Constant-Time Fully Homomorphic Encryption, Decryption & Key Generation

Adrien Bouquet

Spring 2025

1 Introduction

The development of quantum computers presents a significant threat to classical Public-Key cryptographic systems. Many widely deployed cryptographic primitives, such as RSA, Diffie-Hellman, and Elliptic Curve Cryptography (ECC)—rely on the hardness of integer factorization or the discrete logarithm problem. These problems are vulnerable to quantum computers, and therefore enhance research on post-quantum cryptographic scheme.

Plenty of PQC schemes typically rely on lattice-based cryptography because its strong security proofs, efficient implementations, and its ranges of applications such as public-key cryptography, digital signatures, and homomorphic encryption. Fully Homomorphic Encryption (FHE) is a powerful cryptographic primitive that allows arbitrary computation on encrypted data without needing to decrypt it. FHE schemes have based their hardness assumptions on NP-Hard Lattice problem, like Learning With Errors (LWE) and its ring variant (Ring-LWE), both of which are believed to be quantum-resistant.

A FHE scheme is an encryption algorithm based on the property of performing operations on encrypted data while ensuring the result's correctness once decrypted. Such a property allows one (or more) party to encrypt its data and outsources it to an untrusted party (for instance, a cloud server), which will compute the correct result and returns to the sender, without this computing party learning the value of the encrypted data, thus preserving the privacy of the original party.

To achieve this property, this cryptographic primitive leverages a homomorphism between the plaintext domain and the ciphertext domain and respects the following properties:

$$Enc(a + b) = Enc(a) + Enc(b)$$

$$Enc(a \times b) = Enc(a) \times Enc(b)$$

FHE is deployed in system requiring privacy-preserving computation, secure data outsourcing, and applications such as encrypted machine learning, secure voting, and private search. Therefore, there is a strong important interest in ensure that fully homomorphic encryption schemes are secure, cryptographically as well as in their implementations.

In the next section, we will discuss the theoretical background to understand the BGV scheme, an FHE scheme, and present the scheme in itself. Then, we briefly discuss side-channel attacks and the constant-time implementation of BGV as well as the tools used to make this implementation.

2 Preliminaries

2.1 Regev encryption scheme

In [1], Regev introduced the **learning with error (LWE)**, a lattice problem, as well as the cryptographic system resulting from this new problem. The goal is to hide information by using a secret vector and adding noise so that it becomes hard to guess without the secret. The problem is defined as follows:

LWE Definition Let $\mathbf{s} \in \mathbb{Z}_q^n$ a secret vector, $\mathbf{a} \in \mathbb{Z}_q^n$ a vector chosen uniformly at random, $e \leftarrow \chi$ an noise sampled from an error distribution χ (often a discrete Gaussian), and we output the pair $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e \bmod q)$. Given many polynomially pair samples $(\mathbf{a}_i, b_i)$, find the fixed secret vector $\mathbf{s}$

Without the error term e, it would be simple to solve the equation system. However, Regev showed that adding e makes LWE at least as hard to solve as several worst-case lattice problems, such as GapSVP or SIVP. [2]. From this quantum hardness, Regev establishes a new public-key cryptosystem:

- **The private key** is the secret vector $\mathbf{s} \in \mathbb{Z}_q^n$.
- **The public key** is a list of m pairs $(\mathbf{a_i}, b_i)$ where $b_i = \langle \mathbf{a_i}, \mathbf{s} \rangle + e \bmod q$, $a_i \in \mathbb{Z}_q^n$ and $e_i \leftarrow \chi$.
- **To encrypt** a bit x, we choose a random subset S of the m pairs. Then, the encryption of this bit is $(\sum_{i \in S} \mathbf{a_i}, \sum_{i \in S} b_i + \lfloor \frac{q}{2} \rfloor x)$
- **To decrypt** this pair $(\mathbf{a}, b)$, we return 0 if $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is closer to 0 more than $\lfloor \frac{q}{2} \rfloor$ modulo q, or 1 otherwise.

In 2010 [3], Lyubashevsky, Peikert and Regev introduced an algebraic variant *Ring*-LWE, the *ring*-based learning with error problem. From the original paper [3], this new RLWE problem has the same security strength as LWE, while being more efficient. For instance, the polynomial multiplication can be computed efficiently by using the Fast-Fourier Transforms (or variants such as the Number Theoretical Transform) with highly optimized implementations.

LWE had an important impact on post-quantum cryptography and . Many variants of this problem have been introduced and demonstrated, whereas several cryptographic algorithms have been based on this problem (or its variants). For instance, some Public-Key Cryptosystems, such as CRYSTALS-KYBER [4], FHE schemes, as BFV or BGV[5], or Digital Signature schemes, as CRYSTALS-Dilithium, are based on variants of LWE.

Regarding the scope of this paper, we will be focusing primarly on BGV scheme[5].

2.2 BGV encryption scheme

Introduced in [5] by Brakerski, Gentry and Vaikuntanathan, BGV is a Fully Homomorphic encryption scheme based on the R-LWE problem, and is very close to BFV [6] another lattices-based FHE scheme. Compared to BFV, BGV has multiple ciphertext modulus p_i depending on its parameter L (described thereafter). Both algorithms also differ in their message scaling, which prevents the noise from growing to much after executing many operations.

2.2.1 Parameters, Polynomial Rings and Distributions

Regarding BGV's parameters, we define L as the number of prime modulis p_i, $q_l = p_0 \times ... \times p_l$ is the ciphertext modulus corresponding to the encryption level l, where $0 \leq l \leq L$, n as the degree of the polynomials[1], and t the plaintext modulus[2]. From these parameters, the plaintext polynomial ring is defined as $\mathcal{P} = \mathcal{R}_t = \mathbb{Z}_t[x]/(x^n + 1)$ where all coefficients are in $\mathbb{Z}_t$, the ciphertext ring as $\mathcal{C} = \mathcal{R}_{q_l} \times \mathcal{R}_{q_l}$ a pair of polynomials in $\mathcal{R}_{q_l} = \mathbb{Z}_{q_l}[x]/(x^n + 1)$, whose coefficients are in $\mathbb{Z}_q$.

Finally, the protocol relies on three different distributions to sample random polynomials:

- A Uniform ternary distribution over $\mathcal{R}_2 = \{-1, 0, 1\}$[3] mainly for the secret key.
- A Uniform distribution over $\mathcal{R}_q$
- A (truncated) Discrete Gaussian distribution χ for error polynomials, with parameters $\mu = 0$, $\sigma = 3.2$ and $\beta = 19$ according to [7], [8]. β corresponds to the absolute bound of the truncated discrete gaussian i.e., each element x is in $\{-19, ..., 0, ..., 19\}$

2.2.2 Protocol Client's primitive

In the scope of this project, we only considered the client's primitive, i.e. the Key Generation, the Encryption and the Decryption. In fact, we consider that the usage of this BGV client implementation is for instance to encrypt and decrypt data stored on the cloud, without applying any intermediate

[1]For efficiency reasons, n is generally a power of 2.
[2]Generally, $t \ll q_l$
[3]In practice, it will be over $\{q - 1, 0, 1\}$ as secret key's coefficients are also in $\mathbb{Z}_q$

server operation, while offering constant-time computation. These three primitives are similar from Regev's encryption scheme.

The Key Generation takes the client's secret key SK, a random ternary polynomial over $\mathcal{R}_2$, as input and computes the public key $\mathsf{PK} = (\mathsf{PK}_1, \mathsf{PK}_2)$ as follows:

$$\mathsf{PK}_1 = -1(a \cdot \mathsf{PK} + t \cdot e) \bmod q_l$$
$$\mathsf{PK}_2 = a$$

where a is a polynomial sampled uniformly at random over $\mathcal{R}_{q_l}$, and e is a random error polynomial sampled over χ. Here, t is the scaling factor.

The Encryption takes a plaintext $\mathsf{M} \in \mathcal{P}$, and the previously computed public key PK, and outputs the ciphertext $\mathsf{C} = (\mathsf{C}_1, \mathsf{C}_2) \in \mathcal{C}$ as follows:

$$\mathsf{C}_1 = (\mathsf{PK}_1 \cdot u + t \cdot e_1 + \mathsf{M}) \bmod q_l$$
$$\mathsf{C}_2 = (\mathsf{PK}_2 \cdot u + t \cdot e_2) \bmod q_l$$

where u is a ternary polynomial sampled uniformly at random over $\mathcal{R}_2$, and e_1 and e_2 are random error polynomials sampled over χ.

The Decryption returns the original message M by taking the corresponding ciphertext $\mathsf{C} = (\mathsf{C}_1, \mathsf{C}_2)$ and the client's secret key SK in the following way:[4]

$$\mathsf{C}_1 + \mathsf{C}_2 \cdot \mathsf{SK} \bmod t = \mathsf{M} + t \cdot v \bmod t = M \bmod t$$

where v is the noise vector. However, the last equality is correct only if $\|v\|_\infty < \frac{q_l}{2t}$, due to noise growth being to important, thus destroying the message.

Consequently, the security property is based on the hardness for an adversary to recover the plaintext from the ciphertext plus some noise, without knowing the secret key, similarly to LWE.

Remark on $\mathcal{R}_{q_l}$ Previously, we defined all the polynomial's coefficients in $\mathcal{R}_{q_l}$. In practice, we do not multiply each small modulis p_i but rather perform all operation for each modulus individually and send l polynomials. Hence we have an efficient solution, while preserving its correctness. In fact, as the smaller modulis are all primes, and a fortiori pairwise coprime, we can apply the Chinese Remainder Theorem to show that the mapping

$$\mathcal{R}_{q_l} \cong \mathcal{R}_{p_0} \times ... \times \mathcal{R}_{p_l}$$

is a ring isomorphism.

2.3 Side-channel attacks

Although lattices-based cryptography and FHE offer strong cryptographic properties, it is still possible for an adversary to recover the encrypted data. Side-channel attacks can be deployed and targets implementations to leverage physical leakage and exploiting vulnerabilities.

Homomorphic Encryption algorithms can have multiples attack vectors. Single power measurement analysis on BFV Gaussian sampler based on control-flow variations [9], cache-timing attack triggered by a non-constant time Barrett modular multiplication [10], or two single measurement attacks due to ternary value assignment leaking during the Key Generation phase [11] are examples of feasible side-channel attacks on FHE implementations to obtain the secret key with high precision. Therefore, constant-time and formally verified implementations are necessary to ensure the systems' security.

[4]Equation (4) on Inferati's blog post on BGV[8] shows the complete development.

Remark on constant-timeness. When referring to constant-timeness for an implementation or a program, we only consider the constant-timeness of the code, not the hardware. According to definition from Jasmin repository's wiki [12], "a program is said to be constant time when neither the control-flow nor the memory accesses depend on sensitive data". Hence, we assume that hardware operations are correct.

2.4 Project's description

In this project, we implement a BGV scheme in Jasmin, a safe high-assurance and high-speed cryptographic language. We deploy well-known cryptographic efficient or constant-time techniques (Barrett Reduction, NTT, . . .), and we used a Rust wrapper to handle and test the Jasmin implementations. Ensuring and proving constant-timeness and rigorous reasoning require to trust considerably and increase the attack surface.

3 Technical Section

In this section, we detail the Jasmin implementation of our BGV client i.e., the difference with Kyber , our set-up and the implementation of the different primitives we used, as well as the Rust wrapper that will call the assembly extracted from Jasmin to run and test its functionality. Our implementation is based the on MLKEM Jasmin implementation from [13], and therefore will adopt much of their nomenclature and approach, although with some significant differences.

3.1 From Kyber implementation to BGV

Compared to their ML-KEM implementation, BGV requires less cryptographic primitives (KDF, Hash function, . . .), whereas it needs to be more modular than the Kyber . In fact, parameters, such as the modulo q or the polynomial size n, are generally fixed in ML-KEM. For instance, q is often 3329 like for Kyber-512 or Kyber-1024 (c.f. [14]). For BGV, there are small multiple modulis $\{p_0, p_1, ..., p_L\}$, all of which can be modified according to the bit size of the Encryption Level modulo $q_L = p_0 \times ... \times p_L$.

Another difference is the wrapper and testing environment around the Jasmin code. In the original implementation, the Jasmin code is compiled in assembler by the Jasmin compiler, and called at runtime by a C wrapper. For this project, we made the wrapper in Rust to extend the deployment assembly compiled from Jasmin to another language.

On the other hand, plenty of primitives used by the BGV scheme are very similar and already done for Kyber implementation, such as the NTT and INTT, Barrett and Montgomery reduction. Therefore, they needed to be partially adapted to match our implementation. Furthermore, we keep functions and files nomenclature from the original project. In fact, functions having the `export`keyword are meant to be accessible by outside and serves as API for the Jasmin code. Therefore, will be written with a 'j' as the first letter (e.g., `jbgv_encrypt_jazz`); function starting with a `_` (e.g., `_poly_reduce_mod_t`) are functions that performs operation directly on polynomials; and finally the other functions are performing operations on values (e.g., `__barrett_reduce_Q1`). Similarly, if a file contains functions that can be called from outside, its name will begin with a 'j' and end with `.jazz`, while others will only end in `.jinc`.

3.2 Set up

For our project, we aim to make the project work for the parameters $n = 1024 = 2^{10}$, and $log_2(q) = 52$, according to the Homomorphic Encryption Security Standard[7]. However, computing and compiling constants such as the Ψ tables for $n = 1024$ coefficients was too costly for Jasmin compiler. Hence, our default version start at $n = 32$. In addition, we stay with only one modulus (i.e., $L = 1$). As all the constants will be replaced directly in the code assembly by the Jasmin compiler, we have implemented the whole protocol for a modulus q1, and afterwards we could develop the Rust wrapper builder so that it duplicates the Jasmin implementations according to the number of modulis chosen, calculates all the variables depending on each modulus (e.g., the ψ needed for the NTT), and writes them directly to each respective Jasmin project before compiling and using them. We also fixed the plaintext modulus t

to 65537, as described in [15]. In fact, t must be of the form p^r, where p is coprime to $2n$. Hence, we fixed $r = 1$ and $p = 2^{16} + 1$.

In addition, the coefficients are computed on 64-bits unsigned integers. It is necessary to emphasize that the integer representation in $\mathbb{Z}_q$ and in $\{0, ..., 2^{64} - 1\}$ is obviously not the same. Therefore, subtractions and all other overflow/underflow problems must be handled cautiously to maintain the system coherency. Therefore, the Montgomery factor used for its form will be 2^{64}.

Finally, we assume that all the basic operations (addition, multiplication) supported and compiled by the Jasmin compiler are made in constant-time since Jasmin's thread model assume that its generated assembly executed on hardware with constant-time operations as described previously. We also assume that all vectors are in Montgomery form and NTT form upon performing multiplications, additions, or any other operations.

3.3 Basic polynomial operations

Mostly taken from the original implementation, we only have polynomial addition and element-wise multiplication as two basic operations on polynomial. Basically, these two functions performs their operations in $\mathbb{Z}_q$.

3.4 Barrett Reduction

When computing values in $\mathbb{Z}_q$, we need the modulo `%` operation. Even if Jasmin supports the modulo, it's clearly not most efficient and practical operation as it might take a lot of time to compute the result. Hence, we implemented the Barrett reduction algorithm, which reduces an element to its remainder in a field. Although a version exists for ML-KEM, ours is taken from the OpenFHE's project[16], which adapts the algorithm described in [17]. We implemented it to reduce a 128-bit unsigned integer split in two 64-bits integer modulo the ciphertext modulus q, as well as to reduce a 64-bit modulo t our plaintext modulus.

3.5 Montgomery Reduction

Another well known primitive used for its efficiency is the Montgomery reduction [18]. This algorithm allows to speed modular multiplication without modifying the addition. In fact, every coefficient in $\mathbb{Z}_q$ will be represented as $\bar{a} = aR \text{mod} q$, where a is the coefficient in normal form, and R is a radix such that $R \geq q$ and $\gcd(q, R) = 1$. We call $\bar{a}$, the Montgomery form of a. Therefore, addition and multiplication with Montgomery form become:

$$aR + bR = (a + b)R \mod q$$

$$(a \times b)R = (aR \times bR)R^{-1} = (\bar{a} \times \bar{b})R^{-1} \mod q$$

We use Montgomery reduction to speed up algorithm by "allowing efficient implementations of modular multiplication without explicitly carrying out the classical modular reduction step". [17] Chapter 14. As the modulus q will be always odd, we can fix the radix R to be a power of 2. Hence, we choose $R = 2^{64}$.

In `bgvpoly.jinc`, we created the methods `_poly_from_mont` and `_poly_to_mont`, which transforms a polynomial's coefficients from their Montgomery form to their normal form, and vice-versa. Furthermore, we implemented the method `__fqmul` in `bgvreduce.jinc`, which takes two unsigned 64-bits integer in Montgomery form and returns their product in $\mathbb{Z}_q$ according to the multiplication described above.

3.6 NTT & INTT

Multiplying two polynomials in the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ is equivalent to the convolution of the coefficients reduced modulo q, and then performing a long division on the new polynomial of degree $\leq 2n$ by $X^n + 1$. Computing this convolution and this division is costly and clearly not constant-time depending on the input polynomials. To optimize and reduce the cost of the operations, many encryption schemes rely on the Number Theoretic Transform [19], or NTT, (and a fortiori, its inverse) to represent their polynomials into vectors. Indeed, two polynomial in NTT form can be multiplied element-wise, reduced modulo q, and after retransforming the computed into its normal form after applying the Inverse NTT, we obtain the same results as performing the convolution and the long division. Furthermore, the NTT also satisfies the addition and the scalar multiplication, thus keeping the coherency for polynomial addition and Montgomery form.

Similarly to other implementations, we base the NTT on the Cooley-Turkey (CT) butterfly algorithm, and the INTT on Gentleman-Sande (GS) butterfly algorithm from [20] (Algorithm 1 & 2). Both algorithms are optimized variants of these two transforms. The NTT make use of the precomputed table $\Psi \in \mathbb{Z}_q^n$, which stores powers of ψ, a primitive $2n$-th root of unity in $\mathbb{Z}_q$ such that $\psi^n \equiv -1 \bmod q$, while the INTT make use of the precomputed table $\Psi^{-1} \in \mathbb{Z}_q^n$, which stores powers of ψ^{-1}, the multiplicative inverse in $\mathbb{Z}_q$. To speed up the computation, these tables are precomputed by the Rust wrapper and hard-coded in file `psis.jinc` before compiling the code[5]. The Rust wrapper NTT implementation (c.f. `helpers/ntt.rs`), and Ψ computation (in Montgomery form) are adapted from the Lattirust library, maintained by Christian Knabenhans [21].

3.7 Sampling random polynomials

Sampling random polynomials according to multiple distributions is a key primitive of lattice encryption schemes such as BGV or Kyber . The main challenge is preserving the distribution correctness from bytes randomly generated while sampling the coefficient in constant-time. In our BGV implementation based on [8], we consider the three 3 types of distribution in previously mentioned.

All the functions that samples random polynomials according to the three distributions mentioned above are implemented in `src/jasmin/sampler.jinc`. In the following subsection, we described how we handled the Jasmin Syscall generating random bytes, and the reasoning behind each sampling implementations.

3.7.1 Jasmin Syscall

To generates random bytes in Jasmin, there exist the Syscall `#randombytes`, which takes a `reg ptr` (i.e., a pointer morally [12] Section Arrays). However, the Syscall must be handled by the user according to the project's wiki. In our project, we implemented the function `__jasmin_syscall_randombytes__` in Rust (c.f. `main.rs`) taking a raw pointer on unsigned bytes and a length, and returning the same pointer with each byte randomly generated.[6] Because the Syscall's function body is not considered by Jasmin in its threat model, we assume that its in constant time. In practice, we must verify that its implementation is constant-time.

3.7.2 Uniform Distribution Sampling

To ensure uniform distribution sampling, we adopted two methods. For ternary polynomials, we proceed by rejection sampling (c.f. `__sample_random_poly_ternary`). For each coefficient, we generate one byte, and we keep only the last two bits. If these two bits equals 3, we redraw the byte. Otherwise, we subtract one. Hence, we prevent the secret key from being biased because the value 0 and 3 are congruent modulo 3, and will increase the probability of having -1 for each coefficient of the secret key.

For polynomials over $\mathcal{R}_q$ (c.f. `__sample_random_poly_mod_q`), we avoid rejection sampling because it will be clearly inefficient. Instead, we generate randomly $n + 2$ 64-bits unsigned integer. Then, we remove the two extras 64-bits integer, and we apply the modulo on the n integer remaining. In fact, adding the two extra integers reduces the bias probability, and hence we can approximate the uniform distribution as closely as possible.

3.7.3 Gaussian Distribution Sampling

To sample values following a discrete Gaussian distribution, we adapted the `DGS_DISC_GAUSS_UNIFORM_TABLE` from [22]. This sampler uses a precomputed table of probabilities ρ, where the number of entries

[5]In our single modulus implementation, the values are hardcoded manually in the file.

[6]Due to macOS adding an extra '_', we implemented the function a second time starting with only one '_' instead of two.

corresponds to the size of the set covered by of the Gaussian distribution (i.e. $2 * \beta + 1$). In our situation, there are 39 entries from 0 to 38[7]. Each entry's probability is computed as follows:

$$\rho_x = \exp^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad -\beta \leq x \leq \beta$$

As they are probabilities, each ρ_x are in $[0, 1]$. However, Jasmin does not support Floating-point value, and hence, we cannot store the table as it is. By multiplying each value by 2^{64} and rounding them, we keep the same distribution while using unsigned 64-bit integer as comparison. Therefore, we can precompute the table with the Rust wrapper, and we can directly write it in the Jasmin code (c.f. `rhos.jinc`).

To generate our coefficients with respect to the Gaussian distribution, we then proceed as follows (c.f. `__sample_gaussian_distribution`):

- We generate a random byte x, we keep only the six last bit and if the value is greater than 39, we redraw.
- We generate a random 64-bits unsigned integer y.
- If y is smaller than ρ_x, we keep x and subtract β to obtain the real corresponding offset. The result will be our coefficient. Otherwise, we restart at the first step.

We continue this procedure until we obtain a valid coefficient, and we repeat for each coefficient of the vector.

3.8 Client's protocol

Similarly to Inferati's post [8] and according to the BGV's protocol, we split in 3 key function in the `jbgv.jazz`:

- `jbgv_keygen_secret_key` generates the client's secret key according to the ternary distribution.
- `jbgv_encrypt_jazz` takes the client's secret key and a plaintext, and returns the corresponding ciphertext.
- `jbgv_decrypt_jazz`takes the client's secret key, and a ciphertext, and returns the corresponding plaintext.

We choose to generate a random public key for each new plaintexts, and we don't store the public key, only the secret, because we don't need it to decrypt the ciphertext.

To compile the Jasmin code with the Jasmin compiler, go in `src/jasmin/` and run in terminal:

```
> jasminc jbgv.jazz -o jbgv.s
```

To run an example of the protocol, run:

```
> cargo main
```

To run all the test, run:

```
> cargo test -- --include-ignored
```

4 Result

4.0.1 Methodology

At first, we decided to keep the wrapper in C as in the Kyber repository, but in the end, we adopted Rust as our wrapper. To call the assembly compiled from Jasmin from Rust, we use a `build.rs` script, which builds the Rust project each time `jbgv.s`has been modified, and compile such that it can be used as a library by the Rust project.

In `src/helpers/`, there are all the Rust files implementing either cryptographic functions ((I)NTT, Key generation, Encryption, etc), or functions that (pre)compute constant modulus, $2n$th-root of unity,

[7]Negative index are not possible, thus we add $+\beta$ to all the index.

ρ_x, ... To ensure an "informal" functionnal correctness, we implemented the wrapper and the Jasmin project in parallel, and created a test suite in two parts:

- in `bgvpoly.rs`, there are all tests on function that performing operations on polynomials (e.g., addition, ntt, sampling, etc).
- in `bgv.rs`, there are all the tests on the protocol's functions (Key Generation, Encryption, etc)

Finally, the parameters can be found in `src/helpers/config.rs` for Rust, and in `src/jasmin/{params.jinc, psis.jinc, rhos.jinc}` for Jasmin.

4.0.2 Benchmark

Time Benchmark. We made some benchmarks to evaluates the efficiency of the Jasmin implementation compared to Rust. For performing 2000000 NTT in a row, Jasmin has an overall mean of 5.096μs, compared with an overall mean of 11.801μs for Rust. Similarly with INTT, Jasmin took 5.319μs, compared with an overall mean of 14.633μs for Rust.

Gaussian Sampler accuracy. To verify our sampler accuracy, we sampled 200000 polynomials of $n = 32$, we classified them and we computed the estimated mean and standard deviation. We obtain an estimated mean μ' of 0.0022, and an estimated standard deviation σ' of 3.1985.

4.0.3 Jasmin Takeaway

Jasmin is a very interesting language based on a lot of good concepts. The possibility of having a very powerful cryptographic language that allows you to export its functional implementation to a formal verification tool, such as Easycrypt, is practical and powerful. However, the lack of documentation or tutorials was limiting to understand and exploit all the features and advantages of the language. Another example is the error returned by the compiler, which sometimes lacks clear explaination. These projects reflect the complexity of academic research, for which documentation and the scope of development are limited.

Nevertheless, we did not entirely reached the goal of the project, as we were not able to formally verify the constant-timeness of our implementation by extracting the proof from our Jasmin program and running Easycrypt.

5 Future Work

5.0.1 Easycrypt extraction and proving constant-timeness

Afterwards, our intention is to use the Easycrypt framework, by extracting our Jasmin implementation, which can be done natively by Jasmin, to prove its cryptographic security and constant-timeness.

5.0.2 Rust Wrapper extension

In future development, we want to develop the Rust build script to be able to compile all the Jasmin code, as well as handling multiple modulus by duplicates Jasmin projects for each modulus and its respective constants, write in Jasmin and Rust file. The objective would be to have a completely modular program with a default configuration.

6 Conclusion

This project presented a constant-time client-side implementation of the BGV Fully Homomorphic Encryption scheme, targeting both cryptographic soundness and implementation-level security. Built on post-quantum lattice assumptions, FHE ensures privacy-preserving computation, making it a compelling solution in a future shaped by quantum threats. By leveraging the Jasmin language for secure and verifiable low-level programming, we demonstrated the feasibility of implementing BGV primitives in a constant-time manner, while also addressing challenges such as polynomial arithmetic, modular reduction, and secure sampling.

Although full formal verification via EasyCrypt remains future work, our results highlight the practicality and performance benefits of carefully engineered constant-time implementations. As the need for secure data processing continues to grow—particularly in areas like cloud computing and machine learning—FHE, and efficient, secure implementations like the one developed in this project, will be key to building strong, quantum-resistant privacy solutions.

7 References

[1] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the thirty-seventh annual ACM symposium on theory of computing*, in STOC '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 84–93. doi: 10.1145/1060590.1060603.

[2] C. Peikert, "A decade of lattice cryptography." Cryptology ePrint Archive, Paper 2015/939, 2015. Available: https://eprint.iacr.org/2015/939

[3] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, Nov. 2013, doi: 10.1145/2535925.

[4] J. Bos *et al.*, "CRYSTALS – kyber: A CCA-secure module-lattice-based KEM." Cryptology ePrint Archive, Paper 2017/634, 2017. doi: 10.1109/EuroSP.2018.00032.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping." Cryptology ePrint Archive, Paper 2011/277, 2011. Available: https://eprint.iacr.org/2011/277

[6] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Paper 2012/144, 2012. Available: https://eprint.iacr.org/2012/144

[7] M. Albrecht *et al.*, "Homomorphic encryption security standard," HomomorphicEncryption.org; HomomorphicEncryption.org, Toronto, Canada, 2018.

[8] Inferati, "Understanding fully homomorphic encryption (FHE) schemes: BGV." https://www.inferati.com/blog/fhe-schemes-bgv, 2025.

[9] F. Aydin, E. Karabulut, S. Potluri, E. Alkim, and A. Aysu, "RevEAL: Single-trace side-channel leakage of the SEAL homomorphic encryption library," in *2022 design, automation & test in europe conference & exhibition (DATE)*, 2022, pp. 1527–1532. doi: 10.23919/DATE54114.2022.9774724.

[10] W. Cheng, J.-L. Danger, S. Guilley, F. Huang, A. Bel Korchi, and O. Rioul, "Cache-Timing Attack on the SEAL Homomorphic Encryption Library," in *11th International Workshop on Security Proofs for Embedded Systems (PROOFS 2022)*, Leuven, Belgium, Sep. 2022. Available: https://telecom-paris.hal.science/hal-03780506

[11] F. Aydin and A. Aysu, "Leaking secrets in homomorphic encryption with side-channel attacks." Cryptology ePrint Archive, Paper 2023/1128, 2023. doi: 10.21203/rs.3.rs-3097727/v1.

[12] "Jasmin project wiki." 2025. Available: https://github.com/jasmin-lang/jasmin/wiki/

[13] J. B. Almeida *et al.*, "Formally verifying kyber episode v: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt." Cryptology ePrint Archive, Paper 2024/843, 2024. Available: https://eprint.iacr.org/2024/843

[14] R. Avanzi *et al.*, "CRYSTALS-kyber algorithm specifications and supporting documentation," 2017. Available: https://api.semanticscholar.org/CorpusID:198992527

[15] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields." Cryptology ePrint Archive, Paper 2021/204, 2021. Available: https://eprint.iacr.org/2021/204

[16] A. A. Badawi *et al.*, "OpenFHE: Open-source fully homomorphic encryption library." Cryptology ePrint Archive, Paper 2022/915, 2022. Available: https://eprint.iacr.org/2022/915

[17] A. J. Menezes, J. Katz, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. in Discrete mathematics and its applications. CRC Press, 1996. Available: https://books.google.ch/books?id=MhvcBQAAQBAJ

[18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985, Accessed: Jun. 05, 2025. [Online]. Available: http://www.jstor.org/stable/2007970

[19] A. Satriawan, R. Mareta, and H. Lee, "A complete beginner guide to the number theoretic transform (NTT)." Cryptology ePrint Archive, Paper 2024/585, 2024. doi: 10.1109/ACCESS.2023.3294446.

[20] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography." Cryptology ePrint Archive, Paper 2016/504, 2016. Available: https://eprint.iacr.org/2016/504

[21] C. Knabenhans, "Lattirust." https://github.com/cknabs/lattirust; GitHub, 2025.

[22] M. R. Albrecht and M. Walter, "dgs, Discrete Gaussians over the Integers," 2018. Available: https://bitbucket.org/malb/dgs