Bootstrapping Standard-LWE Homomorphic Encryption

Gyana Ranjan Sahu, *Member, IEEE,* and Jane Doe, *Life Fellow, IEEE*

Abstract—

Since the discovery of Gentry's first fully homomorphic scheme it has been a much sought topic in the field of Cyrptography. In brief, FHE is a scheme that allows one to evaluate arbitrary circuits over encrypted data without being able to decrypt. Gentry constructed the first FHE scheme by constructing a somewhat homomorphic encryption scheme in which the resulting ciphertext eventually gets noisy after the evaluation of the circuit. At this stage the ciphertext needs to be refreshed by the bootstrappping algorithm. Unfortunately, Gentry's bootstrapping technique is computationally very expensive, yielding an impractical run-time. The main bottleneck in bootstrapping arises from the need to evaluate homomorphically the decryption circuit. Over the past few years a number of bootstapping algorithms following Gentry's blueprint have been proposed and considerable work has been done to make them practical.

One of the implementations reported in literature is that of Halevi-Shoup (Crypto 2014) in the open source library of HElib. Their work uses a packed ciphertext over field extentions with plaintext modulus p $>$ 2. To give an idea of the runtime they show that a packed vector of 1024 elements can be recrypted in under 5.5 minutes. In another remarkable work, Ducas-Micciancio presented their work on bootstrapping which computes simple bit operations. In their work they make use of a relatively small group element over cyclotomic rings and report a bootstrapping wall clock time of under a second. However, the reported scheme works over a bits with plaintext modulus of 2 and 4 and needs to be bootstrapped after every NAND gate evaluation.

We propose an algorithm to bootstrap the standard LWE ciphertext which remedies some of the drawbacks of bootstrapping scheme proposed by Ducas and Micciancio. Standard LWE homomorphic scheme is based on LWE assumption which enjoys the worst-case hardness of "short vector problems" on arbitrary lattices. The salient feature of ciphertext is that they are defined for an arbitrary plaintext modulus p $>$ 2 and bootstrapping is only needed when the ladded of moduli is exhausted during circuit evaluation.

Index Terms—LWE, Bootstrapping.

1 Introduction

In recent years there has been a growing interest in the area of homomorphic encryption. In a nutshell, a fully homomorphic encryption scheme is an encryption scheme that allows evaluation of arbitrary depth function or circuit on encrypted data. The problem was first suggested by Rivest, Adleman and Dertouzos back in 1978. A number of encryption schemes were proposed which were partially homomorphic such as the encryption systems of Goldwasser and Micali [cite], El-Gamal [cite], Cohen and Fischer [cite], and Paillier [cite]. These schemes supported either adding or multiplying encrypted ciphertexts, but not both operations at the same time. The first scheme that supported both addition and multiplication of ciphertexts was that of Boneh-Goh-Nissim[cite]. However, the scheme could handle only one multiplication and multiple additions and hence not fully homomorphic. This problem remained an open problem until 2009 when Gentry[cite] in a breakthrough work showed the first construction of an encryption scheme capable of performing arbitrary number of additions and multiplications on encrypted data.

Fully homomorphic encryption goes a long way in solving some of the key problems related to data privacy. In businesses and applications where data privacy is a major concern, the consumers or users may not be open to share their data. This is quite contrary to the current day scenario where businesses expect their users to upload their data onto cloud storage systems which may even be controlled by third party vendors. FHE schemes come a long way in remedying such situations by allowing the users to store all data in the cloud encrypted, and perform computations on these encrypted data. Some of the concrete example of such businesses and applications can be found in medical and health care, military and financial sectors, to name a few. Lauter-Naehrig-Vaikuntanathan [cite] have described implementations of simple statistical operations such as mean, standard deviation and logistical regression. Specifically, they make use of Brakerski-Vaikuntanathan [cite] homomorphic encryption scheme in their applications . In another work by Wu-Haven [cite] demonstrate the viability of using leveled homomorphic encryption for large scale statistical analysis on encrypted data. Specifically, they compute the mean and variance of univariate and multivariate data as well as perform linear regression on a multidimensional, encrypted datasets. While both these works show the practical applications and viability of homomorphic encryption, they also avoid the use of bootstrapping procedure in their computation. In other words they compute on a circuit, the depth of which is fixed by the desired application. Moreover, this is primarily done so as to avoid the computationally expensive operation of bootstrapping.

- *M. Shell was with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332. E-mail: see http://www.michaelshell.org/contact.html*
- *J. Doe and J. Doe are with Anonymous University.*

Manuscript received April 19, 2005; revised August 26, 2015.

1.1 Prior Works

It is worth while to mention the pioneering work of Gentry as it paved the foundations of fully homomorphic encryption. Gentry's work starts off with a construction of a new public key encryption scheme based on ideal lattices which provide additive and multiplicative homomorphisms. However, the scheme is homomorphic only for shallow circuits because the "error" grows with addition and (especially) multiplication operations; eventually, it becomes so long that it causes a decryption error. The depth of the scheme is logarithmic in the lattice dimension. Formally, the public key encryption scheme ε is defined by four algorithms $KeyGen_\varepsilon$, $Encrypt_\varepsilon$, $Decrypt_\varepsilon$, and an additional algorithm $Evaluate_\varepsilon$. The $Evaluate_\varepsilon$ algorithm takes in the public key (generated by $KeyGen_\varepsilon$) and a circuit C from the family of circuits C_ε, and a tuple of ciphertexts $\Psi = \langle \psi_1, \psi_2, \cdots, \psi_t \rangle$; it outputs a ciphertext ψ.

Correctness of the public key encryption scheme follows if for a set of plaintexts given by $\langle \pi_1, \pi_2, \cdots \pi_t \rangle$ and corresponding ciphertexts given by $\langle \psi_1, \psi_2, \cdots \psi_t \rangle$ then $|Pr\left(Decrypt_\varepsilon\left(\psi_i\right) \neq \pi_i\right)| \approx negl$ for $i \in [1, 2, \cdots, t]$. Moreover, encryption scheme ε is correct for the class of circuits in C_ε if, for any key-pair (sk, pk) output by $KeyGen_\varepsilon\left(1^\lambda\right)$, any circuit $C \in C_\varepsilon$, any plaintexts $\pi_1, \cdots, \pi_t$, and any ciphertexts $\Psi = \langle \psi_1, \cdots, \psi_t \rangle$ with $\psi_i \leftarrow Encrypt_\varepsilon\left(pk, \pi_i\right)$, then the following holds true :

$$\psi \leftarrow Evaluate_\varepsilon\left(pk, C, \Psi\right)$$

$$\Rightarrow C\left(\pi_1, \cdots, \pi_t\right) = Decrypt_\varepsilon\left(sk, \psi\right)$$

Security of the Somewhat Homomorphic scheme (SwHE) depends upon the Ideal Coset Problem (ICP). In brief, ICP asks to distinguish between an uniformly generated ideal and the one chosen by sampling from a specific distribution. The ideal coset problem can be reduced to decision version of closest vector problem (CVP) which are conjectured to be intractable and can be solved in $2^{\mathcal{O}^{(n)}}$. It is also worth while to note here that for weak choice of parameters, algorithm like LLL lattice reduction techniques make it feasible to break CVP and hence ICP.

Bootstrapping the initial construction of SwHE is achieved by evaluating the decryption circuit homomorphically. This is still the key idea behind most of the recent homomorphic schemes to achieve fully homomorphic encryption scheme. To achieve bootstrapping or refreshing the ciphertext one needs to follow these steps :

- Encrypt the current secret key under a new public key. If sk_1 is the current secret key then we can generate the encryption of bits of sk_1 under pk_2.

$$\overline{\psi_{1j}} = Encrypt_\varepsilon\left(pk_2, sk_{1j}\right)$$

- Use the $Evaluate_\varepsilon$ algortihm on the current ciphertext ψ_1, new public key pk_2 and encryptions of bits of old secret key sk_{1j} to evaluate the decryption circuit. This produces a new ciphertext ψ_2 which can be decrypted under the new secret key sk_2.

$$\psi_2 = Evaluate_\varepsilon\left(pk_2, D_\varepsilon, \overline{\psi_{1j}}, \psi_1\right)$$

$$\pi = Decrypt_\varepsilon\left(sk_1, \psi_1\right) = Decrypt_\varepsilon\left(sk_2, \psi_2\right)$$

This refreshing procedure is considered successful if the noise in the final ciphertext is lower than that in the initial ciphertext. In Gentry's bootstrapping method the original decryption is transformed to re-express decryption as a sparse subset vector sum rather than a full matrix-vector product by taking into account an additional hardness assumption. Doing so reduces the depth of homomorphic computation and in turn reduces final noise level in ciphertext. Specifically, the secret key is split into a set of vectors with a hidden sparse subset whose sum is known from a set that is uniform.

Following Gentry's bootstrapping theorem, Brakerski-Vaikuntanathan [cite] introduced a bootstrapping algorithm based on Learning With Errors with polynomial approximation factors. Bootstrapping theorem states that a scheme that can homomorphically evaluate its family of augmented decryption circuits can be transformed into a leveled fully homomorphic encryption scheme with the same decryption circuit, ciphertext space and public key. Their method uses the Gentry-Sahai-Waters (GSW) cryptosystem (cite) in conjunction with Barringtons "circuit sequentialization" theorem. In particular they convert the decryption algorithm of GSW scheme into an equivalent circuit, the depth of which is found to be of $\mathcal{O}\left(\log N\right)$ where $N = (n+1)\lceil \log q \rceil$, q being the modulus and n the lattice dimension. For converting the decryption algorithm into a circuit they make use of Barrington's theorem which states that every Boolean NAND circuit Ψ that acts on ℓ inputs and has depth d can be computed by a width-5 permutation branching program Π of length 4^d. Given the description of the circuit Ψ, the description of the branching program Π can be computed in $poly(\ell, 4^d)$ time.

It can be easily seen that refreshing a ciphertext by this method can lead to large growth of error due to the homomorphic evaluation of the circuit which has large depth. This is partially remedied by using the GSW scheme which has asymmetric growth of noise term. Specifically, when multiplying two ciphertexts with noise levels e_1 and e_2, the noise in the output turns out to be $e_1 + poly(n).e_2$. This is in contrast to other RLWE based homomorphic schemes where the noise grows as $poly(n)^{depth}$ and a multiplication tree is used to keep the noise to a minimum. To further optimize their scheme a variant of dimension-modulus reduction technique is used. This shrinks the higher noise ciphertext into ones with small noise and optimal lattice parameters. This approach, however, results in very large polynomial runtimes and no known implementations are reported in literature.

In another work reported on bootstrapping, Gentry-Halevi-Smart [cite] construct a recryption technique that improved upon the homomorphic modular-reduction by working with a modulus of the form $2^r + 1$. They also show how to combine their new technique with the SIMD homomorphic computation techniques of Smart-Vercauteren and Gentry-Halevi-Smart, to get a bootstrapping method that works in time quasilinear in the security parameter. Applying their methods to BGV cryptosystem they achieve a scheme capable of evaluating its own decryption circuit, making it a FHE scheme.

BGV cryptosystem is defined over cyclotomic rings of the form $R = \mathbb{Z}_q\left[X\right]/F\left(X\right)$ for a monic irreducible poly-

nomial $F(X)$. To control the noise the scheme maintains a chain of moduli, exhausting one at each level of evaluation. The scheme is parameterized by the number of levels it can compute, denoted by L ($depth = L$) and a set of decreasing odd moduli $q_0 \geq q_1 \geq \cdots \geq q_L$, one for each level. The BGV ciphertext is a 2 vector Ring element represented as $\mathbf{c} = (c_0, c_1)$. In general BGV scheme has a plaintext space of $\mathbb{Z}_p$ and a message $m \in \mathbb{Z}_p$ can be encrypted as a ring element R_p. However, in this work the authors consider a binary plaintext space of R_2. By applying SIMD techniques instead of a single message one encode multiple messages in the plaintext space R_2. Secret key $s \in R_q$ are ring polynomials which are drawn from a narrow discrete gaussian distributions χ_B and have low bounded norms. Decryption proceeds by computing the noise term $[\langle \mathbf{c}, \mathbf{s} \rangle]_q = [c_0 - c_1.s]_q$ and then taking modulo-2 of the resultant. To sum up message is recovered as, $m = \Big[[\langle \mathbf{c}, \mathbf{s} \rangle]_q\Big]_2$. Message is recovered correctly if the noise term doen't wrap around the modulus q.

One of the key observation made in their approach is that if the ciphertext modulus q is of the form $q = 2^r + 1$ then the decryption can be much simplified. To decrypt the ciphertext in this approach one first computes the noise term coefficients, say $Z = \langle \mathbf{c}, \mathbf{s} \rangle \; mod \; F(X) \; mod \; q$. It is assumed that the coefficients are smaller than q^2 in magnitude. If z is one of the coefficients of the polynomial Z then $\Big[[z]_q\Big]_2$ can be computed as $z\langle r \rangle \oplus z\langle 0 \rangle$, where $z\langle i \rangle$ is the i^{th} bit of z.

Assuming circular security, individual coefficients of secret key of the last level (w.r.t q_L) is kept in encrypted form (w.r.t q_0 $(largest \; modulus)$). To encrypt the secret key the plaintext space is temporarily lifted to work in modulo 2^{r+1} ring polynomial. The packed bootstrapping procedure is completed in the following steps:

- Using the encrypted secret key the initial ciphertext is converted into a q_0 ciphertext where the polynomial being encrypted is represented as $Z' \in (\mathbb{Z}/2^{r+1}\mathbb{Z})[X]/\Phi_m(X)$.
- Next they apply a homomorphic inverse-DFT transformation to get encryption of polynomials that have the coefficients of Z' in their plaintext slots.
- Since the coefficients of Z' are in plaintext slots, a bit extraction procedure is performed on all the slots in parallel. The result is encryption of polynomials that have the coefficients of a in their plaintext slots.
- Finally, a homomorphic DFT transformation is applied to get back a ciphertext that encrypts the polynomial a itself.

1.2 Technical Overview

We intend to show our work on bootstrapping on a Standard-LWE public key encryption scheme by Brakerski-Vaikuntanathan [cite]. Our work closely follows the work of Ducas-Micciancio. In their work, the symmetric key encryption scheme is parameterized by a dimension n, a message space $m \in \mathbb{Z}_t$ for t = 4, a ciphertext modulus $q = n^{\mathcal{O}(1)}$ and a randomized rounding function χ: $\mathbb{R} \rightarrow \mathbb{Z}$. The scheme is further facilitated by conventional LWE features such as modulus reduction and key-switching.

One of the limitations of their schemes is that they work in modulo-2 domain and hence messages are encrypted as bits. While one can carry out simple operations like homomorphic NOT($\sim$), OR, XOR operations with low noise growth on ciphertext, performing a homomorphic NAND gate operation leads to a ciphertext that can no more be further computed on. At this stage one needs to bootstrap the ciphertext to bring it back into original form. Thus, every time a homomorphic NAND operation is called, the ciphertext has to be restored by bootstrapping.

In many applications (e.g. financial data, machine learning etc) one needs to work on integer arithmetic rather than binary operations on bits. To further extend the size of the plaintext one can work on a Chinese Remainder Theorem (CRT) based plaintext representation. The LWE scheme of Brakerski-Vaikunthanathan present many such advantages that removes the above discussed limitations. We work on a instance of the scheme by extending the plaintext to p(originally defined for $p = 2$). The scheme has a plaintext space for $m \in \mathbb{Z}_p$ for a small p. It also incorporates the dimension-modulus reduction technique for noise growth. In brief, one can use this property to evaluate many homomorphic computations (specifically many ciphertext multiplications) before the need to bootstrap. This can be considered one of the significant benefits over the scheme of Ducas-Micciancio. Security of the security is based on the classical hardness of solving standard lattice problems in the worst-case.

One of the serious drawbacks of this scheme is exponential increase in dimension upon homomorphic multiplication. Specifically, the dimension goes up from $n + 1$ to roughly $n^2 + n$. At this stage we are required to re-linearize the ciphertext to reduce back the dimension to n. While the re-linearization technique is ingenious in restoring the ciphertext dimension it also adds substantial noise to the ciphertext. Keeping a ladder of moduli to tackle the noise growth only seems natural.

Next we describe our approach to refresh the ciphertext to restore the ciphertext to its initial state.

1.3 Our Approach

Our starting point is the standard LWE ciphertext c of the form $c = (a, b) \in \left[\mathbb{Z}_q^n \times \mathbb{Z}_q\right]$ and a secret key s of the form $s \in \mathbb{Z}_q^n$. Now consider a homomorphic register *HREG* which is capable of holding a finite field element in a small multiplicative group $\mathcal{G}\langle X \rangle$. The homomorphic register *HREG* has the following properties:

- **Initialize**: Given a FF element $a \in \mathbb{Z}_q$ it produces a homomorphic register $HREG_a$.
- **Addition**: Given two homomorphic registers $HREG_a$ and $HREG_b$ it produces a homorphic register $HREG_{(a+b)}$ for $(a + b) \in \mathbb{Z}_q$. Addition is achieved by equivalent multiplication in the group $\mathcal{G}\langle X \rangle$.
- **Multiplication**: Multiplication is here rather expensive and restrictive. We can only compute the multiplication of registers with finite field elements in the clear. Moreover, it is restrictive in the sense that multiplication can only be computed for registers with published auxiliary register sets. Formally, given a

finite field element $a \in \mathbb{Z}_q$ and a auxiliary register set κ_b it produces a register $HREG_{ab}$.

- **Negation**: In a recent work by Micciancio-Sorrell [cite] showed the register can have a negation operation computed in $\mathcal{O}(1)$ operation with just a minimal memory overhead. Given a homomorphic register $HREG_a$ it produces a register $HREG_{-a}$.

Initially we evaluate a circuit on the initial ciphertext and when the levels are exhausted we refresh the ciphertext to produce a fresh encryption. To facilitate the bootstrapping we initially pre-compute and publish the auxiliary register set of the secret key elements s_i as κ_{s_i}.

The decryption is performed as:

$$m = (b - \langle \mathbf{a} \cdot \mathbf{s} \rangle \; mod \; q) \mod p.$$

To perform this decryption homomorphically we compute:

$$HREG_b - \sum_{i=0}^{n-1} HREG_{a_i \cdot s_i} = HREG_{b - \langle \mathbf{a} \cdot \mathbf{s} \rangle}$$

This only computes the decryption is $\mathbb{Z}_q$!!!!!!!!!!!!!!!!!!

2 PRELIMINARIES

Notations. Let $\mathcal{D}$ denote a distribution over some finite set S. Then, $x \xleftarrow{\$} \mathcal{D}$ is used to denote the fact that x is chosen from the distribution D. When we say $x \xleftarrow{\$} \mathcal{S}$, we simply mean that x is chosen from the uniform distribution over S. Unless explicitly mentioned, all logarithms are to base 2.

In this work, we utilize "noise" distributions over integers. The only property of these distributions we use is their magnitude. Hence, we define a B-bounded distribution to be a distribution over the integers where the magnitude of a sample is bounded with high probability. A definition follows.

Definition 2.1. (B-bounded distributions). A $distribution \; ensemble \; \{\chi_n\}_{n \in \mathbb{N}}$ over the integers, is called B-bounded if

$$\underset{e \xleftarrow{\$} \chi_n}{Pr} [|e| > B] \leq 2^{-\widetilde{\Omega}(n)}.$$

We denote scalars in plain (e.g. x) and vectors in bold lowercase (e.g. **v**), and matrices in bold uppercase (e.g. **A**). The ℓ_i norm of a vector is denoted by v_i. Inner product is denoted by $\langle \mathbf{v}, \mathbf{u} \rangle$, recall that $\langle \mathbf{v}, \mathbf{u} \rangle = \mathbf{v}^T \cdot \mathbf{u}$. Let **v** be an n dimensional vector. For all $i = 1, \cdots, n$ the $i^{th} element$ in **v** is denoted $\mathbf{v}[i]$.

2.1 Learning With Errors (LWE)

The LWE problem was introduced by Regev [cite] as a generalization of learning parity with noise. For positive integers n and $q \geq 2$, a vector $\mathbf{s} \in \mathbb{Z}_q^n$, and a probability distribution χ on $\mathbb{Z}_q$, let $\mathbf{A}_{s,\chi}$ be the distribution obtained by choosing a vector $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$ uniformly at random and a noise term $e \xleftarrow{\$} \chi$, and outputting $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$.

Definition 2.2 (LWE.)**.** For an integer $q = q(n)$ and an error distribution $\chi = \chi(n)$ over $\mathbb{Z}_q$, the learning with errors problem $\mathbf{LWE}_{n,m,q,\chi}$ is defined as follows: Given m independent samples from $\mathbf{A}_{\mathbf{s},\chi} \left(for \; some \; \mathbf{s} \in \mathbb{Z}_q^n \right)$, output **s** with noticeable probability.

The (average-case) decision variant of the **LWE** problem, denoted $\mathbf{DLWE}_{n,m,q,\chi,}$ is to distinguish (with non-negligible advantage) m samples chosen according to $\mathbf{A}_{\mathbf{s},\chi}$,(for uniformly random $s \xleftarrow{\$} \mathbb{Z}_q^n$ from m samples chosen according to the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$. We denote by $\mathbf{DLWE}_{n,q,\chi}$ the variant where the adversary gets oracle access to $\mathbf{A}_{\mathbf{s},\chi}$, and is not a-priori bounded in the number of samples.

2.2 Ring-LWE

A ciphertext in the LWE form suffers from the problem of dimension expansion upon homomorphic evaluation. This not only results in greater computational overhead but adds significant noise growth. To accelerate cryptographic constructions based on the Learning With Errors problem (LWE) Lyubashevsky-Peikert-Regev [cite LPR10] introduced the ring learning with error (RLWE) problem[cite]. Lyubashevsky et al. first showed a quantum reduction from approximate SVP (in the worst case) on ideal lattices in R to the search version of ring-LWE, where the goal is to recover the secret $s \in R_q$ (with high probability, for any s) from arbitrarily many noisy products. They also gave a reduction from the search problem to the decision variant, which shows that ring-LWE distribution is pseudorandom assuming worst-case problems on ideal lattices are hard for polynomial-time quantum algorithms.

Definition 2.3 (RLWE)**.** For security parameter λ, let $f(x) = x^d + 1$ where $d = d(\lambda)$ is a power of 2. Let $q = q(\lambda) \geq 2$ be an integer. Let $R = \mathbb{Z}[x]/(f(x))$ and let $R_q = R/qR$. Let $\chi = \chi(\lambda)$ be a distribution over R. The $\text{RLWE}_{d,q,\chi}$ problem is to distinguish the following two distributions: In the first distribution, one samples (a_i, b_i) uniformly from R_q^2. In the second distribution, one first draws $s \leftarrow R_q$ uniformly and then samples $(a_i, b_i) \in R_q^2$ by sampling $a_i \leftarrow R_q$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = a_i.s + e_i$. The $\text{RLWE}_{d,q,\chi}$ assumption is that the $\text{RLWE}_{d,q,\chi}$ problem is infeasible.

2.3 Vector and Polynomial Decomposition

We define the standard tools used for decomposition of vectors and polynomials which aid in low norm key switching and dimension reduction operations.

Vector Decomposition: Given vectors $\mathbf{x}(x_0, x_1, \cdots, x_{n-1}) \in \mathbb{Z}_q^n$, $\mathbf{y}(y_0, y_1, \cdots, y_{n-1}) \in \mathbb{Z}_q^n$ and $\mathbf{z}\left(z_0, z_1, \cdots, z_{(n-1)\cdot\lfloor \log q \rfloor}\right) \in \mathbb{Z}_q^{n \cdot \lceil \log q \rceil}$ we have the following operations

- $\text{BitDecomp}(\mathbf{x})$: It decomposes **x** into its bit representation vectors. That is, we write $\mathbf{x} = \sum_{j=0}^{\lfloor \log q \rfloor} 2^j \cdot \mathbf{u}_j$, where each vector u_i represents the i^{th} bit vector and represented as $u_i = ((x_0)_i, (x_1)_i, \cdots, (x_{n-1})_i)$. The output of the operation is a vector of vectors $\left(u_0, u_1, \cdots, u_{\lfloor \log q \rfloor}\right)$.
- $\text{Powerof2}(\mathbf{y})$: It outputs the vector $\left(\mathbf{y}, 2 \cdot \mathbf{y}, \cdots, 2^{\lfloor \log q \rfloor} \cdot \mathbf{y}\right) \in \left(\mathbb{Z}_q^n\right)^{\lceil \log q \rceil}$
- $\text{BitDecompInverse}(\mathbf{z})$: It is the inverse of bit decompose function BitDecomp operation and produces a

n-length vector by recombines each of the $\lceil \log q \rceil$ interleaved values. The recombination is given by the algebraic expression $\sum_{i=0}^{\lfloor \log q \rfloor} z_{i \cdot \lceil \log q \rceil} \cdot 2^i$.

The operations are defined here deals with representation of 2's but can be easily extended to any other base p. For the vectors **x** and **y** the following property holds true:

$$\langle \text{BitDecomp}\left(\mathbf{x}, q\right), \text{Powerof2}\left(\mathbf{y}, q\right) \rangle$$

$$= \sum_{j=0}^{\lfloor \log q \rfloor} \langle 2^j \cdot \mathbf{u}_j, \mathbf{y} \rangle = \langle \sum_{j=0}^{\lfloor \log q \rfloor} 2^j \cdot \mathbf{u}_j, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{y} \rangle \bmod \text{q}$$

Interestingly, these operations can be extended to cyclotomic polynimails where $x, y \in R_q^n$ and vector inner product is replaced by ring multiplication.

3 Standard LWE private key scheme

In this section we discuss the symmetric-key encryption scheme whose security is based on the LWE assumption. The scheme is parameterized by a dimension $n = n(\lambda)$, a plaintext modulus $p \geq 2$ and a ciphertext modulus $q = \mathcal{O}(\lambda)$ where λ is the security parameter.

The construction is quite straightforward, encryption of a message $m \in \mathbb{Z}_p$ under secret key $\mathbf{s} \in \mathbb{Z}_q^n$ is

$$c = (\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + p.e + m) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$$

Here **a** is a uniformly generated random vector; $\mathbf{a} \xleftarrow{\$} \mathcal{U}_q$. The error e is chosen from a narrow B-bounded Gaussian distribution; $\mathbf{e} \xleftarrow{\$} \chi_B$.

We define decryption as a function which first computes the noise and then reduces it to recover the message m. Since, the error is generated by the discrete Gaussian distribution, the noise is interpreted in the interval $\left[\frac{-q}{2}, \frac{q}{2}\right]$ after reducing in mod q. Moreover, decryption is only successful when the noise term doesn't wrap around the ciphertext modulus q. Decryption is defined as:

$$noise = (b - \langle \mathbf{a}, \mathbf{s} \rangle) \ mod\ q$$

$$m = noise\ mod\ p$$

Formally, the basic symmytric scheme SLWE is defined by tuple of PPT algorithms Keygen, Encrypt and Decrypt as follows:

- SLWE.KeyGen(1^λ): It takes in the security factor λ and generates a secret key $\mathbf{s} \xleftarrow{\$} \chi_B^n$ from a narrow bounded Gaussian distribution.
- SLWE.Encrypt($m \in \mathbb{Z}_p, \mathbf{s}$): It generates a ciphertext $c = (\mathbf{a}, b)$. The first component $\mathbf{a} \xleftarrow{\$} \mathcal{U}_q$ is generated from a n-dimensional uniform random distribution. The next component b is generated by the inner-product of the **a** and **s** which in turn assimilates the message along with the error e as per the definition of LWE.
- SLWE.Decrypt($c = (\mathbf{a}, b), \mathbf{s}$): Message m is recovered by annihilating the two masks one after the other. First the LWE mask is removed by subtracting the inner-product of **a** and **s** from the ciphertext. Then the error mask is destroyed by reducing in modulo-p.

$$m' = (b - \langle \mathbf{a}, \mathbf{s} \rangle \bmod \text{q}) \bmod \text{p}.$$

Definition 3.1. The symmetric scheme SLWE is correct if for all $m \in \mathbb{Z}_p$ and all $\mathbf{sk} \leftarrow \text{SLWE.Keygen}\left(1^\lambda\right)$,

$$\mathbf{Pr}\left[\text{SLWE.}Dec_{sk}\left(\text{SLWE.}Encrypt\left(m\right)\right) \neq m\right] = negl\left(\lambda\right).$$

where the probabilities are obtained by repeating the experiments with SLWE.Encrypt and SLWE.Keygen.

It is easy to see that Homomorphic addition is achieved by adding the corresponding components of the ciphertext. If we have two ciphertexts $\text{SLWE.Encrypt}\left(m_1\right) = c_1 = (\mathbf{a}_1, b_1)$ and $\text{SLWE.Encrypt}\left(m_2\right) = c_2 = (\mathbf{a}_2, b_2)$ then homomorphic addition, $\text{SLWE.Add}(c_1, c_2)$ is given by:

$$c_{add} \leftarrow \text{SLWE.Add}(c_1, c_2)$$

$$c_{add}.\mathbf{a} = \mathbf{a}_1 + \mathbf{a}_2 \ : c_{add}.b = b_1 + b_2\ .$$

Homomorphic addition works because of the linear property of inner products which is: $\langle \mathbf{a}_1, \mathbf{s} \rangle + \langle \mathbf{a}_2, \mathbf{s} \rangle = \langle (\mathbf{a}_1 + \mathbf{a}_2), \mathbf{s} \rangle$ mod-q.

Homomorphic multiplication is not readily achieved because of the non-linear property of inner-product. This is due to the fact that

$$\left(\sum a_1[i] \cdot s[i]\right) \cdot \left(\sum a_2[i] \cdot s[i]\right) =$$

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(a_1[i] \cdot a_2[j]\right) \left(s[i] \cdot s[j]\right).$$

It can be observed that product of the noise terms from the two ciphertexts is approximately equal to the plaintext products of the messages embedded in the ciphertexts.

$$\left(b_1 - \sum \mathbf{a}_1[i]\mathbf{s}[i]\right) \cdot \left(b_2 - \sum \mathbf{a}_2[i]\mathbf{s}[i]\right) = pE + m_1.m_2 \approx m_1.m_2$$

We can leverage the above equation to perform homomorphic multiplication. Assuming circular security, we release the encryption of $s[i]$ and $s[i][j]$ $(i, j \in [n])$terms. Specifically, we compute the encryption of powers of a base $b \leq p$ and publish it as EvalMultKey.

$$\mathbf{a}_{2^\ell i} \xleftarrow{\$} \mathcal{U}_q; b_{2^\ell i} \leftarrow \langle \mathbf{a}_{2^\ell i}, \mathbf{s} \rangle + p.e_{2^\ell i} + 2^\ell.s[i]$$

$$\mathbf{a}_{2^\ell i,j} \xleftarrow{\$} \mathcal{U}_q; b_{2^\ell i,j} \leftarrow \langle \mathbf{a}_{2^\ell i,j}, \mathbf{s} \rangle + p.e_{2^\ell i,j} + 2^\ell.s[i]s[j]$$

$$\ell \in \left\{0, 1, \cdots, \lceil \log_p q \rceil - 1\right\}$$

$$\text{EvalMultKey} \leftarrow \mathbf{a}_{2^\ell i}, \mathbf{a}_{2^\ell i,j}, b_{2^\ell i}, b_{2^\ell i,j}$$

Using EvalMultKey we can evaluate a product $\alpha \cdot s[i]$ for $\alpha \in \mathbb{Z}_q$. The floowing algorithm describes this procedure.

We can write the product of noise of the two ciphertexts as:

$$\left(b_1 - \sum \mathbf{a}_1[i]\mathbf{s}[i]\right) \cdot \left(b_2 - \sum \mathbf{a}_2[i]\mathbf{s}[i]\right)$$

$$= b_1 b_2 - b_1 \sum a_2[i] - b_2 \sum a_1[i] + \sum a_1[i]s[i] \sum a_1[i]s[i]$$

$$= b_1 b_2 - \sum s[i]\left(b_1 a_2[i] + b_2 a_1[i]\right) + \sum_i \sum_j \left(a_1[i]a_2[j]\right)\left(s[i]s[j]\right)$$

Algorithm 1 Plaintext-Ciphertext product

procedure INPUT:($\alpha \in \mathbb{Z}_q$, EvalMultKey$\leftarrow \mathbf{a}_{2^{\ell}i}, b_{2^{\ell}i}, \ell \in [\lceil \log_p q \rceil]$)
  Initialize ciphertext $\mathbf{a}_{out} \leftarrow 0, b_{out} \leftarrow 0$.
  Compute the base-p representation of $\alpha = \sum_\tau p^\tau \cdot \alpha_\tau$
  for $j = \{0, 1, \cdots, \lceil \log_p q \rceil - 1\}$ do
    if $\alpha_\tau \neq 0$
      $a_{out} = a_{out} + a_{2^j,i}; b_{out} = b_{out} + a_{2^j,i}$
  end for
  return (a_{out}, b_{out})

$$= b_1 b_2 - \sum s[i] h_i + \sum_i \sum_j h_{i,j} \left(s[i] s[j] \right),$$

where $h_i = (b_1 a_2[i] + b_2 a_1[i])$ and $h_{i,j} = (a_1[i] a_2[j])$. Using the plaintext-ciphertext multiplcation we can further simply the expression to:

$$b_1 b_2 - \sum_i \left(b_{h_i,s_i} - \langle \mathbf{a}_{h_i,s_i}, \mathbf{s} \rangle \right) + \sum_{i,j} \left(b_{h_{ij},s_{ij}} - \langle \mathbf{a}_{h_{ij},s_{ij}}, \mathbf{s} \rangle \right)$$

$$= b_1 b_2 + \left(- \sum_i b_{h_i,s_i} + \sum_{i,j} b_{h_{ij},s_{ij}} \right) + \left\langle \left(\sum_i \mathbf{a}_{h_i,s_i} - \sum_{i,j} \mathbf{a}_{h_{ij},s_{ij}} \right), \mathbf{s} \right\rangle$$

$$= b_{prod} - \langle \mathbf{a}_{prod}, \mathbf{s} \rangle$$

The values of b_{prod} and a_{prod} can be computed from the ciphertexts c_1, c_2 and EvalMultKey.

3.1 Noise Growth Analysis on Ciphertext Multiplication

To perform ciphertext multiplication "EvalMult", we make use of the individual coefficients $s[i]$ and pair-wise coefficients $s[i] \cdot s[j]$ of secret key $\mathbf{s}$ stored in EvalMultKey. To compute each coefficient $h_i \cdot s[i]$ we add $\lceil \log_p q \rceil$ number of cipertexts indexed from the digit decomposition of h_i in base p. Alternatively, we can write $h_i \cdot s[i] = \left(\lceil \log_p q \rceil \right) \cdot \left(\text{-} \langle \mathbf{a}, \mathbf{s} \rangle \right) \approx \left(\lceil \log_p q \rceil \right) \cdot p.||e||_\infty$. Taking into account the noise incurred from the pairwise terms $s[i].s[j]$ we can see that the total noise growth on ciphertext multiplication is $\left(n^2 + n\right) \cdot p.||e||_\infty \cdot \left(\lceil \log_p q \rceil \right) \approx \left(n^2 + n\right) \cdot pB \cdot \left(\lceil \log_p q \rceil \right)$.

The noise growth term found in the above expression can be considered as worst-case limit. To arrive at an average case noise growth we use the central limit theorem (CLT). Using CLT we can estimate the noise to grow to be the following:

$$\sqrt{(n^2 + n)} \cdot pB \cdot \left(\lceil \log_p q \rceil \right) \approx pnB \cdot \left(\lceil \log_p q \rceil \right)$$

3.2 Modulus Switching

Modulus switching allows the ciphertext working in modulo-Q to be switched to a ciphertext working with another modulus q. An evaluator can apply the modulus switching technique without the knowledge of secret key $\mathbf{s}$. Typically, we use the rounding function $\lfloor \cdot \rceil_{Q:q} : \mathbb{Z}_Q \rightarrow \mathbb{Z}_q$ for modulus switching. The functions is defined as follows:

$$\lfloor x \rceil_{Q:q} = \lfloor x \cdot (q/Q) \rceil$$

The rounding function is applied to vectors and scalars of the ciphertext coordinatewise and is defined as follows:

$$\textbf{ModSwitch}\left(\mathbf{a}, b\right) = \left(\left(\lfloor a_1 \rceil_{Q:q}, \lfloor a_2 \rceil_{Q:q} \cdots \lfloor a_n \rceil_{Q:q} \right), \lfloor b \rceil_{Q:q} \right)$$

3.2.1 *Correctness of Modulus Switch:*

Given two ciphertexts $c = (\mathbf{a}, b)$ and $c' = (\mathbf{a}', b')$ where $c' \leftarrow \textbf{ModSwitch}_{p \leftarrow q}\left(\mathbf{a}, b\right)$ and two modulii p and q then:

$$\textbf{Pr}\left[\textbf{SLWE.Decrypt}\left(c = (\mathbf{a}, b)\right) \neq \textbf{SLWE.Decrypt}\left(c' = (\mathbf{a}', b')\right)\right]$$

$$= negl(\lambda)$$

where the probability is defined over random coins of encryptions of $c \in$ SLWE.

3.3 Key Switching

Key switching is a homomorphic operation of transforming a ciphertext encrypted under a key (original key) into another ciphertext under a new key and preserving the message. Suppose, we have ciphertext c encrypting m under the secret key $\mathbf{s}$ then we can transform the ciphertext c into another ciphertext c' encrypting message m under a new key $\mathbf{t}$. The key switching procedure is parametrized by a base p_{ks}, and requires the holder of the secret key to publish a evaluation key called "KeySwitchHint".

The KeySwitchHint is a container that holds all possible encryptions of powers of secret key $p_{ks}^i \cdot t[i]$ for $i \in \left[\lceil \log_{p_{ks}} q \rceil \right]$ and $\mathbf{t}$ being the new secret key. The ciphertext c encrypting the message m under secret key $\mathbf{s}$ is of the form $(c = (a, b))$ where $b = \sum_i a[i].s[i] + pe + m$. For every $a[i]$ we can compute a encryption of $a[i]s[i]$ using the digits of $a[i]$ in base-p_{ks} representation. Algorithm-2 explains this procedure in detail.

Suppose, we have the ciphertexts $c_{a_i s_i} = (a_{a_i s_i}, b_{a_i s_i})$ then we can re-write the decryption equation as follows:

$$m = \left(b - \left[\sum_i b_{a_i s_i} - \langle \mathbf{a}_{a_i s_i}, \mathbf{t} \rangle \right] \bmod \text{q} \right) \bmod \text{p}$$

$$\Rightarrow m = \left(\left(b - \sum_i b_{a_i s_i} \right) + \left\langle \sum_i \mathbf{a}_{a_i s_i}, \mathbf{t} \right\rangle \bmod \text{q} \right) \bmod \text{p}$$

$$b' = \left(b - \sum_i b_{a_i s_i} \right); \mathbf{a}' = - \sum_i \mathbf{a}_{a_i s_i}$$

Hence, we arrive at a new ciphertext $c' = (\mathbf{a'}, b')$ which can be decrypted by the new secret key $\mathbf{t}$.

The size of the KeySwitchHint is $n.\lceil \log_{p_{ks}} q \rceil$ in number of ciphertexts. The noise grows by a factor of $n\lceil \log_{p_{ks}} q \rceil$ upon key switching in worst case. Using central limit theorem heuristics we can say that noise growth factor is roughly equal to $\sqrt{n}\lceil \log_{p_{ks}} q \rceil$

Algorithm 2 GetCiphertext($m = a_i s_i$)

procedure INPUT:($a_i \in \mathbb{Z}_q$, KeySwitchHint$\leftarrow \mathbf{a}_{2^\ell s_i}, b_{2^\ell s_i}, \ell \in \lceil \lceil \log_p q \rceil \rceil$)
  Initialize ciphertext $\mathbf{a}_{out} \leftarrow 0, b_{out} \leftarrow 0$.
  Compute the base-p_{ks} representation of $a_i = \sum_\tau p_{ks}^\tau \cdot a_{i\tau}$
  for $j = \{0, 1, \cdots, \lceil \log_{p_{ks}} q \rceil - 1\}$ do
    if $\alpha_\tau \neq 0$
      $\mathbf{a}_{out} = \mathbf{a}_{out} + a_{i\tau} \cdot \mathbf{a}_{2^\tau, s_i}; b_{out} = b_{out} + a_{i\tau} a_{2^\tau, s_i}$
  end for
  return $(\mathbf{a}_{out}, b_{out})$

4 RING-GSW SCHEME

This section describes in depth, the concrete scheme used to instantiate our homomorphic register HREG. The scheme is a ring variant of the original scheme proposed by Gentry-Sahai-Waters(GSW).

The original scheme is based on the learning with errors (LWE) problem and employs LWE matrices to encrypt a bit. The secret key in the GSW scheme is a low norm vector $s \in \mathbb{Z}_q^n$. A significant outcome of the scheme was to efficiently perform multiplication without relinearization. Other FHE schemes such as BGV, FV etc mostly rely on relinearization approach to perform homomorphic multiplication. To perform relinearization the evaluator also needs to store a set of evaluation keys the number of which is a function in the logarithm of modulus and security parameter λ. By eliminating the relinearization step the scheme does away with the need to store large evaluation keys.

The ring-GSW scheme is parameterized by a dimension n = n(λ), a ciphertext modulus q = q(λ), a plaintext modulus $p \geq 2$, a B-bounded discrete Gaussian distribution χ_{B,R_q}, a uniform distribution over ring $\mathcal{U}_{q,R_q}$ and $N = 2\ell$ where $\ell = \lceil \log_r q \rceil$ for some $r = 2^k$. Here r is also known as the relinearization window.

The public key encryption scheme is described as follows:

- RGSW.Keygen(1^λ): Draw a polynomial ring **s** from bounded Guassian distribution χ_{B,R_q}. $\mathbf{s} \xleftarrow{\$} \chi_{B,R_q}$. Set the secret key vector as: $\mathbf{sk} \leftarrow [1; \mathbf{s}], \mathbf{sk} \in R_q^{2\times 1}$. To generate the public key **pk** we uniformly sample a ring polynomial **a** from the uniform distribution over rings $\mathcal{U}_{R_q}$. $\mathbf{a} \xleftarrow{\$} \mathcal{U}_{R_q}$. To generate the Ring LWE term **b**, we first generate a small norm ring polynomial **e** from the χ_{B,R_q} and set **b** as follows:

$$e \xleftarrow{\$} \chi_{B,R_q}; \quad b = as + pe.$$

  Set the public key **pk** as:

$$\mathbf{pk} = A_{1\times 2} = \begin{bmatrix} b & a \end{bmatrix}$$

  The noise term generated is similar to that in BGV style encryption and is needed here so as to mimic the noise term of standard LWE encryption scheme.

- RGSW.Encrypt($\mathbf{pk}, \mu$): The message space here is a polynomial ring in a small modulo p. Technically, one can encode polynomial $\mu \in R_p$ using naive encoding or using the slot encoding, however as it will be clear later that we use naive encoding for only a single coefficient of the polynomial ring. To encrypt a message ring μ we generate a random vector of polynomial rings **r** from a ternary distribution $\mathcal{T}_{R_q}$.

$$\{-1, 0, 1\} \xleftarrow{\$} \mathcal{T}_1 \quad \mathbf{r} \xleftarrow{\$} \mathcal{T}_{R_q}^N \quad r \in R_q^{N\times 1}, N = 2\ell.$$

We generate the error vector $\mathbf{E}_{N\times 1}$ from the discrete Gaussian distribution χ_{B,R_q}^N, $\mathbf{E} \xleftarrow{\$} \chi_{B,R_q}^N$ and set the encryption as follows:

$$C = \mu \cdot \mathbf{BDI}\left(I_{N\times N}\right) + \mathbf{r}_{N\times 1} \cdot A_{1\times 2} + pE_{N\times 1}$$

For $r = 2$ ciphertext can be visualized as follows:

$$C_{N\times 2} = \begin{bmatrix} r_0 \cdot b + \mu + pe_0 & r_0 \cdot a \\ r_1 \cdot b + 2\mu + pe_1 & r_1 \cdot a \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ r_{\ell-1} \cdot b + 2^{\ell-1}\mu + pe_{\ell-1} & r_{\ell-1} \cdot a \\ r_\ell \cdot b + pe_\ell & r_\ell \cdot a + \mu \\ r_{\ell+1} \cdot b + pe_{\ell+1} & r_{\ell+1} \cdot a + 2\mu \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ r_{2\ell-1} \cdot b + pe_{2\ell-1} & r_{2\ell-1} \cdot a + 2^{\ell-1}\mu \end{bmatrix}$$

- RGSW.Decrypt(sk, C): Given the ciphertext $C_{N\times 2}$, the plaintext $\mu \in R_q$ is recovered by multiplying the first row of C by the secret key **sk** as follows:

$$\mu' = \left[C_{00} \times \mathbf{sk} \times \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \bmod \mathrm{q}\right] \bmod \mathrm{p}$$

$$= (r_0 \cdot b + \mu + pe_0) - r_0 \cdot a \cdot s$$

$$= r_0 \cdot (b - a \cdot s) + \mu + pe_0$$

$$= r_0 \cdot e + \mu + pe_0 \equiv (\mu \bmod \mathrm{p})$$

Definition 4.1. The public key scheme RGSW is correct if for all $\mu \in R_p$ and all **sk** $\leftarrow$ RGSW.Keygen (1^λ) we have,

$$\mathbf{Pr}\left[\text{RGSW.}Dec_{sk}\left(\text{RGSW.}Encrypt\left(\mu\right)\right) \neq \mu\right] = negl\left(\lambda\right).$$

where the probabilities are obtained by repeating the experiments with RGSW.Encrypt and RGSW.Keygen.

4.1 Homomorphic Operations

To define homomorphic operations we consider two input ciphertexts $C_{N\times 2}$ and $D_{N\times 2} \in R_q^{N\times 2}$ with respective plaintexts μ_1 and $\mu_2 \in R_p$.

- RGSW.ADD(C, D): To add the two ciphertexts $C_{N\times 2}$ and $D_{N\times 2}$ we add them component-wise, and output the sum $C_{N\times 2} + D_{N\times 2}$. The cost of addition is simply $2N$ ring additions.
- RGSW.MULT(C, D): To multiply the two ciphertexts $C_{N\times 2}$ and $D_{N\times 2}$ we compute the bit decomposition of the ciphertext C and multiply it with D. That is we simply output $BD\left(C\right) \cdot D$.

4.2 Correctness of Homomorphic Multiplication and Noise-Growth

The noise growth of homomorphic multiplication can be seen from the following equations:

$$\mathrm{M}_{N\times 2} = \mathbf{BD}\left(C\right)\cdot D$$

Noise growth can be shown by considering only the first row of $\mathrm{M}_0 = [M_{00}\ M_{01}]$. By property of matrix multiplication we only need to consider the first row of C and ciphertext D. Let the first row of ciphertext C, i.e. $[C_{00}\ C_{01}]$ be given as follows:

$$C_{00} = \alpha = r_0'\cdot b + \mu_1 + pe_0'; \quad C_{01} = \beta = r_0'\cdot a$$

Then, bit decomposition of C_0 is shown as follows:

$$\mathbf{BD}\left(C_{00}\right) = [\alpha_0, \alpha_1, \cdots, \alpha_{\ell-1}, \beta_0, \beta_1, \cdots, \beta_{\ell-1}]$$

Let the ciphertext D be given as below:

$$D = \begin{bmatrix} r_0\cdot b + \mu_2 + pe_0 & r_0\cdot a \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ r_{2\ell-1}\cdot b + pe_{2\ell-1} & r_{2\ell-1}\cdot a + 2^{\ell-1}\mu_2 \end{bmatrix}$$

Then, M_{00} is given as:

$$\begin{aligned} M_{00} &= b\cdot\sum_{i=0}^{\ell-1}(\alpha_i r_i) + \mu_2\cdot\sum_{i=0}^{\ell-1}\left(\alpha_i 2^i\right) + p\sum_{i=0}^{\ell-1}(\alpha_i e_i) \\ &\quad + b\cdot\sum_{i=0}^{\ell-1}(\beta_i r_{\ell+i}) + p\sum_{i=0}^{\ell-1}(\beta_i e_{\ell+i}) \\ &= b\cdot\sum_{i=0}^{\ell-1}(\alpha_i r_i) + \mu_2\cdot\alpha + p\sum_{i=0}^{\ell-1}(\alpha_i e_i) \\ &\quad + b\cdot\sum_{i=0}^{\ell-1}(\beta_i r_{\ell+i}) + p\sum_{i=0}^{\ell-1}(\beta_i e_{\ell+i}) \end{aligned}$$

M_{01} is given as:

$$\begin{aligned} M_{01} &= a\sum_{i=0}^{\ell-1}(\alpha_i r_i) + a\sum_{i=0}^{\ell-1}(\beta_i r_{\ell+i}) + \mu_2\cdot\sum_{i=0}^{\ell-1}\left(\beta_i 2^i\right) \\ &= a\sum_{i=0}^{\ell-1}(\alpha_i r_i) + a\sum_{i=0}^{\ell-1}(\beta_i r_{\ell+i}) + \mu_2\beta \end{aligned}$$

Decryption proceeds by computing $\left[M_0\ \times\ \mathbf{sk}\times\left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right]\right] = M_{00} - s\cdot M_{01}$.

$$\begin{aligned} M_{00} - s\cdot M_{01} &= (b-as)\cdot\sum_{i=0}^{\ell-1}(\alpha_i r_i + \beta_i r_{\ell+i}) + \mu_2\left(\alpha-\beta s\right) \\ &\quad + p\sum_{i=0}^{\ell-1}(\alpha_i e_i + \beta_i e_{\ell+i}) \\ &= pE_1 + \mu_2\left(pe' + \mu_1\right) = pE + \mu_1\mu_2 \\ &= \mu_1\mu_2\ (\bmod \mathrm{p}) \end{aligned}$$

Correctness of the homomorphic multiplication is contingent upon the error growth in noise term. Decryption succeeds till the noise term is bounded by $\frac{q}{2}$. The two main components that leads to noise growth are as follows:

- The first component is the error term generated due to the implicit relinearization and hence called relinearization error. Since, the magnitude of the random vector r_i's are much smaller (as they are generated from a ternary distribution) than that of error vectors e_i's we will consider only e_i error vectors in relinearization error term.
  The relinearization error term is a summation term and hence depends on the bounds of e_i's, i.e. $||e||_{\ell_1}$ and the relinearization index r. Here we consider ℓ_1 norm as a means to quantify the noise magnitude. The number of summation terms is given by $\ell = \lceil \log_r q \rceil$. Hence, raising the relinearization index r to a carefully chosen number keeps the ciphertext in a compact form and also mitigates the error growth.

- The second component in the noise term comes directly from the plaintext component of second ciphertext μ_2. Moreover, the error term vanishes if $\mu_2 = 0$. This makes the ciphertext error growth asymmetric.
  Generally, in most RLWE schemes the error growth is symmetric and to avoid exponential growth of noise one has to use a binary tree for multiplication. However, with RGSW's symmetric noise growth there is no need of performing multiplication through a binary tree.

4.3 Faster multiplication in EVALUATION mode using NTT

We note that our homomorphic register **HREG** is always present in EVALUATION domain and we avoid going back to COEFFICIENT domain unless it is really required to. A ciphertext is converted from COEFFICIENT to EVALUATION form using the nega-cyclic number theoretic transform, NTT and back by inverse NTT^{-1}. The following algorithm describes an approach for faster homomorphic multiplication.

Algorithm 3 EvalMult in EVALUATION domain

procedure INPUT:$((C_{N\times 2}, D_{N\times 2}))$
  Compute BD(C)$\rightarrow C_{N\times N}$
  for $C_{i,j},\ i\in[N], j\in\{0,1\}$ do:
    $c_{i,j} \xleftarrow{NTT^{-1}} C_{i,j}$
    Bit decompose $c_{i,j}$; $(\gamma_0,\gamma_1,\cdots\gamma_{\ell-1}) \xleftarrow{BD} c_{i,j}$
    for i in $[\ell]$ do:
      $c_i \xleftarrow{NTT} \gamma_i$
    replace $C_{i,j}$ with $c_0, c_1, \cdots, c_{\ell-1}$
    end for
  end for
  perform matrix-multiplication, $\mathrm{M}_{N\times 2} = C_{N\times N}\cdot D_{N\times 2}$
  return M

5 Bootstrapping Procedure

In this section we describe the bootstrapping procedure to refresh the SLWE ciphertext. We use a homomorphic register

HREG as a core structure for holding an encrypted SLWE ciphertext. Concretely, we utilize the ring GSW scheme for instantiating the register **HREG**.

6 Software Implementation in Palisade Library and Results

We implemented the SLWE bootstrapping scheme in a general purpose multi-threaded C++ library. We design this library to be modular and provide three major software layers, each of which includes a collection of C++ classes to provide encapsulation, low inter-module coupling and high intra-module cohesion. The software layers are the (1) cryptographic primitives, (2) lattice constructs, and (3) arithmetic (primitive math) layers.

- The bottom layer is the primitives layer which contains the

Acknowledgments

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.