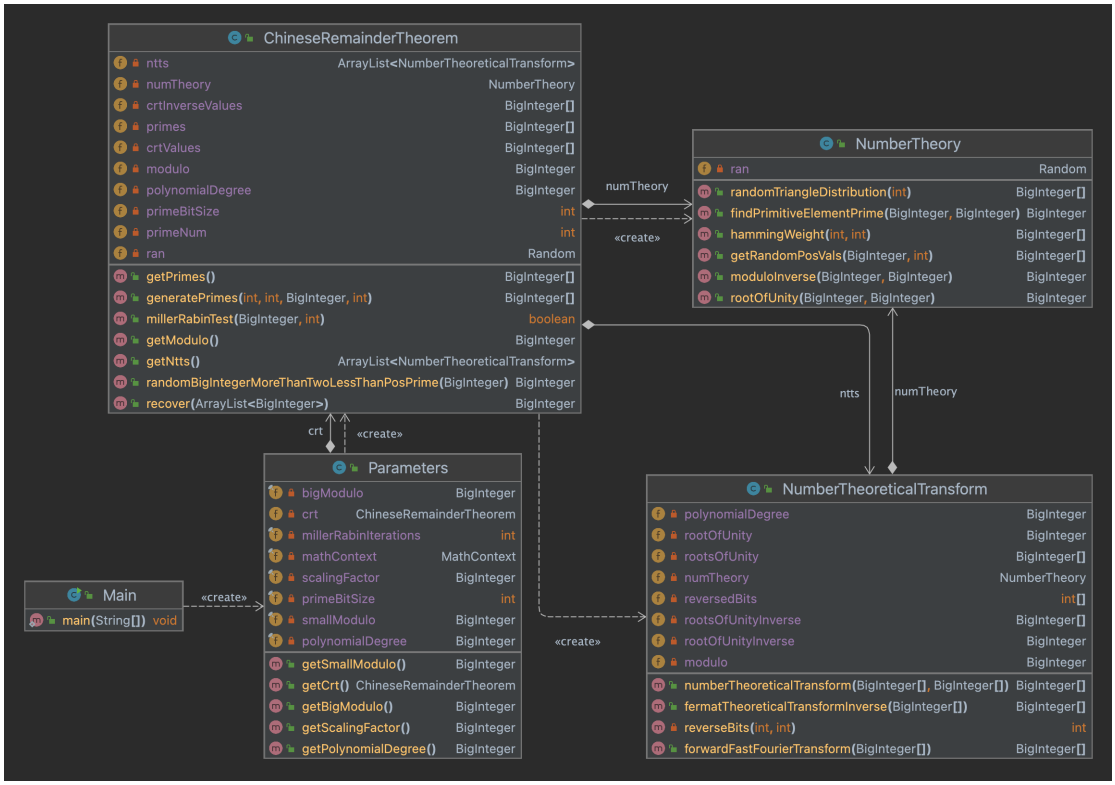

Figure 5.1: Parameters UML

5.1.2 Key generator

The secret key will be created with the Hamming weight of $\frac{n}{4}$, which means that the security of the ciphertext depends on the size of the encrypted vector. When generating the error (e) for the public key ($(a, -as + e)$ and the switching key ($a, -as + e + Q \cdot s'$) I used the $\mathcal{ZO}(0, 5)$ as the Gaussian distribution, which was introduced in 4.3.2.

5.1.3 Encoder

Before we can start encoding the program must first construct the class, which creates and instance of the Fast Fourier Transform (FFT). FFT precomputes the roots of unity and the roots of unity inverse to the cyclotomic $\Phi_{2n}(X)$ as discussed in 4.2.1. The encoder implemented will be more advanced than the one discussed in 4.2. My encoder is based on the latest developments in the HEAAN library [6].

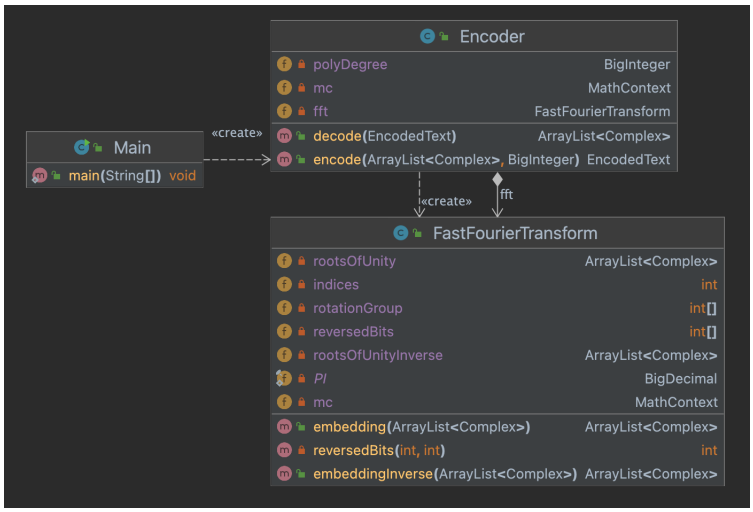

Figure 5.2: Encoder UML

5.1.4 Encryption and Decryption

The values e_0 and e_1 in $ct = v \cdot pk + (m + e_0, e_1)$ will be generated from the distribution $\mathcal{ZO}(0, 5)$. The rest has been implement as described in 4.3.2 and 4.3.3.

5.1.5 Evaluator

Described in 4.3.4.

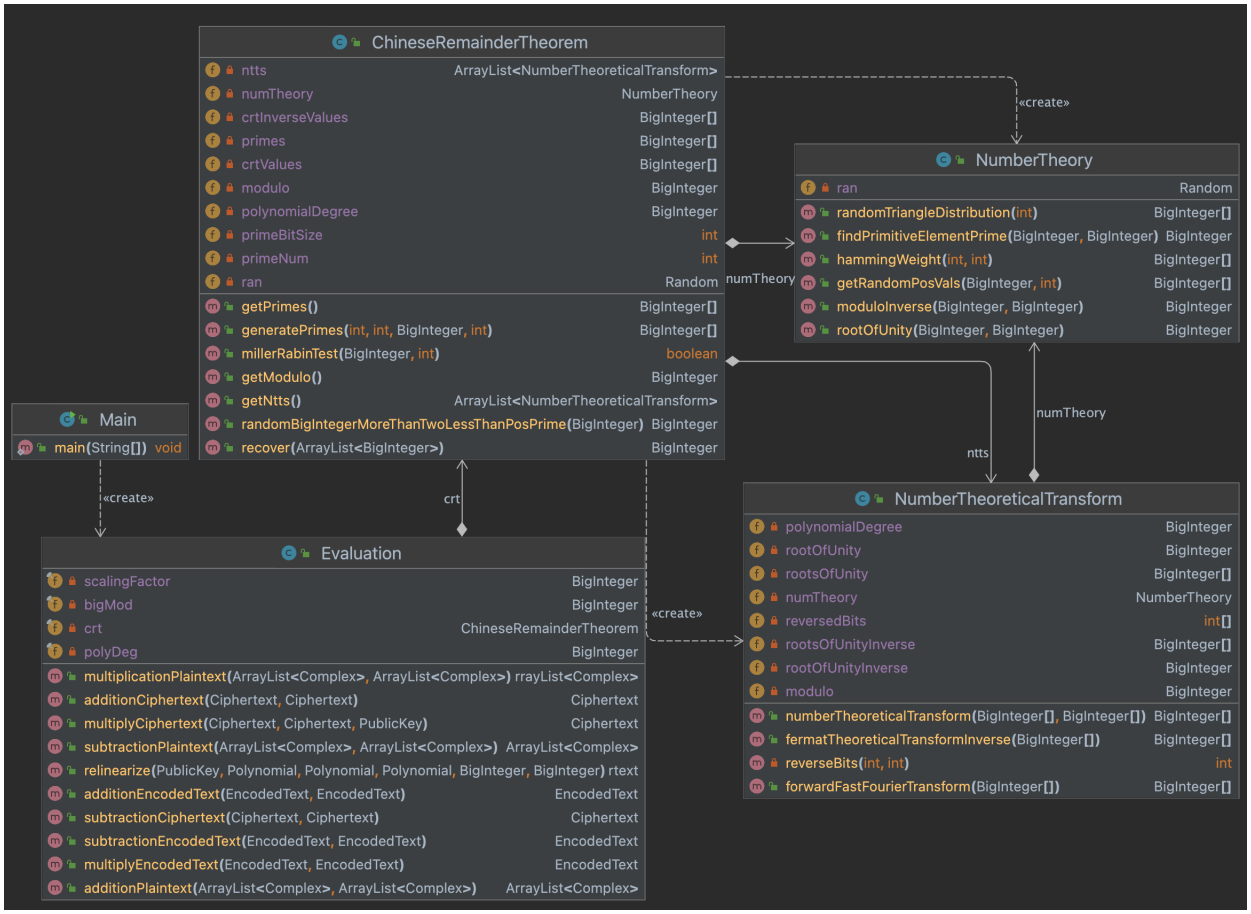

Figure 5.3: Evaluator UML

5.1.6 Graphical User Interface(GUI)

The purpose of the GUI is to offer a basic introduction to the user. The user will be guided through the different steps of the CKKS scheme. This implies that no programming knowledge is needed. The GUI is divided into four frames, that the user fills out sequentially.

In the first frame, the user must fill in the length and size of the vectors. The first frame is illustrated in 5.4.

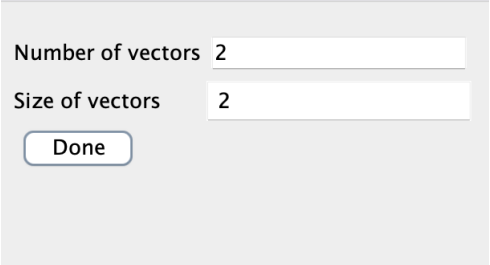

Figure 5.4: Frame one

The next step is to fill in the vector values, which later are available for cryptographic operations. This frame is illustrated in Figure 5.5.

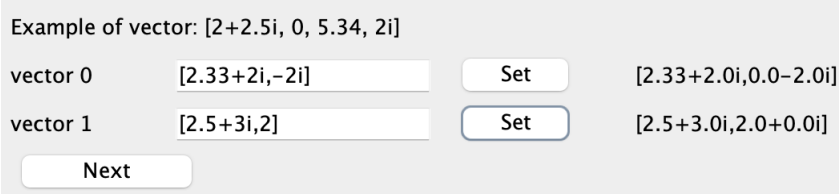

Figure 5.5: Frame two

In the third step, the user will set the parameters which are discussed in 5.1. The program gives the user some suggestions, and the user can make his choices.

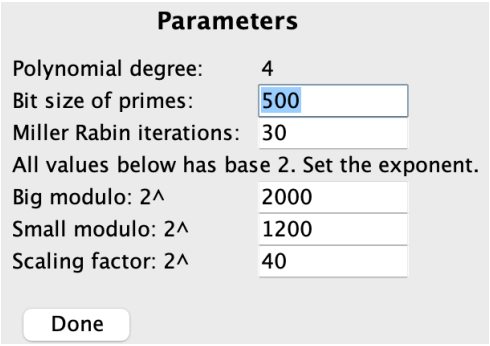

Figure 5.6: Frame three

In the final step, the user can perform encoding, encryption and homomorphic operations like multiplication, addition and subtraction on the vectors from the second step. This frame is illustrated in figure 5.7.

When a user is to encrypt vectors, the user must first press the "Generate keys" button. All the vectors and cryptographic keys values will be printed to the terminal when the user presses one of the "Show buttons".

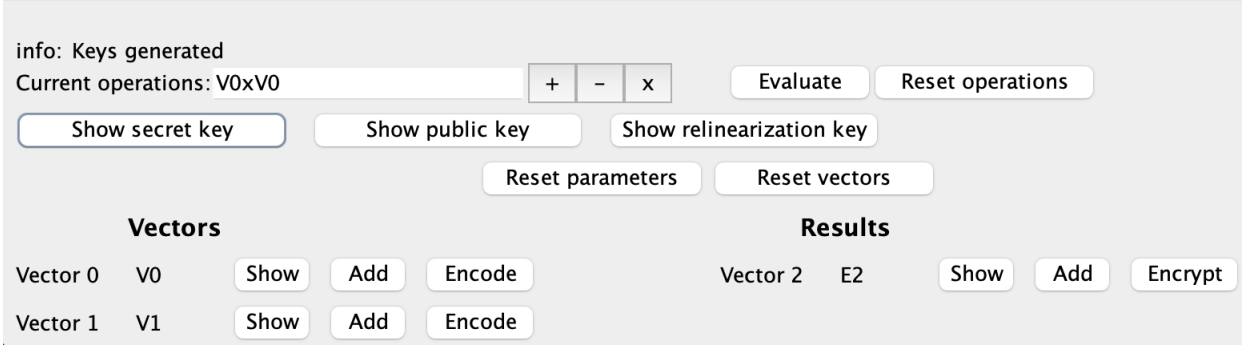

Figure 5.7: Operation frame

To perform evaluation operations, the user must press the "Add" button to the right of one of the vectors, and then select one of the algebraic operations, add another vector and finally press "Evaluate". When the user presses "Evaluate", the result will be printed to the terminal and the resulting vector will be added under "Results".

A problem I faced while creating the GUI, was showing the vectors and keys to the user. It would have been too messy and cluttered to have them in the GUI. Adding scrolling would not help with the user-friendliness. As a solution to this challenge, I chose to combine the GUI with the terminal. This implies that I print these values to the terminal.

Another problem is that when you encode or decode a vector, a small error will be added. If the user pressed a wrong button, the user must go back and set the vector again. To prevent this, the program remembers when a vector has been unencoded and previously encoded. Then it will pick the previously saved state.

Chapter 6

Discussion and conclusion

In this master thesis I implemented a Java library based on the CKKS scheme. This library supports approximate arithmetic on encrypted data consisting of real numbers.

This master thesis is relevant because cloud computing is becoming increasingly popular due to low price and convenience. However, cloud computing is problematic because it can allow 3rd parties to access the stored information. Homomorphic encryption is a good preventive measure. Moreover, we must develop more resilient cryptosystems based on complex mathematical problems to defend ourselves in the coming era of quantum computers.

The field of homomorphic encryption is developing in a fast pace and this does not seem to slow down anytime soon. A cutting edge development is the CKKS encryption scheme, which was first published in 2016 [5]. It has been under continuous development ever since, and this is where this master thesis contributes.

Moreover, in this master thesis I describe various types of encryption and what criteria a good cryptosystem should follow. A fundamental issue is the tradeoff between efficiency and security. I have described the mathematical concept of homomorphism and introduced Ω(NP-Hard) problems based on lattices and showed how these could be applied to cryptography.

Further, I have explained the CKKS scheme and its methods for encoding, encryption and evaluation. However, in addition to the descriptions, the main contribution of this master thesis is my implementation of the CKKS scheme in Java. Java is a popular language, less complex than C++ and faster than Python. To my knowledge, an implementation

in Java has never been done before. In addition, I created a graphical user interphase (GUI) to give the user a basic overview of the CKKS scheme. Moreover, it provides a user-friendliness to encompass users with limited programming experience. A future research avenue is to find a way to set up the library for server-client mode.

To conclude, Fully Homomorphic Encryption (FHE) is cutting edge. It gives us a future ability to process sensitive data safely in a convenient and cost-effective way, also with third parties as cloud computing providers.

Bibliography

[1] *STOC '05: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139608.

[2] Heaan. `https://github.com/snucrypto/HEAAN`, 2022.

[3] Miklós Ajtai. The shortest vector problem in l2 is np-hard for randomized reductions. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 10–19, 1998.

[4] László Babai. On lovász'lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. *Cryptology ePrint Archive*, 2016.

[6] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*, pages 360–384. Springer, 2018.

[7] Joan Daemen and Vincent Rijmen. The design of Rijndael: AES — the Advanced Encryption Standard. page 238, 2002.

[8] DANIEL HUYNH. High level view of ckks, 2020.
URL: `https://blog.openmined.org/content/images/2020/08/Cryptotree_diagrams-2.svg`.

[9] Whitfield Diffie and Martin Hellman. New directions in cryptography. 1976.

[10] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Annual International Conference on the theory and applications of cryptographic techniques*, pages 129–148. Springer, 2011.

[12] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and J-P Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Information Processing Letters*, 71(2):55–61, 1999.

[13] Jeffrey Hoffstein, Jill Pipher, Joseph H Silverman, and Joseph H Silverman. An introduction to mathematical cryptography. page 380, 2008.

[14] Anton A. Huurdeman. The worldwide history of telecommunications. page 529, 2003.

[15] M.Kim J.H.Cheon and K.Lauter. Homomorphic computation on edit distance. *Interantional Conference on Finincial Cryptography and Data Security*, pages 194–212, 2015.

[16] A. López-Alt K. Lauter and M. Naehrig. Private computation on encrypted genomic data. *International Confrence on Cryptology and Information Security in Latin America*, pages 3–27, 2014.

[17] William August Kotas. A brief history of cryptography. 2000.

[18] Arjen K. Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.

[19] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):1–35, 2013.

[20] J.H. Cheon M. Kim, Y. Song. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC medical genomics*, page 10(2): 42, 2017.

[21] K. Lauter M. Naehrig and F.Vercauteren. Can homomorphic encrtyption be practial? *In proceedings of the 3rd ACM workshop on Cloud Computing Secrutry workshop*, pages 113–124, 2011.

[22] Daniele Micciancio. The shortest vector problem is NP-hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, March 2001. doi: 10.1137/S0097539700373039. Preliminary version in FOCS 1998.

[23] Daniele Micciancio. *Closest Vector Problem*, pages 212–214. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_399.
URL: https://doi.org/10.1007/978-1-4419-5906-5_399.

[24] Christof Paar and Jan Pelzl. Understanding cryptography: a textbook for students and practitioners. page 31, 2010.

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

[26] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 (2):120–126, 1978.

[27] Stanford. The history of cryptography.
URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/public-key-cryptography/history.html#:~:text=The%20idea%20of%20public%20key,at%20Stanford%20University%20in%201976. Accessed: 2022-11-8.

[28] A.J Han Vick. Illustration of the idea of the diffie-hellman key exchange, 2011.
URL: https://commons.wikimedia.org/wiki/File:Diffie-Hellman_Key_Exchange.svg. Accessed: 2022-11-8.