< Back Home published on June 29, 2023

The BGV fully homomorphic encryption scheme

This is a sister blogpost to the previous one about a similar scheme (BFV) and it's part of the series that covers fully homomorphic encryption techniques and applications.

Introduction

In this blogpost we will focus on the encryption, decryption, relinearization and the noise analysis of the Brakerski, Gentry, Vaikuntanathan (BGV) fully homomorphic encryption scheme. A Python implementation of the aforementioned will be provided.

> DISCLAIMER: This toy implementation is not meant to be secure or optimized for efficiency. It's for educational purposes only, and for us to better understand the inner workings.

What to expect from this blogpost?

1. We briefly describe the motivations of fully homomorphic encryption.
2. We present the BGV scheme.
3. We offer a Python implementation for the BGV scheme, for educational purposes.

Now, let's make a short, informal, tour before starting. Fully homomorphic encryption (FHE) allows us to perform computation on encrypted data. While there are a few cryptographic schemes to achieve this, in this blogpost we explore the BGV scheme.

Firstly, this scheme allows us to encrypt messages into ciphertexts. Intuitively, a message is hidden in a ciphertext if there is no way to get to retrieve the original message from its encryption. To an attacker that sees a ciphertext, that ciphertext should look like "random garbage". In order to achieve this we hide the message by "masking" it with some noise. We say it's *hard* for an attacker to retrieve a message from its ciphertext if doing so is at least as hard as solving a computationally unfeasable problem. The problem the security of the BGV scheme is based on is called the Ring Learning with Errors problem.

Secondly, the scheme allows us to perform arithmetic operations (additions, multiplications) using noisy ciphertexts. However, we will encounter some problems along the way, such as the noise increasing after computations, or the size of the resulting ciphertext increasing. We'll see how to handle these using **modulus switching** (for the former) and **relinearization** (for the latter).

This post will get a bit mathematical, so before starting we introduce some notation. Any other notation will be defined when introduced.

Notations:

- m, m', m^* - messages.
- c, c', c^* - ciphertexts. If ciphertexts have more components, we denote the components with indices: $c = (c_0, c_1)$
- λ - Security parameter.
- $\mathbb{Z}_q$ - the set of integers from $(-q/2, q/2]$
- R - Ring.
  - Ex: $\mathbb{Z}$ - the integers.
  - Ex: $\mathbb{Z}[X]/(X^n + 1)$ - the quotient polynomial ring with integer coefficients: .
- R_q - Ring modulo the ideal generated by q.
  - Ex: $\mathbb{Z}_q$ - the integers mod q.
  - Ex: $\mathbb{Z}_q[X]/(X^n + 1)$ - the quotient polynomial ring with coefficients being integers in $(-q/2, q/2]$.

- $[a]_q$ - $a \bmod q$, coefficients centered in $(-q/2, q/2]$. When applying $[\cdot]_q$ to a polynomial we mean to apply it to all its coefficients individually. When applying $[\cdot]$ to a vector of elements, we mean to apply it to each element individually. So $[(a, b)]_q = ([a]_q, [b]_q)$. We give the similar treatment to reduction $\bmod q$ operation
- $\chi = \chi(\lambda)$ - Noise distribution, parametrized by λ (usually Gaussian).
- For simplicity, when we say we sample a polynomial from some distribution (or a polynomial comes from some distribution) we mean that its coefficients are sampled independently from that distribution.
- $e \leftarrow \chi$ - An element e sampled from the distribution χ
- $a \xleftarrow{R} S$ - An element sampled uniformly from a set S
  - Ex: $a \xleftarrow{R} R_q$ An element a sampled uniformly from the ring R_q
- $a \cdot b$ denotes the multiplication of 2 elements from the ring R_q
- ta denotes the multiplication of coefficients of $a \in R_q$ by some scalar t.

Fully Homomorphic Encryption (FHE)

Before defining fully homomorphic encryption, let's look at some **applications** of it to get a better intuition:

1. Outsourcing storage and computations

- A company wants to use a cloud provider to store data and do computational heavy lifting.
- *Task*: The company wants to use the information (for example: to do machine learning, extract statistical properties) without locally retrieving it (defeating the purpose of using the cloud company services). Furthermore, the company doesn't want the cloud provider to see the sensitive data.
- *Solution*: Using homomorphic encryption, the company can encrypt and send the data to the cloud provider and then the cloud provider can process the information (as requested by the company) in the encrypted form, without learning anything about the data.
- Ex: finance, medical apps, consumer privacy in ads.

2. Private queries

- A client wants to retrieve a query against a large database held by a provider.
- *Task*: The client wants to retrieve a query **without** the database provider learning the query.
- *Solution*: Using homomorphic encryption the client can encrypt the index of the record and then the database can use this encrypted index to fetch (the comparison operation can be done using FHE) and return the encrypted result to the client. Finally, the client can decrypt the record.

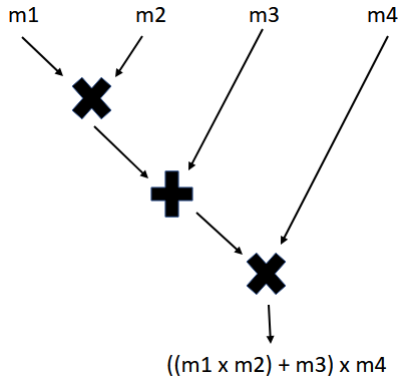
3. Private set intersection

- A client and a server each has a set of elements. The client wants to know which elements from its set are in the server's set.
- *Task*: The client wants to find out the intersection of its elements and the server's, without the server learning anything about the client's elements and vice versa.
- *Solution (Very simplified)*: They translate the comparison operation into a subtraction ($x = y \Rightarrow x - y = 0$). The client encrypts its elements and sends them to the server. Then, the server evalutes these subtractions homomorphically. Because the result of the subtraction is an encrypted result, the server learns nothing. Then the server sends the comparison results back and the client decrypts the results. We have a longer, more in-depth blogpost about it here. Happy reading!
- Examples: Client: user contacts, Server: whatsapp contacts; Client: own passwords, Server: database with breached passwords.

Intuition

- A user has an arithmetic function f and some inputs $(m_1, ...m_n)$.
- He wants to compute $m^* = f(m_1, ...m_n)$ but instead of computing it directly, he wants to encrypt $(m_1, ...m_n) \rightarrow (c_1, ...c_n)$ and do the computation on the

ciphertext, obtaining a result which eventually decrypts to $m^* = f(m_1, ...m_n)$.
- It's helpful to view the functions as tree of operations. For example (taken from the BFV blogpost): $f(m_1, m_2, m_3, m_4) = (m_1 \times m_2 + m_3) \times m_4$



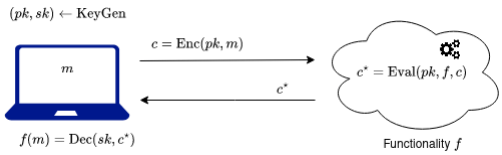
Homomorphic Encryption -- Definition

Let $(\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$ be a tuple of procedures.
Let $f \in \mathcal{F}$ be a function in a family of functions. This function can also be written as a *circuit* from a family of circuits $C \in \mathcal{C}$.

- $(sk, pk) \leftarrow \mathrm{KeyGen}(1^\lambda, 1^d)$ - key generation, λ is a security parameter, d is a functionality parameter (degree of the polynomial or depth of the circtuit).
- $c_i \leftarrow \mathrm{Enc}(pk, m_i)$ - Encryption of a message m_i -- known as fresh ciphertexts.
- $c^* = \mathrm{Eval}(pk, f, c_1, ...c_n)$ - Evaluate the function f on the ciphertexts -- known as evaluated ciphertexts.

Correctness

We require the (FHE) scheme to correctly decrypt both fresh and evaluated ciphertexts: $\mathrm{Dec}(sk, c^*) = m^* = f(m_1, ..., m_n)$



In the image above, the user with the laptop is interested evaluating a certain circuit on its data m. Therefore it sends its data, encrypted, to the cloud so that the cloud computes the circuit. By correctness, the user indeed gets the circuit applied on its data m.

Circuits?

The two important operations we care about are addition and multiplication, because with them we can construct any arithmetic operation! Homomorphic encryption schemes started out by encrypting bits, so the equivalent operations for additon and multiplication on bits, `xor` and and are used.

$$\oplus = + \mod 2$$
$$\otimes = \times \mod 2$$

To convey the fact that a circuit supports both `xor` and and usually a `nand` gate is used, because `nand` gates are composed by 1 multiplication and 1 addition and they are logically complete (you can build any circuit with them).
$\mathrm{nand}(a, b) = a \times b + 1 \mod 2$

By extending to multiple bits, with the gates we discussed above you can represent any computation using boolean circuits.

HE can be classified based on the type of functions f that it supports:

- Partially homomorphic encryption
  Given $\mathrm{Enc}(m_1)$, and $\mathrm{Enc}(m_2)$ you can do limited operations (either additions or multiplications)
- Somewhat (leveled) homomorphic encryption
  Given $\mathrm{Enc}(m_1), ..., \mathrm{Enc}(m_n)$ you can compute $\mathrm{Enc}(f(m_1, ...m_n))$ where f

is a polynomial of a limited degree. One can do limited number of multiplications (Circuits of a maximum depth).

- Fully homomorphic encryption
  One can do unlimited multiplications and additions.

The community put a lot of effort into the analysing FHE schemes: finding good parameters, finding software and hardware optimizations, coming up with possible attacks and adjusting the schemes against them. An interesting read is the homomorphic encryption standard, which compiles good practices and techniques for developing and using existing schemes.

Ring Learning With Errors (RLWE)

In this section we'll focus on briefly introducing the Ring learning with errors (RLWE) problem. The problem is very complex and we'll only attempt to cover the basic blocks for building an intuition. For the interested reader more details can be found in this paper.

Quick recap on working with polynomials

This section is taken from the BFV blogpost, but we'll repeat it here to save you a click. For this subsection we denote with $\mathbb{Z}/q\mathbb{Z} = \{0, ..., q-1\}$.

The HE scheme we are going to describe deals with *adding and multiplying polynomials*. Here we present a quick example of how to work with polynomials, so that we all have the same starting point. If you already know how to do this, you can skip it.

First thing, let's add and multiply polynomials *modulo some polynomial* f. This "modulo f" thing simply means that we add and multiply the polynomials in the usual way, but we take the remainders of the results when divided by f. When we do these additions and multiplications $\bmod f$, we sometimes say in a fancy way that we are working in the *ring* $\mathbb{Z}[x]/(f)$ of reduced polynomials.

Let's take $f = x^4 + 1$. If we look at $p(x) = x^5$, then $p(x) = x \cdot (x^4 + 1) - x$. Therefore, when taking the reminder we get $p(x) = -x \bmod f$. For faster computations $\bmod f$ you can use this trick: when making $\bmod f$, simply replace x^4 by -1, x^5 by $-x$ and so on.

Let's consider two polynomials $a(x) = x^3 + x^2 + 7$ and $b(x) = x^2 + 11x$. Then:

$$a(x) + b(x) \bmod f = x^3 + 2x^2 + 11x + 7 \bmod f.$$

Here nothing special happened. Let's multiply them:

$$\begin{aligned} a(x) \cdot b(x) \bmod f &= (x^3 + x^2 + 7) \cdot (x^2 + 11x) \bmod f \\ &= x^5 + 11x^4 + x^4 + 11x^3 + 7x^2 + 77x \bmod f \\ &= -x - 11 - 1 + 11x^3 + 7x^2 + 77x \bmod f \\ &= 11x^3 + 7x^2 + 76x - 12 \bmod f. \end{aligned}$$

Now let's go one step further and see how we perform these operations of polynomials not just modulo f, *but also modulo some integer* q. As you might expect, the coefficients of these polynomials are also reduced modulo q, so they always take integer values from 0 to $q - 1$. This time, we say that we are working the ring of reduced polynomials $\mathbb{Z}/q\mathbb{Z}[x]/(f)$.

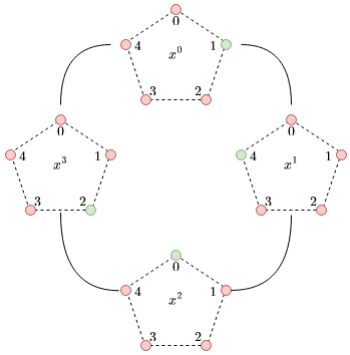
Let's take the previous example, $f = x^4 + 1$, $a(x) = x^3 + x^2 + 7$, $b(x) = x^2 + 11x$ and consider $q = 5$. We can think of a and b as already taken $\bmod f$. If we take them further modulo q, then $[a(x)]_q = x^3 + x^2 + 2$ and $[b(x)]_q = x^2 + x$. Moreover,

$$[a(x) + b(x)]_q = x^3 + x^2 + 2 + x^2 + x = x^3 + 2x^2 + x + 2$$

and

$$
\begin{aligned}
[a(x) \cdot b(x)]_q &= (x^3 + x^2 + 2) \cdot (x^2 + x) \\
&= x^5 + x^4 + x^4 + x^3 + 2x^2 + 2x \\
&= -x - 1 - 1 + x^3 + 2x^2 + 2x \\
&= x^3 + 2x^2 + x + 3
\end{aligned}
$$

To give a powerful visual intuition of polynomials in quotient rings, consider the following polynomial$1 + 4x + 2x^3 \in \mathbb{Z}/5\mathbb{Z}[x]/(x^4 + 1)$ and its visual representation. For this grade 3 polynomial we have four pentagons, for each coefficient. Each pentagon represents $\mathbb{Z}/5\mathbb{Z}$. The value of the coefficient is colored with green:



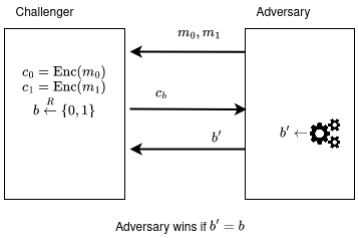
The security of the BGV scheme

Like we said in the introduction, our ciphertexts will be masked with some noise to look as random as possible in order to hide any information an attacker may extract about the message.

In fact it can be proven that the BGV scheme is secure as long as nobody can solve the Ring Learning With Errors (RLWE) problem described below. The parameters of the scheme must be chosen such that there are no known algorithms to solve the underlying RLWE problem in feasable time.

Formally, the idea of "ciphertexts looking like garbage" is called ciphertext indistinguishability. This concept can be formulated using a game between a challenger and an adversary:

- The challenger receives two messages m_0, m_1 from the adversary. Then, the challenger encrypts them into c_0, c_1.
- Then, the challenger secretly flips a bit $b \in \{0, 1\}$ and sends the ciphertext c_b to the adversary.
- The adversary must guess the bit b. After running an efficient algorithm, it outputs a bit b'. It wins if $b' = b$.



If there exists **at least one** adversary that can win the game with a probability significantly larger than 1/2 (a coinflip) using a computationally efficient algorithm (something that can run in our universe's lifetime), the challenger's algorithm is considered insecure. When we prove the security of a scheme, we actually show that if an adversary had such an algorithm that can win the security game, he can take that algorithm and use it to solve a problem that is considered *computationally hard*.

We will now describe the RLWE problem.

Consider the quotient polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. This ring has polynomials with degree at most n and integer coefficients in $(-q/2, q/2]$.

The RLWE problem - decision

The problem asks to distinguish between the following two distributions:

- (a_i, b_i) with $a_i \xleftarrow{R} R_q, b_i \xleftarrow{R} R_q$.
- (a_i, b_i) with $a_i \xleftarrow{R} R_q, e_i \leftarrow \chi$, and $b_i = a_i \cdot s + e_i$ for some "small" secret $s \in R_q$ sampled before.

The RLWE problem also has a search version, where given pairs (a_i, b_i), $b_i = a_i \cdot s + e_i$ you need to find s. In practice, the decision version is more widely used when proving the security of different schemes.

More details about the RLWE problem can be found in this paper. In essence, by adding some small noise e_i, from a Gaussian distribution, we make our secret s unrecoverable even if we have access to many RLWE samples.

The original BGV paper presents the algorithms (encryption, decryption, etc) of a HE scheme based on a more general problem, the *General Learning with Errors (GLWE)* problem. For simplicity, we work with the RLWE case in this blogpost.

The BGV encryption scheme

Generate the following parameters

- n - degree of $X^n + 1$. n is chosen as a power of 2.
- q - ciphertext modulus.
- t - plainext modulus, $t \ll q$.
- χ - the noise distribution, a discrete Gaussian.

We consider $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. This ring has polynomials with degree at most n and integer coefficients in $(-q/2, q/2]$.

Note:

- All polynomial operations are considered $\mathbf{mod} q$ unless otherwise specified, to ease notation.
- When we talk about "small" polynomials, we intuitively think about them having small coefficients.
- Polynomials that come from the noise distribution will be colored with green (ex: e) and the ones that will come from the secret key distribution will be colored with red (Ex: s, u).

We will have one secret key sk, a public key with two components $pk = (pk_0, pk_1)$, a ciphertext with 2 components $c = (c_0, c_1)$ The messsage will come from the ring R_t.

`SecretKeyGen(params) -> sk`

- Draw s from a secret key distribution which outputs "small" elements with overwhelming probability. In practice $s \in \{-1, 0, 1\}^n$, with the probability of sampling 0 specified as a parameter and the probabilities of sampling -1 or 1 being equal.
- Return the secret key $sk = s$.

`PubKeyGen(sk, params) -> (pk0, pk1)`

- Draw a random element $a \xleftarrow{R} R_q$ and the noise $e \leftarrow \chi$.
- Return the public key: $pk = (pk_0, pk_1) = \underbrace{(a \cdot s + te, -a)}_{\text{RLWE instance-like}}$.

Notice that the public key looks like a RLWE sample, but the error e is multiplied by the plaintext modulus t.

`Encrypt(m, pk, params) -> (c0, c1)`

- Draw the noise $e_0, e_1 \leftarrow \chi$ and a "small" polynomial $u \in \{-1, 0, 1\}^n$ in the same way as the secret.

- Compute $c_0 = pk_0 \cdot u + te_0 + m$
- Compute $c_1 = pk_1 \cdot u + te_1$
- Return $c = (c_0, c_1)$.

The ciphertext components (c_0, c_1) are elements in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$.

`Decrypt(c, sk, params)`

- Compute $m = c_0 + c_1 \cdot s \bmod q \bmod t$
- Return m.

▶ Correctness (Click for math)

FHE Operations

Here we will explore the homomorphic properties of the scheme. We refer to the two input message and ciphertext pairs with $(m, c), (m', c')$ and to the resulting message-ciphertext pair with (m^*, c^*).

The goal is to find two functions, $\mathbf{add}, \mathbf{mul}$, that work on ciphertexts, for addition and multiplication, such that:

- when $m^* = m + m'$ we have $c^* = \mathbf{add}(c, c')$ and $\mathbf{Dec}(c^*) = m^*$
- when $m^* = m \cdot m'$ we have $c^* = \mathbf{mul}(c, c')$ and $\mathbf{Dec}(c^*) = m^*$

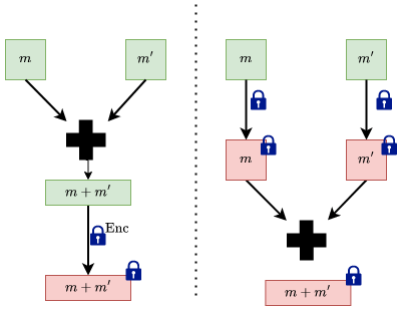
Recall that c and c' encrypt m and m' i.e.:

$$
\begin{aligned}
c_0 &= pk_0 \cdot u + te_0 + m \\
c_1 &= pk_1 \cdot u + te_1
\end{aligned}
$$

$$
\begin{aligned}
c'_0 &= pk_0 \cdot u' + te'_0 + m \\
c'_1 &= pk_1 \cdot u' + te'_1
\end{aligned}
$$

FHE Addition

The image on the left performs the addition before the encryption, with the messages in clear. This is not what we want. The image on the right is the desirable outcome, where we encrypt the messages, then perform addition on the encrypted messages.



We define add: `add(c, c') -> c*`

- $c^*_0 = c_0 + c'_0$
- $c^*_1 = c_1 + c'_1$
- return $c^* = (c^*_0, c^*_1)$

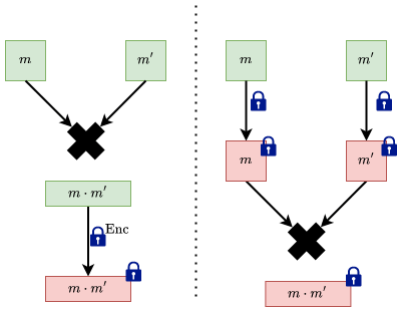
▶ Correctness (click for math)

The resulting ciphertext c^* is not fresh anymore (just encrypted), and has a bigger noise than any of the previous ciphertexts that were inputs in the addition operation. In general, using non-fresh ciphertexts c^* to perform future operations will carry over the noise from all the previous operations.

We'll analyse how the noise grows in a dedicated section at the end.

FHE Multiplication

As in the addition case, we want the multiplication to be done on the encrypted messages.



Before describing how we multiply two ciphertexts we need to reinterpret the decryption equation: $c_0 + c_1 \cdot s \equiv m \bmod t$ as a linear equation: $y = ax + b$ in s.

If we look at adding two such linear equations in s we obtain another linear equation in s: $c_0 + c_1 \cdot s + c'_0 + c'_1 \cdot s = \underbrace{(c_0 + c'_0)}_{c^*_0} + \underbrace{(c_1 + c'_1)}_{c^*_1} \cdot s$

However, when multiplying two linear equations in s we get a quadratic equation ($ax^2 + bx + c$) in s:

$$\underbrace{(c_0 + c_1 \cdot s)}_{=m \bmod t} \cdot \underbrace{(c'_0 + c'_1 \cdot s)}_{=m' \bmod t} = c_1 \cdot c'_1 \cdot s^2 + (c_0 \cdot c'_1 + c_1 \cdot c'_0) \cdot s + c_0 \cdot c'_0$$

We can define `mul` in the following way: `mul(c, c') -> c*`

- $c^*_0 = c_0 \cdot c'_0$
- $c^*_1 = c_0 \cdot c'_1 + c_1 \cdot c'_0$
- $c^*_2 = c_1 \cdot c'_1$
- return $c^* = (c^*_0, c^*_1, c^*_2)$

Remarks

- Similarly, as in addition, the noise of the resulting ciphertext c^* increases.
- We increase the number of ciphertext components by 1. This is not sustainable.
- Our decryption equation is linear, however after `mul` we have a quadratic decryption equation with 3 ciphertexts that doesn't play well with our idea of linear decryption equation.

To solve the remarks the concept of **relinearization** is introduced.

Relinearization

The concept was explained in the BFV blogpost as well.

Intuition: We want to transform the quadratic equation $c^*_0 + c^*_1 \cdot s + c^*_2 \cdot s^2$ in s into some other linear equation $\hat{c}_0 + \hat{c}_1 \cdot \hat{s}$ in some other secret key $\hat{s}$.

In order to do this we need to give some extra information (a "hint") about the key s that will help us get $\hat{s}$.

Consider the hint to be the pair:

$$(ek_0, ek_1) = ((a \cdot s + te) + s^2, -a)$$

This is very similar to how the public key is generated and we can use the `PubKeyGen -> (pk0, pk1)` to generate it: $(pk_0 + s^2, pk_1)$ Intuitively, we use a RLWE sample to hide s^2.

Notice that:

$$ek_0 + ek_1 \cdot s = \cancel{a \cdot s} + e + s^2 - \cancel{a \cdot s} = s^2 + e$$

The easiest way to get the term $c_2^* s^2$ from the above equation is to multiply it by c_2^*. However, c_2^* is a random element in R_q and it will yield some large noise $c_2^* e$. So we have to do something smarter.

Key switching v1

Key switching is a variant of relinearization. We start with a ciphertext c.

Idea:
Split the ciphertext into multiple "small" ciphertexts (for now think of small as the ciphertext polynomials having small coefficents). We can do this by decomposing the ciphertext coefficients into a small base T.

Intuition:
To build an intuition, we can take the most common bases used in representing numbers:

- Base 10: $123_{10} = 3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2$
- Base 2: $12_{10} = 1100_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$

Programmatically, this can be done by dividing the integer by the base T *repeatedly* and collecting the remainders.

In Python code:

```python
def int2base(x: int, base: int) -> List[int]:
    digits = []
    while x > 0:
        q, r = divmod(x, base)
        digits.append(r)
        x = q
    return digits

print(int2base(123, 10))
# [3, 2, 1]
print(int2base(12, 2))
# [0, 0, 1, 1]
```

A ciphertext component $c \in R_q$ is a polynomial of degree less than n, with coefficients in $\{0, 1, .., q-1\}$, which we denote as follows: $c(x) = c[0] + c[1]x + ... + c[n-1]x^{n-1}$

We want to write the coefficients $(c[0], ..., c[n-1])$ in base T. Since any coefficient is less than q, this decomposition has at most $\lfloor \log_T q \rfloor + 1$ terms. For example $c[0] = \sum_{i=0}^{\lfloor \log_T q \rfloor} T^i c[0]^{(i)}$ where $c[0]^{(i)}$ is the corresponding digit $< T$ in the decomposition. Using these digits we can construct the polynomials $c^{(i)}$:

$$c^{(i)}(x) = c[0]^{(i)} + c[1]^{(i)}x + ... + c[n-1]^{(i)}x^{n-1}$$

Then using these polynomials we can decompose the original polynomial c:

$$c = \sum_{i=0}^{\lfloor \log_T q \rfloor} T^i \cdot c^{(i)} \bmod q$$

The polynomials $c^{(i)}$ are elements of R_T and for a reasonable small T they have small coefficients ($< T$). For each one of them we generate the hints:

$$(ek_0^{(i)}, ek_1^{(i)}) = (a_i \cdot s + te_i + T^i s^2, -a_i)$$

with $a_i \xleftarrow{R} R_q$ and $e_i \leftarrow \chi$.

Now we go back to our ciphertext $c^* = (c_0^*, c_1^*, c_2^*)$, the result of `EvalMul`. We decompose c_2^* in the base T and we get:

$$c_2^* = \sum_{i=0}^{\lfloor \log_T q \rfloor} T^i \cdot c_2^{*(i)} \bmod q$$

Then we construct the new ciphertext $\hat{c}$ with the **two** components:

$$\hat{c}_0 = c_0^* + \sum_{i=0}^{\lfloor \log_T q \rfloor} ek_0^{(i)} \cdot c_2^{*(i)} = c_0^* + \sum_{i=0}^{\lfloor \log_T q \rfloor} ((a_i \cdot s + te_i) + T^i s^2) \cdot c_2^{*(i)}$$
$$\hat{c}_1 = c_1^* + \sum_{i=0}^{\lfloor \log_T q \rfloor} ek_1^{(i)} \cdot c_2^{*(i)} = c_1^* + \sum_{i=0}^{\lfloor \log_T q \rfloor} -a_i \cdot c_2^{*(i)}$$

Using the above relations, we obtain the following linear equation:

$$\hat{c}_0 + \hat{c}_1 \cdot s = \underbrace{c_0^* + c_1^* \cdot s + c_2^* \cdot s^2}_{m \cdot m'} + \underbrace{\sum_{i=0}^{\lfloor \log_T q \rfloor} te_i \cdot c_2^{*(i)}}_{\text{relinearization error}}$$

(the terms containing $a_i \cdot s$ from $\hat{c}_0$ and $-a_i \cdot s$ from $\hat{c}_1 \cdot s$ cancel out, c_2^* is reconstructed and we are left with an error term).

Because the relinearization error is a multiple of t, it goes away when we decrypt and reduce $\bmod t$ as long as the relinearization error does not wrap $m^* = m \cdot m'$ around q.

Remarks

- We need to relinearize after a multiplication.
- A smaller T means we have a smaller noise, but more relinearization keys are needed.
- T is chosen based on what we want to optimize. We usually want to choose T to introduce a noise around the size of the noise in the ciphertexts. Once the noise grows we can use T^2 or something else. We can also choose a big T at the start (which introduces lots of starting noise) and we don't need to change it later.

Modulus switching

Task: You want to reduce the noise that accumulates in the ciphertext, after performing homomorphic operations. The modulus switching technique allows us to decrease the noise, to ensure the decryption is still correct after performing homomorphic operations. More analysis is provided in a dedicated section below.

Idea: The technique uses two moduli Q, q with $q < Q$ and $q|Q$. This involves changing the ring where the ciphertexts live, R_Q, with a ring R_q, that is parametrized by the smaller modulus. When you perform this change, the coefficient ring $\mathbb{Z}_Q$ is "approximated" by some ring $\mathbb{Z}_q$, in the intuitive sense that the numbers become smaller (the numbers are scaled down by some factor). As we will see this means that the noise will be smaller in absolute value. Although, the noise will relatively stay the same, since $q < Q$, we mostly care about the magnitude of the noise, especially in multiplications.

You can perform this repeatedly with many moduli $\{q_0, q_1, ..., q_L\}$ with decreasing values to lower the noise after each multiplication.

The easiest way to do this is to scale down the ciphertext's components by q/Q and **smartly round** in a way to keep the decryption equation correct. To ensure this, we want the scaling to keep the following propriety: $\tilde{c}_i = c_i \bmod t$, where $\tilde{c}_i \approx (q/Q)c_i$ where $i \in \{0, 1\}$.

The two requirements that we want to have are

1. The decryption should be correct (decryption mod Q should be the same as decryption mod q): $[c_0 + c_1 \cdot s]_Q = [\tilde{c}_0 + \tilde{c}_1 \cdot s]_q \bmod t$.
2. The noise scales down.

Rounding

In order for the decryption to be correct we must adjust c_i before scaling by q/Q. To do this we add a **small correction term** δ_i per component. We require δ_i to:

1. Only influence the error, hence $\delta_i \equiv 0 \bmod t$ (it will disappear when decrypting)
2. Make the ciphertext to be divisible by Q/q:

$$c_i + \delta_i \equiv 0 \bmod \frac{Q}{q} \Rightarrow \delta_i \equiv -c_i \bmod \frac{Q}{q}$$

One easy choice to set the correction terms such that both properties hold is $\delta_i = t[-c_i t^{-1}]_{Q/q}$ where t^{-1} is the inverse of t modulo Q/q.

In the end, we define the components $\tilde{c}_i$ of the modulus switched ciphertext: $\tilde{c}_i = \frac{q}{Q}(c_i + \delta_i)$

▶ Correctness (Click for math)

Key switching v2

Now that we know about modulus switching we introduce a second version to do key switching in BGV. Recall that the key switching technique is used in relinearization, for decreasing the size of the ciphertext obtained in multiplication.

Idea: Given two moduli q, Q with $q < Q$ and $q|Q$, we start with elements in R_q and we make a detour in a ring R_Q with a bigger modulus Q where we perform our relinearization. Then we perform modulus switching to go back to our small ring R_q.

We generate the *hint*:

$$(ek_0, ek_1) = ((a \cdot s + te + \frac{Q}{q}s^2), -a) \bmod Q$$

Recall that $c^* = (c_0^*, c_1^*, c_2^*)$ is the result of the `mul`. Then we compute the components of the new ciphertext $\hat{c}$:

$$\hat{c}_0 = \frac{Q}{q}c_0^* + c_2^* \cdot ek_0 \bmod Q$$
$$\hat{c}_1 = \frac{Q}{q}c_1^* + c_2^* \cdot ek_1 \bmod Q$$

Lastly, we scale them both using modulus switching: $\hat{c}_i = \frac{q}{Q}(\hat{c}_i + \delta_i)$ with $\delta_i = t[-c_i t^{-1}]_{Q/q}$, like before.

The relinearization equation looks like this:

$$\hat{c}_0 + \hat{c}_1 \cdot s = \underbrace{c_0^* + c_1^* \cdot s + c_2^* \cdot s^2}_{m \cdot m'} + \underbrace{\frac{q}{Q}(tc_2^* \cdot e + \delta_0 + \delta_1 \cdot s)}_{\text{relinearization error}}$$

Because the relinearization error is a multiple of t (remember that we chose δ_i to be multiples of t too, to not influence the error), it will go away when we decrypt and reduce $\bmod t$ as long as the relinearization error does not wrap $m^* = m \cdot m'$ around q.

▶ Correctness (click for math)

Noise analysis

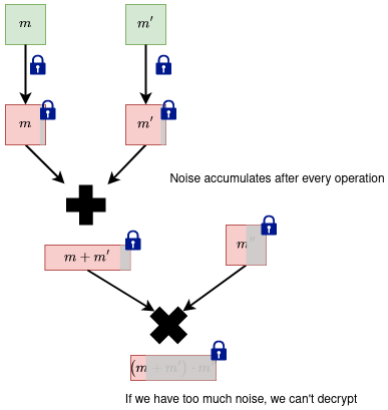
The main motivation of analysing the noise growth is because we want to set the parameters of the scheme based on it.

Relevant papers:

- Ilia Iliashenko thesis
- CS15
- Evaluating the effectiveness of heuristic worst-case noise analysis in FHE
- GHS
- Finding and Evaluating Parameters for BGV

Measuring Noise

When doing operations (such as addition, multiplication) with noisy ciphertexts, the total noise accumulates. In the end we want our decryption to be correct. However, this can only happen if the accumulated noise does not exceed some **bound**.



Intuition: We want to "measure" this accumulated noise by looking at "how much" a polynomial can impact an operation it's involved in. For example, we can analyze the impact of the polynomials e_i which represent the RLWE noise or of the secret s. We can represent the polynomial's impact with a number.

We can "measure" the impact of the polynomials using the:

1. Infinity norm: $\|a\|_\infty = \max |a_i|$, which is the value of the biggest coefficient in absolute value.
2. Canonical norm: $\|a\|^{\text{can}}$. In order to define it we use the so called **canonical embedding** of the polynomial.

While the infinity norm is easy to understand, the canonical embedding is a bit more mathematically involved. Instead of searching for a way to measure the impact of a polynomial $a \in R$, we transfer the polynomial in another space, such as $\mathbb{C}^n$, and we search for a better measure there.

Recall that we work with polynomials from the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with n a power of 2.

Canonical embedding
We have the quotient polynomial $X^n + 1$ of R which has the complex roots $\{\zeta_0, \dots, \zeta_{n-1}\}$ with $\zeta_i^n = -1 \Rightarrow \zeta_i^{2n} = 1$.
Then we take a polynomial $a \in R$:
$a(X) = a_0 + a_1 X + ...a_{n-1} X^{n-1}$ and we evaluate a in every root ζ_i. This leads to a vector $(a(\zeta_0), ...a(\zeta_{n-1})) \in \mathbb{C}^n$. We call this vector the **canonical embedding** of a.
The canonical embedding is defined as

$$\sigma : R \to \mathbb{C}^n, \quad \sigma(a) = (a(\zeta_0), ...a(\zeta_{n-1}))$$

Canonical norm
Since now we have obtained a new vector in $\mathbb{C}^n$ we can look at its infinity norm and define the **canonical norm** $\|a\|^{\text{can}} = \|\sigma(a)\|_\infty$.

In practice, the canonical norm is used to compute the noise bounds, because it provides a better decryption noise bound.

1. There is a relationship between these norms: $\|a\|_\infty \leq c_{2n} \cdot \|a\|^{\text{can}}$ for some constant c_{2n}, which is 1 when n is a power of 2 (see this). With this inequality in

mind, it suffices to assure correct decryption by setting the canonical norm of the noise to be less than $q/2$.
2. When doing multiplications, the noise bound offered by the canonical norm increases slower than the bound offered by the infinity norm. We have these two inequalities:

$$\|ab\|_\infty \leq \gamma \|a\|_\infty \|b\|_\infty$$
$$\|ab\|^{\text{can}} \leq \|a\|^{\text{can}} \|b\|^{\text{can}}$$

where $\gamma > 1$ is the so-called *expansion factor* of the ring R. For more details, see this.

Noise bounds

Suppose we choose $a \in R$ with coefficients sampled independently from one of the distributions (discrete Gaussian, random, ternary). Let V_a be the variance of each coefficient $(a_0, ...a_n)$ in a. In Ilia Iliashenko's thesis and in the Finding and Evaluating Parameters for BGV paper it's shown that we can choose a number D such that $\|a\|^{\text{can}} \leq D\sqrt{nV_a}$ holds with overwhelming probability. In practice, we set $D = 6$.

When analysing the noise growth we'll make use of the following proprieties. For two polynomials $a, b \in R$ with variances V_a, V_b and some constant k we have:

- $V_{a+b} = V_a + V_b$,
- $V_{ka} = k^2 V_a$,
- $V_{a \cdot b} = nV_aV_b$.

For the distributions that we work with we have the **coefficient** variances:

- Discrete Gaussian distribution with standard deviation σ : $V_{\mathcal{DG}} = \sigma^2$.
- Ternary distribution: $V_3 = 2/3$.
- Uniform distribution with integer coefficients in $(-q/2, q/2]$: $V_q \approx q^2/12$.

Noise for operations

We now have the building blocks (starting values and operations) to analyse the noise for each operation. The goal of analysing the noise buildup is to choose the scheme parameters such that the scheme properties (fhe, decryption) hold and the scheme remains secure. Following the paper Finding and Evaluating Parameters for BGV we have:

Fresh ciphertext noise

In order to make sure the decryption is correct we must make sure the error

$$v = c_0 + c_1 \cdot s = \underbrace{t(e \cdot u + e_0 + e_1 \cdot s)}_{r} + m = r + m \bmod q$$

does not wrap around the modulus (i.e $\|v\|_\infty \leq c_{2n}\|v\|^{\text{can}} \leq q/2$). The message m is thought to come uniformly from R_t.

Recall that

- The errors e, e_0, e_1 come from a discrete Gaussian => $V_e = V_{e_0} = V_{e_1} = \sigma^2$.
- The secret s and u come from a ternary distribution => $V_s = V_u = 2/3$.
- The message m comes from R_t and we assume that the coefficients are uniformly distributed in $\mathbb{Z}_t$ => $V_m = t^2/12$.

$$\begin{aligned} V_v &= V_{m + t(e \cdot u + e_0 + e_1 \cdot s)} \\ &= V_m + t^2 V_{e \cdot u + e_0 + e_1 \cdot s} \\ &= \frac{t^2}{12} + t^2 \left(n\sigma^2 \frac{2}{3} + \sigma^2 + n\sigma^2 \frac{2}{3} \right) \\ &= t^2 \left(\frac{1}{12} + \sigma^2 \left(\frac{4}{3} n + 1 \right) \right) \end{aligned}$$

Hence: $\|v\|^{\text{can}} \leq D\sqrt{nV_v}$

The parameters for the scheme and distributions, q, t, σ, n, are chosen such that the decryption is correct.

Addition

For two ciphertexts (c, c') encrypting (m, m') we have the corresponding errors (v, v'). The error after ciphertext addition is v_{add}.

If we use the decryption equation we have

$$v_{\text{add}} = v + v' = (c_0 + c_1 \cdot s) + (c'_0 + c'_1 \cdot s) = r + m + r' + m' \bmod q$$

We have $m + m' = [m + m']_t + gt$ with $g \in R$ and $\|g\|_\infty = 1$ (because both $\|m\|_\infty, \|m'\|_\infty < t/2$). By replacing in the equation above we obtain: $v + v' = [m + m']_t + r + r' + gt \bmod q$

We see that the noise $r + r' + gt$ grows linearly.

Finally, using the canonical norm we have the following:

$$\|v_{\text{add}}\|^{\text{can}} \leq \|v\|^{\text{can}} + \|v'\|^{\text{can}}$$

Multiplication

Similarly for multiplication, if we use the decryption equation we have:

$$\begin{aligned} v_{\text{mul}} &= v \cdot v' = (c_0 + c_1 \cdot s) \cdot (c'_0 + c'_1 \cdot s) \\ &= (r + m) + (r' + m') = mm' + mr' + m'r + rr' \bmod q \end{aligned}$$

If we look at the error term $mr' + m'r + rr'$ we see that in the leading term rr' we multiply the noises, therefore the noise grows quadratically.

Using the canonical norm, we have:

$$\|v_{\text{mul}}\|^{\text{can}} \leq \|v\|^{\text{can}} \|v'\|^{\text{can}}$$

Modulus switching

For modulus switch we scale the noise down and then we add the small error that comes with the correction terms δ_0, δ_1. Recall that

- We set $\delta_i = t[-c_i t^{-1}]_{Q/q}$. We assume the terms $[-c_i t^{-1}]_{Q/q}$ are uniformly distributed mod Q/q, then multiplied by t => $V_{\delta_0} = V_{\delta_1} = \frac{t^2 Q^2}{12 q^2}$.

$$v_{\text{switch}} = [\tilde{c}_0 + \tilde{c}_1 \cdot s]_q = \frac{q}{Q}[c_0 + c_1 \cdot s]_Q + \frac{q}{Q}(\delta_0 + \delta_1 \cdot s) = \frac{q}{Q} v + \frac{q}{Q}(\delta_0 + \delta$$

Using the canonical norm we get:

$$\|v_{\text{switch}}\|^{\text{can}} \leq \frac{q}{Q}\|v\|^{\text{can}} + \frac{q}{Q}\|\delta_0 + \delta_1 \cdot s\|^{\text{can}} \leq \frac{q}{Q}\|v\|^{\text{can}} + Dt\sqrt{\frac{n}{12}\left(1 + \frac{2}{3}n\right.}$$

We can see that the noise is scaled down by almost q/Q.

Key switching - Base decomposition

In relinearization we deal with $c_0 + c_1 \cdot s + c_2 \cdot s^2$. Here we decompose $c_2 \cdot s^2$ in base T and hide the *hints* about s^2 using RingLWE-like samples involving errors e_i. Remember from key switching technique using base decomposition that we have a relinearization error: $\sum_{i=0}^{\lfloor \log_T q \rfloor} te_i \cdot c_2^{*(i)}$. The variance of any term $te_i \cdot c_2^{*(i)}$ from the sum is $V_{te_i \cdot c_2^{*(i)}}$.

Recall that

- The errors e_i come from a discrete Gaussian distribution => $V_{e_i} = \sigma^2$.
- We can assume that the polynomials in base T behave like random polynomials extracted from R_T => $V_{c_2^*} = T^2/12$.

$$v_{\text{ks}} = \underbrace{c_0^* + c_1^* \cdot s + c_2^* \cdot s^2}_{v_{\text{mul}}} + \sum_{i=0}^{\lfloor \log_T q \rfloor} te_i \cdot c_2^{*(i)}$$

If we use the canonical norm:

$$\begin{aligned}
\|v_{\text{ks}}\|^{\text{can}} &\leq \|v_{\text{mul}}\|^{\text{can}} + D\sqrt{n \log_T(q) V_{t \cdot e_i \cdot c_2^{*(i)}}} \\
&= \|v_{\text{mul}}\|^{\text{can}} + Dt\sqrt{n^2 \log_T(q) V_{e_i} V_{c_2^{*(i)}}} \\
&= \|v_{\text{mul}}\|^{\text{can}} + DtnT\sqrt{\log_T(q) \frac{\sigma^2}{12}}
\end{aligned}$$

Key switching - Modulus switch

In the relinearisation version that employs modulus switching, we have the following error term: $\frac{q}{Q}(tc_2^* \cdot e + \delta_0 + \delta_1 \cdot s)$. Recall that:

- The ciphertext component c_2^* comes from R_q => $V_{c_2^*} = q^2/12$.
- The errors e comes from a discrete Gaussian => $V_e = \sigma^2$.
- The secret s comes from a ternary distribution => $V_s = 2/3$.
- We set $\delta_i = t[-c_i t^{-1}]_{Q/q}$. We assume the terms $[-c_i t^{-1}]_{Q/q}$ are uniformly distributed mod Q/q, then multiplied by t => $V_{\delta_0} = V_{\delta_1} = \frac{t^2 Q^2}{12q^2}$.

$$v_{\text{ks}} = \underbrace{c_0^* + c_1^* \cdot s + c_2^* \cdot s^2}_{v_{\text{mul}}} + \frac{q}{Q}(tc_2^* \cdot e + \delta_0 + \delta_1 \cdot s)$$

The variance of this error term is:

$$\begin{aligned}
V_{\frac{q}{Q}(tc_{2_i}^* \cdot e + \delta_0 + \delta_1 \cdot s)} &= \left(\frac{q}{Q}\right)^2 (t^2 n V_{c_2^*} V_e + V_{\delta_0} + n V_{\delta_1} V_s) \\
&= \left(\frac{q}{Q}\right)^2 \left(t^2 n \frac{q^2}{12}\sigma^2 + \frac{t^2 Q^2}{12q^2} + n\frac{t^2 Q^2}{12q^2}\frac{2}{3}\right) \\
&= \left(\frac{q}{Q}\right)^2 \left(t^2 n \frac{q^2}{12}\sigma^2\right) + \frac{t^2}{12} + n\frac{t^2}{12}\frac{2}{3} \\
&= \left(\frac{q}{Q}\right)^2 \left(t^2 n \frac{q^2}{12}\sigma^2\right) + \frac{t^2}{12}\left(1 + \frac{2}{3}n\right)
\end{aligned}$$

Now we can bound:

$$\begin{aligned}
\|v_{\text{ks}}\|^{\text{can}} &\leq \|v_{\text{mul}}\|^{\text{can}} + D\sqrt{n\left(\frac{q}{Q}\right)^2 \left(t^2 n \frac{q^2}{12}\sigma^2 + \frac{t^2}{12}\left(1 + \frac{2}{3}n\right)\right)} \\
&\leq \|v_{\text{mul}}\|^{\text{can}} + D\frac{q}{Q}t\sqrt{n\left(n\frac{q^2}{12}\sigma^2 + \frac{1}{12}\left(1 + \frac{2}{3}n\right)\right)}
\end{aligned}$$

Code

And now that we have discussed all the ingredients and spices of the BGV encryption scheme, it's time to bake it, sorry, implement it.

We provide a proof of concept implementation, in Python. The aim of the code is to mirror the equations easily. This means that if we see $a \cdot s + e$ in the theory part we prefer to see `a * s + e` in code instead of `polyadd(polymul(a, s), e)`.

> DISCLAIMER: The code should be used for educational purposes only, and never in production.

We represent polynomials using a class: `QuotientRingPoly`. Every polynomial is defined by three attributes:

- `coef: np.ndarray` - Vector of the polynomial's coefficients. `coef[-1]` is the free coefficient, `coef[0]` is the leading coefficient. It's always padded to have n elements.
- `coef_modulus: int` - Modulus of the ring where its coefficients live in.

- `poly_modulus: int | np.ndarray` - Modulus of the polynomial ring where it belongs: `X^n + 1`

`QuotientRingPoly` has the `._reduce(self)` method which reduces any polynomial to an element in the ring.

We overloaded the operators `+, -, *, //, %`:

- Operators work between two `QuotientRingPoly` using the default polynomial operation rules.
- Operators work between a `QuotientRingPoly` and an `int | float` coefficient-wise.

The attributes `coef` and `coef_modulus` can be assigned too. After assigning, the `_reduce()` method will be called. This is helpful to easily change the ring of the polynomial.

Remarks

- To work with big integers, we use the python built in `int` class. To use this with numpy we set `dtype = object` for every array creation.
- We have helper functions `roundv` and `intv` (round vector, int vector) that vectorize the `round` and `int` operations, allowing us to help with rounding and converting vector elements to python's builtin `int` class.

Resources

- BGV paper - Original paper
- BV paper
- FHE standard
- CS15 - for comparisons between FHE schemes
- Ilia Iliashenko thesis - for noise analysis
- Evaluating the effectiveness of heuristic worst-case noise analysis in FHE
- GHS
- Finding and Evaluating Parameters for BGV
- Bitdefender BFV blogpost
- Inferati blogpost

written by Dacian Stroia