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 $(-q/2, q/2]$.[7]

2.2 Ring Learning With Errors

The Fan-Vercauteren scheme [7] 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. [7].

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 [10]) 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: [10]

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

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.

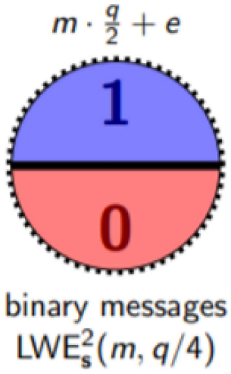

FIGURE 2.1: Binary Message representation.[6]

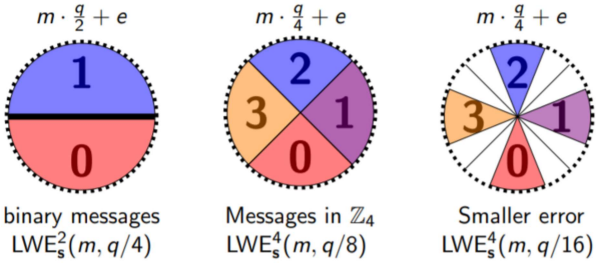

FIGURE 2.2: Messages in $LWE^4_s(m, q/8)$ [6]

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

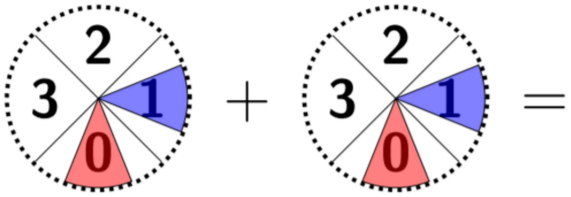

Figure 2.3: Addition of messages [6]

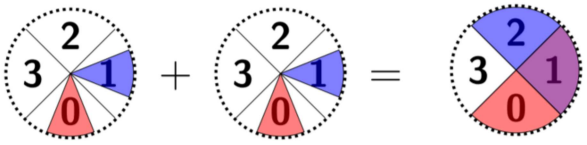

Figure 2.4: Result of the addition [6]

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 2.5 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

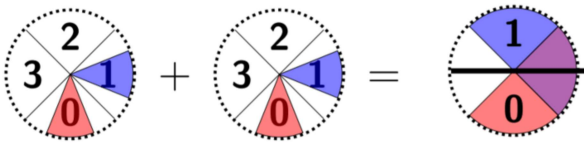

Figure 2.5: Rounding the sum [6]

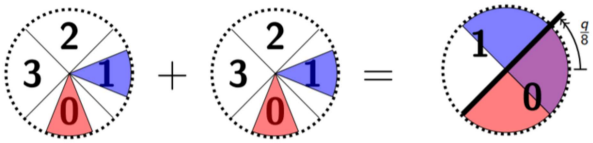

FIGURE 2.6: Creating an AND gate [6]

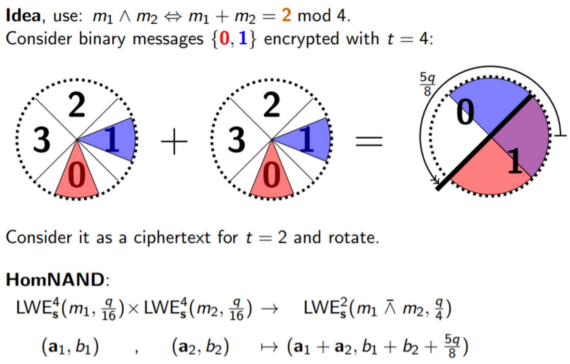

FIGURE 2.7: The final NAND gate [6]

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 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' [5] [10]:

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

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. Multiplications

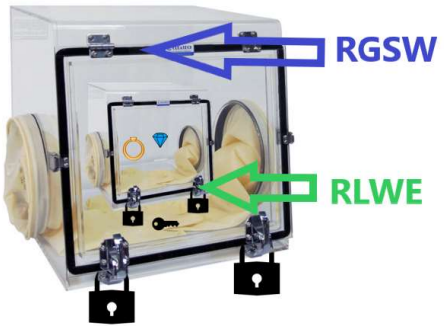

Figure 2.8: Our RLWE ciphertext is encrypted under a RGSW ciphertext so that it can be decrypted using RLWE secret keys, which themselves have been encrypted under RGSW

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.

$$RGSW_s(m) = (RLWE'_s(-s * m), RLWE'_s(m)) \quad (2.10)$$

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 [10]:

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

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

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

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

2.4.5 Bootstrapping Algorithm

We now turn our attention to the bootstrapping process itself, first formally using the algorithm described in [10]. The bootstrapping consists of 3 steps: Initialization

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

$\mathsf{Init}_f(v)$:
  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 [6]

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

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

FIGURE 2.10: Accumulation step [6]

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

(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 2.2: This look-up table is shifted by our initial value so that the result at extraction will be found under index 0.

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

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

all just a single component of a RGSW ciphertext.

2.2: 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([10]) of the FHEW scheme originally introduced in 2014 [5]. 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. [9]

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 [12]. 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.