TFHE

October 8, 2024

1 Introduction

TFHE is a Fully Homomorphic Encryption (FHE) scheme. It is an encryption scheme that allows you to perform computations over encrypted data.

1.1 Integer ring modulo $\mathbb{Z}_q$

An integer ring modulo q (also called integers modulo a prime number), denoted by $\mathbb{Z}_q$, is a mathematical structure known as a cyclic group or finite field in abstract algebra. Specifically, it is the set of residue classes of integers modulo q. This means it's the set of all possible remainders when integers are divided by q. For example, if q equals 2, then $\mathbb{Z}_q$ (which is $\mathbb{Z}_2$ in this case) includes only 0 and 1 because any integer either leaves a remainder of 0 or 1 when divided by 2. In the field of cryptography, we typically work with finite fields. The exciting thing about working in a finite field is that the polynomial coefficients "wrap around" when multiplied. So, regardless of how much polynomial arithmetic we perform, the polynomial coefficients can still be bounded in a fixed range, which is very attractive from an implementation perspective. If q equals 17, then $\mathbb{Z}_q$ (which is $\mathbb{Z}_{17}$ in this case) includes only 0 to 16 because any integer leaves a remainder of 0 to 16 when divided by 17. Hence the vector multiplication for vectors $g = [1, 2, 3, 4]$, and $h = [5, 6, 7, 8]$ with modulo a prime number $q = 17$ is $[5, 16, 0, 9, 10, 1, 15]$

1.2 Congruence

The symbol $\equiv$ denotes congruence and is different from equality. While equality means that items are the same, congruence means items have the same remainder when divided by mod. In a ring $\mathbb{Z}_{7681}$ and $n = 4$, the 4-th root of unity, which satisfy the condition $\omega^4 \equiv 1 \mod 7681$ are $3383, 4298, 7680$.

2 TFHE Ciphertexts

TFHE mainly uses three ciphertext types: LWE, RLWE, and RGSW. All of them have different properties which will be useful in the homomorphic operations

3 GLWE

General LWE, or GLWE includes both of LWE, RLWE

3.1 LWE

LWE encryption supports encrypting small-bit-width integers by placing those bits in the most significant bits of a machine word—for simplicity, say it's a 4-bit integer in the top-most bits of a

32-bit integer with all the other bits initialized to zero, and call that whole encoded plaintext. The secret key is a random binary vector of some fixed length chosen to achieve a specific security. Then, sample a random length vector of 32-bit integers to encrypt, take a dot product with the secret key, add the message, and add some random noise. The encrypted value is both the random samples chosen and the noise-masked result. LWE decryption then reverses this process: re-compute the dot product and subtract it from the output of the encryption. But at the end, you must apply a rounding step to remove the noise added to the ciphertext during encryption. Because the message is in the highest-order bits of the message (say, bits 28-31) and because the noise added was not very large, rounding to the nearest multiple removes it.

3.2 RLWE

In RLWE, the scalar multiplications and additions from LWE are upgraded to polynomial multiplications and additions. We pick a polynomial degree as the maximum degree (say 1024), the coefficients are always integers modulo some chosen modulus q, and finally, we pick a polynomial, usually $x^n + 1$, and represent the result of every operation as a remainder when divided by that polynomial. One or more small integer messages are encoded into a polynomial to encrypt. The secret key is a list of random polynomials with binary coefficients, and the samples are random polynomials with uniformly random mod q coefficients. Then, you take a dot product, add the message, and add a similar "noise polynomial" to mask the result. The main advantage of using RLWE over LWE is that you can pack many messages into a single polynomial and the homomorphic operations you apply to all the messages.

3.3 LWE and RLWE from GLWE

When we instantiate GLWE with $k = n$ and $N = 1$, we get LWE. Observe that R_q is actually Z_q when $N = 1$. We use small letters for (modular) integers (i.e., b, m, e ...).

$$b = \sum_{i=0}^{k-1} a_i.s_i + \triangle m + e \in \mathbb{Z}_q$$

When we instantiate GLWE with $k = 1$, we get RLWE. Here, we use capital letters for polynomials.

$$B = A.S + \triangle M + E \in R_q$$

3.4 Secret key

To generate any ciphertext, we first need a secret key. With GLWE ciphertexts, the secret key is a list of random polynomials from R:

$$\overrightarrow{S} = (S_0, S_1, ...S_{k-1}) \in R^k$$

The coefficients of the elements can be sampled from a uniform binary distribution, a uniform ternary distribution, a Gaussian distribution, or a uniform distribution. Please note that we can find parameters to achieve the desired security level for these secret keys.

3.4.1 Example

Let's choose N (degree or dimension) = 4 and k =2. Let's sample the secret key with a uniform binary distribution of a degree N — 1 polynomial. In this example, the secret keys are [0,1,1,0] and [1,0,1,1].

$$\overrightarrow{S} = ([0,1,1,0],[1,0,1,1]) \in R^2$$
$$\overrightarrow{S} = (0 + x + x^2 + 0x^3, 1 + 0x + x^2 + x^3) \in R^2$$
$$\overrightarrow{S} = (x + x^2, 1 + x^2 + x^3) \in R^2$$

3.4.2 Example TFHEpp lvl1param

For TFHEpp lvl1param, the value for N = 1024, k = 1. The key value maximum is 1, and the minimum is −1. For example:

$$\overrightarrow{S} = ([0,1,0,-1,-1.....1]) \in R^1$$
$$\overrightarrow{S} = (x - x^3 - x^5 - x^6.... + x^{1023}) \in R$$

3.5 Message

The message is a polynomial of degree smaller than N with coefficients whose maximum value depends on the value p.

$$M \in R^p$$

3.5.1 Example

Let's choose N (degree) = 4 and p (plain modulus) = 4. The coefficient values of the message are stored in 2 bits (p=4 = 2^2). The possible coefficient value in binary format is 11, 10, 00, and 01. The possible coefficient value in a signed integer is -2, -1, 0 and 1. The possible coefficient value in an unsigned integer is 3, 2, 0 and 1. Let Message be [-2 1, 0 -1] in this example.

3.5.2 TFHEpp Example

In the TFHEpp library, each message is binary (i.e., 1 or 0 only). For 16-bit integers, we will have 16 messages, one for each bit.

3.6 Mask

To encrypt the message, we must sample a uniformly random mask with coefficients whose maximum value depends on q (modulus).

$$\overrightarrow{A} = (A_0, A_1, ...A_{k-1}) \in R_q^k$$

3.6.1 GLWE Example

Let's choose N (degree) = 4, k =2 and q = 64 (modulus). The coefficient values of the mask are stored in 6 bits (q=64 = 2^6). The possible coefficient value in binary format is 111111, 111110 ..., 100000, 000000, 000001, ... 011111. The possible coefficient value in a signed integer is -31, -30, ...-1, 0, 1, .. 32. The possible coefficient value in an unsigned integer is 64, 63, ... 33, 0, 1, ..32. In this example, let Mask be [17, -2, -24, 9], [-14, 0, -1, 21].

$$\overrightarrow{A} = ([17, -2, -24, 9], [-14, 0, -1, 21]) \in R_{64}^2$$
$$\overrightarrow{A} = (17 - 2x - 24x^2 + 9x^3, -14 - x^2 + 21x^3) \in R_{64}^2$$

3.6.2 RLWE Example

Let's choose N (degree) = 4, k =1 (for RLWE) and q = 64 (modulus). The coefficient values of the mask are stored in 6 bits (q=64 = 2^6). The possible coefficient value in binary format is 111111, 111110 ..., 100000, 000000, 000001, ... 011111. The possible coefficient value in a signed integer is -31, -30, ...-1, 0, 1, .. 32. The possible coefficient value in an unsigned integer is 64, 63, ... 33, 0, 1, ..32. Let Mask be [17, -2, -24, 9] in this example.

$$\overrightarrow{A} = ([17, -2, -24, 9]) \in R_{64}^1$$
$$\overrightarrow{A} = (17 - 2x - 24x^2 + 9x^3) \in R_{64}$$

3.6.3 TFHEpp code

In file tlwe.hpp function tlweSymEncrypt(), it adds mask using std::uniform_int_distribution() C++ utility.

3.7 Error

We must add a discrete Gaussian Error (small coefficients) to encrypt the message. $\chi_{\mu,\sigma}$ is a Gaussian probability distribution with mean μ and standard deviation σ

$$E \in R_q$$

3.7.1 Example

Let's add [-1,1,0,1] error.

$$E = ([-1, 1, 0, 1]) \in R_q$$
$$E = (x + x^2, 1 + x^3) \in R_q$$

3.7.2 TFHEpp code

In TFHEpp, the noise is added for each message using ModularGaussian() in the utils hpp file. i.e. Noise standard deviation for lvl1param is 2^{-25}. i.e. if the $\triangle$ message is 536870912, add some noise and make it 536870870.

3.8 Body

The body of an encrypted message is:

$$B = \sum_{i=0}^{k-1} A_i.S_i + \triangle M + E \in R_q$$

where $\triangle$ = q/p.

3.8.1 Example

Let's continue with previous examples for N (degree) = 4, p (plain modulus) = 4, k = 2, q = 64 (modulo) and $\triangle$ = q/p = 16.

$$\begin{aligned} B &= \sum_{i=0}^{k-1} A_i.S_i + \triangle M + E \in R_q \\ &= \sum_{i=0}^{2} A_i.S_i + 16M + E \in R_q \\ &= A_0.S_0 + A_1.S_1 + 16M + E \in R_q \end{aligned}$$

When we compute R_q, we do polynomial operations modulo $x^N + 1$ and modulo q. To reduce modulo $x^N + 1$, you can observe that:

x^N = $x^N \equiv$ -1 mod x^N + 1

So

$$\begin{aligned}
A_0.S_0 &= (17 - 2x - 24x^2 + 9x^3).(x + x^2) \\
&= 17x + (17-2)x^2 + (-2-24)x^3 + (-24+9)x^4 + 9x^5 \\
&= 17x + 15x^2 - 26x^3 - 15x^4 + 9x^5 \\
&= 17x + 15x^2 - 26x^3 + (-15+9x)x^4 \\
&= 17x + 15x^2 - 26x^3 + (-15+9x)(-1) \\
&= 17x + 15x^2 - 26x^3 + 15 - 9x \\
&= 15 + 8x + 15x^2 - 26x^3 \in R_q
\end{aligned}$$

In the same way:

$$\begin{aligned}
A_1.S_1 &= -13 - 20x + 28x + 7x3 \in R_q \\
\triangle M &= -32 + 16x - 16x^3
\end{aligned}$$

Then:

$$\begin{aligned}
B &= A_0.S_0 + A_1.S_1 + \triangle M + E \in R_q \\
&= -31 + 5x - 21x^2 + 30x^3 \in R_q
\end{aligned}$$

3.9 Encryption

A GLWE ciphertext encrypting the message M under the secret key $\overrightarrow{S}$ is a tuple:

$$GLWE_{\overrightarrow{S},\sigma}(\triangle M) = (A_0, A_1, ...A_{k-1}, B) \subseteq R_q^{k+1}$$

3.9.1 Example

Let's continue with previous examples for N (degree) = 4, p (plain modulus) = 4, k = 2, q = 64 (modulo) and $\triangle$ = q/p = 16.

$$\begin{aligned}
GLWE_{\overrightarrow{S},\sigma}(\triangle M) &= (A_0, A_1, B) \subseteq R_{64}^3 \\
&= (17 - 2x - 24x^2 + 9x^3, -14 - x^2 + 21x^3, -31 + 5x - 21x^2 + 30x^3) \subseteq R_{64}^3
\end{aligned}$$

3.9.2 TFHEpp Example

In the TFHEpp library, each message is binary (i.e., 1 or 0). The following formula transforms $\triangle M$ the message in the function bootsSymEncrypt() tlwe hpp file.

$$message = message?\mu : -\mu;$$

For lvl1param, mu = 1 « 29 = 536870912 (0x2000000). So for message = 1, it is transformed to 536870912 else it is transformed to -536870912 (0xE0000000).

Here are the complete steps for encryption:

1. If the message is 1.
2. Transform the message to 536870912 (0x2000000) by shifting the bits of number 1 to the left 29 places
3. Add a small noise to the transformed message, i.e. make it make it 536870870 (0x1FFFFFF6)
4. Now add the mask to the message. message + $\sum_{i=0}^{1023} A_i.S_i$, where A_i is mask coefficients, and S_i is secret key coefficients. Let the masked message is 1128353980 (0x434150b). It will be the last element of the cypher text. The first 1024 elements of the cypher text are mask coefficients.

3.10 Decryption

We can decrypt the ciphertext using the following equation:

$$B - \sum_{i=0}^{k-1} A_i.S_i = \triangle M + E$$
$$(\triangle M + E)/\triangle = M$$

Observe that the message M is in the MSB part (thanks to the multiplication by $\triangle$) while E is in the LSB part. If $|E| < \triangle/2$ (so if every coefficient of $|e_i| < \triangle/2$), then the second step of the decryption M returns as expected.

3.10.1 Example

Let's continue with previous examples.

$$
\begin{aligned}
B &= -31 + 5x - 21x^2 + 30x^3 \\
A_0.S_0 &= 15 + 8x + 15x^2 - 26x^3 \\
A_1.S_1 &= -13 - 20x + 28x + 7x3 \\
\triangle M + E &= B - \sum_{i=0}^{1} A_i.S_i \\
\triangle M + E &= B - A_0.S_0 - A_1.S_1 \\
\triangle M + E &= -33 + 17x + 49x^2 \\
\triangle M + E &= -33 + 17x + (64 - 15)x^2 \\
\triangle M + E &= -33 + 17x - 15x^2 \\
M &= (-33 + 17x - 15x^3)/\triangle \\
M &= (-33 + 17x - 15x^3)/16 \\
&= -2 + x - x^2
\end{aligned}
$$

3.10.2 TFHEpp code

tlweSymPhase() function in tlwe.hpp removes the mask from the encrypted message. tlweSymDecrypt() checks if MSB bit is set or not. If the MSB bit is not set, it is 1, or else it is 0.

Here are the complete steps for decryption:

1. The masked message is 1128353980 (0x434150b) in our previous example
2. Now remove the mask from the masked message. message - $\sum_{i=0}^{1023} A_i.S_i$, where A_i is mask coefficients, and S_i is secret key coefficients. The message will be 536870870 (0x1FFFFFF6). tlweSymPhase()
3. Function tlweSymDecrypt() checks if the MSB bit is set. For 0x1FFFFFF6 the MSB bit is not set. From it, we deduce that the message is 1.

4 GLev

GLev is an intermediate ciphertext type between GLWE and GGSW ciphertexts, which can be very useful for better understanding GGSW ciphertexts. GLev can be seen as a generalization of the well-known Powers of two encryptions used in BGV. A GLev ciphertext contains redundancy: a list of GLWE ciphertexts encrypting the same message with different and exact scaling factors $\triangle$. Two parameters are necessary to define these special $\triangle$'s: a base β and many levels $l \in \mathbb{Z}$. β and q are the power of 2.

$$
\triangle^j = \frac{q}{\beta^j}
$$

The secret key is the same as for GLWE ciphertexts. To decrypt it is sufficient to decrypt one of the GLWE ciphertexts with the corresponding scaling factor. The set of GLev encryptions of the same message, under the secret key $\overrightarrow{S}$, with Gaussian noise with standard deviation, with base β and level l, will be noted $GLev^{\beta,l}_{S,\sigma}(M)$

$$(GLWE_{\overrightarrow{S},\sigma}(\frac{q}{\beta^1}M) \times ... \times GLWE_{\overrightarrow{S},\sigma}(\frac{q}{\beta^l}M)) = GLev^{\beta,l}_{\overrightarrow{S},\sigma}(M) \subseteq R_q^{l(k+1)}$$

4.1 Body

The body of an encrypted GLev message is:

$$B^j = \sum_{i=0}^{k-1} A_i^j . S_i + \frac{q}{\beta^{j+1}}M + E^j \in R_q$$
$$j = 0, 1, ..., l-1$$

4.2 Lev and RLev from GLev

In the same way that we saw that GLWE was a generalization for both LWE and RLWE, we can observe that GLev can be specialized into Lev and RLev by following the same rules. When we instantiate GLev with $k = n$ and $N = 1$, we get Lev. When we instantiate GLev with $k = 1$, we get RLev.

5 GGSW

Let's put it this way:

- A GLWE ciphertext is a vector of elements from a 1-dimensional matrix from R_q,
- A GLev ciphertext is a vector of GLWE ciphertexts (2-dimensional matrix from R_q)
- A GGSW ciphertext is a vector of GLev ciphertexts (3-dimensional matrix from R_q)

With GGSW ciphertexts, we again add some redundancy thanks to a 3rd dimension in the structure. In particular, in a GGSW, each GLev ciphertext encrypts the product between M and one of the polynomials of the secret key $-S_i$. The last GLev in the list just encrypts the message M:

$$(GLev^{\beta,l}_{\overrightarrow{S},\sigma}(-S_0 M) \times ... \times GLev^{\beta,l}_{\overrightarrow{S},\sigma}(-S_{k-1} M) = GGSW^{\beta,l}_{\overrightarrow{S},\sigma}(M) \subseteq R_q^{(k+1)l(k+1)}$$

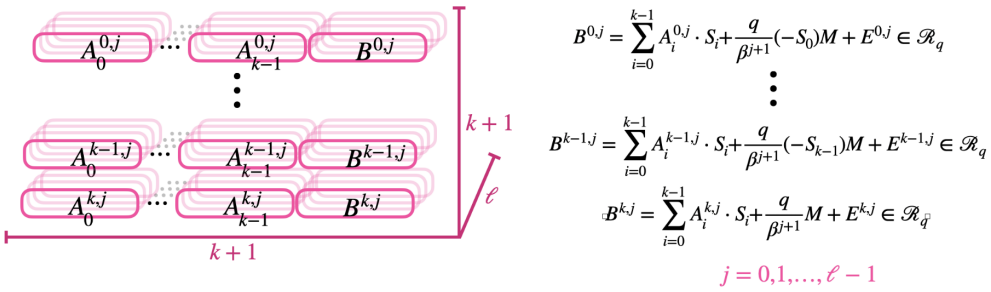

Figure 1: GGSW

5.1 GSW and RGSW from GGSW

In the same way that we saw that GLWE was a generalization for both LWE and RLWE, we can observe that GGSW can be specialized into GSW and RGSW by following the same rules. When we instantiate GGSW with $k = n$ and $N = 1$, we get GSW. When we instantiate GGSW with $k = 1$, we get RGSW.

5.2 GLWE homomorphic addition

A GLWE ciphertext encrypting the message M under the secret key $\overrightarrow{S}$ is a tuple:

$$C = GLWE_{\overrightarrow{S},\sigma}(\triangle M) = (A_0, A_1, \ldots A_{k-1}, B) \subseteq R_q^{k+1}$$

Now, let's consider another GLWE ciphertext encrypting the message M^i under the same secret key

$$C^i = GLWE_{\overrightarrow{S},\sigma}(\triangle M^i) = (A_0^i, A_1^i, \ldots A_{k-1}^i, B^i) \subseteq R_q^{k+1}$$

Then, we can add every component of the two ciphertexts, and the result will be a new GLWE ciphertext encrypting the sum $M + M^i$ under the same secret key with noise that grew slightly.

$$C^+ = C + C^i = GLWE_{\overrightarrow{S},\sigma}(\triangle M + M^i) = (A_0 + A_0^i, A_1 + A_1^i, \ldots A_{k+1} + A_{k-1}^i, B + B^i) \subseteq R_q^{k+1}$$

5.3 GLWE homomorphic multiplication by a constant

Let's now consider a small constant polynomial:

$$\Lambda = \sum_{i=0}^{N-1} \Lambda_i X^i \in R$$

Then, we can multiply the polynomial Λ to every component of the ciphertext and the result will be a new GLWE ciphertext encrypting the product Λ. M under the same secret key with noise that grew a little bit

$$C^{(.)} = C.\Lambda = GLWE_{\overrightarrow{S},\sigma}(\triangle(\Lambda.M)) = (\Lambda.A_0, \Lambda.A_1, ...\Lambda.A_{k+1}, \Lambda.B) \subseteq R_q^{k+1}$$

5.4 Homomorphic multiplication by a large constant

In GLWE homomorphic multiplication by a small constant polynomial, the noise grew proportionally concerning the size of the polynomial coefficients. If we multiply every ciphertext component by a large constant β, the noise grows proportionally concerning its size and will compromise the result.

5.5 Decomposition and Inner products

To solve the noise problem, we do decomposition and inner products. The idea involves decomposing the large constant into a small base β.

$$\gamma = \gamma_1 \frac{q}{\beta^1} + \gamma_2 \frac{q}{\beta^2} .... \gamma_l \frac{q}{\beta^l} Decomp^{\beta l}$$

where where the decomposed elements $(\gamma_1, \gamma_2 ... \gamma_l)$ are in $\mathbb{Z}_\beta$ and are small and both q and β are power of 2.

$$Decomp^{\beta l} = (\gamma_1, \gamma_2 ... \gamma_l)$$

As the elements of the decomposition are small, we should now be able to perform multiplication with the ciphertext and have a negligible impact on the noise. But, to obtain as a result the product of γ and the message M , we need to be able to invert the decomposition, and so to recompose γ . To do that, instead of multiplying the decomposed elements by a GLWE encryption of M, we multiply them by the GLev encryption of M.

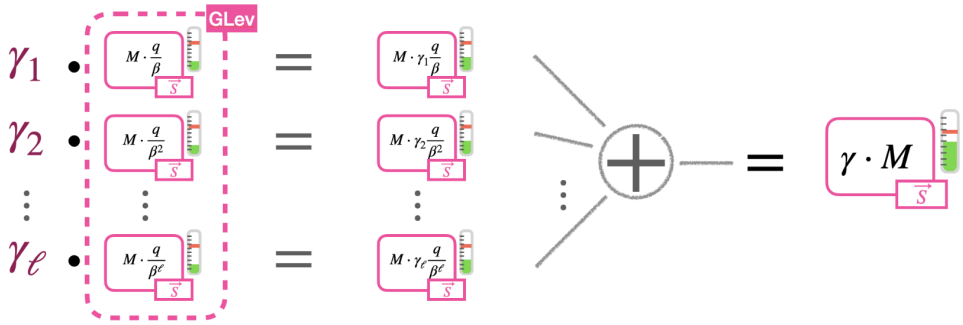

Figure 2: recompose using Glev

In this case, the noise grows very slowly since the multiplication times the small constants have a minor impact on the noise and the following addition.

5.6 Homomorphic multiplication by a large polynomial

Decompose the polynomial into small polynomials and then perform a polynomial inner product with the GLev

5.7 Approximate decomposition

We could do an approximate decomposition and decompose to a fixed precision (using $\beta^l < q$). In practice, this means that we will do a rounding in the LSB part before decomposing: if the decomposition parameters are appropriately chosen, this will not affect the correctness of the computations because, in the LSB part, there is always noise and the information we are interested into keeping – the message – is in the MSB part.

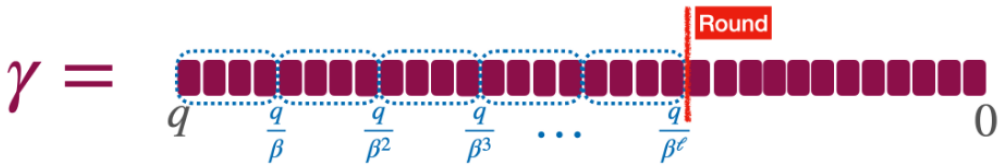

Figure 3: Approximate decomposition

5.7.1 Approximate decomposition example

Let's continue with previous examples for N (degree) = 4, p (plain modulus) = 4, k = 2, q = 64 (modulo) and Δ = q/p = 16.

Now let's choose a random large polynomial in R_q, so a polynomial of degree smaller than N = 4 and with coefficients in -31, -30, ...-1, 0, 1, .. 32:

$$\begin{aligned}\Lambda &= \Lambda_0 + \Lambda_1 X^2 + \Lambda_2 X^3 \\ &= 28 - 5X - 30X^2 + 17X^3\end{aligned}$$

Note that each coefficient consists of six bits. -5 in signed decimal is 59 (64-5) unsigned decimal and (1,1,1,0,1,1) in binary. Similarly, -30 in signed decimal is 34 in unsigned decimal is $2^5 + 2^1$ (1,0,0,0,1,0) in binary.

Let's choose a base for the decomposition $\beta = 4$ and l = 2, so $\beta^l = 16$. Decomposed coefficients will be only 2 bits. We will decompose each coefficient's 4 MSB and round the last 2 LSB.

We need to round all the coefficients. First, we write them in their binary decomposition (MSB on the left and LSB on the right) and then round the LSB.

- $\Lambda_0 = 28 \longmapsto (0,1,1,1|0,0)$ which after rounding becomes $\Lambda_0' \longmapsto (0,1,1,1)$;
- $\Lambda_1 = -5 \longmapsto (1,1,1,0|1,1)$ which after rounding becomes $\Lambda_1' \longmapsto (1,1,1,1)$;
- $\Lambda_2 = -30 \longmapsto (1,0,0,0|1,0)$ which after rounding becomes $\Lambda_2' \longmapsto (1,0,0,1)$;
- $\Lambda_3 = 17 \longmapsto (0,1,0,0|0,1)$ which after rounding becomes $\Lambda_3' \longmapsto (0,1,0,0)$.

Figure 4: Rounding

The next step is the decomposition. We start from the LSB and, since the base $\beta = 4$ We need to extract 2 bits for every round. We want the decomposition to be signed, so we want coefficients in -2,-1,0,1. So, in the binary decomposition,

- (0,0) corresponding to 0
- (0,1) corresponding to 1
- (1,0) corresponds to -2 (i.e. 2 - 4), and we add 4 to the next block in the decomposition, like a carry.
- (1,1) corresponds to -1 (i.e. 3 - 4), and we add 4 to the next block in the decomposition, like a carry.

Every carry that goes beyond the MSB is thrown away.

In $\Lambda_0' \longmapsto (0,1,1,1)$:

- The two LSB are $(1,1)$, corresponding to the value 3. We subtract 4 and obtain $3-4=-1$ as the first element of the decomposition.

- The next block is $(0,1)$, but since we subtracted 4 before, we need to add it back now, which corresponds to add 1 to $(0,1)$, that becomes $(1,0)$, corresponding to the value 2. As before we subtract 4, obtaining $2-4=-2$ as the second element of the decomposition. Since we reached the MSB, the $+4$ that should have been performed in the next block is simply thrown away.

In $\Lambda_1' \longmapsto (1,1,1,1)$:

- The two LSB are $(1,1)$, corresponding to the value 3. We subtract 4 and obtain $3-4=-1$ as the first element of the decomposition.
- The next block is $(1,1)$, but since we subtracted 4 before, we need to add it back now, which corresponds to add 1 to $(1,1)$, that becomes $(0,0)$, corresponding to the value 0, which will be the second element of the decomposition.

In $\Lambda_2' \longmapsto (1,0,0,1)$:

- The two LSB are $(0,1)$, corresponding to the value 1, which will be the first element of the decomposition.
- The next block is $(1,0)$, corresponding to the value 2. As before we subtract 4, obtaining $2-4=-2$ as the second element of the decomposition.

In $\Lambda_3' \longmapsto (0,1,0,0)$:

- The two LSB are $(0,0)$, corresponding to the value 0, which will be the first element of the decomposition.
- The next block is $(0,1)$, corresponding to the value 1, which will be the second element of the decomposition.

Figure 5: decomposition of the coefficients

Now that the decomposition of the coefficients is done, we can write explicitly the decomposed polynomials as:

$$\begin{cases} \Lambda^{(1)} = -2 - 2X^2 + X^3 \\ \Lambda^{(2)} = -1 - X + X^2 \end{cases}$$

Figure 6: Decomposed polynomial

The coefficients of $\Lambda^{(2)}$ are the first elements of the decomposition, while the coefficients of $\Lambda^{(1)}$ are the second elements of the decomposition. To verify that the decomposition is correct, we can invert it by computing:

$$\begin{aligned}\Lambda^{(1)}.\frac{q}{\beta^1}+\Lambda^{(2)}.\frac{q}{\beta^2} &= (-2-2X^2+X^3).\frac{64}{4^1}+(-1-X+X^2).\frac{64}{4^2}\\ &= (-2-2X^2+X^3)16+(-1-X+X^2)4\\ &= 28-5X-28X^2+16X^3 \in R_{64}\end{aligned}$$

6 Key switching

This homomorphic operation is primarily used in all the (Ring)LWE-based schemes. As the name suggests, it switches the secret key to a new one. To switch the key, we cancel the current secret key and homomorphically re-encrypt it under a new secret key.

$$KSK_i = (GLWE_{\overrightarrow{S'},\sigma_{KSK}}(\frac{q}{\beta^1}S_i)\times\ldots\times GLWE_{\overrightarrow{S'},\sigma_{KSK}}(\frac{q}{\beta^l}S_i)) = GLev^{\beta,l}_{\overrightarrow{S'},\sigma_{KSK}}(S_i)\subseteq R_q^{l(k+1)}$$

The key switching is performed as follows:

$$C' = \underbrace{\overbrace{(0,\ldots,0,B)}^{\text{Trivial GLWE of } B} - \sum_{i=0}^{k-1}\overbrace{\langle \mathsf{Decomp}^{\beta,\ell}(A_i),\mathsf{KSK}_i\rangle}^{\text{GLWE encryption of } A_iS_i}}_{\text{GLWE encryption of } B-\sum_{i=0}^{k-1}A_iS_i=\Delta M+E} \in GLWE^{\vec{}}_{S',\sigma'}(\Delta M)\subseteq \mathcal{R}_q^{k+1}.$$

Figure 7: Key switching

The secret key has switched from $\overrightarrow{S}$ to $\overrightarrow{S'}$, but the message is the same.

7 External Product

The external product is to homomorphically multiply two ciphertexts such that the result is an encryption of the product of messages.

We will use a similar approach as the one used for key switching, so we will take one of the two ciphertexts as a GLWE (the ciphertext we will decompose), and the other ciphertext will be a list of GLev ciphertexts. The difference, this time, is that with key-switching, we wanted only the mask of the first ciphertext to be multiplied by the GLev's encrypting the secret key, while this time, we want both mask and body. A ciphertext composed by GLev ciphertexts is GGSW. So, to summarize, the external product we will build is an operation that allows multiplying two ciphertexts – a GLWE and a GGSW –, and that returns in output a new ciphertext – a GLWE one – encrypting the product of the two messages encrypted in the inputs. The two inputs are:

- a GLWE ciphertext encrypting a message $M_1 \in \mathcal{R}_p$ under the secret key $\vec{S} = (S_0, \ldots, S_{k-1}) \in \mathcal{R}^k$:

$$C = (A_0, \ldots, A_{k-1}, B) \in GLWE^{\vec{}}_{S,\sigma}(\Delta M_1) \subseteq \mathcal{R}_q^{k+1}$$

  where the elements A_i for $i \in [0..k-1]$ are sampled uniformly random from $\mathcal{R}_q$, and $B = \sum_{i=0}^{k-1} A_i \cdot S_i + \Delta M + E \in \mathcal{R}_q$, and $E \in \mathcal{R}_q$ has coefficients sampled from a Gaussian distribution χ_σ, as we have already seen before.

- a GGSW ciphertext encrypting a message $M_2 \in \mathcal{R}_p$ under the same secret key $\vec{S} = (S_0, \ldots, S_{k-1}) \in \mathcal{R}^k$:

$$\overline{\overline{C}} = (\overline{C}_0, \ldots, \overline{C}_{k-1}, \overline{C}_k) \in GGSW^{\beta,\ell}_{\vec{S},\sigma}(M_2) \subseteq \mathcal{R}_q^{(k+1)\times\ell(k+1)}$$

  where $\overline{C}_i \in GLev^{\beta,\ell}_{\vec{S},\sigma}(-S_i M_2)$ for $i \in [0..k-1]$ and $\overline{C}_k \in GLev^{\beta,\ell}_{\vec{S},\sigma}(M_2)$.

Then the external product is noted with the symbol $\boxdot$ and is computed as:

$$\begin{aligned} C' = \overline{\overline{C}} \boxdot C &= \langle \mathsf{Decomp}^{\beta,\ell}(C), \overline{\overline{C}} \rangle \\ &= \underbrace{\overbrace{\langle \mathsf{Decomp}^{\beta,\ell}(B), \overline{C}_k \rangle}^{\text{GLWE encrypt. of } BM_2} + \sum_{i=0}^{k-1} \overbrace{\langle \mathsf{Decomp}^{\beta,\ell}(A_i), \overline{C}_i \rangle}^{\text{GLWE encrypt. of } -A_i S_i M_2}}_{\text{GLWE encrypt. of } BM_2 - \sum_{i=0}^{k-1} A_i S_i M_2 \approx \Delta M_1 M_2} \in GLWE^{\vec{}}_{S,\sigma''}(\Delta M_1 M_2) \subseteq \mathcal{R}_q^{k+1} \end{aligned}$$

Figure 8: External Product Equation

Figure 9: External Product

- The external product is like a key switching with an additional element to the key switching key
- The external product is like a key switching where we do not switch the key. In fact, in the GGSW ciphertext, the secret key used for encryption and the one used inside the GLev ciphertexts are the same.

- An external product that takes as input a GGSW ciphertext, that uses a different secret key for encrypting and uses the same secret key as the GLWE ciphertext inside the encryption, is called functional key switching. In fact it applies a function (multiplication by an encrypted constant) and switches the key at the same time.

8 Internal Product

A GGSW ciphertext is a list of Glev ciphertexts, and each GLev is a list of GLWE ciphertexts. So a GGSW is a list of GLWE ciphertexts. Since the external product is a product between GLWE and GGSW ciphertexts, we can define the internal product as a list of independent external products between one of the GGSW ciphertexts in input and all the GLWE ciphertexts composing the second GGSW input. The result of all these external products will be the GLWE ciphertexts that will write the GGSW output.

Figure 10: Internal Product

9 CMux

The CMux operation is the homomorphic version of a Mux gate, also known as a multiplexer gate. In practice, a Mux gate is an if condition: it takes three inputs, a selector (a bit b) and two options (d_0 and d_1), and, depending on the value of the selector, it chooses between the two options.

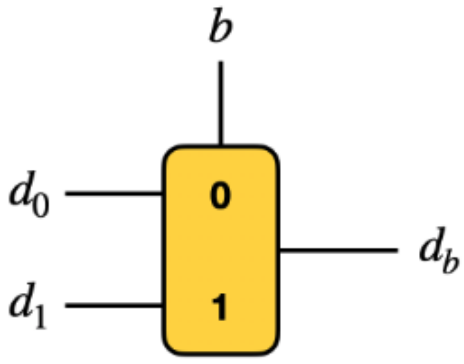

Figure 11: CMux

It is evaluated by computing:

$$b.(d_1 - d_0) + d_0 = d_b$$

To evaluate it homomorphically, we encrypt b as a GGSW ciphertext, d_0 and d_0 and d_1 as GLWE ciphertexts. Then the multiplication in the cleartext formula is evaluated as an external product, while the other operations (addition and subtraction) are assessed as homomorphic additions and subtractions. The result is that d_b is encrypted as a GLWE ciphertext.

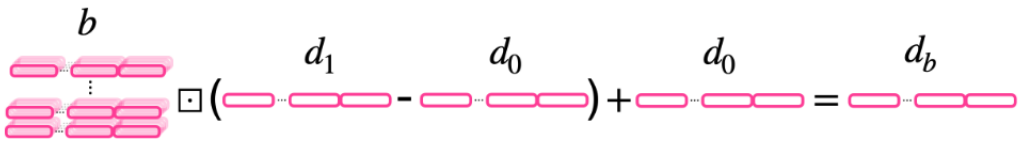

Figure 12: CMux homomorphically

10 Modulus Switching

It is switching the ciphertext modulus to a new one. Then, let's consider another positive integer ω such that p < ω < q and ω, p and q are all equal to a power of 2: the modulus switching from q to ω is the operation that takes all the components of the LWE ciphertext and switches their modulus from q to ω as follows:

$$\tilde{a_i} = \frac{\omega.a_i}{q} \in \mathbb{Z}_\omega$$

This operation keeps the MSB of the original ciphertext as the message is encoded in it. However, the noise grows considerably, so it should be used carefully if we do not want to compromise the message.

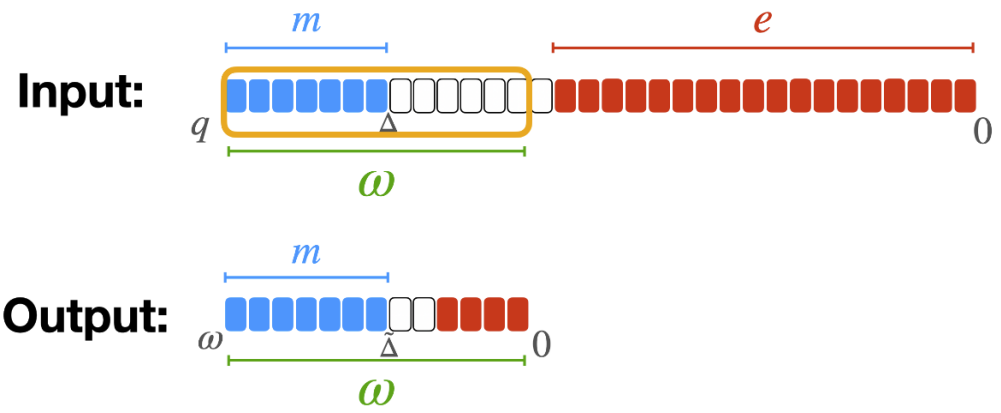

Figure 13: Modulus Switching

10.1 Modulus Switching Example

Let's choose n = 4, q = 64, p = 4 and $\triangle$ = q/p = 16. The coefficient values of the mask are stored in 6 bits (q=64 = 2^6). A signed integer's possible mask coefficient value is -31, -30, ...-1, 0, 1, .. 32. In this example, let the message be 1, the mask coefficient be (-25, 12, -3, 7), the secret key be (0.1.1.0), and the error is 1.

$$\begin{aligned} b &= a_0 s_0 + a_1 s_1 + a_2 s_2 + a_3 s_3 + \triangle m + e \\ &= 12 - 3 + 16 + 1 \\ &= 26 \end{aligned}$$

So the encryption is:

$$\begin{aligned} c &= (a_0, a_1, a_2, a_3, b) \in \mathbb{Z}_q^{n+1} \\ &= (-25, 12, -3, 7, 26) \in \mathbb{Z}_{64}^5 \end{aligned}$$

Let's perform the modulus switching by choosing $\omega = 32$.

$$\tilde{a_i} = \frac{\omega . a_i}{q} = \frac{32 . a_i}{64} = \frac{a_i}{2}$$

The new encryption is:

$$\begin{aligned} \tilde{c} &= (\tilde{a_0}, \tilde{a_1}, \tilde{a_2}, \tilde{a_3}, \tilde{b}) \in \mathbb{Z}_\omega^{n+1} \\ &= (-13, 6, -2, 4, 13) \in \mathbb{Z}_{32}^5 \end{aligned}$$

The new scaling factor is $\tilde{\triangle} = \frac{\omega}{p} = \frac{32}{4} = 8$. So, let's try to decrypt this new ciphertext.

$$\begin{aligned} \tilde{\triangle} m &= \tilde{b} - \sum_{i=0}^{n-1} a_i . s_i \\ 8m &= 13 - (6 - 2) \\ m &= \frac{9}{8} \\ m &= 1 \end{aligned}$$

This is the message we originally encrypted. The new error equals 1 ($Tildebigtriangleup$m + e = 9). It is not bigger than the original error in absolute value, but it has increased because the modulus switching has reduced its distance to the message (the scaling factor has changed).

11 Sample Extraction

A sample extraction operation takes a GLWE ciphertext as input and extracts the encryption of one of the message's coefficients as an LWE ciphertext.

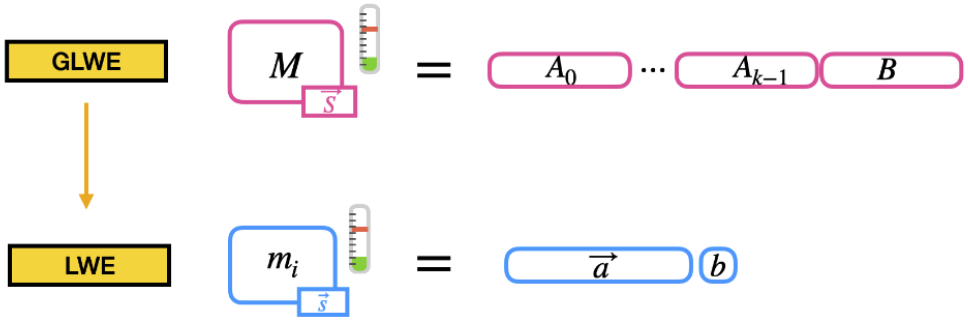

Figure 14: Sample Extraction

The operation does not increase the noise and consists of "copy-pasting" some of the coefficients of the GLWE ciphertext into the output LWE ciphertext. Imagine that we want to extract the h-th coefficient of the message M, with $0 \leq h < N$. The result LWE ciphertext will have n=kN (k=1 for RLWE) and be encrypted under the extracted secret key. We build the result LWE ciphertext c = $(a_0, a_1, ...a_{n-1}, b) \in \mathbb{Z}_q^{n+1}$ by simply copying the coefficients of the GLWE as follows:

$$\begin{cases} a_{N \cdot i+j} \leftarrow a_{i,h-j} & \text{for } 0 \leq i < k, 0 \leq j \leq h \\ a_{N \cdot i+j} \leftarrow -a_{i,h-j+N} & \text{for } 0 \leq i < k, h+1 \leq j < N \\ b \leftarrow b_h \end{cases}$$

Figure 15: Sample Extraction equation

11.1 Sample Extraction Example

Let's use a generic RLWE ciphertext (so k = 1) and N = 4.

$$\begin{cases} (a_0, -a_3, -a_2, -a_1, b_0) & \in LWE^{\vec{}}_{s,\sigma}(\Delta m_0) \subseteq \mathbb{Z}_q^{n+1} \\ (a_1, a_0, -a_3, -a_2, b_1) & \in LWE^{\vec{}}_{s,\sigma}(\Delta m_1) \subseteq \mathbb{Z}_q^{n+1} \\ (a_2, a_1, a_0, -a_3, b_2) & \in LWE^{\vec{}}_{s,\sigma}(\Delta m_2) \subseteq \mathbb{Z}_q^{n+1} \\ (a_3, a_2, a_1, a_0, b_3) & \in LWE^{\vec{}}_{s,\sigma}(\Delta m_3) \subseteq \mathbb{Z}_q^{n+1} \end{cases}$$

Figure 16: Sample Extraction example equation

12 Blind Rotation

It is called blind rotation because it realizes a rotation of the coefficients of a polynomial in a blind way, meaning that we will rotate the coefficients. Still, the number of rotated positions is encrypted.

12.1 Rotation of coefficients of a polynomial

It simply means that we want to shift the coefficients towards the left (or the right) and to do so, we multiply the polynomial by a monomial $X^{-\pi}$ (or X^{π}) where $0 \leq \pi < N$ is the number of positions we want to rotate. In practice, the rotation brought the coefficient initially in position π down to the constant coefficient position. Of course, since the rotation is computed modulo $X^N + 1$, the coefficients from m_0 to $m_{\pi-1}$ passed "on the other side" with a minus sign.

$$\cdot X^{-\pi} \begin{cases} M = m_0 + m_1 X + \ldots + m_\pi X^\pi + \ldots + m_{N-1} X^{N-1} \\ M \cdot X^{-\pi} = m_\pi + m_{\pi+1} X + \ldots + m_{N-1} X^{N-\pi-1} - m_0 X^{N-\pi} - \ldots - m_{\pi-1} X^{N-1} \end{cases} \quad \text{mod } X^N + 1$$

Figure 17: Rotation of coefficients of a polynomial

12.2 Rotation of coefficients of an encrypted polynomial

We can encrypt polynomials with GLWE ciphertexts: the rotation is performed by simply rotating all the components of the GLWE by multiplying them times $X^{-\pi}$. The result is then a GLWE ciphertext encrypting the rotated message polynomial. This operation does not increase the noise.

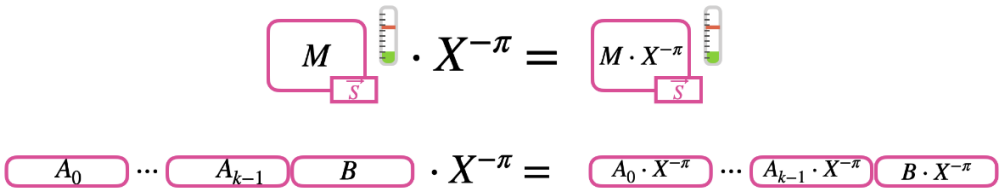

Figure 18: Rotation of coefficients of an encrypted polynomial

12.3 Blind rotation of coefficients of an encrypted polynomial

The encryption of the value π finally makes a blind rotation of coefficients of an encrypted polynomial. π is an integer value, so we need to express it in its binary decomposition:

$$\pi = \pi_0 + \pi_1.2 + \pi_2.2^2 + ... + \pi_\delta.2^\delta$$

where:

$$\delta = log_2(N)$$

What we want to compute is:

$$M.X^{-\pi} = M.X^{-(\pi_0+\pi_1.2+\pi_2.2^2+.....+\pi_\delta.2^\delta)}$$
$$M.X^{-\pi} = M.X^{-\pi_0}.X^{-\pi_1.2}..X^{-\pi_2.2^2}....X^{-\pi_\delta.2^\delta}$$

Since π_j is binary, two options are possible when we compute $M.X{-}^{\pi_j.2^j}$. This is exactly an if condition, and we have seen that we can evaluate it as a CMux that takes as inputs:

$$M \cdot X^{-\pi_j \cdot 2^j} = \begin{cases} M & \text{if } \pi_j = 0 \\ M \cdot X^{-2^j} & \text{if } \pi_j = 1 \end{cases}$$

Figure 19: CMux π_j as bit selector

The result of this operation is a GLWE encryption of $M.X^{-\pi_j.2^j}$ as we wanted.

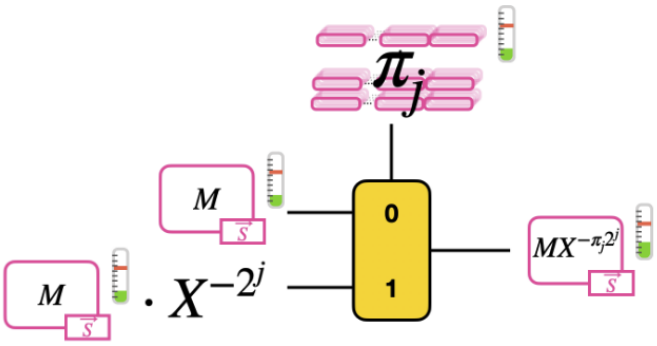

Figure 20: CMux π_j as bit selector

The blind rotation computes recursively the product of the GLWE encryption of M by each of these monomials, so it performs a sequence of δ CMuxes. The result of a CMux becomes the new message option of the following CMux.

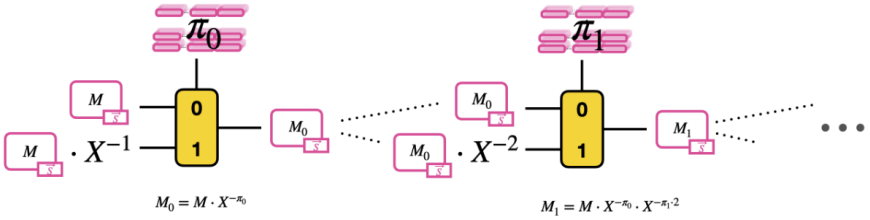

Figure 21: Recursively CMux π_j as bit selector