Chapter 1

Introduction to homomorphic encryption

1.1 An introduction to Encryption

Let's say it's 1943 and you are the head of a resistance movement that helps downed pilots escape. To get the pilots out of Europe, you need to be able to communicate to Britain to be able to arrange a pick-up in Spain. The only way to do this quickly and reliably is via the radio. However you are met with a serious problem: anyone can listen in to your broadcast. Therefore you need to send your message in a code. The British Special Operations Executive provided booklets called "one-time pads". These contained long rows of letters, as seen on Figure 1.1 and were used to turn the message, which we call "plain-text" into a jumble of seemingly meaningless letters. This process of turning plain-text messages into messages that cannot be understood by eavesdroppers is called encryption. Only the receiver in Britain, with the only copy of the one-time pad in the world, would be able to turn the encrypted message back into the original message.

What has been described is called symmetric key encryption, because both the sender and receiver are using the same key. The use of it in WW2 posed certain problems: records were kept of all intercepted messages by the occupying forces, even when they were not understood. This way, when a radio operator and his one-time pad were captured, earlier messages could still be deciphered. Secondly, some way had to be found to provide radio operators with the one-time pad booklet, and a new booklet had to be created for every radio operator.

How to provide a user with a key without accidentally giving an eavesdropper the opportunity to acquire this key has always been a problem. To solve this, public key cryptography was created. In public key cryptography, there is both a secret and a public key. The receiver (we'll call her Alice) makes his public key available to anyone who wants it, including potential eavesdroppers. However, the public key can only be used to *encrypt* the data. Once a potential sender (we'll call him Bob)

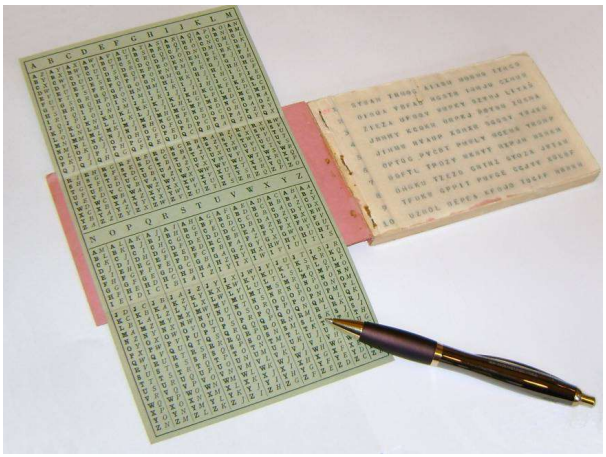

Figure 1.1: One Time Pad [16]

has encrypted his message, he cannot reverse this process.

It is as if Alice has sent every person in the world a safe with a lock but without a key. If someone like Bob wants to send Alice a message, they simply take the safe, place their message inside of it, and lock the safe. Bob can't unlock the safe anymore, but he can send the safe to Alice. Alice has a key that fits for all the safes she sent out, so she can unlock the safe and retrieve Bob's message, without an eavesdropper (who we'll call Eve from now on) being able to read the message, even if Eve has full access to the safe which contains Bob's message.

1.2 An introduction to Learning With Errors

Because there are many uses for cryptography, from securing messages to securing bank accounts, there are many cryptographic schemes, i.e. ways of encrypting and decrypting information. As previously mentioned, public/secret key schemes need to have a public key (i.e. a safe in which a sender can store information) and a secret key (a key to that safe that the receiver can use). If you are the receiver and you've chosen a secret key, you should be able to create a public key from this secret key. However the opposite should obviously be impossible! After all, the public key is being given away to anyone and should not give (much) information about the secret key. Given enough time, you might be able to reverse engineer the public key into a secret key, so we call a scheme "secure" (i.e. good enough) if it takes so long and requires so many resources that it might as well be impossible. Impossible meaning that the required time to break the secret key is on the order a small country dedicating all its computers for millennia for high enough security needs.

The schemes that were used over the last 50 years did a pretty good job of this. However, quantum computers, which may become a commercial reality in the future, could speed up the process of breaking these schemes massively. This means that there is a pressing need for new schemes that are based on public keys that cannot be reverse engineered into their secret keys.

Learning With Errors is a basis for many of these new schemes. The scheme approximately consists of taking the secret key and (part of) the public key, multiplying them together and then adding some random error. The result of this operation doesn't contain enough information to reverse engineer the secret key , and so the result can be distributed to anyone safely. In addition to being resistant to quantum attacks, LWE has another neat feature: it can be used for Homomorphic Encryption.

1.3 Homomorphic Encryption

Homomorphic encryption means encryption on which functions can still be executed. If you encrypt the value 3 and you encrypt the value 5, you can send both these encrypted values to a server and ask the server to add them together, then return the result back to you. Normally, the decrypted result would be some useless random value. Under a homomorphic encryption scheme however, the result after decryption would actually be 8! Fast, working homomorphic encryption would have a wide-ranging impact on the financial sector, the web services sector and many other sectors in which a company in need of computation cannot trust the providers of computation with their data.

There are 2 types of homomorphic encryption: Somewhat Homomorphic Encryption and Fully Homomorphic Encryption. Somewhat Homomorphic Encryption allows for a limited number of computations, Fully Homomorphic Encryption allows for an infinite number of computations. In 2009 Gentry [9] showed that Fully Homomorphic encryption, i.e. doing an arbitrary amount of operations on encrypted data, is possible.

Gentry explained this using an analogy: imagine a jeweler, Alice, who wants to let her workers work on precious gems without them being able to steal the gems. She creates a box that allows the workers to manipulate the jewels. This box represents Somewhat Homomorphic Encryption, because while the workers can do some operations, eventually the gloves stop working (so that the workers can no longer continue the work, although the jewels are still safe). Thus, the workers have to bring the box back to Alice so that she can unlock the box and retrieve the finished jewels. If the workers haven't finished their work yet, she must take the half-finished jewels from the box with the "worn" gloves and place them in a box with new gloves so that the workers can continue their work.[2]

Figure 1.2: The jewel box [2]

The “box” in this analogy is of course our encryption scheme. “Alice” is the user while the workers represent a server that can do operations (such as addition, multiplication or others) on the encrypted data without being able to access it. However, because the “box” in our case is the encryption scheme “Learning With Errors”, every operation that is done will increase the amount of error added to the final result. In the beginning, this error is small, meaning that we can still recover our result simply by rounding. At some point, the number of operations will cause the noise to exceed the threshold at which we can remove it. At that point, our “worker” (i.e. the server which is doing the operations) has to stop or the output it gives will be wrong.

The idea behind "bootstrapping" is to turn Somewhat Homomorphic Encryption into Fully homomorphic encryption. This can be done by performing the decryption, i.e. the unlocking of the box, while everything is encrypted under another encryption. In our analogy, this would be like locking our box in another box that already has a key for our box inside of it. (Since we are using public/secret key encryption, we can place things into a locked box with the public key without having to unlock said box. This is a feature of public key encryption).

We call this unlocking while keeping the box locked under a different box "bootstrapping". We call it "bootstrapping" because it allows us to keep working on the encrypted message.

Then our workers can open the inner box and continue working on the jewels. (see Figure 1.2) For an arbitrary amount of boxes, we can thus compute an arbitrary amount of functions on data that stay encrypted. The fact that gloves wear out in our analogy is representative of the fact that the homomorphic encryption schemes use some form of numerical noise to mask the message. As we do addition, or multiplication, the amount of noise increases. Multiplication increases the noise at a much greater rate than addition. Due to this noise increase, decryption of the data (whose last step is usually a sort of rounding operation to remove noise) will at some

point no longer return the correct answer, unless of course we bootstrap in time to reduce the noise by performing a decryption.

1.4 Encryption Libraries and homomorphic encryption schemes

Because Homomorphic Encryption would be so widely applicable, several software libraries have been created to provide software developers with the tools required to use schemes in their messenger apps/ database programs/ financial application software/ etc ....

We can split the homomorphic schemes that are currently the focus of most research into 2 (or 3) generations. The first generation consists of older schemes that are so inefficient they are no longer being used. The second generation consists of schemes like the Fan-Vercauteren [8] scheme, which is primarily used for Somewhat Homomorphic Encryption. The second-generation schemes allow for multiplication and addition but have a very compute-intensive bootstrapping process. The third generation schemes only allow for 1 gate (such as a AND/OR gate), and bootstrap immediately, but the bootstrapping happens relatively fast ($\tilde{1}$00ms)[12]. In this master thesis, I have accelerated one of the third generation schemes, namely Fastest Homomorphic Encryption in the West (FHEW) but it is still worth to first consider the Fan Vercautren scheme.

1.4.1 Fan-Vercauteren: usually Somewhat Homomorphic Encryption

FV has been the subject of hardware acceleration by multiple groups [13] [19] [17]. It is also implemented in the homomorphic encryption libraries SEAL and PALISADE among others.
The problem with the second generations schemes as previously mentioned is that bootstrapping takes so long the schemes usually forgo the bootstrapping for practical reasons. This means that any user of the libraries has a limited number of multiplications to work with (as we have seen, the limited noise growth of additions is usually negligible). This imposes constraints on the programmer.

In summary then: second-generation schemes execute fairly quickly, and as such schemes like FV are good for a certain class of programs with a limited amount of operations but a large amount of data. However, they only allow for a limited number of computations, and algorithms that are already out there and require a lot of instructions will thus not easily translate into programs that can homomorphically execute data. This can also be seen in a hardware acceleration for the Fan-Vercauteren scheme done by COSIC in 2019 [17]. This implementation could do

FV very quickly, at 400 homomorphic multiplications per second, but is limited to a multiplicative depth of 4. Although this is enough for a lot of applications, it can limit its flexibility in a general role.

1.5 FHEW (Fastest Homomorphic Encryption in the West)

FHEW (the name is a reference to the Fastest Fourier Transform in the West library) is part of the so-called third-generation schemes[5] [3]. It attempts to tackle the long length of the bootstrapping process by only executing one NAND gate (other simple gate functions are also possible) and then immediately bootstrapping. This makes it a true fully homomorphic scheme, as the bootstrapping can actually be done within a reasonable amount of time, namely 137 milliseconds for NAND + bootstrap for 128-bit security on an intel i7 [12]. In other words, while only one NAND gate can be done at a time (later papers show parallelisation is possible), there is no limit on the depth of our circuit, and as all functions can be written as a combination of logic gates, no limit on the functions that can be executed.

Because of its flexibility, a FHEW scheme that operated fast enough would have a large advantage in usability over the FV scheme. Programmers using FHEW do not need to worry about the noise growth of the scheme they are using, or when they should return their data back to the client. The entire FHEW scheme can be treated as a black box by those using it, and it is for this reason that I chose this scheme to accelerate.

1.6 Hardware Acceleration

We have previously mentioned the term *Hardware Acceleration* when talking about an implementation of Fan-Vercauteren. Hardware acceleration means building a specialized electronic circuit to make a certain function run faster (in less time) or using less energy. Electronics built for one function only are faster than electronics built to execute software. The reason for this is simple: general purpose electronics needs to occupy itself with reading the instructions that are sent to it. Specialized electronics don't need to do this, since they would always get the same instructions, and as such don't need to waste time on understanding what they have to execute and can dedicate all their resources to execute very specific operations. In this thesis, we use a FPGA, which is a form of *programmable* electronics, where different electrical components are wired together according to a netlist that can be programmed from a computer.

The main advantage of hardware acceleration is that it can provide a speedup of a factor 10 to a factor 100 for a given task.[17] The disadvantage is that by its very nature, it requires specialized electronics to be used. Those electronics might

one day end up in a consumer device, such as laptop, smartphone, or in the case of homomorphic encryption, a server, but this is unlikely to occur soon, and requires significant investment. Hardware requires a new chip to become part of the server hardware, and so would normally only start to function once the servers are replaced with new servers that contain the chip. Additionally, the encryption schemes which are hardware accelerated are still in flux, and there is no guarantee that a much faster scheme isn't just over the horizon, rendering those investments pointless. Software is much easier to deploy, and requires no expensive investment. However, recently web hosting companies such as Amazon Web Services have started making servers with programmable FPGA's available [18], which makes deploying hardware almost as easy as deploying software. Given that the computational bottlenecks for Homomorphic Encryption schemes mainly lie at the server side, hardware acceleration becomes a perfect solution to the problems faced by the homomorphic encryption schemes.

Chapter 2

Preliminaries

2.1 Definitions

A lot of lattice-based cryptography uses polynomial rings for various optimizations. The polynomials rings we use are defined as $R = \mathbb{Z}[x]/f(x)$ with $f(x) = x^d + 1$ and $d = 2^m$ with $m \in \mathbb{N}$. We also define $R_q = R/qR = \mathbb{Z}_q[x]/f(x)$, where again $f(x) = x^d + 1$ and $d = 2^m$ but $\mathbb{Z}_q$ defines the set of integers between $(-q/2, q/2]$.[8]

2.2 Ring Learning With Errors

The Fan-Vercauteren scheme [8] is based on the Ring Learning With Errors Problem, which states the following: For many vectors $(a_i, b_i) \in (R_q, R_q), b_i = a_i * s + e_i$ with e_i being a randomly sampled error term and s being our secret key and a_i being a randomly sampled but known vector, it is not possible to find s. In other words, (a_i, b_i) is a good public key for since we can't reverse engineer the secret from the public key. If we consider the problem from the case of the server: we cannot reverse engineer (a_i, b_i) into s, even if we have a very large quantity of a_i's and b_i's.

For our error terms, we consider the discrete Gaussian sampling function $D_{\mathbb{Z},\sigma}$. To get our error term, we can simply sample our coefficients from this Discrete Gaussian function, so that the error distribution $\chi = D^d_{\mathbb{Z},\sigma}$.

2.3 Fan Vercauteren

Consider the following simplified Fan Vercauteren scheme: (equations showing addition and multiplication are still valid).

The first part of our public key is:

$$b = (-a * s - e) \bmod q \tag{2.1}$$

with

$$a \in R_q, s \in R_q \text{ and } e \in \chi \tag{2.2}$$

which we put together with a to form (a, b).
Our cipher text is then:

$$c = ((b \bmod q) * u + e_1 + m, a * u + e_2) = (c_0, c_1) \tag{2.3}$$

with u randomly sampled and e, e_0 and e_1 randomly sampled error terms.
To decrypt , we then simply multiply the second part of the ciphertext c_1 with the secret key s and add this to the first part c_0.

$$m = \lfloor c_0 + c_1 * s \rceil \tag{2.4}$$

Adding or multiplying ciphertexts together will increase the noise but will (for low enough noise values) still return the correct answer when decrypted. [8].

2.4 FHEW (Fastest Homomorphic Encryption in the West)

2.4.1 Additional definitions

Our thesis focuses on FHEW. To explain how it works in detail, we must therefore create some additional definitions. Since, for a hardware engineer, the main interest lies in the parameter sizes, we will the numbers along with every parameter that we define for the parameter set (STD 128 [12]) that we have decided to use.

First we redefine the RLWE encryption of a message $\tilde{m} \in R_q$ with a secret key s as: [12]

$$RLWE_s(\tilde{m}) = (a, as + e + \tilde{m}) \tag{2.5}$$

with as previously defined, $a \leftarrow R_q$ and randomly sampled and $e \leftarrow \chi_\sigma^d$ with chi_σ being a discrete Gaussian distribution with σ as parameter which is sampled d times (once for every coefficient) to form χ_σ^d. The secret key s can be sampled from R_q or from a smaller distribution. This impacts the performance, but also of course the security. For the purpose of this work we focus on Gaussian (i.e. the most general) secret keys.

We have previously mentioned that our actual message m is a bit, not a integer. We have $\tilde{m} = \frac{q}{t} * m$ with q as always our modulus (=512 in our parameter set) and t being dependent on which LWE scheme we use (we write LWE_s^t). Figure 2.1 shows a representation of such a message. From the representation, it is immediately obvious that the error should not exceed $\frac{q}{4} = 128$.

We can now define our decryption function as:

$$RLWE_s^{-1}(a, b) = b - a * s = \tilde{m} + e \tag{2.6}$$

We can then recover m from $\tilde{m}$ by multiplying with $\frac{t}{q}$ and rounding the error away. We can now describe how to perform a NAND game using this scheme is such a way that there is only a little bit of error growth.

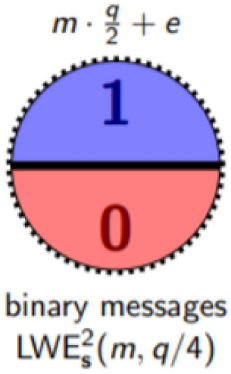

FIGURE 2.1: Binary Message representation.[7]

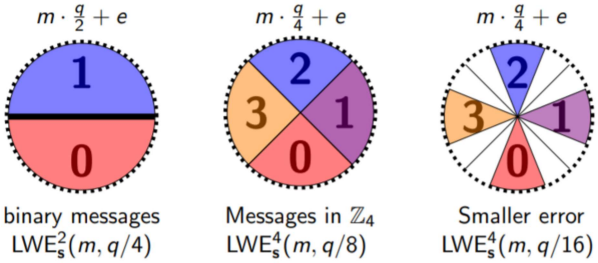

FIGURE 2.2: Messages in $LWE^4_s(m, q/8)$ [7]

2.4.2 The decryption function

As previously mentioned, our message is a single bit. If it is 0, we could represent it as some error, if it is 1, we could represent it as ($\frac{q}{2} = 256$ with some error added to it, as shown in Figure 2.1. In this case, our maximum error is equal to +/- $\frac{q}{4} = 128$. Thinking back to our parameter t, we see that this scheme uses $t = 2$. We can now define The representation that we uses $t = 4$. This representation allows for 4 possible messages, with the message being multiplied by $\frac{q}{t} = 128$, which is $\frac{q}{4}$ (Figure 2.2). Our maximum error value in this case is +/- $q/8 = 64$. Nothing stops us from reducing the possible error further though, as can been seen on the right of Figure 2.2, so that there are values that are never reached by a given message and error. (white space on Figure 2.2).

We can use these messages for addition (Figure 2.3). After addition, the result now has more error than the beginning (Figure 2.4), because the noise adds up.

Our interest lies in making a functional NAND gate. For a NAND gate, consider the following table: (table 2.1)

Result of the Sum	0	1	2	3
Result of a AND gate	False (0)	False (0)	True (1)	N/A

Table 2.1: Creating an AND gate from the sum of 2 bits

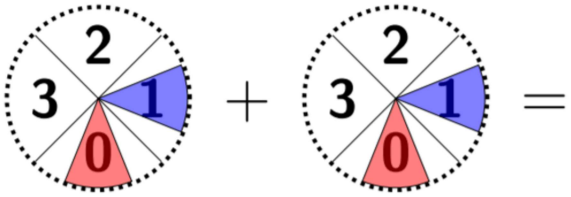

Figure 2.3: Addition of messages [7]

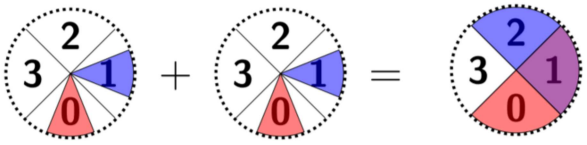

Figure 2.4: Result of the addition [7]

If we consider everything on the top left to be 1 and everything on the bottom right to be 0, we have an AND gate (Figure 2.6 and table 2.1). To make this a NAND, we rotate by $\frac{q}{2}$, in other words, we consider the bottom right to be 1 and the other half to be a 0, and this results in the results that we expect from a NAND gate for given input values (Figure 2.7).

2.4.3 Introduction to bootstrapping in FHEW

What we've discussed until now is the *underlying* encryption system. If we go back to our analogy from the introduction, our RLWE scheme is the box that the workers use to assemble the jewels. The gloves on this box are one-use only (we only do one NAND gate, or a similarly small binary gate operation). The workers now need

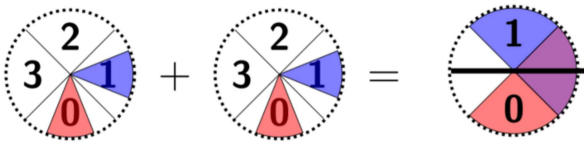

Figure 2.5: Rounding the sum [7]

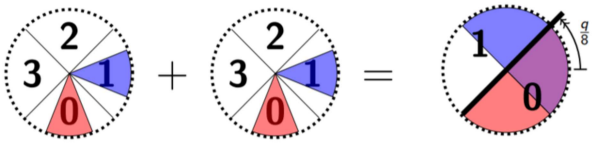

FIGURE 2.6: Creating an AND gate [7]

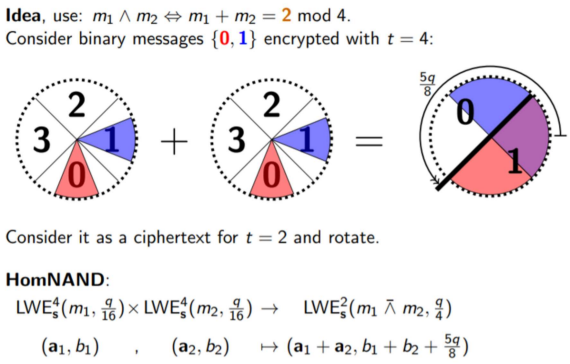

FIGURE 2.7: The final NAND gate [7]

a larger box in which take can put this RLWE box, so that they can unlock the RLWE box and put the jewels in a new (small, RLWE) box with new gloves. This (larger) box will be provided by RGSW, which we will define in the next paragraph. By encrypting our secret keys under RGSW, we can safely send them to the server without breaking security by giving the server the means to decypther the RLWE ciphertext into plaintext. Instead, we will decypher our $RGSW(RLWE(m))$ into a $RGSW(m)$, just like the jewels never leave the security of the outer box once the inner box has been unlocked.

As mentioned before this process is called bootstrapping. Bootstrapping is done by executing decryption operations homomorphically, i.e. doing the decryption operations with an encrypted secret key. This reduces the noise back down to the original level, completing our algorithm.

2.4.4 Definitions for bootstrapping

We define RLWE' [6] [12]:

$$\mathrm{RLWE}'_s(m) = (\mathrm{RLWE}_s(m), \mathrm{RLWE}_s(B*m), \mathrm{RLWE}_s(B^2*m), ..., \mathrm{RLWE}_s(B^{k-1}*m)) \tag{2.7}$$

with $B^k = q$.(In our parameter set,$B = 128 = 2^7$). The reason for this definition is that for the outer box, we want a system that creates as little noise as possible to ensure that the bootstrapping step doesn't add more noise than it removes.

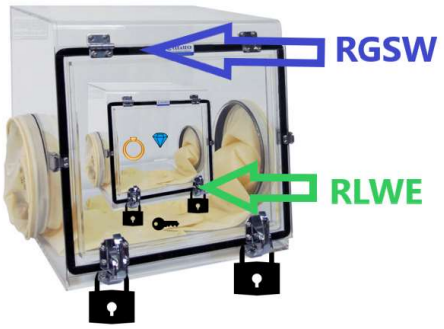

Figure 2.8: Our RLWE ciphertext is encrypted under a RGSW ciphertext so that it can be decrypted using RLWE secret keys, which themselves have been encrypted under RGSW

Multiplications are defined as:

$$(\odot) : R \times \text{RLWE}' \to \text{RLWE} : d \in R, \mathbf{c} \in \text{RLWE}' : d \odot (\mathbf{c}_0, ..., \mathbf{c}_{k-1}) = \sum_i d_i \mathbf{c}_i \quad (2.8)$$

We can extend this multiplication definition to a multiplication that results in RLWE':

$$(\odot') : R \times \text{RLWE}' \to \text{RLWE}' : d \odot \mathbf{C} = ((B^0 * d) \odot \mathbf{C}, (B^1 * d) \odot \mathbf{C}, ..., (B^{k-1} * d) \odot \mathbf{C}) \quad (2.9)$$

And finally, using these definitions, we can define our RGSW scheme, which will allow us to multiply ciphertexts together. We need this ability if we want to be able to run our RLWE decryption while everything is encrypted under RGSW.

$$\text{RGSW}_s(m) = (\text{RLWE}'_s(-s * m), \text{RLWE}'_s(m)) \quad (2.10)$$

To be able to use our secret key s encrypted under RGSW, we need to be able to multiply a RGSW ciphertext $RGSW(m_1) = (\mathbf{c}, \mathbf{c'})$ with a RLWE' ciphertext $RLWE_s(m_0, e_0) = (a, b)$ and get a result that when decrypted gives the product of the 2 messages [12]:

$$(a, b) \diamond (\mathbf{c}, \mathbf{c'}) = \langle (a, b), (\mathbf{c}, \mathbf{c'}) \rangle = a \odot \mathbf{c} + b \odot \mathbf{c}' \quad (2.11)$$

$$= a \odot \text{RLWE'}_s(-s * m_1) + b \odot \text{RLWE'}_s(m_1) \quad (2.12)$$

$$= \text{RLWE}_s((b - a * s) * m_1) = \text{RLWE}_s((m_0 + e_0) * m_1) \quad (2.13)$$

This $\text{RGSW} \times \text{RLWE}' \to \text{RLWE}$ can be turned into a $\text{RGSW} \times \text{RLWE}' \to \text{RLWE'}$, and then we have everything we need to do our bootstrapping.

$$\mathsf{ACC}_f[v] = \mathsf{RLWE}\left(\sum_{i=0}^{q/2-1} f(v-i)\cdot X^i\right) \in R_Q^2$$

$\mathsf{Init}_f(v)$:
  for $i = 0, \ldots, q/2-1$
    $m_i = f(v-i)$
  $\mathbf{m} = \sum_{i<q/2} m_i X^i = (m_0, \ldots, m_{q/2-1})$
  return $(\mathbf{0}, \mathbf{m})$

$\mathsf{Extract}_f(\mathbf{a}, \mathbf{b})$:
  let $(b_0, \ldots, b_{q/2-1}) = \mathbf{b}$
  return $(\mathbf{a}, b_0)$

FIGURE 2.9: Initialization and extraction step [7]

$$E(s) = \{\mathbf{Z}_{j,v} = \mathsf{RGSW}(X^{vB_r{}^j \cdot s}) \mid j < \log_{B_r} q, v \in \mathbb{Z}_{B_r}\} \in \mathsf{RGSW}^{B_r \log q / \log B_r}.$$

$\mathsf{Update}(c, \{\mathbf{Z}_{j,v}\}_{j,v} = E(s))$:
  for $j = 0, \ldots, \log_{B_r} q - 1$
    $c_j = \lfloor c / B_r{}^j \rfloor \bmod B$
    if $c_j > 0$
      $\mathsf{ACC} \leftarrow \mathsf{ACC} \diamond \mathbf{Z}_{j,c_j}$
  return ACC

FIGURE 2.10: Accumulation step [7]

Index of Coefficient	7	6	5	4	3	2	1	0
Value of Coefficient	0	0	0	0	f(-2) = f(2)	f(-1) = f(3)	f(0)	f(1)

TABLE 2.2: Initial polynomial for v=1 and n=8

2.4.5 Bootstrapping Algorithm

We now turn our attention to the bootstrapping process itself, first formally using the algorithm described in [12]. The bootstrapping consists of 3 steps: Initialization (see Figure 2.9), Accumulation (Figure 2.10) and Extraction (Figure 2.9).

The way that we implement both our rounding function and NAND gate that we previously described is as a look-up table. In a look-up table, the result of an input i is stored at index i and executing the function thus simple requires looking at the value at index i. In this scheme, a polynomial is used as look-up table. To explain our scheme however, we will use tables, and reduce the length of the table from $n = 512$ to $n = 8$.

Our look-up table is initialized to the formula seen on 2.9 (see table 3.1): This look-up table is shifted by our initial value so that the result at extraction will be found under index 0.

Index of Coefficient	7	6	5	4	3	2	1	0
Value of Coefficient	0	0	f(-2) = f(2)	f(-1) = f(3)	f(0)	f(1)	0	0

Table 2.3: Initial polynomial for v=1 and n=8

What we have after initialization is effectively a noiseless RLWE encryption of this polynomial. We can then perform the decryption of this RLWE encryption using secret keys encrypted under the RGSW scheme because a RLWE cipthertext is after all just a single component of a RGSW ciphertext.

3.1: Now, addition on the RLWE level is then performed using multiplication on the RGSW level. The reason for this is again our look-up table. When we rotate the look-up table, from our RLWE perspective we are adding $a_j * s_j$ from the index (which was originally v). From a RGSW perspective, a rotation is a multiplication of the polynomial with a value $X^{a_j * s_j}$. This is how we can use the multiplication $RGSW \odot' RLWE' \rightarrow RLWE'$ to perform the decryption $b - \mathbf{a} * \mathbf{s}$.(Table 2.3) shows this rotation through a look-up table.

After extracting the correct value from the polynomial (i.e. the coefficient of X^0), which is the constant term or the rightmost value in our table, a key-switching step occurs to change from the keys used in the Accumulator back to the keys of the RLWE. This step does not concern this thesis as we are mainly interested in resolving the bottleneck operation of FHEW, which is in the accumulator loop.

2.5 Notes on previous optimisations

The version that is accelerated in this thesis is an optimized version([12]) of the FHEW scheme originally introduced in 2014 [6]. This was chosen because it was the simplest of all third-generation schemes. The only optimisation paper on this scheme yet was an effort to make FHEW run on multicore CPU and GPU's, where they achieved a speedup of about 2.2 using CUDA on the 2015 version of FHEW. [10]

Another optimisation paper introduced NuFHE, which uses another third-generation scheme called TFHE with GPU acceleration and succeeded in a 100 times speedup, bringing the cost of homomorphic encryption down to 0.13 ms/bit for binary gates (for FFT implementation). For NNT implementations it is 0.35 ms/bit [14]. However, this scheme implemented full adders and took advantage of the fact that multiple bits can be packed into one cipher text, then divided the total bootstrapping speed by the number of bits in one adder. No attempt at packing is made in this thesis.