Symmetric Ciphers for Fully Homomorphic Encryption

Håkon Thorvaldsen

Master thesis at the University of Bergen, Norway

June 2020

Year: 2020
Title: FHE Master thesis at the University of Bergen
Author: Håkon Thorvaldsen

Acknowledgements

I would like to thank my supervisor Håvard Raddum for his support, and ability to simplify and make some of the hard topics included in the thesis understandable. I would also like to thank the people of Simula@UiB for being there for valuable discussions and providing an excellent workplace environment.

Abstract

Fully homomorphic encryption is the latest addition to the world of cryptography. It is a type of encryption that allows operations to be done on ciphertexts, which is not possible with traditional encryption. The field has gained a lot of traction since it was first theoretically proved possible in 2009.

This thesis goes through how fully homomorphic encryption works, from making a somewhat homomorphic encryption scheme, into a fully homomorphic scheme. We also explain in detail the different aspects required, such as bootstrapping and noise.

Since 2009 several schemes and libraries to optimize homomorphic encryption have been suggested, so that it one day may be feasible to implement it in regular modern-day applications. Some libraries target regular developers without an extensive cryptographic background, so they may still be able to use homomorphic encryption in applications, while others aim for researchers to implement and discover the possibilities that come with fully homomorphic encryption.

With a focus on the use for fully homomorphic encryption within cloud computing, this thesis focuses on how symmetric ciphers can make fully homomorphic encryption possible, also for use with small IoT devices. We look at several such ciphers that have been suggested and focus on the family of stream ciphers called Rasta. We have implemented one variant of the Rasta cipher using the software library HElib and timed its performance.

Contents

Chapter 1

Introduction

In modern human society, many parts of our lives have become digital. Most of us are walking around with a pocket computer, and WiFi signals can be found everywhere in urban areas. Given all this digitization most of our communication is thereby digital. To ensure that our communication is only read by our intended recipients and that our recipients know that the message is indeed from us, we need confidentiality, integrity, and authentication. Confidentiality ensures that the message is secret, while integrity is to ensure that there has been no tampering of the message after it was sent. Authentication enables us to know that the sender of the message is whom he or she claims to be. To enable these features of modern communication, we need cryptography.

1.1 Cryptography

Cryptography is the science of secret writing with the goal of hiding the meaning of a message [14]. **Cryptanalysis** is the science of breaking and discovering weaknesses in cryptographic security systems. Together they form the two main branches of the more general term **cryptology**. This thesis will focus on the cryptography part of the science and not on the cryptanalysis.

When we want to send a secret message to someone, we will scramble our message and provide our recipient with a *key* so that our message can be unscrambled to a readable message, which makes sense to the recipient. This scrambling and unscrambling are called encryption and decryption. It is crucial for security that the key used does not fall into the wrong hands. Because an adversary with knowledge of the cryptosystem and the key will be able to decrypt the ciphertext and read the message in its plaintext form.

In general, an encryption scheme is based on two or three algorithms. The first algorithm encrypts the plaintext into ciphertext. The second, usually closely related, decrypts the ciphertext back into plaintext. Furthermore, a third algorithm may be used, depending on the scheme, to generate different the keys used in the previous algorithms.

Different cryptosystems vary a lot, but all modern cryptosystems follow to some degree the principles that the Dutch cryptographer Auguste Kerckhoff published in the paper **Le Journal des Sciences Militaires** in 1883:

1. The System must be indecipherable, at least in practice if not mathematically.

2. The system must not be required to be secret, and must be able to fall into the hands of an enemy without inconvenience.
3. The encryption key for the system must be capable of being stored and communicated without the help of written notes, and to be changed or modified at the will of the communicating parties
4. The system must be capable of being applied to telegraph.
5. Equipment and documents of the system must be portable, and their usage and function must not require the gathering or collaboration of several people
6. Lastly, the system must be easy to use and should not be stressful or require the user to know and comply with a long list of rules.

Not all of the principles are as relevant today, of course, but especially the second principle still stands strong. The idea behind this principle is that with a publicly known system, where all details are public knowledge except for the key, the adversaries still will not be able to decipher the messages. Hence the security of the scheme should not rely on the obscurity of the cipher to provide security. Another security technique is called ”security through obscurity”, which relies on the adversary not knowing how the encryption scheme works to provide security. However, this practice is not recommended and rarely used. Today most schemes are made public before being deployed in practice so that cryptanalysts may test and discover weaknesses or flaws of a system before an adversary may.

To ease into some examples of encryption schemes, we will start by explaining a few of the classical ciphers.

1.1.1 Classical Ciphers

Humans have used cryptography to hide one’s messages from an unwanted reader for thousands of years. Maybe the most famous of the classical ciphers is the **Caesar cipher** credited to Julius Caesar(100BC-44BC) as one of the first encryption techniques. It is a very simple substitution cipher which moves the letters of an alphabet by k positions, making the number of positions k the key.

A	B	C	D	...	Y	Z
D	E	F	G	...	B	C

Figure 1.1: Table showing encryption/decryption of letters in the Latin alphabet, where the key $k = 3$.

A plaintext message will have all its characters encrypted with a right or left shift k times in the alphabet to get a corresponding ciphertext. For the recipient to decrypt the message, they would shift the characters of the ciphertext k times in the opposite direction from encryption. Hence if we used a left shift of $k = 3$ for encryption, we would use a right shift of $k = 3$ for decryption.

The biggest flaw of the cipher is its key space. The number of possible keys for the Caesar cipher is only 25 for the Latin alphabet. Shifting letters more than this is pointless since a shift of 26 is the same as not shifting at all. This leaves the cipher vulnerable to brute force attacks on the key. One could imagine trying all different 25 shifts on a small piece of the ciphertext. Then with pen and paper, it should not take too long to find the key. Another attack that the cipher is vulnerable to is the letter frequency attack. Given a language, some letters are used more frequently than others. Within a few tests, we can get a high probability of figuring out the key by comparing the most frequent letters in the ciphertext to the most frequent letters in that language.

The Caesar cipher is a substitution cipher that replaces each character in the plaintext with another in the ciphertext. Another type of cipher is the transposition ciphers. Here, instead of substituting the characters, they are instead scrambled in some smart way. A classic example of this would be the Greek Scytale. The entire message was written in one line on a parchment piece that could be rolled around a cylinder. If the cylinder had the right width, the parchment would create a meaningful text while wrapped around the cylinder. However, without the cylinder, it would be hard to determine in what order the letters should be read. So in this scheme, the width of the cylinder is considered the key k. See Figure 1.2

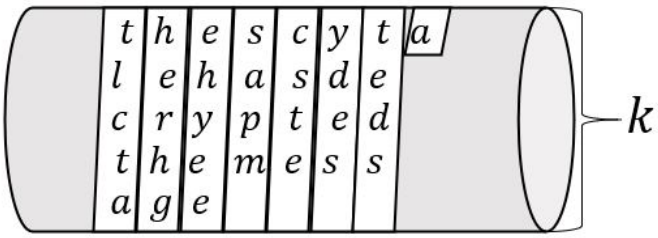

Figure 1.2: Figure showing how a message could be read of a Scytale.

1.1.2 Modern Cryptographic Primitives

All classic ciphers have in common that they use the same key for both encryption and decryption. Having one key was enough since the only goal of the classical ciphers was confidentiality. This way of encryption and decryption, where there is only one key, is called symmetric key cryptography. It was not until after World War II and the development of the modern computer that the cryptography term was expanded to include primitives such as asymmetric encryption, hash functions, and digital signatures, to mention a few. These primitives are combined to create robust encryption systems to secure modern communication.

Asymmetric cryptography, or public-key cryptography, was publicly introduced in 1976 by Whitfield Diffie, Martin Hellman, and Ralph Merkle. The revolutionary idea behind asymmetric cryptography is that the key possessed by the person who encrypts the message is not required to be secret. The crucial part is that the receiver is the only one able to decrypt the message with their secret key. For this to work, the receiver needs to have a public key so that everyone can encrypt messages and send it to the receiver. However, only the receiver can decrypt the messages using their corresponding secret key. This means that

in an asymmetric cryptography scheme, the keys are pairs of keys that share a mathematical relationship that is infeasible to solve. Hence why the public key can indeed be public.

The principle behind asymmetric encryption is called the *one-way function*. a function $f()$ is a one-way function if:

$$y = f(x) \text{ is easy to compute, and the inverse}$$
$$x = f^{-1}(y) \text{ is computationally infeasible.}$$

Two widely used one-way functions are the integer factorization problem and the discrete logarithm problem.

Asymmetric cryptography makes a wide variety of applications possible over the internet today. Some of these applications would not exist without it, such as digital signatures. In our modern world, it would not be sufficient with a single encryption algorithm if we wanted a secure system. So what we do is combine several cryptographic primitives to make more robust and secure systems. Hence primitives such as digital signatures, hash functions, MACs, and symmetric and asymmetric cryptography can be viewed as building blocks for our modern communication systems.

1.1.3 Digital Signatures

Digital signatures are widely used and an essential cryptographic tool used in the modern world of communication. Digital signature applications range from digital certificates for secure browsing to the signing of legal contracts and software updates on our computers. Digital signatures share with handwritten signatures the feature that they provide a method to assure that a message is authentic. Digital signatures must be based on asymmetric cryptography to work. The Basic idea is that the person who signs the message signs it using their private key, and the receiver verifies it using the corresponding public key.

Hash Functions

Hash functions are another widely used and an important cryptographic primitive. They compute a short, fixed-length bit string that is a unique representation of a message. In contrast to traditional cryptographic algorithms, hash functions do not have keys. They are an essential part of digital signatures and message authentication codes. However, they are also used for a wide variety of applications, such as storing password hashes or key derivation and is an essential part of the block-chain technology.

Message Authentication Codes

A message authentication code (MAC) can be viewed as a *keyed* hash function. In terms of security functionality, MACs and digital signatures share some properties. Because MACs also provide message integrity and message authentication, however a difference from digital signatures is that MACs are symmetric-key schemes and can not provide non-repudiation. Given that they are based on symmetric primitives, they are faster than digital signatures to compute.

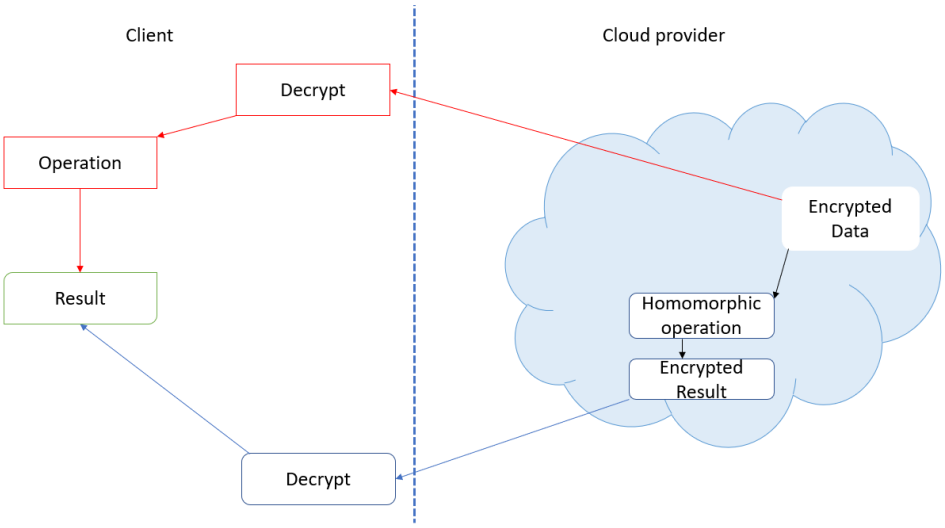

Figure 1.3: FHE allows for the cloud to operate on encrypted data.

1.1.4 Fully Homomorphic Encryption

One of the latest forms of encryption is fully homomorphic encryption (FHE). This form of encryption allows for operations to be done on encrypted data, and when decrypted, the result is as if the operations were done on unencrypted data. This allows for some exciting use cases within cloud computing. However, given we are still somewhat in the infancy of fully homomorphic encryption, it is still being developed for the future. A more detailed description of how FHE works will be described in chapter 2.

Cloud computing has been mentioned as the primary use case for FHE. In traditional cloud storage and computation solutions, the cloud has been required to have access to the unencrypted data or the decryption key to be able to do computations on the customer's data. This necessarily exposes the data to the cloud provider. Therefore data privacy relies on access controls implemented by the cloud provider, and the customers have to trust these.

With FHE we can move this trust over to cryptography. With the use of FHE the cloud provider will never have access to the unencrypted data they are storing and doing computations on. We can see in Figure 1.3 that two paths are leading to the same result. The red arrows show a path where traditional encryption has been used, and the blue path shows where FHE has been utilized. In traditional encryption (red), the client has to download and decrypt the data before doing any meaningful operations on the data. Only then can the client get their result.

Meanwhile, the blue path shows a scenario where the data has been encrypted using FHE. In this case, the cloud can do homomorphic operations on the data to get an encrypted result, and the client only needs to download and decrypt this result. Hence the client will not need to do any of the computations on the data.

Chapter 2

Fully Homomorphic Encryption

This thesis's primary focus is on fully homomorphic encryption, a form of encryption that enables computations on encrypted data. FHE allows for some exciting applications where we do not have to decrypt the data before working on it, or even better, outsource the work to a cloud without compromising privacy. There are, however, some limitations on homomorphic encryption.

The idea of FHE was first introduced in 1978 by Rivest, Adleman, and Dertouzos [17]. In their paper they discuss one of the limitations of standard encryption of data, that all encrypted data needs to be decrypted before any operations on the data can be performed. The scientists thought that a solution to the problem existed with functions that allowed for operations on data without decryption. A working solution was not found until Gentry found a solution in his 2009 paper [9]. Rivest, Adleman, and Dertouzos write about special encryption functions that would allow such operations without decryption. In their paper they are called "privacy homomorphisms", today we call them homomorphic encryption schemes.

2.1 Homomorphic Encryption

Since 1978 several schemes have been able to permit operations on encrypted data. However, the early schemes only allow for either addition or multiplication, and can not handle both. These schemes are called homomorphic encryption schemes, and allow for either addition or multiplication, such that when operations are done on encrypted ciphertext the result after decryption would be the same as if the operations had been done directly on the plaintext. For example, the unpadded version of RSA is homomorphic with regards to multiplication. The following list shows how RSA key generation works:

1. Select two large primes p and q
2. Calculate $n = p \cdot q$
3. Calculate $\phi(n) = (p-1) \cdot (q-1)$
4. Chose an e such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$
5. Find d such that $d \cdot e \equiv 1 \mod \phi(n)$

Here e is the public encryption key and d the private decryption key. The plaintext is m with corresponding ciphertext c. The public key is the pair (e, n) that anyone can use to encrypt their message and send it to the recipient. Only the recipient can decrypt with their private decryption key d. Also, p and q must be kept secret.

Encryption and decryption are done as follows:

$$\begin{aligned} \text{Encryption:} Enc(m) &\equiv m^e \equiv c \mod n \\ \text{Decryption:} Dec(c) &\equiv c^d \equiv m \mod n \end{aligned}$$

Let m_1 and m_2 be two arbitrary plaintexts, with e as the public key and d as the private key:

$$Enc(m_1) \cdot Enc(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e \mod n = Enc(m_1 \cdot m_2) \tag{2.1}$$

We can see from equation 2.1 that it does not matter whether we encrypt the plaintext or not before the multiplication, as the result would be the same. Therefore we can say that the scheme is homomorphic in terms of multiplication. Let us see what the equation looks like for addition:

$$Enc(m_1) + Enc(m_2) = m_1^e + m_2^e \neq (m_1 + m_2)^e \mod n = Enc(m_1 + m_2) \tag{2.2}$$

As we can see, equation 2.2 does not hold, so RSA is homomorphic for multiplication, but not for addition.

2.2 Fully Homomorphic Encryption

In 2009 Craig Gentry presented the first fully homomorphic encryption scheme that worked for both addition and multiplication [9], using a lattice-based scheme. Since the scheme could handle both addition and multiplication, it follows that any operation on the ciphertext would be the same after decryption as if the operations had been done on the plaintext. This trait of FHE gives some very interesting use cases where security and privacy are of importance, like cloud computing.

There was still a significant problem in his scheme. The encryption was very inefficient and therefore had very little practical use. Since 2009 there have been many improvements within FHE, such as the BGV cryptosystem, which we will describe in detail in a later section. The BGV system still has performance issues, but it is closer to a system that has practical use cases and could bring FHE closer to commercial use. In 2013 the system was implemented in Halevi and Shoup's homomorphic encryption library HElib [11].

Users of cloud computing services may have sensitive data that they are reluctant to share in plaintext with the cloud, such as medical information about patients or business-sensitive data. The classic approach here would be to encrypt the data before sending it to the cloud, but this would not make much sense since it would have to be redownloaded and decrypted before one could operate on the data. Another approach would be to give the cloud access to the decryption key. The cloud could then operate on the client's data, but this is only a safe solution if the client trusts the cloud provider. Here FHE provides an additional option. If the cloud can operate on the data without decrypting it, one would not compromise security.

The cloud provider does not know what the ciphertext in its possession represents. Therefore FHE will remove the constraint of having to trust the cloud provider, while still being able to utilize a cloud's high hardware power to operate on one's data.

Gentry's 2009 scheme starts with a somewhat homomorphic encryption scheme. An encryption scheme being somewhat homomorphic means that a limited amount of operations can be done on the data while still being decrypted correctly. After this limit has been reached, the decryption would become incorrect. Some schemes are called leveled homomorphic encryption schemes. These are also somewhat homomorphic, but the limit on the number of operations is a predetermined parameter. Hence a leveled homomorphic encryption scheme can, in principle, evaluate arbitrary circuits, but the circuit size must be known beforehand. In contrast, a fully homomorphic scheme can evaluate arbitrary circuits without needing to know the size of the circuits at the time of system setup.

2.2.1 Noise

The noise of a ciphertext is a random element added while encrypting, and is needed for security in all existing schemes. This value increases for every operation done on the ciphertext, and if it becomes too big, decryption will no longer be correct. Hence why the somewhat homomorphic encryption scheme of Gentry will not be able to decrypt correctly after a finite number of operations. To illustrate this, let us look at a simple scheme over the integers from [19, p. 50–51], which is homomorphic for both multiplication and addition:

1. Pick an odd number k as the private key
2. Pick a random q and r, where $r \ll \sqrt{k}$

The plaintext space in this scheme is only $\{0, 1\}$, and the ciphertext c corresponding to a message m is encrypted and decrypted as follows.

$$\textbf{Encryption: } Enc(m) = c = kq + 2r + m$$
$$\textbf{Decryption: } Dec(c) \equiv m \equiv (c \mod k) \mod 2$$

In decryption we reduce the ciphertext modulo k, leaving the remainder $2r + m$, as long as $2r + m < k$. The reason for choosing $r \ll \sqrt{k}$ will become clear below. After reduction modulo k, the result is again reduced mod 2, leaving only the plaintext, which means that the ciphertext has been correctly decrypted into the corresponding plaintext. However, this is not true if the noise of the ciphertext is too great. The noise is represented by $2r$ in the equation. So if the noise is greater than k, reducing modulo k would not necessarily leave us with the remainder $2r + m$. Thus, making decryption impossible.

We now look at why the scheme is homomorphic with respect to both addition and multiplication. At the same time, we will learn how the noise grows when using the two operations. Let m_0 and m_1 be two plaintext bits with corresponding ciphertexts c_0 and c_1:

$$c_0 = kq_0 + 2r_0 + m_0$$
$$c_1 = kq_1 + 2r_1 + m_1$$

Addition:

$$c_0 + c_1 = k(q_0 + q_1) + 2(r_0 + r_1) + (m_0 + m_1) \tag{2.3}$$

Multiplication:

$$c_0 \cdot c_1 = k(kq_0q_1 + 2r_1q_0 + m_1q_0 + 2r_0q_1 + m_0q_1) + 2(2r_0r_1 + m_1r_0 + m_0r_1) + m_0m_1 \tag{2.4}$$

When:

$$r_0 + r_1 < k/2$$
$$2r_0r_1 + m_0r_1 + m_1r_0 < k/2$$

We can see from the addition of c_0 and c_1 that the sum of the ciphertexts correctly encrypts the sum of the plaintext bits m_0 and m_1. The same is true for the multiplication, where the product of the ciphertexts correctly encrypts the product of the plaintext bits. Therefore, we can conclude that this scheme has the homomorphic properties required. When we are looking at the noise of the scheme, we can notice a difference in the growth of the noise from the two operations. The term $2(r_0 + r_1)$ represents the noise added during addition. The noise term for multiplication is $2(2r_0r_1 + m_1r_0 + m_0r_1)$. We see that multiplication is far more expensive in the terms of growth of noise. Since the noise terms for addition are only added together, we will get linear growth of noise. After n additions, the noise would be approximately $n \cdot r$. For multiplication the growth is exponential, and after n multiplications, we get a noise term of approximately r^n.

2.2.2 Bootstrapping

In order to obtain the fully homomorphic property, schemes must be able to reduce the noise component that compounds with every addition and multiplication. Bootstrapping is a method in which the scheme runs its decryption circuit to remove the current noise while adding some new noise from the decryption circuit itself. For this to be fully homomorphic, the new noise must be smaller than the noise removed to retain the fully homomorphic properties. Gentry proposed his bootstrapping technique as a solution to this problem. By taking the decryption algorithm of the encryption scheme and turning it into a circuit with the ciphertext and the encryption of the private key as input, the outcome of the circuit would be a re-encryption of the ciphertext, see Figure 2.1. If this decryption circuit is cheap enough to evaluate, then our new ciphertext will have less noise than the original. Hence, allowing more operations on the ciphertexts to be done, giving us the fully homomorphic property. It is then possible to evaluate any circuit, running the bootstrapping procedure whenever necessary in the computation.

2.3 Learning With Errors and Ring Learning With Errors

The schemes we will look at later are based upon the Learning with errors (LWE) problem, which is a quantum robust method in cryptography, meaning that we could avoid quantum computers from breaking public-key cryptosystems with this method [10]. The problem can be stated as follows: For the ring $\mathbb{Z}_q^n$, given a secret vector s, we can calculate as many inner products $a \cdot s = b$ as we want, but we will only learn the value of b distorted by a value e. Here

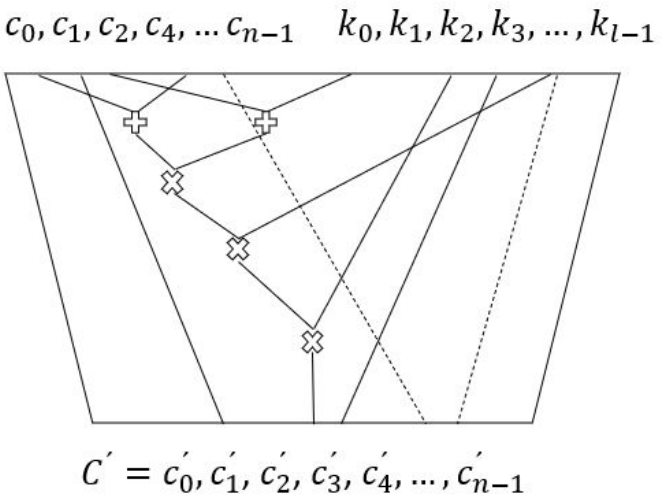

Figure 2.1: An arbitrary decryption circuit with encryptions of the bits of the ciphertext and corresponding key as input. The output is a re-encryption of the ciphertext, with less noise than the input.

e is a small error value that is unknown but taken from a known distribution. The problem comes down to finding s from the known a and $b + e$ values.

$$\begin{aligned} \mathbf{a} &\in \mathbb{Z}_q^n \\ (a_{11}, a_{12}, \ldots, a_{1n}) \cdot (s_1, s_2, \ldots, s_n) &= b_1 + e_1 \\ (a_{21}, a_{22}, \ldots, a_{2n}) \cdot (s_1, s_2, \ldots, s_n) &= b_2 + e_2 \\ &\vdots \\ (a_{N1}, a_{N2}, \ldots, a_{Nn}) \cdot (s_1, s_2, \ldots, s_n) &= b_N + e_N \end{aligned}$$

The problem can be set up as matrix multiplication where matrix A and vector $\mathbf{b}$ are known, as in equation 2.5.

$$\mathbf{b} = A \cdot \mathbf{s} - \mathbf{e} \tag{2.5}$$

If there was no error added to the equation, the problem could easily be solved using Gaussian elimination. Also, the error values e must be quite small as this will make sure that the problem has a unique solution for s, given enough samples. Hence the added error is what makes the problem hard. Defined by Oded Regev defined in 2005 [16].

To expand LWE into Ring-LWE, we use polynomials instead of vectors of integers. The ring we are working with changes from $\mathbb{Z}_q^n$ to $\mathbb{Z}_q[x]/(x^d + 1)$ and the vectors now become polynomials.

$$a(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_1 x + a_0$$

Hence the secret vector s from LWE is a secret polynomial $s(x)$ in RLWE. Instead of finding the inner products, we now need to use polynomial multiplication. The error polynomials $e(x)$ are taken from a special distribution where the roots in the complex numbers are small. RLWE can then be given as follows:

$$a_1(x) \cdot s(x) = b_1(x) + e_1(x)$$
$$\vdots$$
$$a_N(x) \cdot s(x) = b_N(x) + e_N(x)$$

The problem is still to find the secret polynomial $s(x)$, given many samples of $a_i(x) \cdot s(x)$ distorted by $e_i(x)$.

2.4 FHE Schemes

Since Gentry's original scheme from 2009, there have been proposed new schemes which are considerably more efficient than Gentry's original. Several of these have been implemented and tested in various libraries. We are going to focus on some of the more popular ones.

2.4.1 BGV Scheme

The BGV scheme is named after Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan [10], who published their scheme in 2011. The BGV scheme is implemented in HElib and is a leveled FHE scheme without bootstrapping. Table 2.1 is a description of the user-defined parameters needed to initialize a BGV scheme in HElib. The BGV scheme is based on the RLWE problem mentioned in the previous section.

Description	Parameter
The specific modulus	m
Plaintext base [default=2]	p
Hensel lifting [default=1]	r
Number of bits of the modulus chain	$bits$
Number of columns in the key switching matrices	c

Table 2.1: User defined parameters of the BGV scheme.

These user defined parameters will define the scheme's other parameters, such as the security level, the number of slots s and the size of each element in a slot d. We will only focus on the field $\mathbb{F}_2$, as the plaintext space only contains the binary values 0 and 1. This means that the user defined parameter r will always be 1, and p will always be 2.

Slots

In HElib and the BGV scheme the plaintext bits are stored in an array, whose size is determined by the user defined parameters. The size of this array is the number of slots in the instantiation of the scheme. It is determined by equation 2.6:

$$s = \frac{\phi(m)}{d}, \tag{2.6}$$

where ϕ is Euler's ϕ function.

Every position in the array represents a slot, and the elements in each slot can be from the finite field $\mathbb{F}_{2^d}$. An array is used to store the plaintext bits because BGV supports Single Instruction Multiple Data (SIMD) operations. SIMD allows for encryption on multiple plaintext bits in a ciphertext object, where the number of encrypted bits in the ciphertext object is the number of slots. We will use the notation:

$$C = \{(p_1, p_2, ..., p_s)\}$$

To illustrate that our ciphertext object C encrypts the plaintext bits $p_1, p_2..., p_s$.

The homomorphic operations are then computed slot wise in the BGV scheme, meaning that given two ciphertext objects:

$$C_1 = \{(a_1, a_2, .., a_s)\}$$
$$C_2 = \{(b_1, b_2, .., b_s)\}$$

Addition and multiplication homomorphically will be:

$$C_1 + C_2 = \{(a_1 \oplus b_1, ..., a_s \oplus b_s)\}$$

$$C_1 \times C_2 = \{(a_1 \otimes b_1, ..., a_s \otimes b_s)\}$$

Here $\oplus$ is the regular bitwise XOR operation and $\otimes$ is the AND bitwise operation. By combining the slots property and the slot-wise operations, we can use ciphertext objects to encrypt multiple bits and perform several operations simultaneously. This is desired as without this property one would have to encrypt each plaintext bit $a_1, a_2, .., a_s$ and $b_1, b_2, ..., b_s$ into their own ciphertext objects. In BGV one can encrypt them into only two ciphertext objects and can add and multiply homomorphically on them using only a single operation $C_1 + C_2$ or $C_1 \times C_2$. This property is advantageous when implementing a decryption circuit homomorphically since operations done during decryption are usually simple and repeated multiple times. Therefore it is easy to take advantage of the parallelism provided in the BGV scheme.

2.4.2 CKKS Scheme

The CKKS [5] scheme was published in 2016 by its authors Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song, hereby the abbreviation CKKS. They all have an affiliation with the Seoul National University in the Republic of Korea. The CKKS scheme is based upon the BGV scheme, but it supports approximate arithmetic over complex numbers, and in particular real numbers, as opposed to the BGV scheme which only works over the integers. A compelling use case for the scheme are cases where the data is not exact or where data is being sampled and discretized to an approximate value.

The idea behind using approximate arithmetic in the scheme is to allow for approximate computations within HE. The main idea is that during approximate computations, we can treat the encryption noise as part of the errors occurring in said computations.

A ciphertext c of a plaintext m will have the following decryption structure, with the secret key sk:

$$\langle c, sk \rangle = m + e (mod\ q) \tag{2.7}$$

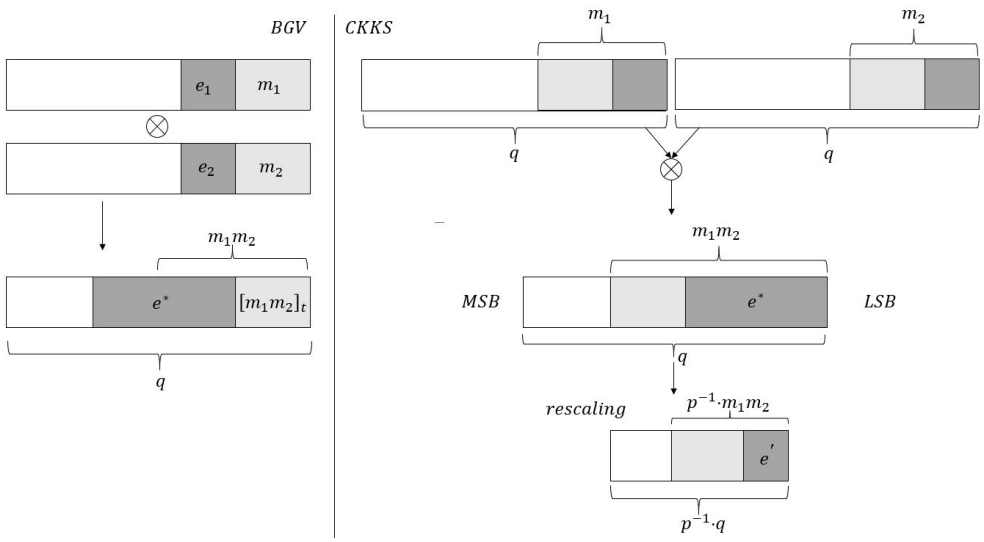

Figure 2.2: HE multiplication in BGV. HE multiplication and rescaling in CKKS.

In equation 2.7 e is a small error inserted to guarantee the security based on RLWE. Given that e is small enough in comparison to the message, this noise is not likely to destroy the significant figures of m, and a new value $m' = m + e$ can replace the original message m for approximate arithmetic. Hence m' is not exactly the same value as m, but since the error e is small enough, m' is close enough to the original value.

For homomorphic operations, the error in decryption is maintained small compared to the ciphertext modulus, such that the computational result is still smaller than q. However, there is still a problem with bit sizes of messages increasing exponentially with the depth of a circuit without rounding. The researchers behind CKKS, suggest a solution to this problem that they call *rescaling*, which manipulates the message in a ciphertext. For a ciphertext c of a plaintext m where $\langle c, sk \rangle = m + e (mod\ q)$, the rescaling procedure outputs a ciphertext $\lfloor p^{-1} \cdot c \rceil (mod\ q/p)$ which is an encryption of m/p where the noise has size e/p. This procedure reduces the size of the ciphertext modulus and, as a result, removes the error located in the least significant bits of the message. At the same time the precision of the plaintext is almost preserved. Figure 2.2 illustrates the procedure and the difference between BGV and CKKS. This composition of homomorphic operations and rescaling mimics regular approximate arithmetic. As a result, the required ciphertext modulus bit size grows only linearly with the depth of a circuit, as opposed to exponential.

The CKKS scheme also uses the same technique as the BGV scheme when it comes to encrypting multiple messages in a single ciphertext and parallel processing using SIMD. BGV uses a ring of finite characteristic as the plaintext space, with a small error located in a separated place in the ciphertext modulus inserted for security. In the BGV scheme this error can be removed after homomorphic operations. Meanwhile, in the CKKSscheme a plaintext is a polynomial contained in a cyclotomic ring of characteristic zero, and the small error resides in the LSBs. This error can not be removed after decryption, but the decrypted value will be close to the original message.

With the scheme's ability to handle real numbers, there have been suggestions to apply it for data science and machine learning.

2.5 FHE Libraries

Today several libraries can be used for research within FHE. HElib was the first widely used open-source library, but today we have some alternatives in Microsoft's SEAL and PALISADE.

2.5.1 HElib

HElib is an open-source software library that implements homomorphic encryption, launched in 2013. HElib implements both the BGV and CKKS schemes mentioned previously. It can be viewed as an assembly language for Homomorphic encryption because most of the library is relatively low level, meaning that it provides low-level routines such as set, multiply, add, shift, etc. It is written in C++ and builds on the Number Theory Library (NTL) and Gnu MultiPrecicion Library (GMP). HElib is distributed with the Apache 2.0 license.

HElib uses leveled HE schemes, meaning that the parameters need to be predetermined by the user, as mentioned in Section 2.4.1 about BGV. Due to the low-level approach in HElib, it should be considered mostly as a tool for researchers, but they claim to hopefully be able to provide higher-leveled features in the future. HElib initially only implemented the BGV scheme, before including support for the CKKS scheme.

2.5.2 Microsoft SEAL

Microsoft SEAL(Simple Encrypted Arithmetic Library) [18] is an easy to use open-source (MIT-licensed) homomorphic encryption library developed by the Cryptography and Privacy research group at Microsoft. Microsoft SEAL has support for the CKKS scheme and the BFV scheme. The latter is more closely related to the BGV scheme as it also works on exact values. Microsoft SEAL is written in C++ and should be easy to run and compile in many different environments. They provide easy to understand code examples for software engineers to enable them to build end-to-end encrypted data storage and computation services.

The developers goal was to make FHE available to everyone and not just the researchers within the cryptographic field, due to the many complicated mathematical properties of cryptography and FHE. They claim to provide a simple and convenient API with state of the art performance. They also provide several examples of code snippets to teach the user how to use the library correctly and securely.

2.5.3 PALISADE

The PALISADE homomorphic encryption software library [15] is a project intended to provide an efficient implementation of lattice cryptography building blocks and leading homomorphic encryption schemes. PALISADE is designed to provide usability, simpler APIs, modularity and cross-platform support. The library is offered under the 2-clause BSD open-source license. PALISADE supports the BGV and CKKS schemes previously mentioned, but also the BFV-scheme. The library is implemented in C++. The project leaders all have an affiliation to the New Jersey Institute of Technology, but the project has expanded with a large community of contributors.

The architecture of PALISADE is divided into several layers where each layer provides a service to the layer ”above” or makes use of the services provided by the layer ”below”. The stack is divided into the following layers:

1. **Application:** All programs that call the PALISADE library services are in this layer.
2. **Encoding:** Implementations of methods to encode data.
3. **Crypto:** Implementations of cryptographic protocols.
4. **Lattice Operations:** Higher-level lattice-crypto mathematical building blocks.
5. **Primitive Math:** Low-level generic mathematical operations, such as multi-precision arithmetic implementations.

Chapter 3

Symmetric Ciphers Designed for FHE

In recent years researchers have been developing solutions based on symmetric cryptography to ease the heavy burden of HE operations. The heavy operations are outsourced to the cloud, and weaker devices can run cheap symmetric ciphers while still being able to utilize FHE. In this chapter we are going to take a look at how this is done.

3.1 Use Case Scenario

The schemes previously mentioned in Chapter 2 are quite expensive to run, due to several factors such as ciphertext expansion from operating on the data. The ciphertext expansion could be from thousand-fold to million-fold, which obviously will have a considerable effect on the efficiency of schemes.

Small devices used in the Internet of Things (IoT) will generally not be able to run FHE schemes on their own. However, with the use of symmetric cryptography, they might be able to run a symmetric cipher, which is computationally much cheaper. Symmetric encryption has proved to give strong encryption with lightweight ciphers even for low processing power devices.

The use case is such that a device will run a symmetric cipher to encrypt the plaintext P using key K into the ciphertext C without FHE. The ciphertext C is sent to the cloud where the cloud re-encrypts the bits in C using FHE to create C^*.

At system setup, the bits in K have been encrypted with FHE and stored in the cloud as K^*. The heavy operation of FHE-encrypting the key K does not need to be done by the same device that symmetrically encrypts the plaintext without the use of FHE. Both C^* and K^* are encrypted using the same public key. With C^* and K^* the cloud can run the homomorphic decryption circuit of the symmetric cipher, resulting in data encrypted only by FHE. Since the cloud now has access to FHE-only encrypted data, it can do operations on the client's data. Figure 3.1 illustrates the process.

Since noise gets added during the homomorphic decryption, we need to find symmetric ciphers that are cheap and efficient for FHE purposes. A scheme that needs to run bootstrapping for evaluating the decryption circuit will not be useful. Instead, the decryption circuit must have low multiplicative depth. In recent years it has been a research challenge to find such symmetric ciphers that use few AND gates but are still secure.

In this section we will look at several proposed symmetric ciphers intended to be used for symmetric FHE encryption.

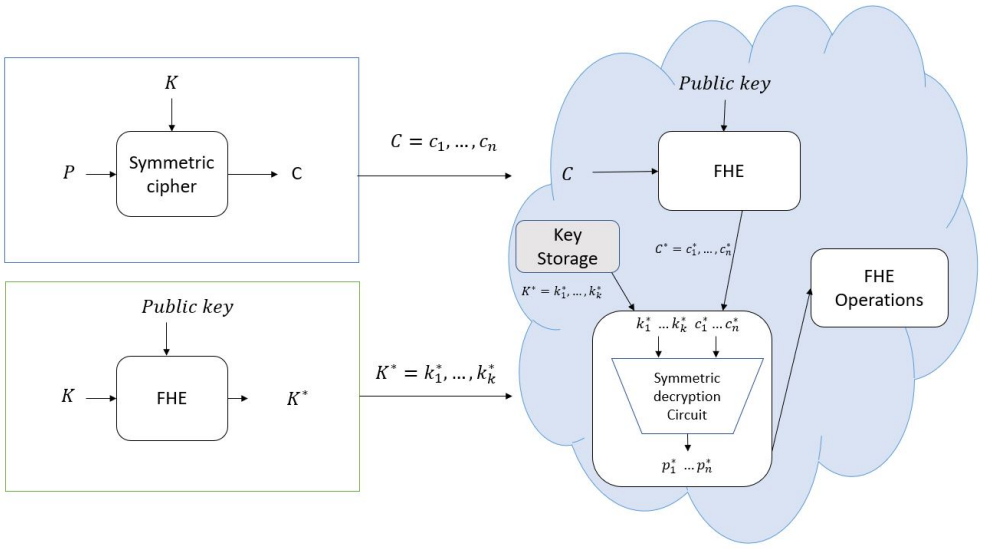

Figure 3.1: FHE cloud scenario with symmetric cipher.

3.2 Designing Ciphers for use in FHE

When designing ciphers for use in FHE the decryption circuit of the cipher needs to be homomorphically evaluated. With the use of the noise-based cryptography in FHE schemes, every operation on homomorphically encrypted data increases the noise. In most schemes the noise grows fast with the increase of the multiplicative depth of the circuit. Hence the usual design strategy for these ciphers has been to reduce the multiplicative depth of the decryption circuit. In the following sections we will look at some symmetric ciphers which aim to make FHE with symmetric ciphers feasible.

3.3 FLIP

FLIP is a family of filter permutators designed as stream ciphers to allow constant and small(er) noise compared to previous block ciphers and stream ciphers. Block ciphers typically have been allowing constant but small homomorphic capacity, due to a relatively large number of rounds leading to significant noise from complex Boolean functions. Stream ciphers typically allow for large homomorphic capacity on the first ciphertext blocks. However, this capacity decreases with the number of ciphertext blocks due to the increasing Boolean complexity in stream ciphers' output. The idea is to combine the best of both worlds in a stream cipher, where the Boolean filter function is applied to a public bit permutation of a constant key register. In this way the Boolean complexity of the key stream that the cipher outputs are constant [13].

The general structure of the filter permutators consists of three parts: A register where the N-bit key K is stored, a bit permutation generator that is parameterized by a pseudo-random number generator (PRNG) initialized with a public IV. The permutation generator produces an N-bit permutation P_i that shuffles the bits right before entering the filtering function F. Lastly, the filtering function produces the key stream bits z_i, see Figure 3.2. The filter permutators' structure can be compared to a filter generator where a permuted key register replaces the LFSR. The register is no longer updated by an LFSR but by pseudo-

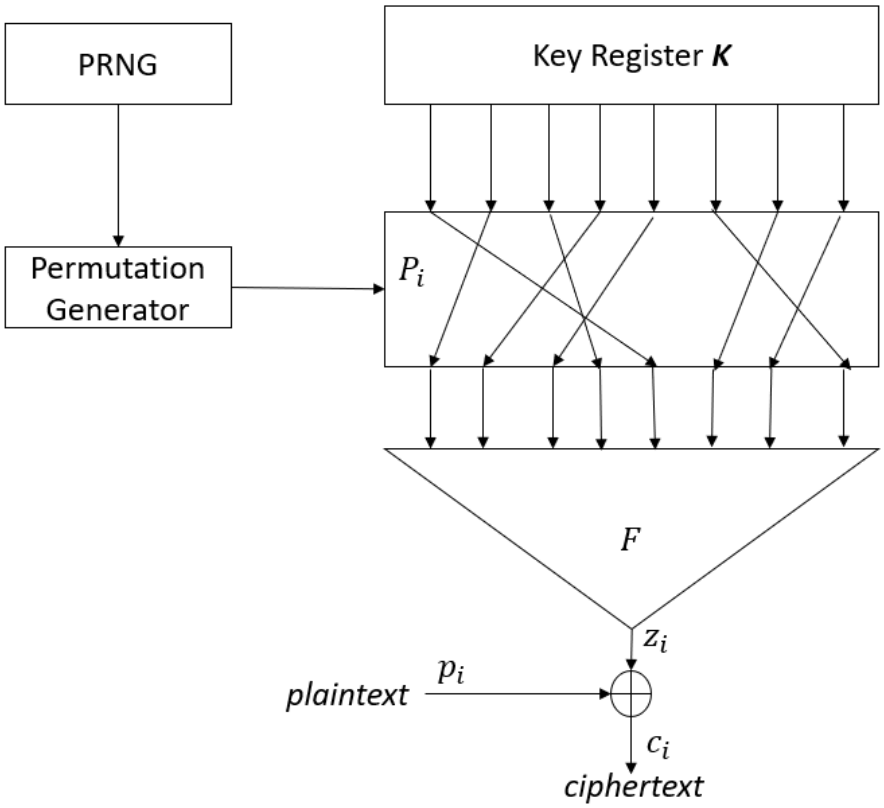

Figure 3.2: FLIP, filter permutator basic structure.

random bit permutations. More precisely, at each cycle or bit-output from the filter register, a pseudo-random permutation is applied to the register, and the permuted key register is filtered. Finally, encryption with a filter permutator consists of XORing the key stream bits z_i output by the filtering function with the plaintext bits p_i, to produce the ciphertext bits c_i [8]. To recover the plaintext, the same procedure is simply repeated, and the key stream bits z_i are XORed to the ciphertext bits c_i to recover the plaintext p_i. Every part of the scheme is public except for the key.

In [13] the authors of the paper specify FLIP as a family of stream ciphers using a feed-forward secure PRNG based on AES-128. The Knuth Shuffle as the bit permutation generator [12] and the filter function F is the N-variable Boolean function defined by the direct sum of three Boolean functions f_1, f_2, f_3 of respectively n_1, n_2, n_3 variables, defined as follows:

$$f_1(x_0, \ldots, x_{n_1-1}) = L_{n_1}$$
$$f_2(x_{n_1}, \ldots, x_{n_1+n_2-1}) = Q_{\frac{n_2}{2}}$$
$$f_3(x_{n_1+n_2,\ldots,x_{n_1+n_2+n_3-1}}) = T_k$$

$$f(x_0, \ldots, x_{n_1+n_2+n_3-1}) = f_1(x_0, \ldots, x_{n_1-1}) + f_2(x_{n_1}, \ldots x_{n_1+n_2-1}) + f_3(x_{n_1+n_2,\ldots,x_{n_1+n_2+n_3-1}})$$

Here L_n is the linear function in n variables defined as:

$$L_n(x_0, \ldots, x_{n-1}) = \sum_{i=0}^{n-1} x_i$$

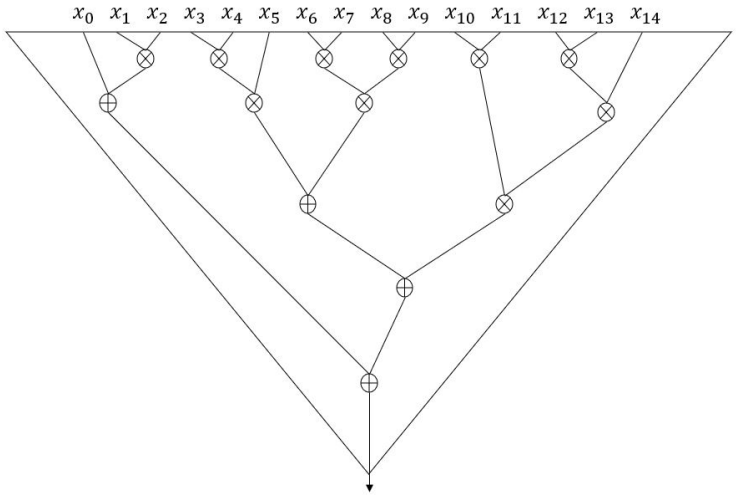

Figure 3.3: The circuit for the triangular function T with $k = 5$, with 15 variables.

Q_n is a quadratic function in $2n$ variables defined as:

$$Q_n(x_0, \dots x_{2n-1}) = \sum_{i=0}^{n-1} x_{2i}x_{2i+1}.$$

The k'th triangular function T_k is a $\frac{k(k+1)}{2}$-variable Boolean function defined as

$$T_k(x_0, \dots, x_{\frac{k(k+1)}{2}-1}) = \sum_{i=1}^{k} \prod_{j=0}^{i-1} x_{j+\sum_{\ell=0}^{i-1}\ell}$$

When evaluating these functions L, Q, T as a circuit, we can see how the circuit's multiplicative depth will be kept quite low. L is linear and has no multiplications at all. Q is quadratic and thus has multiplication depth 1. The triangular function T_k has degree k and is the heaviest one, but its multiplicative depth is only $\lceil \log_2 k \rceil$. For example, the 5'th triangular function T_5 has 15 variables and multiplicative depth $\lceil \log_2 5 \rceil = 3$, which we can see from Figure 3.3.

3.4 Kreyvium

Kreyvium [4] is a symmetric stream cipher published in 2018, based on the cipher Trivium [6]. Trivium is also a symmetric stream cipher which was recommended by the eSTREAM project after a five year competition. Due to the small number of non-linear operations in its transition function, they claim it would be a natural candidate in the HE context. Both Kreyvium and Trivium can be decomposed into:

- The resynchronization function, Sync, which takes as input the IV and the key (possibly expanded by some precomputation phase), and outputs some n-bit initial state.
- The transition function Φ, which computes the next state of the generator
- The filtering function f, which computes a key stream segment from the current internal state

Both the key and IV of Trivium consists of 80 bits each. The internal state of Trivium consists of 3 registers of sizes $93, 84$, and 111 bits, respectively, making the entire internal state 288 bits. The notion of the designers is that the leftmost bit in the 93-bit register is s_1, and the rightmost one is s_{93}. Of the second register of size 84 the leftmost bit is s_{94} and rightmost s_{177}. The last register of size 111 bits the leftmost bit is s_{178} and the rightmost bit is s_{288}. The initialization and generation of an N-bit key stream can be described by Algorithm 3.4.

Algorithm 1 Trivium pseudo code

$(s_1, s_2, \ldots, s_{93}) \leftarrow (K_0, \ldots, K_{79}, 0, \ldots, 0)$
$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (IV_0, \ldots, IV_{79}, 0, \ldots 0)$
$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$
for $i = 1$ to $1152 + N$ **do**
  $t_1 \leftarrow s_{66} + s_{93}$
  $t_2 \leftarrow s_{162} + s_{177}$
  $t_3 \leftarrow s_{243} + s_{288}$
  if $i > 1152$ **then**
    output $z_{i-1152} \leftarrow t_1 + t_2 + t_3$
  end if
  $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
  $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
  $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$
  $(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
  $(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
  $(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
end for

The multiplicative depth grows quite slowly with the number of iterations. Another important observation is that in the internal state, only the first 80 bits in the first register (the key bits) are initially encrypted under HE. Consequently, performing hybrid clear text and encrypted data calculations is possible. With the square brackets denoting encrypted bits, this is done under the following quite simple rules:

$$\begin{aligned} 0 \cdot [x] &= 0 \\ 1 \cdot [x] &= [x] \\ 0 + [x] &= [x] \\ 1 + [x] &= [1] + [x] \end{aligned}$$

In all but the last case, a homomorphic operation is avoided, which is very desirable. This is an optimization that allows for an increase in the number of bits that can be generated at a given multiplicative depth. The relevant quantity in Trivium's context is the multiplicative depth of the circuit, which computes N key stream bits from the 80-bit key.

In Trivium, the key stream length $N(d)$ which can be produced from an 80-bit key with a circuit of multiplicative depth $d \geq 4$ is given by 3.1.

$$N(d) = 283 \times \lfloor \frac{d}{3} \rfloor + \begin{cases} 81 & \text{if } d \equiv 0 \mod 3 \\ 160 & \text{if } d \equiv 1 \mod 3 \\ 269 & \text{if } d \equiv 2 \mod 3 \end{cases} \tag{3.1}$$

The Kreyvium cipher is very similar to Trivium, except that it accommodates key and IV sizes of 128 bits instead of 80. It also has a bigger internal state, where two 128-bit registers have been added to the original 288-bit internal state of Trivium. These extra states hold the 128-bit secret key and IV, respectively. This newly added part of the state aims to make both the filtering and transition functions dependent on both key and IV. More precisely, the two functions f and Φ depend on the key and IV bits through the successive outputs of the two shift-registers K^* and IV^* initialized by key and IV, respectively. The internal state of Kreyvium is then composed of five registers of sizes: $93, 84, 111, 128$, and 128 bits. In total the internal state has a size of 544 bits. We use the same notation for Kreyvium as we did for Trivium, with the added registers' bits denoted as K^*_{127} for the leftmost bit and K^*_0 for the rightmost one (the output). Likewise, for the IV where IV^*_{127} is the leftmost bit and IV^*_0 is the rightmost one (the output). Both of these two added registers rotate independently from the rest of the "Trivium" part of the cipher. The initialization and generation of an N-bit key stream are described in Algorithm 3.4.

Algorithm 2 Kreyvium pseudocode

$(s_1, s_2, ..., s_{93}) \leftarrow (K_0, ..., K_{92})$
$(s_{94}, s_{95}, ..., s_{177}) \leftarrow (IV_0, ..., IV_{83})$
$(s_{178}, s_{179}, ..., s_{288}) \leftarrow (IV_{84}, ..., IV_{127}, 1...1, 0)$
$(K^*_{127}, K^*_{126}, ..., K^*_0) \leftarrow (K_0, ..., K_{127})$
$(IV^*_{127}, IV^*_{126}, ..., IV^*_0) \leftarrow (IV_0, ..., IV_{127})$
for $i = 0$ to $1152 + N$ **do**
  $t_1 \leftarrow s_{66} + s_{93}$
  $t_2 \leftarrow s_{162} + s_{177}$
  $t_3 \leftarrow s_{243} + s_{288} + K^*_0$
  if $i > 1152$ **then**
    output $z_{i-1152} \leftarrow t_1 + t_2 + t_3$
  end if
  $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} + IV^*_0$
  $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
  $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$
  $t_4 \leftarrow K^*_0$
  $t_5 \leftarrow IV^*_0$
  $(s_1, s_2, ..., s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$
  $(s_{94}, s_{95}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$
  $(s_{178}, s_{179}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$
  $(K^*_{127}, K^*_{126}, ..., K^*_0 \leftarrow (t_4, K^*_{127}, ..., K^*_1)$
  $(IV^*_{127}, IV^*_{126}, ..., IV^*_0) \leftarrow (t_5, IV^*_{127}, ..., IV^*_1)$
end for

As with Trivium, the number of key stream bits generated from a key at a given depth can be computed. The only difference is that the first register now contains 93 bits instead of the 80 from Trivium. For a fixed depth $d \geq 4$, $N(d)$ is given by:

$$N(d) = 282 \times \lfloor \frac{d}{3} \rfloor + \begin{cases} 70 & \text{if } d \equiv 0 \ mod\ 3 \\ 149 & \text{if } d \equiv 1 \ mod\ 3 \\ 258 & \text{if } d \equiv 2 \ mod\ 3 \end{cases} \tag{3.2}$$

3.5 LowMC

LowMC is a family of block ciphers designed with a focus on providing a symmetric-key primitive with low multiplicative depth and size [2]. The design is motivated for use with secure multi-party-computation (MPC), fully homomorphic encryption (FHE), and zero-knowledge proofs (ZK), where the cost of doing linear operations can be considered virtually free, in comparison to non-linear computations. The different variants of LowMC differ by choice of the parameters block size n, key size k, and the number of S-boxes per layer, denoted by m. They all start with key whitening before the encryption rounds. The rounds are built the same way for every variant of LowMC. The encryption rounds consist of an S-box layer, an affine layer and a key addition.

3.5.1 S-box of LowMC

An unusual attribute of the S-box layer of the LowMC ciphers is that the S-boxes do not have to cover the entire cipher block. So the number of S-boxes m in the S-box layer can differ for various variants of the LowMC ciphers. There is a trade-off between the number of S-boxes and the number of rounds due to security reasons. In the S-box layer, the cipher block goes through m S-boxes with 3 bits per S-box, where bits not going through an S-box remain unchanged. The S-box contains three multiplications, with a single multiplication in each output:

$$\begin{aligned} S_0(A,B,C) &= A \oplus B \cdot C \\ S_1(A,B,C) &= A \oplus B \oplus A \cdot C \\ S_2(A,B,C) &= A \oplus B \oplus C \oplus A \cdot B \end{aligned} \tag{3.3}$$

3.5.2 Linear Layer

In the linear layer of the cipher, each block is multiplied by an $n \times n$ matrix, with different matrices for each round. The only restriction of the matrices is that each matrix has to be invertible for the decryption phase to work. The matrices should be chosen at random or be generated using the key stream from the Grain stream cipher [2].

Constant and Key addition

In the encryption phase a constant binary vector of length n is added to the cipher block in each round. These constants should also be chosen at random. As for key addition, a subkey of length n will be added to the cipher block at the end of each round. These subkeys are constructed from the master key of size k by multiplying the master key by random $n \times k$ matrices.

LowMC Variants

Table 3.5.2 show the parameters of some of the suggested variants of the LowMC cipher. Several more variants were suggested, but these five were chosen by Martin Albrecht in his

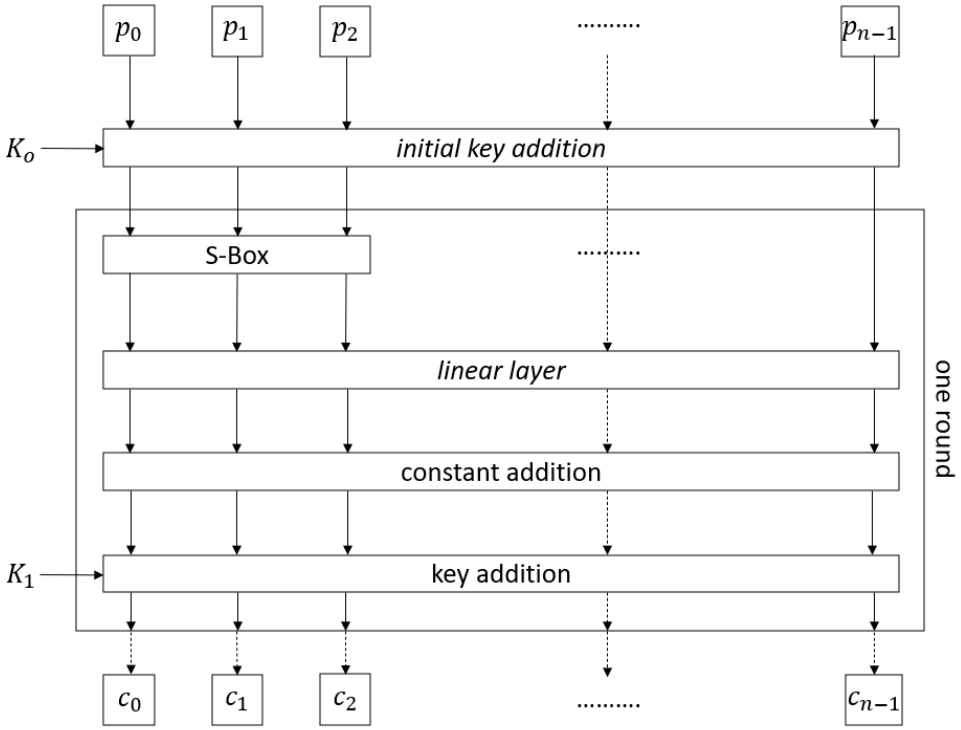

Figure 3.4: Figure showing overall structure of a LowMC cipher.

HElib implementation of the LowMC ciphers. The two first variants aimed for 80-bit security. The third and fourth for 128-bit security parameters, and the last for a 256-bit security level [1].

Block size n	number of s-box, m	Key size, k	# of rounds r
256	49	80	12
128	31	80	12
256	63	128	14
196	63	128	14
512	66	256	18

Table 3.1: Some variants of the LowMC cipher.

3.6 Rasta

Rasta [7] is a family of stream ciphers based on a design strategy for symmetric encryption where the AND depth d is low, while only needing d AND gates per encrypted bit. To produce a key stream, Rasta applies a permutation with feed-forward, where the input to the permutation is the secret key K, and the key size matches the block size n of the underlying family of permutation $P_{N,i}$. The key stream is generated by using different permutations $P_{N,i}$

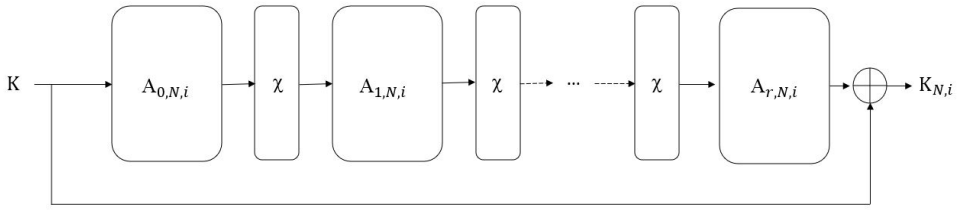

Figure 3.5: Structure of the Rasta cipher.

for each encrypted block, which is parameterized by a nonce N and the block counter i. To ensure confidentiality for each new encryption a new and unique nonce N is required. The original key K is added to the output after the rounds of an instance of the Rasta cipher. Thus the resulting key block is the key stream produced by the cipher. See Figure 3.5 for the overall structure of Rasta. Note that the authors of Rasta consider the security parameters of Rasta quite conservative and the suggested parameters could potentially be reduced while still maintaining adequate security.

3.6.1 Permutation

Rasta's family of permutations $P_{N,i}$ applies r rounds of different affine layers $A_{j,N,i}$ and a non-linear layer χ. After r rounds, a final affine layer is applied.

$$P_{N,i} = A_{r,N,i} \circ \chi \circ A_{r-1,N,i} \circ \cdots \circ \chi \circ A_{1,N,i} \circ \chi \circ A_{0,N,i} \tag{3.4}$$

Each affine layer is unique and depends on the nonce N and the block counter i. The permutations are parameterizable in the number of rounds r, and the block size of the permutation n must be odd because χ is not a permutation for an even n.

3.6.2 Affine layers

The affine layer is simply a matrix multiplication of a binary invertible $n \times n$ matrix $M_{j,N,i}$ to the cipher block x, followed by the addition of a round constant $c_{j,N,i}$, as seen in 3.5.

$$y = M_{j,N,i} \cdot x \oplus c_{j,N,i} \tag{3.5}$$

Non-linear layer

For the non-linear layer, the χ transformation previously used in Keccak [3] is applied to the entire block. This transformation is invertible for any odd number of bits n. For input bits x_i and output bits y_i, $0 < i < n$, the non-linear layer is defined as in equation 3.6 with all indices modulo n.

$$y_i = x_i \oplus x_{i+2} \oplus x_{i+1}x_{i+2} \tag{3.6}$$

3.6.3 Generation of Matrices and Constants

The authors of the Rasta paper suggest generating matrices and the round constants using a pseudo-random generator (PRNG) seeded with r, n, N and i. The output of the PRNG is first used to generate $M_{0,N,i}$. The rows are filled iteratively with the PRNG output while checking that they are linearly independent. If they are not, a new random row is added and examined, etc. Afterward, the output of the PRNG is used to create the constants $c_{0,N,i}$ and then the next matrix $M_{1,N,i}$ and so forth. To ensure the security of the permutation, the PRNG must be cryptographically secure. For instance, it should not be feasible for an attacker to find inputs to the PRNG for the attacker's choice of outputs. The PRNG internal state must also be so substantial that internal collisions within its state will not occur in practice.

Chapter 4

Implementing Rasta in HElib

4.1 Previous Implementations and Variants

In addition to the implementation in [7], there is as far as we know only one other implementation of the Rasta cipher in HElib. This has been done by researchers from the Ruhr University Bochum. They are also the ones behind the related Dasta cipher, which we will mention in the next subsection.

4.1.1 Dasta

Dasta is a variant of Rasta where the same parameters for block length and number of rounds are used, but the use of randomly generated linear layers is avoided. Instead a bit-permutation that depends on a nonce is the only variable part of the linear transformation. In Rasta, an attacker may modify the nonce, but only as an input to a PRNG, so it effectively gives no control of the resulting linear mappings. Hence the probability of choosing a weak instance for a desired linear mapping is negligible. Since Dasta removes this PRNG, the attacker may choose the specific linear layer it wants to use. Therefore the creators of Dasta have to restrict the choices of possible linear layers to a subset that does not involve any weak instances.

To solve the problem of weak instances, the authors of the paper have chosen a modular approach of dividing the linear layer into two parts. The two parts are a variable bit permutation and a fixed linear transformation. This approach gives some optimization possibilities that would not be possible in a regular Rasta variant.

The block-wise key stream generation of Dasta is defined by 4.1.

$$B_i(k) = L \circ P_{r,i} \circ \chi \circ L \circ P_{r-1,i} \circ \cdots \circ \chi \circ L \circ P_{1,i} \circ \chi \circ L \circ P_{0,i} + k \tag{4.1}$$

Here k is the secret key, and $0 \leq i \leq D$ is the block counter with the data limit $D = \lceil \frac{2^{s/2}}{n} \rceil$. L is a fixed linear transformation that depends on the Dasta variant. The block counter i determines a $(r+1)$-tuple of bit permutations, which the authors call an *instance*. The parameters for block length and number of rounds used in Dasta variants corresponds to the same variants of Rasta.

The linear transformations are created using a $[2n, n, d]$ code C, with the generator matrix $G = (I_n|A)$, where the linear transformation $L = A^T$, and the linear branch number $B = d$.

4.2 Our Implementation in HElib

4.2.1 Parameters

When implementing Rasta in HElib, it is required of the developer to use the HElib class EncryptedArray for storing ciphertext objects. For an 80-bit security level, a four-round variant of Rasta with block length of 327 is suggested. Hence we had to adjust the parameters to get an EncryptedArray with 327 slots. We did a search for m, such that the equation $s = \frac{\phi(m)}{d}$ gives the exact solution 327, but no such $m < 36001$ was found. Our solution to this problem was to find an m such that the number of slots s was slightly bigger than the suggested block length of 327.

To find such an m, we ran the initialization step of the BGV scheme for all values of m from 10001 up too 36001. All pairs of slot sizes and m values were stored in a text file if they met the requirements of having a slot size bigger than 327. We were then able to choose an m that gave a desired number for the block-length/slot-size.

When $m = 30269$, the ϕ function gives us $s = 329$. This means that the implementation would run with a block length of size 329. The last two slots in the Encrypted array will be treated as ”dummyslots”, only containing the value 0. Then we had to adapt the implementation, such that these extra slots will not impact any calculations in the cipher.

The remaining user defined parameters of the BGV-scheme was left at their suggested default values. The plaintext base $p = 2$, Hensel lift $r = 1$, number of bits in the modulus chain $bits = 300$ and number of columns in the key switching matrix $c = 2$.

4.2.2 Initialization in HElib

When initializing a HElib program we first need to set the previously mentioned user defined parameters. Then the steps in the list 4.2.2 are required:

- The FHEcontext is initialized using the user defined parameters, giving a HElib instance convenient access to these parameters with access methods and some utility functions.
- The modulus chain is built by using buildModChain(context, bits, c) with the parameters $bits$ and c for the amount of bits and columns in the key-switching matrices.
- The secret key is generated with the associated context.
- We compute the key-switching matrices we need based upon the secret key.
- Then we get our public key from the secret key.
- We then get the EncryptedArray of our context object.
- The ciphertext object is created based upon our public key.
- Finally we use our EncryptedArray and public key to encrypt our plaintext into the ciphertext object.

Then we are able to create our Rasta instance based upon these objects and parameters. The EncryptedArray is a class used to store plaintexts as mentioned in chapter 2. This provides us with methods to treat ciphertext objects as regular one-dimensional arrays.

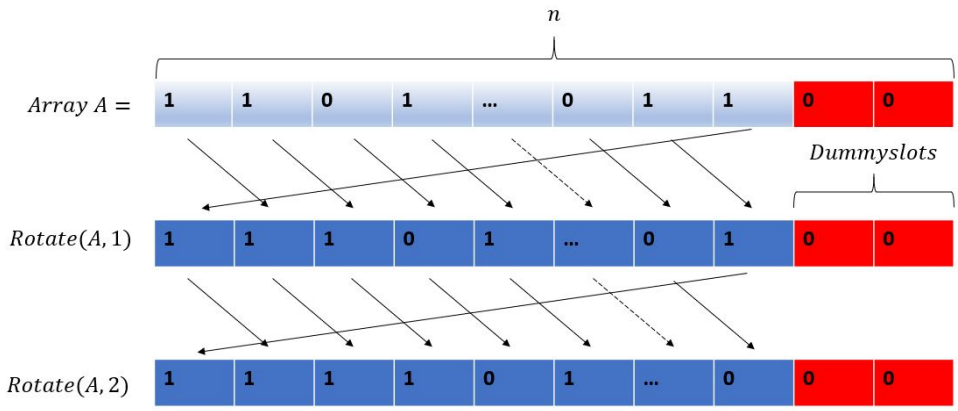

Figure 4.1: Figure showing how we need the rotation function to work.

4.2.3 Affine Layer

For the affine layer in each round of Rasta, the inclusion of dummyslots in the EncryptedArray object needs to be addressed. So the affine layers' matrices and constants are created as size 329×329 0-matrices and vectors. Then they are filled with NTL's PRNG up to a size of 327×327, leaving the last two rows and columns as zeroes.

The 327×327 part of the matrices are checked for non-singularity by the determinant function of the mat_GF2 class and stored in the text file if they are non-singular. Since this procedure is not guaranteed to succeed, our implementation runs by trial and error until it has created the required amount of matrices. Hence creating the binary non-singular matrices that are required for the affine layer of Rasta. We precompute and store these matrices and constants in text files, prior to running the Rasta implementation. Then in the initialization phase of the Rasta implementation, the matrices are read in and stored as vectors of MatMulFull objects for the computations in the affine layer of the different rounds of Rasta.

HElib supports matrix multiplication, by using the built-in matrix multiplication class. However it is required to create a subclass for the abstract MatMulFull class in order to make use of the built-in functions for matrix multiplication with a ciphertext object. The cipher block can then be multiplied with the round matrix and have corresponding round constants added homomorphically for each round.

4.2.4 χ Function

The application of the χ function used in Rasta comes with some issues in the implementation due to the previously mentioned dummyslots. HElib comes with a built-in *rotation* function, but this function does not adhere to our use of dummyslots. Hence we were required to make a modified version that expands upon the built-in rotation function, but where our rotation function does not rotate the bits into the last slots, i.e. the dummyslots of the array, see Figure 4.1.

We achieve this by making two helper arrays where the first array is an all zeroes array except for position $n-2$. The second array is an all-ones array except for the positions $n-1$, and $n-2$ (positions of the dummyslots). We also need a copy t of the Ctxt object c. Now we first rotate c once so that the first dummyslot holds our value for position 0 in c. We then make the actual copy t and multiply t by the first helper array so that t only holds this

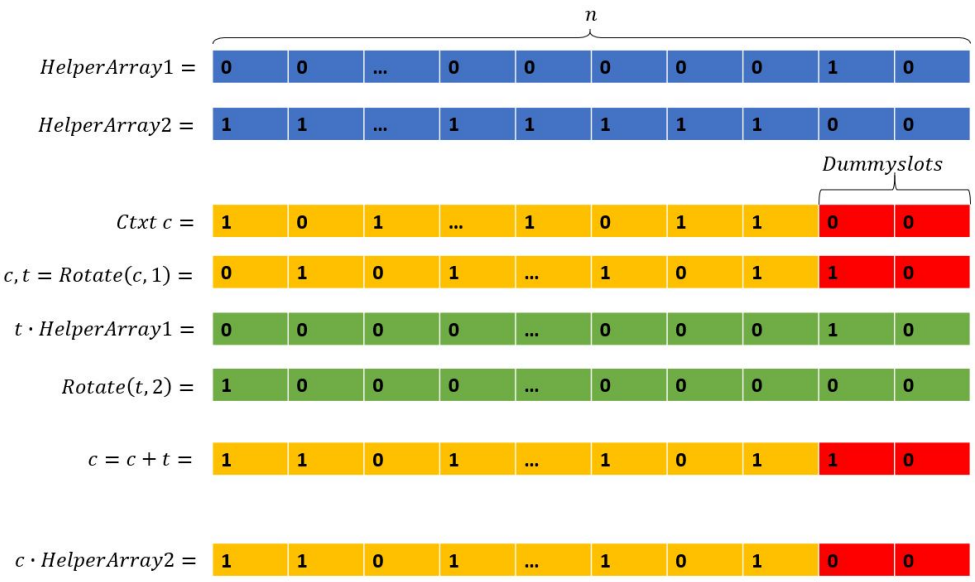

Figure 4.2: Figure showing how rotation is done.

value in position $n-2$. Then we rotate t twice, such that the value is in t's first position and add it together with c. Lastly, we multiply c together with the second helper array to remove a potential 1 value in the dummyslot of c. Note that these multiplications are done with constants and using HElib's multByConstant function, which is much cheaper than ordinary multiplication of ciphertexts. The resulting c will have had its rotation done by one position as intended. See Figure 4.2 for the exact procedure. Rotations by larger amounts can be done by repeating this step.

Algorithm 3 custom rotation pseudocode

Input: $Ctxt\ c$
$n = block\ length$
$HelperArray_1 = (0, 0, \ldots, 0, 1, 0)$
$HelperArray_2 = (1, 1, \ldots, 1, 0, 0)$
Do:
rotate(c,1)
$Ctxt\ t = c$
$t = t \times HelperArray1$
rotate(t,2)
$c = c + t$
Return: $c \times HelperArray2$

Multiplication of ciphertexts is a heavy operation in HElib. This is why we want to run all multiplications of the χ function in parallel. From the χ function 3.6, we can see that every output bit is only dependent on three neighboring bits. So we make two copies of the ciphertext c and rotate them once and twice, respectively. Hence we have all the bits required for the chi function aligned, i.e. the bits x_i, x_{i+1} and x_{i+2} are now aligned in their respective arrays. We are now able to add and multiply in parallel, with only one multiplication required, see Algorithm 4.

Algorithm 4 χ function pseudocode

Input: Ctxt c
$temp1 = c$
CustomRotate$(temp1, 1)$
$temp2 = c$
CustomRotate$(temp2, 2)$
$c = c + temp2$
$temp1 = temp1 \times temp2$
Return: $c + temp1$

4.3 Timed Performance

Table 4.1 shows the timed result for our implementation of our 4-round, 80-bit security, block length 327 variant of Rasta. It has been run ten times, where the minimum time, maximum time and the average time are being displayed. Note that the timings for matrix multiplication, adding constants and the χ function are added together for the functions' total times.

The same matrices and constants have been used for all tests with an all-one key. In addition to the computer specs, the randomness of RLWE used in the BGV-scheme can also affect the results. The source code of the implementation can be found in Appendix 5.

	Average time (seconds)	Min. time	Max. time
Total time	314.37	313.93	314.81
BGV initialization	19.88	18.78	20.05
Read and initialize matrices and constants	7.92	7.91	7.94
Rasta time	286.57	286.20	286.88
Matrix multiplication	270.59	270.26	270.92
Adding constants	0.524	0.523	0.525
Chi function	15.20	15.16	15.27

Table 4.1: Timed results for our Rasta implementation.

Computer spec	
Memory	7.6 GiB
Processor	Intel® Core™ i5-7200U CPU @ 2.50GHz × 4
OS-type	64-bit Ubuntu 18.04

Table 4.2: Computer specs

In [7] the designers of Rasta also timed their implementation in BGV. It should be noted that they used a bit-sliced implementation, using the slots to generate several blocks of key stream in parallel. The advantage of their implementation is that the affine layer consists only of addition of ciphertexts, and thus they produce many key stream blocks at the same time. The drawback of this implementation is that the χ function must be done serially with 327

multiplications. This implementation runs faster, with 110 seconds using 378 BGV-slots, but it is still interesting to see the trade-off between an easy χ layer and an easy affine layer. To get the best of both worlds, one could choose an affine layer that is easy to evaluate for the corresponding FHE scheme, when the cipher block allows χ to be done in parallel.

Chapter 5

Conclusion

With the introduction of FHE in the last years, new possibilities within privacy and cryptography can be achieved in cloud computing and other fields. Where we would have to give up privacy to the cloud earlier if we wanted the cloud to do any operations on the data for us, we can now keep our data private and still outsource computations and the heavy duty operations of FHE to the cloud.

With the publication of several software libraries there are also more possibilities for developers to learn and use FHE in practice. Microsoft SEAL has stated that they want to make FHE understandable and usable for developers who do not necessarily have a cryptographic background. Previously FHE libraries were harder to grasp and mainly directed at researchers who had a good understanding of homomorphic encryption. One such library is HElib which is described as an assembly language for FHE. This means that FHE is moving closer to becoming a tool which regular developers can use in applications, and not only as a research field for the future.

With the development of different symmetric ciphers for use in FHE, there is a steady line of improvements with new ciphers. Multiplicative depth or AND's per bit have been reduced, hence making the ciphers more efficient. With the use of cloud computing weak devices can run cheap traditional symmetric encryption while outsourcing the heavy computations of FHE to the cloud, and still keeping confidentiality intact.

For the case of Rasta the security parameters are chosen quite conservatively, so that performance could potentially be improved with weaker, yet still acceptable security parameters. Optimizations within schemes, libraries or the ciphers themselves can make them efficient enough for commercial use.

The biggest time consumer in our implementation of Rasta is in the affine layer, where matrix multiplications done homomorphically take up the largest chunk of time. Most of the other phases of the implementation have considerably faster times in comparison. So improvements on how the affine layer is designed could increase the efficiency of the cipher drastically. For instance, the χ layer in the implementation of [7] is the heaviest operation, while their implementation only uses $\approx 35.1\%$ of our computing time. The FHE scheme in question should be taken into consideration while designing a symmetric cipher to be used in tandem with it. It is possible to both have an easy non-linear layer and design the affine transformation to be equally easy.

For future work it would be interesting to review Dasta and variants of Rasta with weaker security parameters which may be quicker, yet still retain sufficient security.

Bibliography

[1] M. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for mpc and fhe. Cryptology ePrint Archive, Report 2016/687, 2016. https://eprint.iacr.org/2016/687. 24

[2] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for mpc and fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 430–454. Springer, 2015. 23

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013. 25

[4] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *Journal of Cryptology*, 31(3):885–916, 2018. 20

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017. 13

[6] C. De Canniere and B. Preneel. Trivium. In *New Stream Cipher Designs*, pages 244–266. Springer, 2008. 20

[7] C. Dobraunig, M. Eichlseder, L. Grassi, V. Lallemand, G. Leander, E. List, F. Mendel, and C. Rechberger. Rasta: a cipher with low anddepth and few ands per bit. In *Proceedings of CRYPTO 2018*, pages 662–692. Springer, 2018. LNCS 10991. 24, 27, 31, 33

[8] S. Duval, V. Lallemand, and Y. Rotella. Cryptanalysis of the flip family of stream ciphers. In *Annual International Cryptology Conference*, pages 457–475. Springer, 2016. 19

[9] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009. 7, 8

[10] C. Gentry, Z. Brakerski, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Security*, 111(111):1–12, 2011. 10, 12

[11] S. Halevi and V. Shoup. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 6:12–15, 2013. 8

[12] D. E. Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014. 19

[13] P. Méaux, A. Journault, F.-X. Standaert, and C. Carlet. Towards stream ciphers for efficient fhe with low-noise ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 311–343. Springer, 2016. 18, 19

[14] C. Paar and J. Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. 1

[15] Y. Polyakov, K. Rohloff, and G. W. Ryan. Palisade lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, 2017. 15

[16] O. Regev. The learning with errors problem. *Invited survey in CCC*, 7:30, 2010. 11

[17] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978. 7

[18] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Apr. 2020. Microsoft Research, Redmond, WA. 15

[19] X. Yi, R. Paulet, and E. Bertino. *Homomorphic Encryption and Applications*. Springer, 2014. 9

Appendix Source Code

Source code can be downloaded from https://github.com/Haakonthor/MBackup and run. HElib must be installed in the same folder (or edit Cmakelist.txt to your path). Run the cmake command: ”make” to compile and create executables, when in the folder.

```
#include <iostream>
#include <stdio.h>
#include <helib/FHE.h>
#include <helib/EncryptedArray.h>
#include <helib/matmul.h>
#include <NTL/BasicThreadPool.h>
#include "MyMatMulFull.h"
#include "readConstant.h"
#include "CyclicRotation_chi.h"
#include <sys/time.h>
#include <sys/resource.h>
#include "Rasta_80_327.h"


//void rasta_327_80(int rounds, vector<long> user_key, vector<vector<lo


int main(int argc, char *argv[]){


    vector<long> test_key(329,1);
    test_key[328] = 0;
    test_key[327] = 0;

  //################################initializing ##########################
  clock_t BGVinit_begin = clock();

        //BGV Initialization
  // Plaintext prime modulus
  unsigned long p = 2; // p = 2 (mod 2)
  // Cyclotomic polynomial - defines phi(m)
  unsigned long m = 30269; // over 10.000 oddetall, default: 32109
  // Hensel lifting (default = 1)
  unsigned long r = 1; //stay as one
```

```
// Number of bits of the modulus chain
unsigned long bits = 300; //100-300 , default: 300
// Number of columns of Key-Switching matrix (default = 2 or 3)
unsigned long c = 2; // c = 2,3? default: 2

std::cout << "Initialising context object..." << std::endl;
// Intialise context
FHEcontext context(m, p, r);
// Modify the context, adding primes to the modulus chain
std::cout  << "Building modulus chain..." << std::endl;
buildModChain(context, bits, c);

// Print the security level
std::cout << "Security: " << context.securityLevel() << std::endl;

// Secret key management
std::cout << "Creating secret key..." << std::endl;
// Create a secret key associated with the context
FHESecKey secret_key(context);
// Generate the secret key
secret_key.GenSecKey();
std::cout << "Generating key-switching matrices..." << std::endl;
// Compute key-switching matrices that we need
addSome1DMatrices(secret_key);

// Public key management
// Set the secret key (upcast: FHESecKey is a subclass of FHEPubKey)
const FHEPubKey& public_key = secret_key;

// Get the EncryptedArray of the context
const EncryptedArray& ea = *(context.ea);

  // Get the number of slots (phi(m))
long nslots = ea.size();
std::cout << "Number of slots: " << nslots << std::endl;

  // Create a vector of long with nslots elements
  //std::vector<long> ptxt(nslots);


// Create a ciphertext
Ctxt ctxt(public_key);

// Encrypt the plaintext using the public_key
ea.encrypt(ctxt, public_key, test_key);
```

```
// Create a plaintext for decryption
std::vector<long> decrypted(nslots);

clock_t BGVinit_end = clock();
//#########bgv end of initzialise ######################

clock_t read_matrix_timer_start = clock();

//creates a vector of matmulFullexec matrices for use in the affine
vector<MatMulFullExec> MMFE_matrices;

//reads in matrices from different files
MyMatMulFull my_MMF_matrix0(ea, "matrix329_0.txt");
MyMatMulFull my_MMF_matrix1(ea, "matrix329_1.txt");
MyMatMulFull my_MMF_matrix2(ea, "matrix329_2.txt");
MyMatMulFull my_MMF_matrix3(ea, "matrix329_3.txt");
MyMatMulFull my_MMF_matrix4(ea, "matrix329_4.txt");
//MyMatMulFull my_MMF_matrix5(ea, "matrix329_5.txt");

//converts the MatMulFull objects into MarMulFullExec objects
MatMulFullExec MMFE_matrix0(my_MMF_matrix0, false); //create the mat
MatMulFullExec MMFE_matrix1(my_MMF_matrix1, false); //create the mat
MatMulFullExec MMFE_matrix2(my_MMF_matrix2, false); //create the mat
MatMulFullExec MMFE_matrix3(my_MMF_matrix3, false); //create the mat
MatMulFullExec MMFE_matrix4(my_MMF_matrix4, false); //create the mat
//MatMulFullExec MMFE_matrix5(my_MMF_matrix5, false); //create the m

//adds the different matrices to the vector of MatMulFullExec matric
MMFE_matrices.push_back(MMFE_matrix0);
MMFE_matrices.push_back(MMFE_matrix1);
MMFE_matrices.push_back(MMFE_matrix2);
MMFE_matrices.push_back(MMFE_matrix3);
MMFE_matrices.push_back(MMFE_matrix4);
//MMFE_matrices.push_back(MMFE_matrix5);

//vector<long> constants = readConstantsFromFile(ea,"constants329.tx

vector<vector<long>> constants = readConstantsFromFile_all(ea, "cons

clock_t read_matrix_timer_end = clock();
double read_matrix_time = (double)(read_matrix_timer_end-read_matrix


//########################initiliaze end ############################

cout << "starting Rasta: " << endl;
```

```
  //cout << "starting Rasta, with key: " << test_key << endl; //shows
  clock_t Rasta_start = clock();

  rasta_327_80(4, test_key, constants, MMFE_matrices, ctxt, ea);

  clock_t Rasta_end = clock();
  clock_t total_time_end = clock();

  //ea.decrypt(ctxt, secret_key, test_key);
  //cout << "end decrypted: " << endl;
  //cout << "end decrypted: " << ptxt << endl; //shows decrypted resul

  double total_time = (double)(total_time_end-BGVinit_begin)/CLOCKS_PE
  double initBGV_time =  (double)(BGVinit_end-BGVinit_begin)/CLOCKS_PE
  double rasta_time =  (double)(Rasta_end-Rasta_start)/CLOCKS_PER_SEC;


  cout << "timed results:" << endl;
  cout << "total time:" << total_time << endl;
  cout << "initBGV  time:" << initBGV_time << endl;
  cout << "read in matrix phase timer" << read_matrix_time << endl;
  cout <<  "rasta time:"<< rasta_time << endl;


    return 0;
}

//read constant from file
//also includes cyclicbitshift and chi functions.
#include <stdio.h>
#include <vector>
#include <fstream>
#include <NTL/ZZ.h>
#include <NTL/GF2X.h>

NTL_CLIENT


std::vector<long> readConstantsFromFile(const EncryptedArray& ea, cons
    /*
    * reads in a single vector of constants from file, obsulete for ou
    */

    long n = ea.size();
    FILE *fp;
    int i, bb;
```

```
    vector<long> constants(n,0);

    fp = fopen(fname,"r");
    if(fp == NULL) {cout<<"could not find constants" << endl; exit(0);

    for(i=0; i<n; ++i){
        fscanf(fp, "%d", &bb);
        constants[i] = bb;
    }
    fclose(fp);

    return constants;
}
 //test method to read severa lines form a file

std::vector<std::vector<long>> readConstantsFromFile_all(const Encrypt
    /*
    * reads in multiple lines of constants and return them as a matrix
    */
    long n = ea.size();
    FILE *fp;
    int i,j, bb;
    vector<long> tempData(n,0);
    vector<vector<long>> constants;
    constants.resize(n);
    for(i=0; i<n; ++i){
        constants[i].resize(n);
    }


    fp = fopen(fname,"r");
    if(fp == NULL) {cout<<"could not find constants" << endl; exit(0);
    for(j=0; j<10; ++j){
        for(i=0; i<n; ++i){

            fscanf(fp, "%d", &bb);
            tempData[i] = bb;
        }
        constants[j] = tempData;
    }
    fclose(fp);

    return constants;
}
```

```
void addConstants(Ctxt &c, const EncryptedArray& ea, std::vector<long>
    /*
    * adding a full array worth of constants to a ciphertext object.
    */
    long n = ea.size();
    ZZX emask;
    vector<long> pmask(n, 0);
    for(int i=0; i<n; ++i){
        pmask[i] = (constants)[i];
    }
    ea.encode(emask,pmask);
    c.addConstant(emask); //using HElibs built in function
}

void addKey(Ctxt &c, const EncryptedArray& ea, std::vector<long> key){
    /*
    * adding a full array worth of constants to a ciphertext object. t
    */
    long n = ea.size();
    ZZX emask;
    vector<long> pmask(n, 0);
    for(int i=0; i<n; ++i){
        pmask[i] = (key)[i];
    }
    ea.encode(emask,pmask);
    c.addConstant(emask);
}


void rasta_327_80(int rounds, vector<long> user_key, vector<vector<lon
  int i;
  int counter = 0;
  double chi_time =0;
  double matmul_time = 0;
  double addConst_time = 0;

  cout << "starting␣rasta␣rounds." << endl;
  for(i = 0; i<rounds; i++){
    cout << "round:␣" << i << endl;


    clock_t matmul_timer_start = clock();
    MMFE_matrices[i].mul(c);
    clock_t matmul_timer_end = clock();
    matmul_time += (double)(matmul_timer_end-matmul_timer_start)/CLOCK
```

```
    cout << "matrix.mul(c) ended" << endl;

    clock_t adding_const_timer_start = clock();
    addConstants(c, ea, constants[i]);
    clock_t adding_const_timer_end = clock();
    addConst_time += (double)(adding_const_timer_end-adding_const_time

    cout << "add constants ended." << endl;


    clock_t chi_timer_start = clock();
    keccak_chi(c,ea);
    clock_t chi_timer_end = clock();
    chi_time +=  (double)(chi_timer_end-chi_timer_start)/CLOCKS_PER_SE


    cout << "keccak chi ended" << endl;
    counter = counter + 1;
  }
  cout <<  "counter: " <<counter << endl;
  cout << "last round without keccak i:  " << i << endl;

  clock_t matmul_timer_start = clock();
  MMFE_matrices[counter].mul(c);
  clock_t matmul_timer_end = clock();
  matmul_time += (double)(matmul_timer_end-matmul_timer_start)/CLOCKS_

  cout << "multiplying in C last time" << endl;

  clock_t adding_const_timer_start = clock();
  addConstants(c, ea, constants[counter]);
  clock_t adding_const_timer_end = clock();
  cout << "last  add constant" << endl;
  addKey(c, ea, user_key);
  cout << "added key again" << endl;

  //double matmul_time =  (double)(matmul_timer_end-matmul_timer_start
  //double addConst_time =  (double)(adding_const_timer_end-adding_co
  cout <<"matMul timer:" << matmul_time << endl;
  cout <<"addingConst timer:" << addConst_time << endl;
  cout <<"chi timer: " << chi_time << endl;
}

//MyMatMulFull
//helper class to create MatMulFull objects to be used for MatMulFulle
```

```
#include <stdio.h>
//#include <helib/matmul.h>
NTL_CLIENT

class MyMatMulFull : public MatMulFull_derived<PA_GF2> {
    PA_INJECT(PA_GF2)
    const EncryptedArray& ea;
    vector<vector<RX>> data;

public:
    MyMatMulFull(const EncryptedArray& _ea, const char *fname): ea(_ea
        long n = ea.size();
        RBak bak; bak.save(); ea.getContext().alMod.restoreContext();
        FILE *fp;
        int i,j, bb;

        data.resize(n);
        for(i = 0; i <n; ++i){
            data[i].resize(n);
        }

        fp= fopen(fname,"r");
        if(fp==NULL) {cout<<"could␣not␣find␣matrix␣" <<fname<<endl; ex
        for(i=0;i<n; ++i){
            for(j=0;j<n; ++j){
                fscanf(fp,"%d",&bb);
                data[i][j] = GF2X(bb);
            }
        }
        fclose(fp);
    }
    bool get(RX& out, long i, long j) const override {
        long n = ea.size();

        if(i>= 0 && i<n && j>= 0 && j<n){
            out = data[i][j];
            if(IsZero(data[i][j]))
                return true;
            return false;
        }
        else{
            cout<< "indices␣(" << i <<","<< j <<")␣out␣of␣range␣(" <<
            exit(0);
        }
    }
```

```
    void print(){
        int i, j;
        long n = ea.size();

        for(i=0; i<n; ++i){
            for(j=0; j<n; ++j){
                cout << data[i][j] << "␣";
            }
            cout << endl;
        }
    }
    const EncryptedArray& getEA() const override {return ea;}


};


void cyclicBitShift(Ctxt &c ,const EncryptedArray& ea){
    // working cyclic bitshift for a encrypted array of size 2 more th

    long n = ea.size();

    std::vector<long> h1Arr(n,0);
    std::vector<long> h2Arr(n,1);
    h1Arr[n-2] = 1;
    h2Arr[n-1] = 0;
    h2Arr[n-2] = 0;

    ZZX h1; //obejct polynomial over zz used for encoding the long vec
    ZZX h2;
    ea.encode(h1, h1Arr);
    ea.encode(h2, h2Arr);

    //HElibs built in rotate function.
    ea.rotate(c, 1);
    Ctxt t = c;
    t.multByConstant(h1); //HElib built in multiply constant function

    //Ctxt s = t;
    ea.rotate(t,2);
    c += t;
    c.multByConstant(h2);
}


void keccak_chi(Ctxt &c, const EncryptedArray& ea){
```

```
    // chi function used in rasta

    Ctxt temp1 = c; //create copies of c
    cyclicBitShift(temp1, ea);

    Ctxt temp2 = temp1;
    cyclicBitShift(temp2, ea);

    c += temp2;
    temp1 *= temp2;
    c += temp1;

}

//generate_NS_matrix.cpp
#include <iostream>
#include <NTL/mat_GF2.h>
#include <NTL/matrix.h>
#include <NTL/vec_vec_GF2.h>
#include <vector>
#include <fstream>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>

NTL_CLIENT


using namespace std;

//function to take a matrix and write it to a txt file
void writeMatToFile(vector<vector<long>>& a, const char *fname){

    long n = a.size();
    ofstream myfile;
    myfile.open(fname,ios::app);
    int i, j;

    for(i=0; i<n; i++){
        for(j=0;j<n;j++){
            myfile << a[i][j] << ' ';
        }
        myfile << endl;

    }
    myfile << endl;
```

```
    myfile.close();


}
//writes the constants to a txt file.
void writeConstToFile(vector<long>& v, const char *fname){
    long n = v.size();
    ofstream myfile;
    myfile.open(fname, ios::app);
    int i;

    for(i=0; i<n; i++){
        myfile << v[i] << ' ';
    }
    myfile << endl;
    myfile.close();
}
//pseudo random generation of constants
void generateConstants(vector<long>& v){ //pseudo random generation of
    long n = v.size();
    for(int i=0; i< n-2; i++){
        long r = rand() %2;
        v[i] = r;
    }

}
//converts a NTL GF2 matrix to a vector of vectors.
void convertGF2_to_vec(mat_GF2& gf2mat, vector<vector<long>>& matrix){
    long n = gf2mat.NumRows();
    //vector<vector<long>> matrix_y(n, vector<long>(n));
    GF2 g;
    long a;

    for (long i = 0; i < n; i++)
    {
        for(long j = 0; j < n; j++)
        {
            g = gf2mat.get(i,j);
            a = conv<long>(g);
            (matrix[i])[j] = a;

        }
    }
}
```

```
void printVecMat(vector<vector<long>> const &matrix){
    for(const vector<long> &v : matrix){
        for (long x : v)cout << x << ' ';
        cout << endl;
    }
}

int main(){

    long n = 327;

    mat_GF2 x;
    GF2 d;

    double matrix_time = 0;
    clock_t matrix_gen_timer_start = clock();

    vector<vector<long>> myVec(n+2,vector<long>(n+2)); //creates a n+2
    vector<long> myConst(n+2,0);

    int numOfNSMatrices = 0;

    //generate 10 vectors of constants (will only need 4 to 6)
    for(int i = 0; i < 10; i++){
        generateConstants(myConst);
       // writeConstToFile(myConst, "constants329.txt"); //uncomment t
    }


    //cout << x << endl;
    //creates matrices and checks if they are non-singular, if they ar
    for (int i = 0; i < 100; i++) //set to 100, loop will end when eno
    {
        random(x,n,n); //NTL creates a random nxn matrix
        determinant(d,x); //checks the determinant for the matrix
        //x.SetDims(n+2,n+2);
        /*
        for(long i= 0; i<n-2; i++){
            x.put(i,n-1,0);
            x.put(i,n-2,0);
        }

        for(long j= 0; j<n; j++){
            x.put(n-1,j,0);
            x.put(n-2,j,0);
        } */
```

```
        if(d ==1){
            numOfNSMatrices += 1;
            cout<< "matrix␣{" << i << "}␣is␣nonsingular" << endl;
            //cout<< x << endl;

            convertGF2_to_vec(x,myVec);
            cout<< "matrix␣{" << i << "}␣in␣convertet␣form" << endl;
            //printVecMat(myVec);
            //writeMatToFile(myVec,"matrices329.txt"); //uncomment to


        }
        if(numOfNSMatrices == 5){ //set to the required amount of matr
            cout << "found␣5␣NS␣matrices,␣breaking␣out␣of␣loop" << end
            break;
        }

    }
    clock_t matrix_gen_timer_end = clock();
    matrix_time = (double)(matrix_gen_timer_end-matrix_gen_timer_start
    cout << "matrix␣timer:␣" << matrix_time << endl;
    return 0;
}
```