# Efficient implementation of LowMC in HElib


**Isabel Thevahi Francis**

Supervisor: Håvard Raddum

Department of Informatics
University of Bergen

June 2018

# Acknowledgements

I would like to thank my supervisor, Håvard Raddum. I am deeply grateful for the guidance and encouragement he has given me, throughout writing this thesis.

I would also like to thank the people at Simula@UiB, for providing a supportive and inclusive workplace. Our daily lunches and "Fikas" will be greatly missed.

# Abstract

LowMC is a symmetric block cipher designed for fully homomorphic encryption. This thesis focuses on Martin Albrecht's implementation of the cipher in the FHE library HElib, and how his implementation can be improved when encrypting a single plaintext. We have succeeded in getting faster encryption by changing the packing of the plaintext bits and focusing on a rotation-based linear layer. When only encrypting a single plaintext Albrecht's implementation takes 217.17 seconds, while our alternative implementation takes 11.53 seconds.

# Table of contents

# Chapter 1

# Introduction

Every day, millions of people share sensitive information through the internet. Passwords, credit card numbers, and social security numbers are examples of data that could lead to severe consequences if fallen into the wrong hands. How can we prevent others from taking a peek at our private details, while still allowing insight for a select few? What happens when the ones who are meant to process our information, cannot be trusted? Is it possible to perform computations on data without having direct access to it?

When storing data in the cloud today, clients are forced to put trust in their cloud providers. Is it possible to eliminate this demand? In this thesis, we will discuss how encryption, or more specifically fully homomorphic encryption (FHE), can be a solution to this problem. Efficiency concerning practical applications has been an issue often raised regarding FHE. However, we believe that with this thesis we have taken a step closer to solving this concern.

## 1.1 Encryption

Encryption is the process of scrambling a message such that only the intended recipient is capable of retrieving the original message. The intended receiver gets a 'key' which unscrambles the encrypted message or, in other words, decrypts it. It is crucial that the key does not fall into the wrong hands, as anyone in possession of it is capable of decrypting the encrypted message. We often refer to the decryption key as the private key, as it is kept private from the rest of the world. When encrypting a message, we also use a key. Depending on the encryption used, it varies whether this key is kept private or not.

How safely the message is encrypted is reliant on the encryption scheme used. An encryption scheme consists of three algorithms; one that transforms a message (plaintext) into a scrambled message (ciphertext), one that decrypts the ciphertext back to the plaintext,

and one that generates the keys used for encryption and decryption. Though the different encryption schemes vary considerably, the majority of them follow Kerckhoffs' principle.

In 1883, Auguste Kerckhoffs published two articles in the French "Le Journal des Sciences Militaires" stating six design principles for military ciphers.

1. The system must be indecipherable at least in practice, if not mathematically.
2. The system must not be required to be secret, and it 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 communications via telegraph (the prevailing technology of the time).
5. Equipment and documents for the system must be portable, and their usage and function must not require the gathering or collaboration of several people.
6. The system must be easy to use, requiring neither mental strain nor the knowledge of a long series of rules to implement it.

Not all of the six principles remain relevant in modern times, but the second principle still stands strong today. The basic idea behind the second principle is that all details around the encryption scheme, except for the secret key, should be public without destroying the security of the scheme. If the components of the scheme become public, it should not compromise the entire scheme. In current times encryption schemes are made available to the public before being used, this is to catch any flaws in the scheme before being deployed in real applications.

To get a better understanding of encryption schemes and their security we will give two examples of previously used historical ciphers; the Caesar cipher, and the substitution cipher.

### Caesar Cipher

One of the most well-known encryption schemes is the Caesar cipher. The Caesar cipher is named after Julius Caesar (100-44 BC), who according to the historian Suetonius used the cipher when sending letters of military significance. In the Caesar cipher, every letter in the original message shifts with a fixed number of positions.

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | A | B | C |

Table 1.1 Caesar cipher with 3 position shift

Table 1.1 shows how the letters in the English alphabet shifts in a Caesar cipher with a right shift of 3. The top row represents the letters in the original message, and the bottom row represents the letters after the encryption. Given the message "SECRET MESSAGE," we can use this table to see that it encrypts to "VHFUHW PHVVDJH." To decrypt the message, we would shift the letters in the encrypted message in the opposite direction. Had the message been encrypted with a Caesar cipher with a right shift of 7 positions, we would decrypt the encrypted message with a left shift of 7 positions.

In every encryption scheme, we have a keyspace which refers to the set of all possible keys. In the Caesar cipher, the key is the number of positions in which we shift the letters. The keyspace would, therefore, contain 25 unique keys since there are 25 ways we can shift the letters of the alphabet without getting the same output. A shift of 50 positions would be the same as a shift of 24. Due to this limited keyspace, it is easy to perform a brute-force attack to find the original message. In a brute-force attack, we would try all possible keys until we find the right one. With the Caesar cipher having only a keyspace of 25 keys this would take a short time even with pen and paper.

### Substitution Cipher

The Caesar cipher is a particular type of substitution cipher. Where in the Caesar cipher a letter shifts, the substitution cipher replaces a letter. It is not essential whether the letter is replaced by another letter, a symbol, a group of symbols or a group of letters, as long as the replacement is one-to-one, meaning each letter has its unique replacement.

The keyspace for the substitution cipher is much larger than for the Caesar cipher. Since every letter has a unique replacement, the keyspace for the English alphabet contains $26! \approx 2^{88}$ keys. The size of the keyspace makes it challenging to run a brute-force attack, even with the most powerful computers we have today. However, it is possible to run frequency analysis to break the cipher.

Frequency analysis is the study of how often letters or symbols repeat in a ciphertext. Most languages are built in such a way that some letters are used more often than others. In the English language the three most frequently used letters are 'E,' 'T,' and 'A,' while 'Z,' 'Q,' and 'X' are the least frequently used. By studying the frequencies in the ciphertext and comparing them with the known frequencies of the letters in our alphabet, we can make some assumptions on what parts of the plaintext may be. There is no guarantee that the

frequency analysis will work; however, it has a high chance of success when the plaintext is long enough. The substitution cipher is typically broken quite quickly when the plaintext has more than 500 letters.

While neither the Caesar cipher nor the substitution cipher is safe enough to use today, there are plenty of safe-to-use encryption schemes that have taken their places. Two prominent examples are AES (Advanced Encryption Standard) and RSA (Rivest-Shamir-Adleman), where AES is a symmetric cipher, and RSA is an asymmetric cipher [11].

### 1.1.1 Asymmetric vs. Symmetric Encryption

Encryption schemes are divided into two categories; asymmetric, and symmetric encryption. In symmetric encryption, both the sender and the receiver possess the same key, and the key is used both for encrypting and decrypting messages. In asymmetric encryption, there are two keys, a private and a public key. The public key is used for encryption, while decryption is done with the private key.

#### Symmetric Encryption

We categorise symmetric encryption in two groups; stream ciphers and block ciphers.

A stream cipher is an encryption algorithm that encrypts a single plaintext bit at a time. The given key is used to produce an arbitrarily long stream of pseudorandom bits, also called a keystream, that is continuously added bit by bit with the plaintext to produce the ciphertext. For the stream cipher to remain secure, it is essential that the keystream remain unpredictable and never reused. If the keystream becomes predictable, the stream cipher becomes insecure.

While the stream cipher is not 100% secure in the information theoretical sense, it is modelled after a cipher that is unbreakable, the one-time pad. The keystream in the one-time pad is not generated from a given key but is the actual key, which means that the key is at least as large as the plaintext. One of the essential features of the one-time pad is that the key must be truly random, unlike the stream cipher which has a pseudorandom keystream. The one-time pad is unbreakable because for every conceivable plaintext, there is a key that decrypts the given ciphertext into the particular plaintext. The attacker might as well guess the plaintext itself and gets no information from the ciphertext

Though the one-time pad provides an unbreakable cipher, it is quite troublesome to use. Creating a truly random key can be quite tricky, especially since we are not able to reuse it, and the key can be difficult to send because of its size.

While the stream cipher encrypts a single plaintext bit at a time, the block cipher encrypts a set number, or a block, of plaintext bits at a time. Instead of using a keystream, a key of

predetermined length is chosen when encrypting the plaintext. The typical block length used in practice is either 128 bits, such as in AES, or 64 bits, such as in DES (Data Encryption Standard) or Prince [5]. Because the block cipher works on a chunk of data, it requires more memory than the stream cipher that encrypts a single bit at a time. As well as needing more memory, the block cipher is also more prone to errors. If one bit of an encrypted block is damaged, the whole block will be decrypted incorrectly. If the same happens with the stream cipher, only a single bit will be damaged.

### Asymmetric Encryption

Historically most encryption schemes have been symmetric ciphers. It was not until 1976 that Whitfield Diffie, Martin Hellman, and Ralph Merkle publicly introduced asymmetric encryption [11]. The problem with the symmetric ciphers was the key exchange. To securely communicate with a symmetric cipher, the secret key had to be exchanged over a secure channel. Most of the time the communicating parties had to exchange the key physically, which was very unpractical and was not always safe. By creating an asymmetric scheme, Diffie and Hellman made it possible to share the symmetric encryption key over an insecure line.

In asymmetric encryption, also referred to as public-key encryption, each party has two keys, a public, and a private key. The public key is shared with the rest of the world, while the private key is kept secret. The public key is used for encryption of a message, and only the corresponding private key can be used to decrypt the message. The private key can also be used to encrypt a message, where the public key decrypts it, however, this is used in digital signatures and not in encryption schemes.

The asymmetric encryption takes longer time than the symmetric encryption. It is therefore preferred to use symmetric encryption when communicating over the Internet. As mentioned, the symmetric encryption has problems regarding the key exchange, but this can be solved using the asymmetric encryption. If Alice wants to share a symmetric encryption key with Bob, she could encrypt it using Bob's public key. Bob would then be the only one who could obtain the symmetric encryption key since he is the only one who has the private key. This process is what happens in the Transport Layer Security protocol. Previously it had been difficult to obtain a secure line of communication between strangers over the Internet, as the shared private key had to distributed between the communicating partied ahead of time. With asymmetric encryption this is no longer required.

Asymmetric encryption can also help to provide non-repudiation in essential documents or messages. Non-repudiation is the property which prevents a sender of a message to change the content of the message after it is sent or deny being the actual sender. With

digital signatures, both the identity of the sender and the original content of the message is verifiable.

## 1.2 Fully Homomorphic Encryption

In 1978 Rivest, Adleman and Dertouzos first introduced the idea of fully homomorphic encryption [12]. In their paper they addressed one of the main limitations of encryption; a system working with encrypted data can store or retrieve the encrypted data for the user, but if they want to perform any operations on the data it must first be decrypted. Rivest, Adleman, and Dertouzos were quite optimistic about finding a solution to this problem. They believed there existed encryption functions that would permit encrypted data to be operated on without having to be decrypted. In the paper these special encryption functions were called "privacy homomorphisms," but today we refer to them as fully homomorphic encryption schemes.

Various encryption schemes throughout the years have been able to permit operations on encrypted data. The operations we have in mind are addition and multiplication. Earlier schemes only respected one of the operations, and we define these schemes as homomorphic encryptions concerning either addition or multiplication. Homomorphic encryption schemes allow either multiplication or addition to be performed on the ciphertext, that when decrypted would give the identical result if those same operations had been applied on the plaintext. The unpadded version of RSA and ElGamal are encryptions that are homomorphic concerning multiplication, and we show this for the case of RSA.

The unpadded version of RSA cryptosystem works in the following way.

1. Select two distinct primes, $p$ and $q$
2. Calculate $n = pq$
3. Calculate $\phi(n) = (p-1)(q-1)$
4. Choose an $e$ such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$
5. Find $d$ such that $d \equiv e^{-1} (mod\ \phi(n))$

The public key is $e$, the private key is $d$, the plaintext is $m$, and the ciphertext is $c$.

$$\text{ENCRYPTION: } Enc(m) \equiv c \equiv m^e (mod\ n)$$

$$\text{DECRYPTION: } Dec(c) \equiv m \equiv c^d (mod\ n)$$

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

$$\begin{aligned} Enc(m_1) \cdot Enc(m_2) = m_1^e (mod\ n) \cdot m_2^e (mod\ n) = m_1^e \cdot m_2^e (mod\ n) = \\ (m_1^e \cdot m_2^e)(mod\ n) = (m_1 \cdot m_2)^e (mod\ n) = Enc(m_1 \cdot m_2) \end{aligned} \tag{1.1}$$

As we can see from equation 1.1 it does not matter if we encrypt the plaintexts first and then multiply them together or if we multiply the plaintexts first and then encrypt them, the outcome will be the same. The scheme is therefore homomorphic with respect to multiplication. The same cannot be said regarding addition, as seen in equation 1.2.

$$\begin{aligned} Enc(m_1) + Enc(m_2) = m_1^e (mod\ n) + m_2^e (mod\ n) = m_1^e + m_2^e (mod\ n) \\ = (m_1^e + m_2^e)(mod\ n) \neq (m_1 + m_2)^e (mod\ n) = Enc(m_1 + m_2) \end{aligned} \tag{1.2}$$

It was not until 2009 that the first fully homomorphic encryption scheme was published by Craig Gentry, using lattice-based cryptography [7]. Unlike the earlier homomorphic encryption schemes, which only allow either multiplication or addition to be applied on the ciphertexts, the fully homomorphic encryption scheme allows both. By managing both multiplication and addition, the scheme can handle any operations applied on the ciphertexts, such that when the result is decrypted it would give the same output as if the same operations had been done on the plaintext. Though Gentry managed to solve the problem, there was one major drawback with his solution. The encryption was very inefficient and had, therefore, little practical use.

Since 2009 there have been made many improvements on fully homomorphic encryption. One of the schemes that proved to be more efficient than Gentry's original scheme was the BGV cryptosystem. The BGV cryptosystem was published in 2011 by Brakerski, Gentry, and Vaikuntanathan [6]. While the system was still suffering from significant performance costs, it was closer to practical use. In 2013 the BGV encryption scheme was implemented in Halevi and Shoup's homomorphic encryption library, HElib [8].

## 1.3 Problem statement of the thesis

This thesis examines how the combination of a symmetric cryptosystem and FHE can increase efficiency. We will take a closer look at the LowMC symmetric encryption scheme, that is designed primarily for use with FHE. Up until now, Martin Albrecht is the only person, to the best of our knowledge, who has tested the LowMC encryption in the HElib library where his implementation encrypts multiple plaintexts in parallel. Though this is effective when we have a set of plaintexts that we would like to encrypt simultaneously, many applications

need to encrypt only a single plaintext as quickly as possible. Is it possible to implement the LowMC scheme such that the encryption of a single plaintext will be more efficient than it is with Albrecht's method?

# Chapter 2

# Fully Homomorphic Encryption

## 2.1 What is FHE?

As mentioned in Section 1.2, fully homomorphic encryption makes it possible to perform arbitrary operations on encrypted data, without having to decrypt it. This unique trait can be very useful in applications that are reliant on high security.

A widely talked about use for FHE is in cloud computing. Clients of cloud providers often have sensitive data and are reluctant to send it unencrypted to the cloud. Examples of this may be medical data, business sensitive data, identification data, and so on. A possibility would be encrypting the data before uploading it to the cloud. However, this would be counterproductive if the data needs to be decrypted to perform computations on it. The current solution to the problem is sending the decryption key along with the encrypted data. Access to the decryption key allows the cloud provider to decrypt the data and perform operations on it; however, it is not an optimal solution when the client does not trust the cloud provider. FHE offers a solution that lets the cloud perform operations on the encrypted data without having access to the decryption key, which removes the client's need to trust the cloud provider.

In 2009 Craig Gentry constructed the first FHE scheme in his dissertation [7], proving it theoretically possible to achieve. Gentry's starting point is a "somewhat" homomorphic encryption scheme. This scheme can apply some operations on the ciphertext, while still being able to decrypt it correctly. However, it can only handle a certain number of operations before the decryption becomes incorrect. The limitation in the number of operations can be attributed to the *noise* in the ciphertext.

### 2.1.1 Noise

Noise is a small random value that is added to the ciphertext while encrypting to enhance security. The noise also increases when we perform operations on the ciphertext. Since there is a maximum limit of how significant the noise can be before the decryption of the ciphertext becomes incorrect, there is also a limit of how many operations we can perform on the ciphertext.

We now present a simple scheme, to get a better understanding of the noise and how it increases. The following is a simple symmetric encryption scheme over the integers.

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

The plaintext $m \in \{0,1\}$, and the ciphertext $c$ is produced 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 the decryption process, we reduce the ciphertext modulo $k$, which leaves us with the remainder, $2r + m$. When the remainder is reduced modulo 2, the plaintext is the only value left. When the ciphertext is reduced to the plaintext alone, the decryption is correct and has not been affected by the noise. The value $2r$ is the noise in this example. If the noise were to increase to more than $k$, the decryption would fail. The reason for this is that when reducing the ciphertext modulo $k$, the remainder would be something different than $2r + m$, making it impossible to decrypt to the original plaintext.

If we look at the two operations, addition and multiplication, we can see that the noise produced differs between the operations. Let $m_0$ and $m_1$ be two plaintext bits and $c_0$ and $c_1$ be two ciphertext bits, where

$$c_0 = kq_0 + 2r_0 + m_0$$

$$c_1 = kq_1 + 2r_1 + m_1$$

**ADDITION**

$$c_0 + c_1 = k(q_0 + q_1) + 2(r_0 + r_1) + (m_0 + m_1)$$

**MULTIPLICATION**

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

When looking at the addition of $c_0$ and $c_1$ we can see that their sum is a ciphertext that encrypts $m_0 + m_1$. The same goes for the multiplication of $c_0$ and $c_1$ where the product is a ciphertext that encrypts $m_0 \cdot m_1$. Using these observations, we therefore know that the homomorphic property is present in this scheme.

When comparing noise from the addition, $2(r_0 + r_1)$, with the noise from the multiplication, $2(2r_0r_1 + m_1r_0 + m_0r_1)$, we can conclude that the multiplication operation is a lot more expensive than the addition operation. Where the noise terms from the ciphertexts in the addition operation are only added together, the noise terms from the ciphertexts in the multiplication operation are multiplied together. After $n$ additions the noise will have a linear growth, $r \cdot n$, while after $n$ multiplications the noise will have exponential growth, $r^n$.

### 2.1.2 Bootstrapping

To transform this "somewhat" homomorphic encryption scheme into a fully homomorphic scheme the noise must be reduced while still maintaining the homomorphic property of the ciphertext. Gentry proposed his bootstrapping technique as a solution to this problem. By taking the decryption algorithm for the encryption scheme and converting it into a circuit with the ciphertext and the encryption of the private key as input, the outcome of the circuit would be a recryption of the ciphertext. If the decryption circuit is cheap enough to evaluate, then the output ciphertext will have less noise than the input ciphertext.

Figure 2.1 illustrates a decryption circuit of a scheme where $C = (c_0 \ \ldots \ c_{n-1})$ is an encryption of a single plaintext bit, $m \in \{0,1\}$, and $k = (k_0 \ \ldots \ k_{l-1})$ is the associated key. The operations are done on single bits.

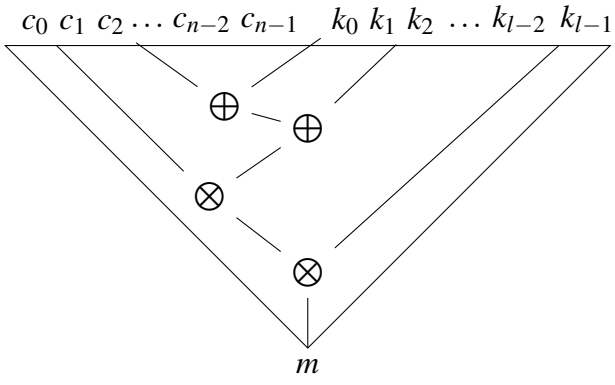


Fig. 2.1 Decryption circuit of a scheme, operations are done on single bits.

In the bootstrapping process both the bits of the ciphertext $C$ and the associated key $K$ will be encrypted with a public key, $Pk$.

$$Enc(C, Pk) = C^* = (c_0^* \ \ldots \ c_{n-1}^*)$$

$$Enc(K, Pk) = k^* = (k_0^* \ \ldots \ k_{l-1}^*)$$

$C^*$ and $k^*$ will both be used as input to the new decryption circuit that is being evaluated homomorphically.

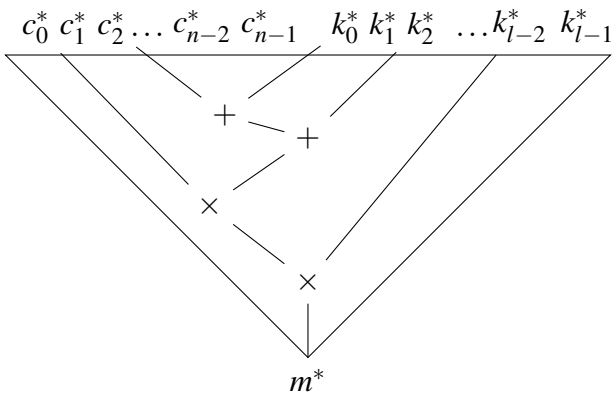


Fig. 2.2 Decryption circuit being evaluated homomorphically, operations are done on elements of some mathematical ring.

By encrypting the bits of the ciphertext and the key and run the homomorphic decryption circuit on them, we can remove all previous noise. The only noise that will remain is the noise introduced in the decryption circuit itself. A scheme is said to be bootstrappable if it can evaluate its decryption circuit homomorphically, and handle one additional operation without the noise growing too big. The problem with most schemes is that they are not able to evaluate their decryption circuit homomorphically. The decryption circuit can be very deep and costly, which will lead to a significant amount of noise being introduced to the ciphertext when the decryption circuit is evaluated. The noise added after going through the bootstrapping process may be so large that the decryption would fail on the recrypted ciphertext.

There have been some attempts at decreasing the noise added in the decryption circuit, such as augmenting the circuit, but this may affect the security of the scheme. Gentry's scheme could not handle its decryption circuit; however, he fixed this by adding "hints" about the secret decryption key in the public encryption key [7]. Introducing this information made the decryption circuit simpler, and the entire scheme became bootstrappable.

## 2.2 Using FHE in Practice

As mentioned, FHE has been widely discussed regarding its use within cloud computing. By eliminating the need to hand over the decryption key to the cloud provider, the clients do not have to trust their cloud provider.

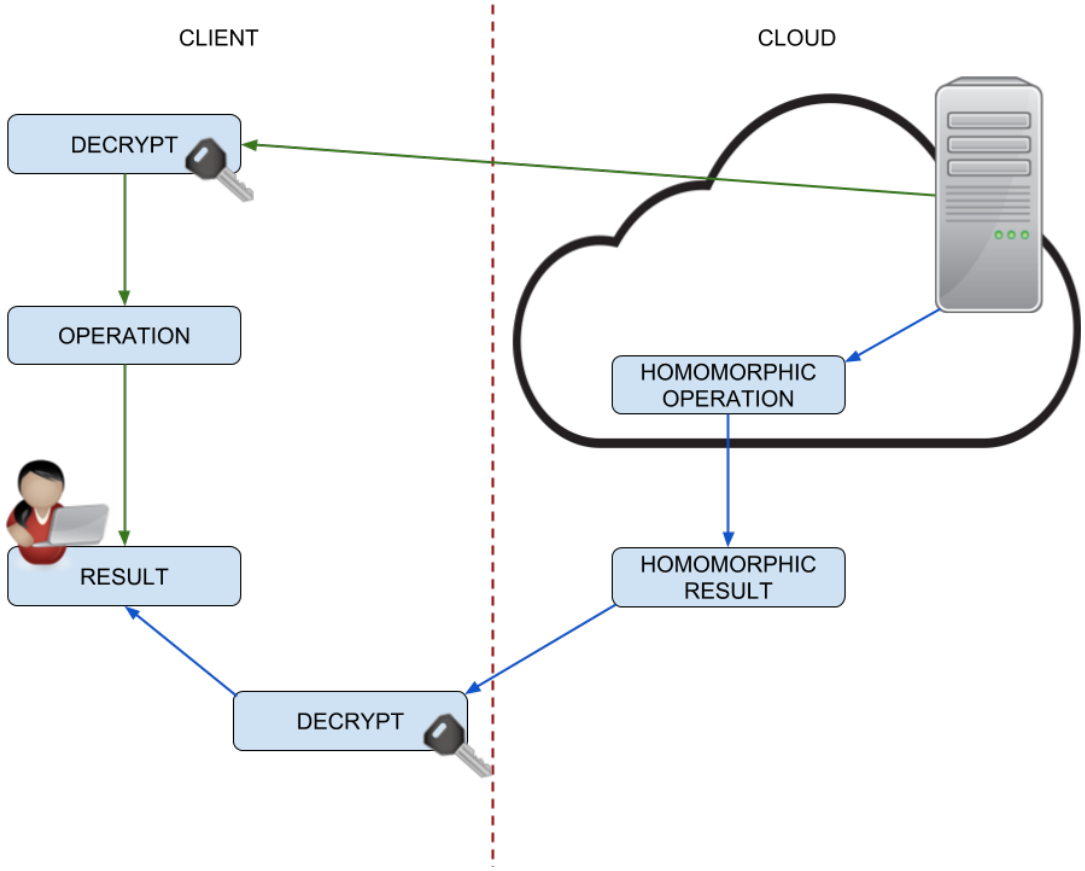


Fig. 2.3 Two different methods of extracting data from the cloud, based on the encryption of the data. Both end in the same result.

Figure 2.3 depicts two different approaches to extracting the result of some operation done on encrypted data, without giving the cloud provider access to the decryption key.

The green coloured path represents encrypting the data without FHE. In this approach, the client has to download and decrypt the entire data set from the cloud provider before they can perform operations on it. While the cloud provider only has to store the data, the client has to do most of the work.

The blue coloured path represents encrypting data with FHE. By using this approach, most of the work is handed over to the cloud provider. If a client asks for the result of an operation done on a specific part of the data, the cloud provider can perform the operations

on the ciphertexts and get an encrypted result. While still encrypted under FHE, the cloud provider will send only the result to the client who has to decrypt it themselves.

In Figure 2.3 the data is encrypted using FHE; however, not all applications are capable of performing this type of encryption. FHE requires a lot of processing powers, which not all devices may have. Is there a way such that devices with low processing power can store data encrypted under FHE in the cloud, without having to perform the FHE encryption itself?

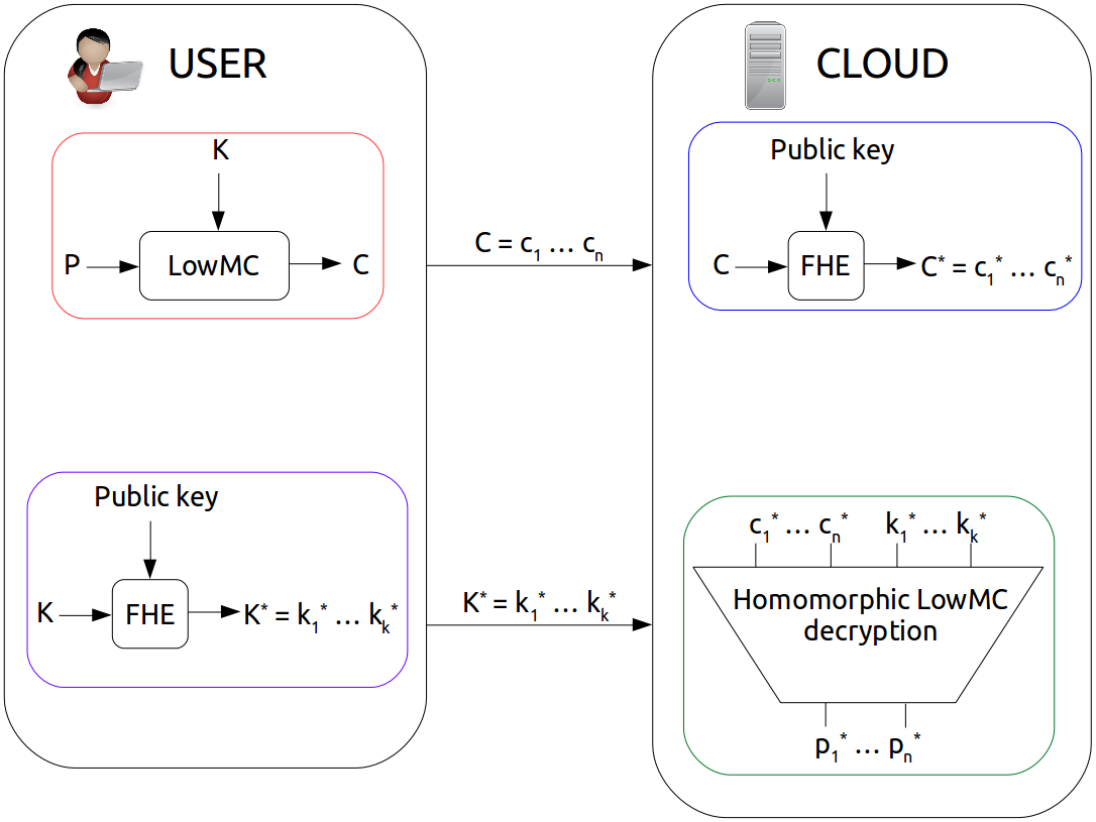


Fig. 2.4 How devices with low processing power passes the task of encrypting data with FHE to the cloud.

A proposed solution is shown in Figure 2.4. The user encrypts the data $P$ with the encryption key $K$ using a symmetric cipher and produces $C$, as shown in the red bordered box on the left. In this example, the data is encrypted with the symmetric cipher LowMC. The reason a symmetric cipher is chosen is that it is a lot cheaper to use than FHE and asymmetric encryption. For a device with low processing power, this attribute can be quite helpful.

The ciphertext $C$ is then sent to the cloud, where it is the cloud provider that again encrypts it, but now using FHE. The user, using FHE, encrypts the key $K$ with the same public key used by the cloud provider and produces $K^*$. The FHE encryption of the key $K$ needs only to be done once, and the encryption does not necessarily have to be done by

the device that encrypts the plaintext $P$. After being sent to the cloud, the key $K^*$ is passed through a homomorphic LowMC decryption circuit alongside the data $C^*$ as input.

The output from the circuit is the original plaintext $P$ from the user, encrypted with FHE. With this procedure, the user does not have to encrypt the data using FHE, as this task is left to the cloud provider. The user only has to encrypt the data using a symmetric cipher and encrypt the key using FHE, which both take little processing power.

## 2.3 HElib and BGV

HElib is a software library, created by Shai Halevi and Victor Shoup [8, 9], that implements homomorphic encryption. The specific scheme used in the library is the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [6].

### 2.3.1 (Levelled) FHE without Bootstrapping

The BGV scheme is a levelled FHE without bootstrapping. A scheme without bootstrapping must have its multiplicative depth of the circuits specified before being created. By removing the need for bootstrapping the performance improves, however, the security is dependent on the predetermined parameters of the scheme.

#### Parameters of the scheme

Table 2.1 gives a brief description of the parameters needed to initiate a BGV scheme in HElib. In HElib the parameter $m$ represents a specific modulus, however, to avoid confusion with a different parameter, which will later be defined, this parameter will be represented by *LWE dim* instead of $m$.

| Parameter | Description |
|---|---|
| *LWE dim* | a specific modulus |
| $p$ | plaintext base [default=2] |
| $r$ | lifting [default=1] |
| $L$ | number of primes in the modulus chain (i.e., the number of levels in the scheme) |
| $c$ | number of columns in the key-switching matrices |
| $B$ | bits per level |

Table 2.1 Parameters in HElib that must be chosen by the user.

The user-defined parameters in Table 2.1 will effect several other parameters in the BGV scheme in HElib, such as *security level*, the number of slots $s$, and the size of each element in a slot, $d$. We will focus only on the field $\mathbb{F}_2$ as the plaintext space containing exclusively the elements 0 and 1, so we always have $p = 2$ and $r = 1$.

### 2.3.2 Slots

HElib stores the plaintext in an array, where the size of the array is dependent on the predetermined parameters of the scheme,

$$s = \frac{\phi(LWE\ dim)}{d}.$$

Every position in the array represents a slot. The elements in each slot can be elements over the finite field $\mathbb{F}_{2^d}$.

The reason HElib uses an array to store the plaintext is that the BGV scheme supports SIMD (Single Instruction Multiple Data) operations. SIMD allows us to encrypt multiple plaintext bits in a single ciphertext object, where the number of encrypted bits in the ciphertext object is referred to as the number of slots. We will use the notation

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

to illustrate that ciphertext object $C$ encrypts the plaintext bits $p_1,\ p_2,\ \ldots\ ,\ p_s$.

The homomorphic operations in HElib are applied slot-wise. This means that given two ciphertext objects

$$C_a = \{(a_1,\ a_2,\ \ldots\ ,\ a_s)\}$$

$$C_b = \{(b_1,\ b_2,\ \ldots\ ,\ b_s)\}$$

adding or multiplying them homomorphically would produce the following

$$C_a + C_b = \{(a_1 \oplus b_1,\ \ldots\ ,\ a_s \oplus b_s)\}$$

$$C_a \times C_b = \{(a_1 \otimes b_1,\ \ldots\ ,\ a_s \otimes b_s)\}$$

where $\oplus$ is the regular XOR bitwise operation and $\otimes$ is the regular AND bitwise operation.

By combining HElib's property of slots and the slot-wise operations, we can use ciphertext objects to encrypt multiple bits and perform several operations simultaneously. Instead of having to encrypt each of the plaintext bits $a_1,\ a_2,\ \ldots\ ,\ a_s$ and $b_1,\ b_2,\ \ldots\ ,\ b_s$ in their own ciphertext objects, we are able to encrypt them in only two ciphertext objects and

multiply/add them together homomorphically using a single operation. This can be very useful when implementing a decryption circuit homomorphically, since operations done during decryption are simple and repeated many times. Thus it is easy to take advantage of the parallelism offered by HElib and the BGV scheme.

# Chapter 3

# LowMC

LowMC (Low Multiplicative Complexity) is a family of block ciphers proposed by Martin Albrecht et al., designed for FHE [4]. The goal is to minimise the number of multiplications, while still having a secure cipher. In FHE multiplications will cause much more substantial growth in noise than addition. Therefore, a cipher with low multiplication complexity can be a lot more efficient to evaluate homomorphically.

## 3.1 Description of LowMC

There are multiple variants of the LowMC block cipher, which differ based on how we choose the following parameters; the block size $n$, the key size $k$, and the number of S-boxes per round $m$. Despite these differences, the LowMC block cipher variants have many shared traits. Every LowMC cipher starts with a key whitening before continuing with the encryption rounds, where the rounds are built the same for every LowMC cipher variant. The encryption round consists of an S-box layer, an affine layer (which contains a linear layer and a constant addition), and a key addition.

### 3.1.1 S-box layer

The S-box layer in the LowMC block cipher is quite special since the S-boxes do not have to cover the whole cipher block. The number of S-boxes, $m$, in the S-box layer will differ between the various versions of LowMC ciphers.

While there are no strict rules for how many S-boxes we should use, having too few will affect the security, and having too many may affect the efficiency of the cipher. The number of rounds in the LowMC cipher compensates the number of S-boxes. With few S-boxes the number of rounds increases and with many S-boxes the number of rounds decreases.

Algebraic Normal Form of S-box

$$S_0(A,B,C) = A \oplus BC$$
$$S_0(A,B,C) = A \oplus B \oplus AC$$
$$S_0(A,B,C) = A \oplus B \oplus C \oplus AB$$

Fig. 3.1 Description of S-box.

In the S-box layer, the cipher block passes through the $m$ S-boxes, with 3 bits per S-box. The bits that do not pass through any of the S-boxes remain unchanged in this layer. Every S-box has three multiplications, as shown in figure 3.1, with a single multiplication in each output. Though the S-box is minimal, it retains the necessary non-linear properties.

### 3.1.2 Linear layer

In the linear layer, the cipher block is multiplied with a binary matrix of size $n \times n$. The matrices differ for each round, and the LowMC cipher does not have any strict restrictions on the matrices, except that they have to be invertible to make it possible to decrypt the ciphertext. It is, however, recommended that the matrices be either chosen at random or generated using the keystream from the Grain stream cipher [4]. Later in this thesis, we will propose predetermined matrices that work efficiently with HElib, without compromising the security.

### 3.1.3 Constant addition

In the encryption round, a constant binary vector of length $n$ is added to the cipher block. The vectors differ from each round and are randomly chosen.

### 3.1.4 Key addition

In the encryption round, the subkey for the round, a binary vector of length $n$, will be added to the cipher block. The keys, which differ for each round, are constructed by multiplying the master key, of length $k$, with a random binary matrix, of size $n \times k$. The round keys used in the encryption rounds will, therefore, have a length of $n$.

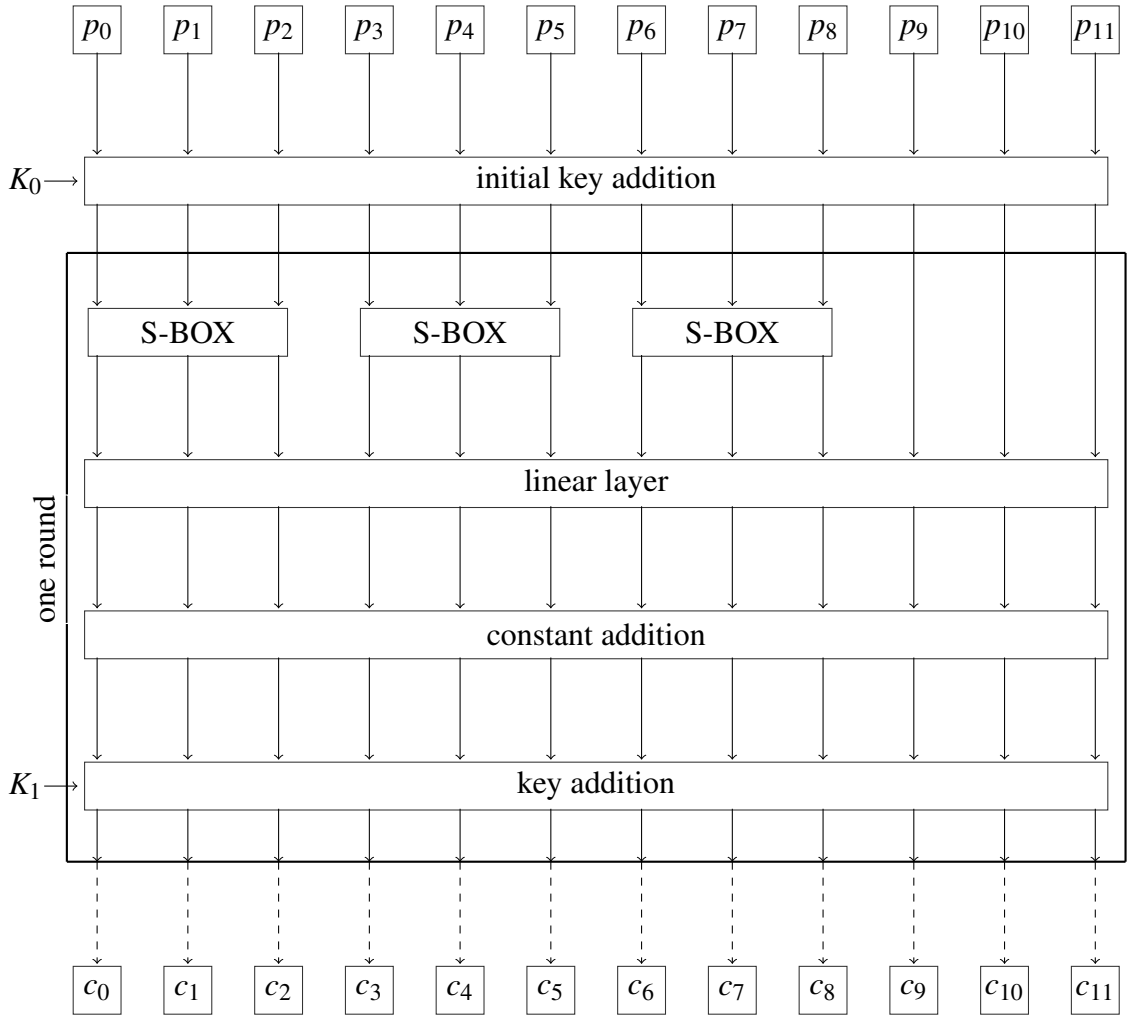


Fig. 3.2 Down-scaled LowMC with $n = 12$ and $m = 3$.

Figure 3.2 displays the first round of a down-scaled LowMC cipher, with a block size of twelve bits and three S-boxes. The plaintext bits that we want to encrypt are represented by $p_0, \ldots, p_{11}$ and the resulting ciphertext bits are represented by $c_0, \ldots, c_{11}$. In this cipher, the S-boxes cover 75% of the cipher block, where the last three plaintext bits are not passed through any S-boxes.

Though we are interested in evaluating the decryption circuit of the LowMC block cipher homomorphically, the encryption and decryption circuits are so similar that there would be little difference between evaluating the two. We have chosen to focus on the encryption circuit, something Martin Albrecht also decided to do in his paper, and will in the following only discuss homomorphically evaluating the LowMC encryption circuit.

| **Block size** $n$ | **# of S-boxes** $m$ | **Key size** $k$ | **# of rounds** $r$ |
|---|---|---|---|
| **256** | **49** | **80** | **12** |
| **128** | **31** | **80** | **12** |
| 64 | 1 | 80 | 164 |
| 1024 | 20 | 80 | 45 |
| 1024 | 10 | 80 | 85 |
| **256** | **63** | **128** | **14** |
| **196** | **63** | **128** | **14** |
| 128 | 3 | 128 | 88 |
| 128 | 2 | 128 | 128 |
| 128 | 1 | 128 | 252 |
| 1024 | 20 | 128 | 49 |
| 1024 | 10 | 128 | 92 |
| **512** | **66** | **256** | **18** |
| 256 | 10 | 256 | 52 |
| 256 | 1 | 256 | 458 |
| 1024 | 10 | 256 | 103 |

Table 3.1 Proposed variants of the LowMC block cipher.

Table 3.1 presents the different proposed variants of the LowMC block cipher. Albrecht focused on five different variants in his HElib implementation of LowMC [2], which are all marked in red in Table 3.1. We will focus mainly on his implementation of the LowMC cipher with block size 128, 31 S-boxes, key size 80, and 12 rounds. Our implementations will be similar to this variant, except that it has 32 S-boxes instead of 31. The reason for this alteration will be explained later in the thesis.

## 3.2 Martin Albrecht's implementation

Figure 3.2 shows how a single plaintext is being encrypted with the LowMC block cipher, however it is possible to encrypt multiple plaintexts at the same time. As previously mentioned the BGV scheme in the HElib library supports packing multiple plaintext bits in a single ciphertext object. Using this attribute in HElib, Martin Albrecht presented a method to homomorphically encrypt multiple plaintexts in parallel. In Albrecht's method, he packs

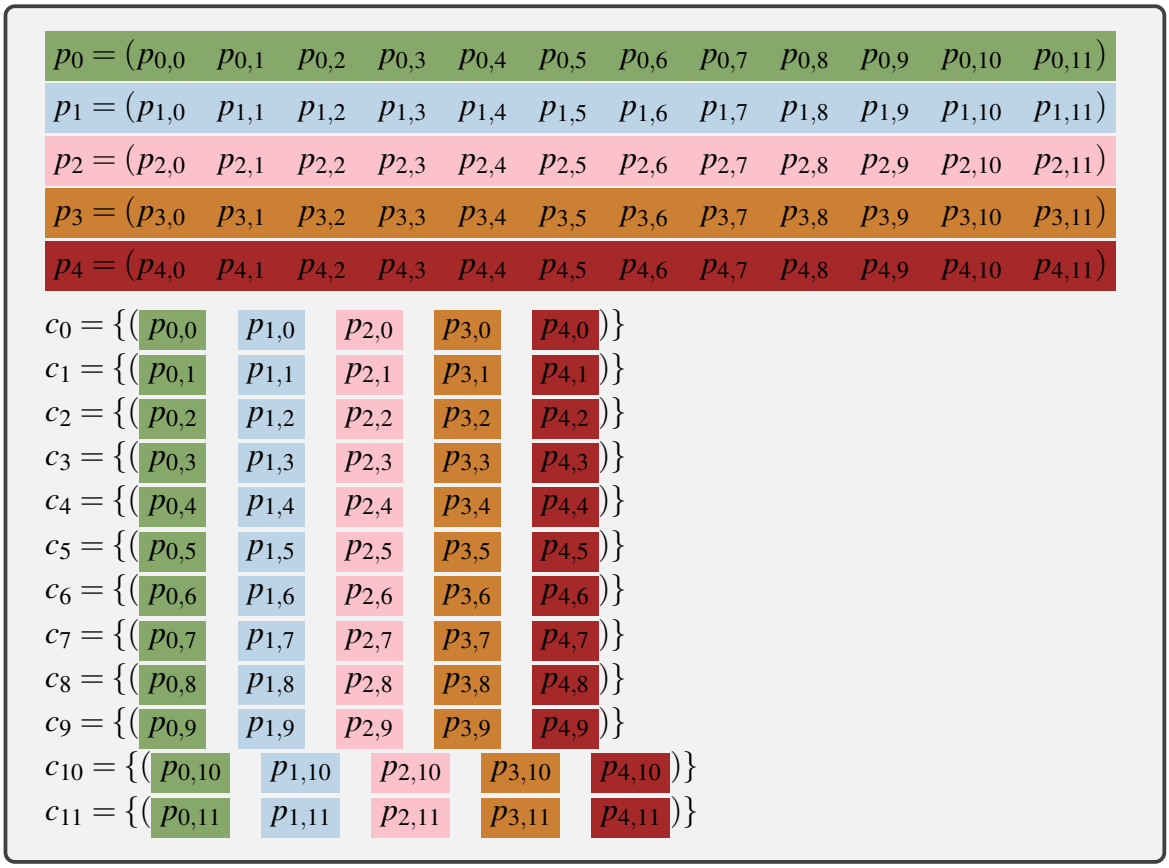


Fig. 3.3 Martin Albrecht's packing of plaintext bits.

the encrypted plaintext bits of multiple plaintexts, into the ciphertext objects in a particular manner.

Figure 3.3 shows how five plaintexts, $p_0$, $p_1$, $p_2$, $p_3$, and $p_4$ would be packed into ciphertext objects using Albrecht's method. As we can see from the figure, twelve ciphertext objects are used, the same amount as the number of bits in each plaintext. Instead of packing each encrypted plaintext bit in its own ciphertext object, the plaintext bits that occur in the same position in every plaintext, $p_0$, $p_1$, $p_2$, $p_3$, and $p_4$, are packed in the same ciphertext object. Had the plaintext bits been encrypted individually into their ciphertext object, the number of ciphertext objects used would be $12 \times 5 = 60$ instead of the 12 that are used in this scenario.

By using this packing technique, Albrecht can perform homomorphic operations on multiple plaintexts at the same time. However, since each ciphertext object contains bits from all the plaintexts, specific built-in methods of HElib cannot be used. One of these methods is the matrix-vector multiplication. If this method were to be used, bits from different plaintexts would affect each other, which is something we would not want to happen. Though Albrecht

cannot use all the built-in methods in HElib, his method needs only the homomorphic addition and multiplication methods to be able to encrypt homomorphically in HElib.

### 3.2.1 m4ri

M4ri stands for "Method of the 4 Russians" and is a library for fast arithmetic with dense matrices over $\mathbb{F}_2$, written by Martin Albrecht, Gregory Bard and William Hart [3]. In Albrecht's implementation of LowMC in HElib, he uses the library m4ri for the matrix-vector multiplications in the affine layer instead of the straightforward method. The library m4ri has a runtime of $\mathcal{O}(\frac{n^2}{\log n})$, unlike the straightforward method which has a run-time of $\mathcal{O}(n^2)$. With a block size 128, m4ri becomes 7 times faster than the straight-forward method.

### 3.2.2 Results

Table 3.2 shows the runtimes of Albrecht's implementation of the LowMC block cipher in HElib[2]. As mentioned in Chapter 2 the parameter *LWE dim* stands for a specific modulus, while *s* stands for the number of slots in the ciphertext object. The time spent on evaluating the entire LowMC procedure is represented by *eval*, and the time spent on evaluating the S-box layers is represented by *S-box*.

From Table 3.2 we can see that block wise Albrecht's method is quite fast, though the total time spent on the entire LowMC encryption does take at least a couple of minutes. Most of the time is spent on the S-box layers, which take up around 72%-77% of the total encryption evaluation time.

In January 2018 Albrecht updated his code for the implementation of the LowMC cipher to the current HElib API. The implementation with *LWE dim* $= 14351$ was run using his updated code, alongside the updated HElib library from January 2018. The implementation with *LWE dim* $= 13981$ was run using Albrecht's old code for the implementation of LowMC, which was from January 2015. This implementation used a version of the HElib library that was dated back to April 2014.

| **LWE dim** | **s** | **S-box** | **eval** | **eval per block**$(\frac{eval}{s})$ | **Security level** |
|---|---|---|---|---|---|
| 14351 | 504 | 173.04 (s) | 240.47 (s) | 0.4771 (s) | 92 |
| 13981 | 600 | 166.92 (s) | 217.71 (s) | 0.3629 (s) | 81 |

Table 3.2 Martin Albrecht's results from 2018 and 2015 with user-defined HElib parameters $p = 2$, $r = 1$, $L = 14$, $c = 1$, $B = 28$, of the LowMC cipher with $n = 128$, $m = 31$, $k = 80$, and $r = 12$.

Table 3.3 shows the times we got after reconstructing Albrecht's implementation of the LowMC cipher in HElib. The runtimes are calculated by averaging the times from ten different runs of the LowMC cipher in HElib. When reconstructing Albrecht's implementation of the LowMC cipher in HElib we only used his updated code from January 2018, and the HElib library from March 2018. It is not stated in Albrecht's documentation which versions of the GNU MP Bignum (GMP) library and the Number Theory Library (NTL) were used in his implementation. The libraries we used in our reconstruction were GMP v.6.1.2 and NTL v.10.5.0.

The reconstructed runtimes are a bit faster than the original ones when *LWE dim* $= 14351$, however, this can be explained by different processing powers. Albrecht had four cores with a frequency of 2.36GHz, while we had two cores with a frequency of 2.40GHz. The implemented code does not seem to take advantage of multiple of cores, and the 0.04GHz difference may have made our reconstruction run faster.

The reconstructed times are a bit slower than the original ones when *LWE dim* $= 13981$, with a lower security level. The original and reconstructed code use different versions of the HElib library, which may explain the different runtimes as well as the difference in security level.

| **LWE dim** | **s** | **S-box** | **eval** | **eval per block**$(\frac{eval}{s})$ | **Security level** |
|---|---|---|---|---|---|
| 14351 | 504 | 163.79 (s) | 236.13 (s) | 0.47 (s) | 92 |
| 13981 | 600 | 164.2 (s) | 233.16 (s) | 0.39 (s) | 61 |

Table 3.3 Reconstruction of Albrecht's implementation with code from 2018 with user-defined HElib parameters $p = 2$, $r = 1$, $L = 14$, $c = 1$, $B = 28$, of the LowMC cipher with $n = 128$, $m = 31$, $k = 80$, and $r = 12$.

# Chapter 4

# Alternative implementation of LowMC in HElib

An issue not addressed when discussing Albrecht's implementation of the LowMC cipher in HElib is what happens with the LowMC plaintexts after being homomorphically decrypted in the cloud. After the decryption, the plaintexts will have the same packing as before, however now encrypted under the fully homomorphic encryption. In Albrecht's implementation, multiple bits from different plaintexts are, therefore, stored in the same ciphertext object, which can be a problem in further processing. Before the cloud can continue their process of the plaintexts, the bits in the ciphertext objects may have to be rearranged such that each ciphertext object contains bits from only one plaintext. This rearrangement can be very time-consuming, making it very inefficient to encrypt multiple plaintexts in parallel. The authors do not mention this problem in paper [4]. Encrypting a single plaintext at a time could, therefore, be a better solution.

If we disregard this issue, Martin Albrecht's implementation is effective when encrypting multiple plaintexts simultaneously. However what happens when we only want to encrypt a single plaintext, or very few? The encryption times for Albrecht's implementation is not affected by the number of plaintexts we want to encrypt. Therefore encrypting one plaintext will take roughly the same time as encrypting $s$ plaintexts, when the ciphertext objects have $s$ slots. We show a real-world example, where it is necessary to encrypt a single plaintext at a time, instead of multiple.

For many people struggling with diabetes, it can be quite troublesome to keep track of their glucose levels. In recent years continuous glucose monitoring (CGM) has been used to help diabetes patients monitor their blood sugar, to manage their disorder better [1]. Patients who use CGM receive a small sensor that is placed right under their skin. This

sensor measures their glucose levels every five minutes, transmits this data wirelessly to a local display device (e.g. smartphone) and then sends it encrypted to the cloud.

While many countries have embraced this technology, some countries are hesitant to do the same because of strict privacy laws [10]. Since the data is decrypted in the cloud, there is no absolute insurance that the vast amount of medical data is stored safely and it is unclear who has access to it. FHE can eliminate a lot of the issues regarding storage of sensitive data. When it is not necessary for the cloud provider to be in possession of the decryption key, the need for trust in the cloud provider is eliminated.

With CGM it is the local device that is responsible for encrypting the data. Since FHE requires large amounts of processing power, the local device must encrypt the data using a symmetric cipher, like the LowMC block cipher. If Albrecht's implementation of the LowMC cipher, with $n = 128$ and $s = 600$, was used in the HElib library, the cloud provider would have to wait, $600 \times 5$ minutes $= 3000$ minutes $\approx$, 2 days to process the 600 ciphertexts from the local device before starting the decryption circuit. For patients who are dependent on near real-time results, waiting almost two days would defeat the purpose of the system. The cloud provider could run the decryption circuit immediately after receiving a single ciphertext and fill the remaining 599 slots with dummy data, however, this would take approximately the same amount of time as decrypting 600 ciphertexts and would be a waste of resources. Is it possible to run the LowMC cipher faster than Albrecht's implementation, when encrypting a single ciphertext at a time?

## 4.1 Parallel S-boxes

As we can see from the results of Albrecht's implementation, most of the time spent on the LowMC encryption is used in the S-box layers. The S-box layers cover 72-76% of the entire time taken for the LowMC encryptions. Our main goal is to lower the time taken on the S-box layers, for the overall time taken to decrease. Our idea for this challenge is to run the S-boxes in parallel in each round, instead of evaluating $m$ S-boxes serially. By taking advantage of the slots in HElib, we can pack the bits into four ciphertext objects, where three of them will pass through multiple S-boxes simultaneously. Instead of having to pack every bit into an individual ciphertext object, we can cut down the number of ciphertext objects used to only four, regardless of the block size.

The crucial part is the way the bits are packed, and we will use a down-scaled version of the LowMC cipher with $n = 12$ and $m = 3$ to present this.

$$p = (p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_6 \quad p_7 \quad p_8 \quad p_9 \quad p_{10} \quad p_{11})$$

$$\begin{aligned} C_0 &= \{(p_0 \quad p_3 \quad p_6)\} \\ C_1 &= \{(p_1 \quad p_4 \quad p_7)\} \\ C_2 &= \{(p_2 \quad p_5 \quad p_8)\} \\ C_3 &= \{(p_9 \quad p_{10} \quad p_{11})\} \end{aligned}$$

Fig. 4.1 Our packing using a down-scaled version of the LowMC cipher with $n = 12$ and $m = 3$.

In Figure 4.1, the plaintext $p$ is used as an example to show how bits from the plaintext are packed into four different ciphertext objects $C_0$, $C_1$, $C_2$ and $C_3$. The bits that will not pass through the S-boxes are packed into a single ciphertext object $C_3$. The remaining three ciphertext objects contain the bits that will pass through the S-boxes. The ciphertext object $C_0$ contains every third bit starting from $p_0$, $C_1$ contains every third bit starting from $p_1$, and $C_2$ contains every third bit starting from $p_2$. By encrypting the plaintext bits that will pass through the same S-box in the same slots in $C_0$, $C_1$, $C_2$, we can evaluate the S-boxes in parallel.

**S-box**

$$C_0 + C_1 \times C_2$$
$$C_0 + C_1 + C_0 \times C_2$$
$$C_0 + C_1 + C_2 + C_0 \times C_1$$

Fig. 4.2 Our S-box layer in the alternative implementation of the LowMC cipher in HElib.

Continuing with the example from Figure 4.1, Figure 4.2 shows the implementation of S-box layer. Because of the packing, we need fewer operations to evaluate the same amount of S-boxes. Had we used Albrecht's implementation we would have needed $m$-times the operations to evaluate the same amount of S-boxes, and twelve ciphertext objects instead of four.

It is necessary for efficiency that the S-boxes to cover exactly $\frac{3}{4}$ of the cipher block, such that the last ciphertext object can have the same amount of filled slots as the rest. In our implemented version of the LowMC cipher with $n = 128$, we need 32 S-boxes in order for them to cover $\frac{3}{4}$ of $n$.

## 4.2 The linear layer

In the linear layer of the LowMC cipher, the ciphertext objects will be multiplied with a random binary matrix of size $n \times n$. To do so, we may use HElib's built-in method, *mul*. Initially, we used the method called *matMul*, however when the HElib library was updated in February/March 2018 the built-in method *matMul* was optimised and changed its name to *mul*. Our tests have been done using *mul*, being the faster of the two methods.

The method *mul* takes a ciphertext object with number of slots $s$ and a matrix of size $s \times s$ and alters the given ciphertext, while it retains the same size. Given a ciphertext object

$$C = \{(a \quad b \quad c)\}$$

and a matrix

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix},$$

the method *mul* alters the given ciphertext object to

$$C = \{(a \cdot m_{11} + b \cdot m_{21} + c \cdot m_{31} \quad a \cdot m_{12} + b \cdot m_{22} + c \cdot m_{32} \quad a \cdot m_{13} + b \cdot m_{23} + c \cdot m_{33})\}$$

$$= \{(a \quad b \quad c) \cdot M\}$$

Following the syntax of HElib, we also write this as $M.mul(C)$.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} & a_{0,9} & a_{0,10} & a_{0,11} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} & a_{2,11} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} & a_{3,11} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & a_{4,9} & a_{4,10} & a_{4,11} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} & a_{5,11} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} & a_{6,11} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} & a_{7,11} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} & a_{8,11} \\ a_{9,0} & a_{9,1} & a_{9,2} & a_{9,3} & a_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} & a_{9,11} \\ a_{10,0} & a_{10,1} & a_{10,2} & a_{10,3} & a_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} & a_{10,11} \\ a_{11,0} & a_{11,1} & a_{11,2} & a_{11,3} & a_{11,4} & a_{11,5} & a_{11,6} & a_{11,7} & a_{11,8} & a_{11,9} & a_{11,10} & a_{11,11} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \\ p_{10} \\ p_{11} \end{bmatrix} = \begin{bmatrix} p_0' \\ p_1' \\ p_2' \\ p_3' \\ p_4' \\ p_5' \\ p_6' \\ p_7' \\ p_8' \\ p_9' \\ p_{10}' \\ p_{11}' \end{bmatrix}$$

Fig. 4.3 Linear layer of the down-scaled LowMC cipher with $n = 12$ and $m = 3$.

Figure 4.3 is a representation of the linear layer in the down-scaled LowMC cipher with $n = 12$ and $m = 3$, where $p_0, \ldots, p_{11}$ are the incoming bits and $p_0', \ldots, p_{11}'$ are the outgoing bits. We would like to implement this layer in HElib using the method *mul*, however, because of our alternative packing, this can be a bit tricky. Since our implementation has divided the encrypted plaintext bits into four different ciphertext objects, we cannot simply multiply each of them with a random matrix. Given the incoming plaintext bits $p_i$'s and matrix from Figure 4.3, we would like to implement a linear layer that produces the $p_i'$'s given in Figure 4.3, while still keeping the same packing as shown in Figure 4.1.

We solved this problem by splitting the incoming matrix into 16 smaller matrices. The reasoning behind this will be shown in the following two figures, where $C_0$, $C_1$, $C_2$, and $C_3$ are copies of the incoming ciphertext objects and $D_0$, $D_1$, $D_2$, and $D_3$ are the outgoing ciphertext objects.

$$\begin{bmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} & a_{0,9} & a_{0,10} & a_{0,11} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} & a_{2,11} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} & a_{3,11} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & a_{4,9} & a_{4,10} & a_{4,11} \\
a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} & a_{5,11} \\
a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} & a_{6,11} \\
a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} & a_{7,11} \\
a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} & a_{8,11} \\
a_{9,0} & a_{9,1} & a_{9,2} & a_{9,3} & a_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} & a_{9,11} \\
a_{10,0} & a_{10,1} & a_{10,2} & a_{10,3} & a_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} & a_{10,11} \\
a_{11,0} & a_{11,1} & a_{11,2} & a_{11,3} & a_{11,4} & a_{11,5} & a_{11,6} & a_{11,7} & a_{11,8} & a_{11,9} & a_{11,10} & a_{11,11}
\end{bmatrix}
\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \\ p_{10} \\ p_{11} \end{bmatrix}
=
\begin{bmatrix} p_0' \\ p_1' \\ p_2' \\ p_3' \\ p_4' \\ p_5' \\ p_6' \\ p_7' \\ p_8' \\ p_9' \\ p_{10}' \\ p_{11}' \end{bmatrix}$$

$$\begin{bmatrix} \{p_0'\} \\ \{p_3'\} \\ \{p_6'\} \end{bmatrix}
=
\begin{bmatrix} a_{0,0} & a_{0,3} & a_{0,6} \\ a_{3,0} & a_{3,3} & a_{3,6} \\ a_{6,0} & a_{6,3} & a_{6,6} \end{bmatrix}
\begin{bmatrix} \{p_0\} \\ \{p_3\} \\ \{p_6\} \end{bmatrix}
+
\begin{bmatrix} a_{0,1} & a_{0,4} & a_{0,7} \\ a_{3,1} & a_{3,4} & a_{3,7} \\ a_{6,1} & a_{6,4} & a_{6,7} \end{bmatrix}
\begin{bmatrix} \{p_1\} \\ \{p_4\} \\ \{p_7\} \end{bmatrix}
+
\begin{bmatrix} a_{0,2} & a_{0,5} & a_{0,8} \\ a_{3,2} & a_{3,5} & a_{3,8} \\ a_{6,2} & a_{6,5} & a_{6,8} \end{bmatrix}
\begin{bmatrix} \{p_2\} \\ \{p_5\} \\ \{p_8\} \end{bmatrix}
+
\begin{bmatrix} a_{0,9} & a_{0,10} & a_{0,11} \\ a_{3,9} & a_{3,10} & a_{3,11} \\ a_{6,9} & a_{6,10} & a_{6,11} \end{bmatrix}
\begin{bmatrix} \{p_9\} \\ \{p_{10}\} \\ \{p_{11}\} \end{bmatrix}$$

Fig. 4.4 Calculating $D_0$ in the alternative implementation of the down-scaled LowMC cipher with $n = 12$ and $m = 3$.

The last equation in Figure 4.4 shows how the ciphertext object $D_0 = \{(p_0' \quad p_3' \quad p_6')\}$, in our alternative implementation of the linear layer in HElib, is calculated. The first equation is the representation of the linear layer from Figure 4.3, and we will use this equation to explain how the ciphertext object $D_0$ is calculated.

In the matrix from the linear layer representation, the three rows that affect the calculations of $p_0'$, $p_3'$, and $p_6'$ are extracted to form four smaller $3 \times 3$ matrices. The red coloured bits form

the matrix $m_{0,0}$, blue bits form $m_{0,1}$, green bits form $m_{0,2}$ and the purple bits form the matrix $m_{0,3}$. When we look closer at the incoming plaintext bits these matrices are multiplied with, we can see that when these bits are extracted based on colour, they form the ciphertext objects $C_0$, $C_1$, $C_2$ and $C_3$. The red coloured incoming plaintext bits are the same bits represented in $C_0$, the blue bits are represented in $C_1$, the green bits are represented in $C_2$ and the purple bits are the same as the ones represented in $C_3$. The calculation of ciphertext object $D_0$ can therefore be rewritten as

$$D_0 = m_{0,0}.mul(C_0) + m_{0,1}.mul(C_1) + m_{0,2}.mul(C_2) + m_{0,3}.mul(C_3)$$

The calculation of ciphertext objects $D_1$ and $D_2$ is done in a similar manner, where $D_1$ uses the bits affecting the calculation of $p'_1$, $p'_4$, $p'_7$ and $D_2$ uses the bits affecting the calculation of $p'_2$, $p'_5$, $p'_8$.

$$\begin{bmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} & a_{0,9} & a_{0,10} & a_{0,11} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} & a_{1,11} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} & a_{2,11} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} & a_{3,11} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & a_{4,9} & a_{4,10} & a_{4,11} \\
a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} & a_{5,11} \\
a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} & a_{6,11} \\
a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} & a_{7,11} \\
a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} & a_{8,11} \\
a_{9,0} & a_{9,1} & a_{9,2} & a_{9,3} & a_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} & a_{9,11} \\
a_{10,0} & a_{10,1} & a_{10,2} & a_{10,3} & a_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} & a_{10,11} \\
a_{11,0} & a_{11,1} & a_{11,2} & a_{11,3} & a_{11,4} & a_{11,5} & a_{11,6} & a_{11,7} & a_{11,8} & a_{11,9} & a_{11,10} & a_{11,11}
\end{bmatrix}
\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \\ p_{10} \\ p_{11} \end{bmatrix}
=
\begin{bmatrix} p_0' \\ p_1' \\ p_2' \\ p_3' \\ p_4' \\ p_5' \\ p_6' \\ p_7' \\ p_8' \\ p_9' \\ p_{10}' \\ p_{11}' \end{bmatrix}$$

$$\begin{bmatrix} \{p_9'\} \\ \{p_{10}'\} \\ \{p_{11}'\} \end{bmatrix} = \begin{bmatrix} a_{9,0} & a_{9,3} & a_{9,6} \\ a_{10,0} & a_{10,3} & a_{10,6} \\ a_{11,0} & a_{11,3} & a_{11,6} \end{bmatrix} \begin{bmatrix} \{p_0\} \\ \{p_3\} \\ \{p_6\} \end{bmatrix} + \begin{bmatrix} a_{9,1} & a_{9,4} & a_{9,7} \\ a_{10,1} & a_{10,4} & a_{10,7} \\ a_{11,1} & a_{11,4} & a_{11,7} \end{bmatrix} \begin{bmatrix} \{p_1\} \\ \{p_4\} \\ \{p_7\} \end{bmatrix} + \begin{bmatrix} a_{9,2} & a_{9,5} & a_{9,8} \\ a_{10,2} & a_{10,5} & a_{10,8} \\ a_{11,2} & a_{11,5} & a_{11,8} \end{bmatrix} \begin{bmatrix} \{p_2\} \\ \{p_5\} \\ \{p_8\} \end{bmatrix} + \begin{bmatrix} a_{9,9} & a_{9,10} & a_{9,11} \\ a_{10,9} & a_{10,10} & a_{10,11} \\ a_{11,9} & a_{11,10} & a_{11,11} \end{bmatrix} \begin{bmatrix} \{p_9\} \\ \{p_{10}\} \\ \{p_{11}\} \end{bmatrix}$$

Fig. 4.5 Calculating $D_3$ in the alternative implementation of the down-scaled LowMC cipher with $n = 12$ and $m = 3$.

Figure 4.5 presents the calculation of ciphertext object $D_3 = \{(p'_9 \quad p'_{10} \quad p'_{11})\}$, which is approximately the same as when calculating the ciphertext object $D_0$. The main difference is that the extracted rows from the main matrix and the encrypted plaintext bits from the incoming ciphertext object, are the ones that affect the calculations of $p'_9$, $p'_{10}$, and $p'_{11}$.

We can therefore conclude that our alternative packing, using only four ciphertext objects, can pass through any arbitrary linear layer using the method *mul* 16 times. Though the previous figures 4.3, 4.4 and 4.5 use the down-scaled LowMC cipher with $n = 12$, this can

easily be scaled up to the LowMC cipher with $n = 128$ which is the version we implemented in HElib.

### 4.2.1 Timing results

There are multiple values of *LWE dim* that give the needed number of slots, $s = 32$, but we tried finding the *LWE dim* that gave a good security level with the lowest evaluation time for the given implementation. The user-defined HElib parameters have a major effect on both timings and security, and should, therefore, be picked carefully. Almost all of the parameters are equal to the ones used in Albrecht's implementation, except for *LWE dim*, $L$ and $m$. The number of slots, $m$, was increased by one to get a more efficient implementation and the number of levels in the scheme, $L$, was increased by one to ensure correct decryption of the bits.

Table 4.1 shows the results of our alternative implementation of the LowMC cipher using *mul*. As we can see the timings for the S-box layer has decreased a great deal, though the timings for the linear layer has increased considerably. It turns out that the method *mul* is quite slow, and the time saved in the S-box layer is not sufficient to justify the time spent in the linear layer.

| **LWE dim** | **S-box** | **eval** | **eval per bit**$(\frac{eval}{n})$ | **Security level** |
|---|---|---|---|---|
| 26849 | 12.61 (s) | 531.78 (s) | 4.15 (s) | 127 |

Table 4.1 Results of implementation using *mul* in linear layer, with user-defined HElib parameters $p = 2$, $r = 1$, $L = 15$, $c = 1$, $B = 28$, of the LowMC cipher with $n = 128$, $m = 32$, $k = 80$, and $r = 12$.

The reason Albrecht's linear layer was so efficient, was that it used the m4ri library. With Albrecht's implementation, only the bits in the same slot positions in the different ciphertext objects will be added together. In our alternative implementation, we are not able to apply the m4ri library since the different slots from one ciphertext object need to be mixed. Albrecht never performs any addition between any of the bits in the same ciphertext object, making it possible to use the m4ri library.

## 4.3 Rotation-based linear layer

LowMC does not have any specifications for the matrices in the linear layer, except that they need to be invertible and should be random to ensure good diffusion. Because we want to

keep the configuration of the bits in the cipher block for the sake of the S-boxes, we should look for a linear layer that better fits this organisation. We can try to find an alternative implementation of the linear layer that is better suited for HElib by taking advantage of the lack of specifications of the matrices used in the linear layer, and choosing matrices that work well with the HElib library.

After checking the different built-in methods in HElib, we found the method *rotate*. The *rotate* method takes in two parameters, a ciphertext object $C$ and an integer $r$, and returns the ciphertext object $C'$ where the slots have been cyclically rotated $r$ positions to the right. Rotating a ciphertext object

$$C = \{(p_1,\ p_2,\ \dots\ ,\ p_{n-1},\ p_n)\}$$

by $r = 2$ with the method *rotate*, gives

$$C' = \{(p_{n-1},\ p_n,\ p_1,\ p_2,\ \dots\ ,\ p_{n-3},\ p_{n-2})\}.$$

Since the rotation is cyclic, the encrypted bits $p_{n-1}$ and $p_n$ will continue its rotation to the beginning of the ciphertext object.

The *rotate* method is a lot faster than the *mul* method, and the execution time is independent of how many positions we rotate by. We have therefore chosen to use this method to optimise the linear layer, by proposing specific linear layers for the different rounds. We have created five unique linear transformations that are repeated as often as necessary to cover all the rounds in the LowMC cipher.

$$\begin{aligned}
rotations &= [16,8,4,2,1,16,8,4,2,1,\dots,16,8,4,2,1] \\
D_0^{\,j} &= C_0^{\,j} + (C_2^{\,j} >>> rotations[r]) + (C_3^{\,j} >>> rotations[r+3]) \\
D_1^{\,j} &= C_1^{\,j} + (C_0^{\,j} >>> rotations[r+1]) + (C_3^{\,j} >>> rotations[r+3]) \\
D_2^{\,j} &= C_2^{\,j} + (C_1^{\,j} >>> rotations[r+2]) + (C_3^{\,j} >>> rotations[r+3]) \\
D_3^{\,j} &= C_0^{\,j} + C_1^{\,j} + C_2^{\,j} + C_3^{\,j}.
\end{aligned} \tag{4.1}$$

Equations 4.1 depicts the linear layer for an arbitrary round $j$, where the size of the list *rotations* is equal to the number of rounds in the LowMC cipher plus three. The ciphertext objects $C_0^{\,j}, C_1^{\,j}, C_2^{\,j}, C_3^{\,j}$ form the input to the linear layer in round $j$, and $D_0^{\,j}, D_1^{\,j}, D_2^{\,j}, D_3^{\,j}$ represent the ciphertext objects after. These rotations and additions can be represented by multiplying the encrypted plaintext bits with a specific matrix and therefore still follow the LowMC specifications of the linear layer, except that the matrices are no longer random but predefined.

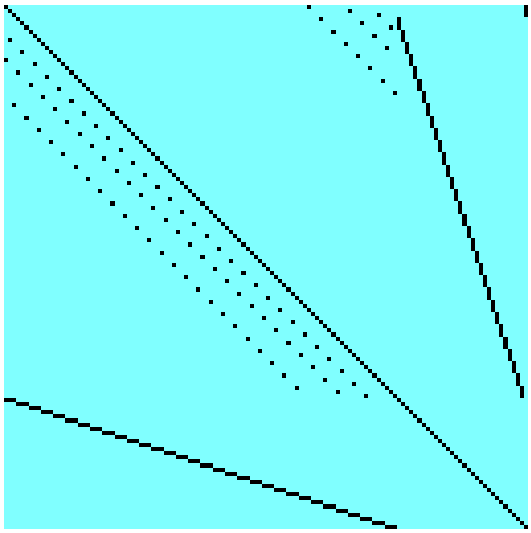

Fig. 4.6 One of the five proposed matrices for the linear layers

Figure 4.6 represents one of the five $128 \times 128$ matrix representations of the rotation-based linear layers. The blue pixels represent the 0-bits, while the black pixels represent the 1-bits. The matrix in Figure 4.6 is quite sparse with few 1-bits, similar to the other four matrices. This property makes the matrices fitting to use in the decryption process since the homomorphic decryption circuit is the only one that needs to be implemented, as seen in Figure 2.4, and should thus be kept as simple as possible.

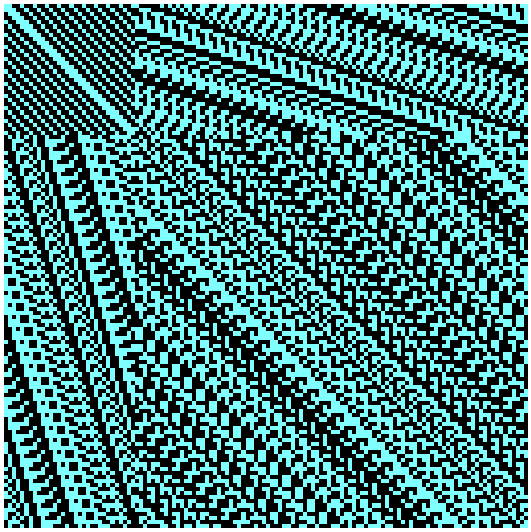

Fig. 4.7 The inverse of the matrix in Figure 4.6

Figure 4.7 shows the inverse of the matrix in Figure 4.6. Unlike the matrix in Figure 4.6, the inverse consists of a lot more 1-bits making it quite dense. This property will not cause

any problem when applying these matrices in the encryption process, since a basic matrix multiplication outside HElib will not be time-consuming.

### 4.3.1 Diffusion analysis of the linear layer

According to Shannon, a good cryptosystem holds two essential properties; diffusion, and confusion. Confusion means that the relationship between the ciphertext and the plaintext should be complicated and non-linear. In the LowMC cipher, it is the S-box layer that provides the confusion of the encryption.

Diffusion means that if a single bit in the plaintext changes, this bit should contribute to several bit changes in the ciphertext. The diffusion property in the LowMC cipher is achieved in the linear layer. Since we propose specific matrices used in the linear layer in our implementation of the LowMC cipher, we need to argue they give good diffusion.

We will use the notation ${C_i}^j$ to represent the incoming ciphertext objects in the linear layer in round $j$, where $i$ indicates which of the four ciphertext objects is being represented. ${D_i}^j$ represents the outgoing ciphertext objects of the linear layer.

Continuing with equations (4.1), we will take a closer look at how a single bit difference introduced in ciphertext object ${C_0}^j$ will spread from one active S-box to 32 active S-boxes. After going through the linear layer with the single bit difference in ${C_0}^j$, this bit will spread to ${D_0}^j$, ${D_1}^j$, ${D_3}^j$ based on the equations (4.1). Because ${D_1}^j$ gets influenced by a rotated ${C_0}^j$, unlike ${D_1}^j$ and ${D_3}^j$, the single bit difference will be placed in different parts of the ciphertext object than ${D_1}^j$ and ${D_3}^j$. The ciphertext object ${C_3}^j$ could have been used to cancel out the one-bit difference introduced by ${C_0}^j$ in either ${D_0}^j$ and ${D_1}^j$. However, ${C_3}^j$ rotated is also added to ${D_0}^j$, ${D_1}^j$, ${D_2}^j$ and introduces more one-bit differences than it would manage to cancel out. Therefore when ${D_0}^j$, ${D_1}^j$, ${D_2}^j$ pass through the S-box layers, at least two S-boxes will be activated by the single bit difference introduced.

In the next round, the one-bit differences existing in ${C_0}^{j+1}$, ${C_1}^{j+1}$, ${C_3}^{j+1}$ will be spread to at least four active S-boxes. Since ${C_0}^{j+1}$ and ${C_3}^{j+1}$ have their one-bit differences in different parts of the ciphertext objects, there may be a chance that ${C_3}^{j+1}$ rotated may cancel out the one-bit difference in ${C_0}^{j+1}$ which both contribute to ${D_0}^{j+1}$. However ${C_3}^{j+1}$ rotated will introduce a one-bit difference in ${D_1}^{j+1}$ and ${D_2}^{j+1}$. Both ${C_3}^{j+1}$ and ${C_1}^{j+1}$ have their one-bit difference in the same position, and when they are rotated with different values their contributions to ${D_1}^{j+1}$ and ${D_2}^{j+1}$ cannot cancel each other out.

Because of the different rotation amounts in each linear layer, each one-bit difference will spread to at least two other S-boxes. With this happening in each round, it will take five rounds for the single bit difference to spread and activate all $2^5 = 32$ S-boxes.

Our proposed matrices in the linear layer are safe from attacks starting with a one-bit difference, as long as the cipher has at least five rounds. However, is it possible to introduce the one-bit difference in the middle of the cipher? If the spread of the bits is the same when encrypting and decrypting, an attacker can start with a one-bit difference in the fourth round and move three rounds in both directions with the bits spreading to only $2^3 = 8$ S-boxes in each direction.

When we look at the matrices used in encryption and decryption, the decryption matrices are a lot denser. The bit differences will therefore not have the same amount of spread, as the decryption process will lead to a broader spread of bits. For an attacker to be able to produce a cipher block with a one-bit difference in the fourth round, they would have to start with a significant amount of bit differences in the previous round(s).

In conclusion, five rounds of LowMC with our proposed linear layer should be safe against linear and differential cryptanalysis. Since the actual cipher has twelve rounds, using our proposed linear layer gives more than sufficient security margin.

### 4.3.2 Timing results

After running our implementation of the LowMC cipher in HElib, using the linear transformations given in (4.1), we got the runtimes shown in Table 4.2. The overall time used to evaluate the cipher homomorphically has decreased considerably, and a rotation-based linear layer is the best solution for encrypting single plaintexts when using HElib. The code of our implementation is in Appendix A, and the associated makefile is in Appendix B.

While the alternative linear layer does decrease the runtimes a great deal, we can see from the table that the user-defined HElib parameters play a huge role when it comes to timing results and security level. By changing the *LWE dim* value, from 26849 to 12641, the overall evaluation time for the encryption is more than halved. While the bits per level, $B$, does not seem to have a substantial effect on the timings, the security level considerably decreases when increasing the $B$ value from 14 to 28.

When *LWE dim* $= 10057$, we notice that the security level decreases by a small amount when changing the value of $B$ from 14 to 28, compared to the other two values of *LWE dim*. The change in security level may be attributed to the increased $L$ when $B = 14$, however, we cannot explain the sudden need to increase $L$. The sudden change in parameters help illustrate how sensitive the BGV scheme is when choosing the values of *LWE dim* and $B$.

| **LWE dim** | **S-box** | **eval** | **eval per bit**$(\frac{eval}{n})$ | **B** | **L** | **Security level** |
|---|---|---|---|---|---|---|
| 26849 | 13.86 (s) | 26.29 (s) | 0.21 (s) | 14 | 15 | 304 |
| 26849 | 11.84 (s) | 22.50 (s) | 0.18 (s) | 28 | 13 | 144 |
| 12641 | 6.09 (s) | 11.53 (s) | 0.09 (s) | 14 | 15 | 108 |
| 12641 | 5.40 (s) | 10.20 (s) | 0.08 (s) | 28 | 13 | 9 |
| 10057 | 11.16 (s) | 32.78 (s) | 0.26 (s) | 14 | 25 | -3 |
| 10057 | 5.94 (s) | 17.83 (s) | 0.14 (s) | 28 | 13 | -17 |

Table 4.2 Results of implementation using *rotate* in linear layer, with user-defined HElib parameters $p = 2$, $r = 1$, $c = 1$, of the LowMC cipher with $n = 128$, $m = 32$, $k = 80$, and $r = 12$.

# Chapter 5

# Conclusion

Fully homomorphic encryption is still a very new area within cryptography. There have been many improvements since the first FHE scheme was proposed in 2009, and in years to come further advances will be made. One of the recent developments has been the use of slots in the BGV scheme. Each bit used to be encrypted by themselves in a single ciphertext object, however with the existence of slots it became possible to encrypt multiple bits simultaneously.

The primary focus of this thesis has been on evaluating the encryption circuit of the LowMC block cipher as efficiently as possible, in the FHE library HElib. With the use of slots in HElib, we were able to improve evaluation times for the encryption circuit. The slots in HElib are utilised differently based on the number of plaintexts we would like to encrypt. If several plaintexts need to be encrypted simultaneously it would be wisest to use Martin Albrecht's implementation of the LowMC cipher in HElib, provided the problem of disentangling all LowMC ciphertexts from each other can be solved. Albrecht packed several bits from different plaintexts into the same ciphertext object, making it possible to encrypt multiple plaintexts simultaneously. If there is only a single plaintext, or few, that needs to be encrypted, then it would be more efficient to use our alternative implementation. In our implementation we divide a single plaintext into four ciphertext objects, where three of them contain bits that will pass through the S-boxes, making it possible to run the $m$ S-boxes in parallel.

Up until now, when designing symmetric ciphers that can be used in FHE, it has been focused mainly on reducing the number of multiplications in the encryption algorithm. We believe it would be better to as well take into consideration the actual FHE schemes and libraries in the design process. By analysing the various methods and their efficiency, ciphers can be built to use the most efficient methods of the FHE scheme and libraries it was designed for. In our process of finding a more effective implementation of the LowMC cipher in HElib, we discovered that the built-in *mul* method was a lot slower than the *rotate* method. By

altering our implementation, in consideration of the BGV scheme and HElib library, we were able to cut down the evaluation time of the encryption circuit considerably.

When implementing the linear layer of the LowMC cipher, using the method *mul*, we only managed to get an encryption time of 531.78 seconds. While the S-box layers only used 12.61 seconds, most of the time was spent in the linear layer. By switching to the *rotate* method, our encryption time decreased to 22.50 seconds. Both of these implementations applied the same value for two of the user-defined HElib parameters $LWE\ dim = 26849$ and $B = 28$. However, we were able to further decrease the encryption time to 11.53 seconds by altering these parameters to $LWE\ dim = 12641$ and $B = 14$.

Through trial and error the user-defined HElib parameters, *LWE dim*, $B$, and $L$, applied in our alternative implementations of the LowMC cipher were found. Our goal was to find a combination of these parameters that gave an adequate security level and a decent runtime of the LowMC encryption circuit. Since we were not able to test all the different combinations, there may exist a combination of parameters that gives an adequate security level with a shorter encryption time.

For further work, we propose looking closer at other symmetric ciphers recommended for FHE use. Though we used the LowMC cipher when working with the BGV scheme and HElib library, there may be other ciphers that are more efficient when used in HElib. There may also be other FHE schemes and libraries that suits the LowMC cipher even better. What is the most optimal combination of a symmetric cipher, FHE scheme, and library?

# References


[1] (2018). Dexcom. http://www.dexcom.com. Accessed on 2018-05-09.

[2] Albrecht, M. (2018). lowmc-helib. https://bitbucket.org/malb/lowmc-helib, commit = f36db03.

[3] Albrecht, M., Bard, G., and Hart, W. (2010). Algorithm 898: Efficient multiplication of dense matrices over gf(2). *ACM Trans. Math. Softw.*, 37(1):9:1–9:14.

[4] Albrecht, M. R., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. (2015). Ciphers for mpc and fhe. In Oswald, E. and Fischlin, M., editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 430–454, Berlin, Heidelberg. Springer Berlin Heidelberg.

[5] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knezevic, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S., and Yalçın, T. (2012). Prince – a low-latency block cipher for pervasive computing applications. In Wang, X. and Sako, K., editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 208–225, Berlin, Heidelberg. Springer Berlin Heidelberg.

[6] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2011). Fully homomorphic encryption without bootstrapping. *IACR Cryptology ePrint Archive*, 2011:277.

[7] Gentry, C. (2009). *A fully homomorphic encryption scheme*. Stanford University.

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

[9] Halevi, S. and Shoup, V. (2018). Helib. https://github.com/shaih/HElib.

[10] Magnus, P. C. and Farestveit, E. (2017). Mobilapp kan hjelpe diabetessyke oliver (12) – helseregionene sier nei. https://www.nrk.no/hordaland/mobilapp-kan-hjelpe-diabetessyke-oliver-_12_-_-helseregionene-sier-nei-til-a-bruke-den-1.13685969. Accessed 14.05.2018.

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

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

# Appendix A

# Source Code

```
#include "matmul.h"
#include <iomanip>

vector<vector<Ctxt> > subkeys;
vector<vector<Ctxt> > constants;
vector<int > rotations = {16,8,4,2,1,16,8,4,2,1,16,8,4,2,1,16,8,4,2,1};
int nRounds;
double sbox_elapsed_secs = 0;
double linear_layer_elapsed_secs = 0;
double const_add_elapsed_secs = 0;
double key_add_elapsed_secs = 0;
double round_sbox_elapsed_secs = 0;
double round_linear_layer_elapsed_secs = 0;
double round_const_add_elapsed_secs = 0;
double round_key_add_elapsed_secs = 0;



vector<Ctxt> constant_addition(vector<Ctxt> ciphers,int round){
  for(int i = 0; i < ciphers.size(); i++){
    ciphers[i].addCtxt(constants[round][i]);
  }
  return ciphers;
}

vector<Ctxt> key_addition(vector<Ctxt> ciphers,int round) {
  for(int i = 0; i < ciphers.size(); i++){
    ciphers[i].addCtxt(subkeys[round][i]);
  }
  return ciphers;
```

```
}

vector<Ctxt> linear_layer(vector<Ctxt> ciphers,int round,
  const EncryptedArray& ea,const FHEPubKey& publicKey){

  Ctxt c0(publicKey), c1(publicKey), c2(publicKey), cR(publicKey),
  c0_non_rotation(publicKey), c1_non_rotation(publicKey),
  c2_non_rotation(publicKey);

  c0 = ciphers[0], c1 = ciphers[1], c2 = ciphers[2], cR = ciphers[3],
  c0_non_rotation = ciphers[0], c1_non_rotation = ciphers[1],
  c2_non_rotation = ciphers[2];

  ea.rotate(c0,rotations[round+1]);
  ea.rotate(c1,rotations[round+2]);
  ea.rotate(c2,rotations[round]);
  ea.rotate(cR,rotations[round+3]);

  // c0 + (c2 << rotations[round]) + (cR << rotations[round+3])
  ciphers[0].addCtxt(c2);
  ciphers[0].addCtxt(cR);

  // c1 + (c0 << rotations[round+1]) + (cR << rotations[round+3])
  ciphers[1].addCtxt(c0);
  ciphers[1].addCtxt(cR);

  // c2 + (c1 << rotations[round+2]) + (cR << rotations[round+3])
  ciphers[2].addCtxt(c1);
  ciphers[2].addCtxt(cR);

  // c0 + c1 + c2 + cR
  ciphers[3].addCtxt(c0_non_rotation);
  ciphers[3].addCtxt(c1_non_rotation);
  ciphers[3].addCtxt(c2_non_rotation);

  return ciphers;
}

vector<Ctxt> sbox(vector<Ctxt> ciphers,const FHEPubKey& publicKey){
  Ctxt s2(publicKey), s0(publicKey), s1(publicKey);
  s2 = ciphers[0], s0 = ciphers[1], s1 = ciphers[2];

  //c2*c3+c1
  s0.multiplyBy(ciphers[2]);
```

```
  s0.addCtxt(ciphers[0]);

  //c3*c1+c1+c2
  s1.multiplyBy(ciphers[0]);
  s1.addCtxt(ciphers[0]);
  s1.addCtxt(ciphers[1]);

  //c1*c2+c1+c2+c3
  s2.multiplyBy(ciphers[1]);
  s2.addCtxt(ciphers[0]);
  s2.addCtxt(ciphers[1]);
  s2.addCtxt(ciphers[2]);

  vector<Ctxt> outputFromSBoxes;
  outputFromSBoxes.push_back(s0);
  outputFromSBoxes.push_back(s1);
  outputFromSBoxes.push_back(s2);
  outputFromSBoxes.push_back(ciphers[3]);

  return outputFromSBoxes;
}

vector<Ctxt> lowmc_round(vector<Ctxt> ciphers,
  const FHEPubKey& publicKey,const EncryptedArray& ea,int round){

// ----------SBOX BEGIN----------
  clock_t sbox_begin = clock();
  ciphers = sbox(ciphers,publicKey);
  clock_t sbox_end = clock();
// ----------SBOX END----------

// ----------LINEAR LAYER BEGIN----------
  clock_t linear_layer_begin = clock();
  ciphers = linear_layer(ciphers,round,ea,publicKey);
  clock_t linear_layer_end = clock();
// ----------LINEAR LAYER END----------

// ----------CONSTANT ADDITION BEGIN----------
  clock_t const_add_begin = clock();
  ciphers = constant_addition(ciphers,round);
  clock_t const_add_end = clock();
// ----------CONSTANT ADDITION END----------

// ----------Key ADDITION BEGIN----------
```

```
  clock_t key_add_begin = clock();
  ciphers = key_addition(ciphers,round);
  clock_t key_add_end = clock();
// ----------Key ADDITION END----------

  round_sbox_elapsed_secs =
  double(sbox_end - sbox_begin) / CLOCKS_PER_SEC;

  round_linear_layer_elapsed_secs =
  double(linear_layer_end - linear_layer_begin) / CLOCKS_PER_SEC;

  round_const_add_elapsed_secs =
  double(const_add_end - const_add_begin) / CLOCKS_PER_SEC;

  round_key_add_elapsed_secs =
  double(key_add_end - key_add_begin) / CLOCKS_PER_SEC;

  sbox_elapsed_secs += round_sbox_elapsed_secs;
  linear_layer_elapsed_secs += round_linear_layer_elapsed_secs;
  const_add_elapsed_secs += round_const_add_elapsed_secs;
  key_add_elapsed_secs += round_key_add_elapsed_secs;

  return ciphers;
}

void get_constants(const EncryptedArray& ea,const FHEPubKey& publicKey){
  long n = ea.size();

  for(int round = 0; round < nRounds; round++){
    vector<Ctxt> constant;

    for(long i = 0; i < 4; i++) {
      vector<long> vector_constant;
      vector_constant.resize(n);

      for(int j = 0; j < n; j++) {
        vector_constant[j] = rand() % 2;
      }

      NewPlaintextArray plaintext_constant(ea);
      encode(ea,plaintext_constant,vector_constant);
      Ctxt cipher_constant(publicKey);
      ea.encrypt(cipher_constant,publicKey,plaintext_constant);
      constant.push_back(cipher_constant);
```

```
    }
    constants.push_back(constant);
  }
}

void get_subkeys(const EncryptedArray& ea,const FHEPubKey& publicKey){
  long n = ea.size();

  for(int round = 0; round <= nRounds; round++){
    vector<Ctxt> roundKey;

    for(long i = 0; i < 4; i++) {
      vector<long> vector_roundKey;
      vector_roundKey.resize(n);

      for(int j = 0; j < n; j++) {
        vector_roundKey[j] = rand() % 2;
      }

      NewPlaintextArray plaintext_roundKey(ea);
      encode(ea,plaintext_roundKey,vector_roundKey);
      Ctxt cipher_roundKey(publicKey);
      ea.encrypt(cipher_roundKey,publicKey,plaintext_roundKey);
      roundKey.push_back(cipher_roundKey);
    }
    subkeys.push_back(roundKey);
  }
}


int main(int argc, char *argv[]){
  long m, p, r, L, c, w,B;

  m=26849;
  p=2;
  r=1;
  L=15;
  c=1;
  w=80;
  B=14;
  nRounds=12;


// create text-file to store results
```

```
  ofstream myfile;
  stringstream filename;
  filename << "m=" << m << ",p=" << p << ",r=" << r <<
  ",L=" << L << ",c=" << c << ",w=" << w << ",B=" << B <<
  ",nRounds=" << nRounds << ".txt";
  myfile.open(filename.str());


// ----------SETUP BEGIN----------
  clock_t setup_begin = clock();

  FHEcontext context(m,p,r);
  context.bitsPerLevel=B;
  buildModChain(context,L,c);
  FHESecKey secretKey(context);
  const FHEPubKey& publicKey=secretKey;
  secretKey.GenSecKey(w/2);
  addSome1DMatrices(secretKey);
  EncryptedArray ea(context);
  long nslots=ea.size();

  clock_t setup_end = clock();
// ----------SETUP END----------

  myfile << "slots=" << nslots << ", m=" << m <<  "\n";
  myfile << "security level=" << context.securityLevel() << "\n\n";


// ----------READ DATA BEGIN----------
  clock_t reading_data_begin = clock();

  NewPlaintextArray p1(ea),p2(ea),p3(ea),pRest(ea);

  vector<long> plaintext_1=
  {1,1,0,1,0,0,1,1,0,1,1,0,1,1,1,1,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,1};

  vector<long> plaintext_2=
  {0,0,1,1,1,0,0,1,1,1,1,0,0,0,0,1,1,1,0,1,1,0,1,1,1,0,1,0,0,1,0,1};

  vector<long> plaintext_3=
  {1,0,0,1,0,0,0,1,0,1,1,0,1,0,1,1,1,1,1,1,0,0,0,0,1,1,1,0,0,1,0,0};

  vector<long> plaintext_rest=
  {0,0,1,1,0,0,0,1,0,0,1,0,0,1,0,0,1,0,0,0,1,0,1,1,1,0,1,0,0,0,1,0};
```

```
  encode(ea,p1,plaintext_1);
  encode(ea,p2,plaintext_2);
  encode(ea,p3,plaintext_3);
  encode(ea,pRest,plaintext_rest);

  get_constants(ea,publicKey);
  get_subkeys(ea,publicKey);

  clock_t reading_data_end = clock();
// ----------READ DATA END----------


// ----------CTXT ENCRYPTION BEGIN----------
  clock_t ctxt_encryption_begin = clock();

  Ctxt c1(publicKey),c2(publicKey),c3(publicKey),cRest(publicKey);
  ea.encrypt(c1,publicKey,p1);
  ea.encrypt(c2,publicKey,p2);
  ea.encrypt(c3,publicKey,p3);
  ea.encrypt(cRest,publicKey,pRest);

  vector<Ctxt> ciphers;
  ciphers.push_back(c1);
  ciphers.push_back(c2);
  ciphers.push_back(c3);
  ciphers.push_back(cRest);

  clock_t ctxt_encryption_end = clock();
// ----------CTXT ENCRYPTION END----------


// ----------LOWMC ENCRYPTION BEGIN----------
  clock_t lowmc_begin = clock();

  // ----------INITIAL KEY ADDITION BEGIN----------
  clock_t initial_key_addition_begin = clock();
  ciphers = key_addition(ciphers,nRounds);
  clock_t initial_key_addition_end = clock();
  // ----------INITIAL KEY ADDITION END----------

  for(int round = 0; round < nRounds; round++){
    myfile << "round " << setw(2) << round << "  ";
    myfile << "  baselevel: " << setw(2)
```

```
    << ciphers[0].findBaseLevel() << "␣␣";

    // ----------LOWMC ROUND BEGIN----------
    clock_t round_time_begin = clock();
    ciphers = lowmc_round(ciphers,publicKey,ea,round);
    clock_t round_time_end = clock();
    // ----------LOWMC ROUND END----------

    double round_elapsed_secs =
    double(round_time_end - round_time_begin) / CLOCKS_PER_SEC;

    myfile << "␣␣sbox:␣" << setw(8)
    << round_sbox_elapsed_secs << "(s)␣␣";
    myfile << "␣␣linear:␣" << setw(8)
    << round_linear_layer_elapsed_secs << "(s)␣␣";
    myfile << "␣␣const_add:␣" << setw(8)
    << round_const_add_elapsed_secs << "(s)␣␣";
    myfile << "␣␣key_add:␣" << setw(8)
    << round_key_add_elapsed_secs << "(s)␣␣";
    myfile << "␣␣t:" << setw(8)
    << round_elapsed_secs << "(s)␣␣";
    myfile << "␣␣baselevel:␣" << setw(2)
    << ciphers[0].findBaseLevel() << "␣␣";
    myfile << "\n";
  }
  myfile << "\n";

  clock_t lowmc_end = clock();
// ----------LOWMC ENCRYPTION END----------

  double setup_elapsed_secs =
  double(setup_end - setup_begin) / CLOCKS_PER_SEC;

  double reading_data_elapsed_secs =
  double(reading_data_end - reading_data_begin) / CLOCKS_PER_SEC;

  double ctxt_encryption_elapsed_secs =
  double(ctxt_encryption_end - ctxt_encryption_begin) / CLOCKS_PER_SEC;

  double initial_key_addition_elapsed_secs =
  double(initial_key_addition_end - initial_key_addition_begin)
  / CLOCKS_PER_SEC;

  double lowmc_elapsed_secs =
```

```
  double(lowmc_end - lowmc_begin) / CLOCKS_PER_SEC;


  myfile << setw(25) << "setup: " << setup_elapsed_secs << "s" << "\n";
  myfile << setw(25) << "reading data: " << reading_data_elapsed_secs
  << "s" << "\n";
  myfile << setw(25) << "encryption: " << ctxt_encryption_elapsed_secs
  << "s" << "\n";
  myfile << setw(25) << "initial key addition: "
  << initial_key_addition_elapsed_secs << "s" << "\n";
  myfile << setw(25) << "lowmc: " << lowmc_elapsed_secs << "s" << "\n";
  myfile << setw(25) << "sboxes: " << sbox_elapsed_secs << "s" << "\n";
  myfile << setw(25) << "linear layer: " << linear_layer_elapsed_secs
  << "s" << "\n";
  myfile << setw(25) << "constant additions: " << const_add_elapsed_secs
  << "s" << "\n";
  myfile << setw(25) << "roundkey additions: " << key_add_elapsed_secs
  << "s" << "\n";

  for(int i = 0; i < ciphers.size(); i++){
    if(!ciphers[i].isCorrect()){
      myfile << "\n" << "Ctxt is invalid: ";
    }
  }

  myfile.close();
}
```

# Appendix B

# Script for Makefile

```
#compiler to use
CC = g++

#compiler options
# -I are directories to search for included files
CFLAGS = -O3 -std=c++14 -w -I../HElib/src/

#linker options
# -L path to search for libcrary files
# -l name of libraries with -lTEST will link with libTEST.so
LDFLAGS = -L ../ntl-10.5.0/src/ -lntl -lm -lgmp

#list of source files separated by space
SOURCES = lowmc-helib.cpp

#list of object files to link with
OBJECTS = $(SOURCES:.cpp=.o) ../HElib/src/fhe.a

#name of executable that should be made
EXECUTABLE = runme

all: $(EXECUTABLE)

%.o : %.cpp
	$(CC) $(CFLAGS) -c $<

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(CFLAGS) $(OBJECTS) -o $@ $(LDFLAGS)

clean:
```

```
	rm -f $(EXECUTABLE)
	rm -f *.o
```