

TDK-dolgozat

Korpa Péter Zsolt,
Tot Bagi Márton

ELTE - Eötvös Lóránd University
Informatikai Kar, Komputeralgebra Tanszék

Szerzők: Korpa Péter Zsolt - 2021/21/1,
Tot Bagi Márton - 2022/23/1

Témavezető: Kutas Péter, egyetemi adjunktus, Phd

Pályamunka címe:

Implementation of a Dark Market algorithm in Fully
Homomorphic Encryption

Implementation of a Dark Market algorithm in Fully Homomorphic Encryption

Tot Bagi, Márton
mtotbagi@gmail.com

Korpa, Péter Zsolt
korpa.peter0723@gmail.com

ELTE - Eötvös Lóránd University

Abstract

Fully Homomorphic Encryption (FHE) has long been a "holy grail" among cryptographers, but only been realized by Gentry in 2009. FHE allows for operations on encrypted data while being secure, even against quantum computers. Potential applications of FHE include privacy-preserving machine learning, secure cloud computing and e-voting. The original scheme, proposed by Gentry, was too slow for any real-world application. Since then there were multiple schemes proposed, with greatly improved performances, one of which is Fully Homomorphic Encryption over the Torus (TFHE). Using Zama's TFHE implementation in rust, the TFHE-rs library, we show that its possible to achieve good performance with FHE libraries. In particular, we provide an implementation for one of the challenge problems issued by Zama and achieve competitive running times.

Supported by the ÚNKP-22-1 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund



1 Introduction

Fully homomorphic encryption enables one to perform computations on encrypted data. It is a powerful tool for many applications ranging from secure cloud computing to privacy-preserving machine learning. For a long-time it was considered the holy grail of cryptography as only schemes with simpler homomorphic property (supporting only addition and not multiplication) were known. Well-known examples of partially homomorphic schemes are Paillier’s encryption scheme [23] and El Gamal [9]. These are useful for certain applications (e.g., electronic voting) but apply to a much more restricted set of cryptographic problems. In 2009 Gentry proposed the first fully homomorphic encryption scheme based on ideal lattices [11]. The result was revolutionary from many different aspects. First it provided a solution to a long-standing open problem (for which Gentry was awarded the prestigious Gödel prize in 2022). Second it relied on an assumption far from the usual assumptions of discrete log and factoring, namely on finding short vectors in structured lattices. Interestingly, this provides potential post-quantum security for these kind of schemes which was not an initial requirement. Furthermore, even if quantum computers never get built, lattice-based cryptography shows applicability in this context. The main idea of Gentry was to start with a scheme that is somewhat homomorphic, i.e., can support a bounded number of operations and turn it into a scheme that can support an unconditional number of homomorphic operations with a procedure called bootstrapping. Gentry’s original construction was very slow (mainly because bootstrapping was slow) and it wasn’t clear whether it could evolve into something practical.

Since Gentry’s original result the area of fully homomorphic encryption became an active area of research. Important schemes include [2] [14] [7] [3] [4]. A breakthrough result came in 2018 with the development of TFHE [4] which sped up bootstrapping considerably. Zama is an emerging FHE startup that built two open-source FHE libraries from the TFHE construction called Concrete and TFHE-rs (see here). They also announced certain implementation challenges that people can try to solve using said libraries. Each challenge has an associated bounty, the higher the prize the more difficult the problem.

Our contributions We worked on a challenge that asks to implement Algorithm 3 and its goal set out was to achieve a performance of 20 seconds. We were able to reach a runtime of 110 seconds with a parallelized version of the algorithm. This challenge had a reward of 5000€, which made it an expert level bounty. See our implementation here

Structure In section 2 we give an overview of the bases of lattice based cryptography. Then in section 3 we explain what fully homomorphic encryption is and what its current state is. Then we dive deeper into a concrete FHE scheme TFHE, which we used in our implementation. Then we quickly introduce MPC and the algorithm that we implemented in section 6. We explain how we optimized on this algorithm, then in section 7 we present the runtimes we achieved.

2 Lattices, lattice based cryptography

Lattices have become increasingly popular in the last decades because you can build efficient and secure cryptographic systems from them. In this section we will take a look at what lattices are, their properties and the basic algorithmic problems. We also discuss what kind of cryptosystems one can build from them.

To get an idea what lattices are, we will look at $\mathbb{R}^n$, as an n -dimensional vector space. If we take all members of $\mathbb{R}^n$ such that they only have components (coordinates) in $\mathbb{Z}$ then we get $\mathbb{Z}^n$ which is a lattice. So lattices are a form of discrete vector spaces.

Definition 2.1. For a basis $B = \{b_1, b_2, \dots, b_n\}$ in $\mathbb{R}^n$ the lattice $\mathcal{L}$ in $\mathbb{Z}^n$ generated by B is:

$$\mathcal{L} = \left\{ \sum_{i=1}^n a_i b_i \mid a_i \in \mathbb{Z} \right\}.$$

We call the elements of $\mathcal{L}$ lattice points or lattice vectors interchangeably. Thus informally lattices are just finitely generated abelian groups embedded into $\mathbb{R}^n$ which provides a euclidean distance notion.

We will mostly be looking at lattices in $\mathbb{R}^n$ or $\mathbb{Z}^n$.

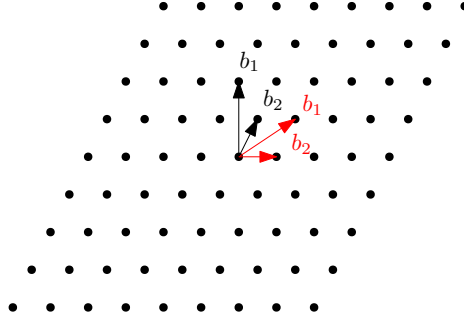


Fig. 1. Example lattice generated by two different b_1, b_2

2.1 Properties of lattices

If we write the basis vectors of $\mathcal{L}$ next to each other then we get a matrix $B \in \mathbb{R}^{n \times n}$ which we can use to denote the basis which generated $\mathcal{L}$. We then define the *fundamental region* of $\mathcal{L}$.

Definition 2.2. The *fundamental region* of a lattice $\mathcal{L}$ generated by a basis B is the parallelepiped formed by the column space of B :

$$fr(\mathcal{L}) = \{Bx \mid x \in [0, 1)^d\}$$

The volume of $fr(\mathcal{L})$ is :

$$vol(fr(\mathcal{L})) = \sqrt{\det B^T B}$$

also denoted with $vol(\mathcal{L})$.

Remark 2.3. This definition also covers the cases of lattices which are not full lattices (i.e., generated by less elements than the dimension of the underlying vector space). In the case of full lattices the volume is just the absolute value of $\det(B)$.

Multiple basis can generate the same lattice but the volume of the fundamental region is uniquely determined by $\mathcal{L}$ (because two matrices generate the same lattice if and only if they differ by an integral matrix of determinant 1 or -1). We will refer to this invariant as the determinant of the lattice and denote it by $\det(\mathcal{L})$. We are going to look at the notion of multiple bases and why they are important in subsection 2.2.

An important property of a lattice is its shortest vector which was introduced in 1896 by Minkowski. . The shortest-vector is the first successive minima which is:

Definition 2.4. The i -th successive minima of $\mathcal{L}$, $\lambda_i(\mathcal{L})$ is defined to be the smallest r such that the n -dimensional ball with radius r contains exactly i linearly independent vectors.

Definition 2.5. $\lambda_1(\mathcal{L})$ is the smallest vector of the lattice $\mathcal{L}$.

The following theorem of Minkowski is useful to provide an upper bound on the size of the shortest vector of the lattice.

Theorem 2.6 (Minkowski). Let C be a centrally symmetric convex body and let $\mathcal{L}$ be a lattice in $\mathbb{R}^n$. If $vol(C) > 2^n \det(\mathcal{L})$, then C contains a lattice point other than the origin.

In order to apply our results to bound the shortest vector of a lattice in terms of $\det(\mathcal{L})$ and the dimension we can take C to be an appropriately sized ball centered around the origin.

Corollary 2.7. $\lambda_1(\mathcal{L}) \leq \sqrt{n} \det(\mathcal{L})^{1/n}$.

No bound is known for higher successive minima, but there is an upper bound for product of the first n successive minima [17] .

One would think that obtaining short vectors is easy as one just looks for linear combinations with small coefficients. This is not the case and in fact is the property why we have 'good' and 'bad' bases.

2.2 Good and bad bases

As we saw in the previous section, multiple bases can generate the same lattice and due to the problems we would like to solve in lattices we can find 'good' or 'bad' bases to work in. The motivation behind working in good bases can be thought of as working with vector spaces with orthonormal bases. Many lattice problems can be solved easily in good bases but become hard in bad ones. The 'goodness' and 'badness' of a basis is close to the notion of orthogonality, we call a basis for $\mathcal{L}$ 'good' if

$$\prod_i \frac{\|b_i\|_*}{\text{vol}(fr(L))} \quad (1)$$

is not much bigger than 1. So $\prod_i \|b_i\|_*$ is not much bigger than $\text{vol}(fr(L))$ in the $\|\cdot\|_2$ norm. They are only equal if the basis is orthogonal. One can also say that this means that if we take a small vector in the lattice then the coordinates of the vector in that basis are small. So

$$v_i \leq c \frac{\|v\|_*}{\|b_i\|_*} \quad (2)$$

for some small constant c .

There exist algorithms that given a basis (be it a bad one) for a lattice calculates a better basis for it, such as the LLL-algorithm [19]. We will look at LLL in subsection 2.3.

One can also take the HNF (Hermite normal form) of a basis B , call it B' , then B' will uniquely determine $\mathcal{L}$, just as the fundamental region, and is efficiently computable. The HFN of B is a good choice for a canonical basis of $\mathcal{L}$. It turns out that it is actually the worst basis for $\mathcal{L}$ (in some quantifiable sense) which can be used when creating cryptographic primitives from lattice problems.

Many hard problems can be stated in lattices which are also easy to understand but very hard to solve algorithmically, we will take a look at the most used ones.

2.3 Lattice problems: SVP, CVP

The two main problems we will look at are SVP, the shortest vector problem and CVP, the closest vector problem. In both we are given a basis B , mostly in HNF, for a lattice $\mathcal{L}$ and a norm $\|\cdot\|_*$.

Definition 2.8. Search-SVP

Given a basis of a lattice and a norm, what is the smallest vector $0 \neq v \in \mathcal{L}$ so that $\forall x \in \mathcal{L} : \|v\|_ \leq \|x\|_*$.*

SVP asks us to find $\lambda_1(\mathcal{L})$. One can also ask, given $r > 0$, if there exists $v \in \mathcal{L}$ such that $0 < \|v\|_* < r$, this problem is called the *Decision-SVP*.

Definition 2.9. γ approximate-SVP

Given a basis, and an approximation factor γ and a norm, find $0 \neq v \in \mathcal{L}$ such that $\forall x \in \mathcal{L} : \|v\|_* \leq \gamma \|x\|_*$.

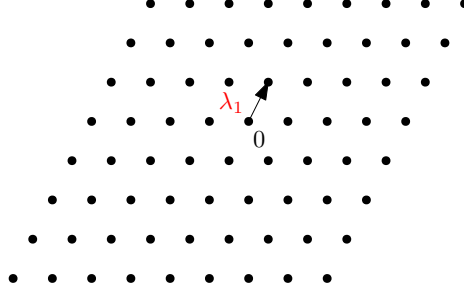


Fig. 2. The smallest vector of a lattice in two dimensions

CVP on the other hand asks, given a vector, to find a vector in the lattice closest to it. While SVP might seem easy CVP already seems harder because the input vector can be outside of our lattice. We will see that SVP and CVP are both somewhat equally hard problems.

Definition 2.10. Search-CVP

Given a basis, a target vector t and a norm, find $0 \neq v \in \mathcal{L}$ such that $\forall x \in \mathcal{L} : \|v - t\|_* \leq \|x - t\|_*$.

Definition 2.11. γ approximate-CVP

Given a basis, a target vector t and an approximation factor γ and a norm, find $0 \neq v \in \mathcal{L}$ such that $\forall x \in \mathcal{L} : \|v - t\|_* \leq \gamma \|x - t\|_*$.

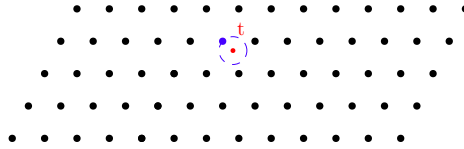


Fig. 3. The closest vector to t in a lattice

CVP also has a decision version, and both problems have close relatives which are similar, all these problems and how they reduce to each other can be seen in [26]. It can also be shown that SVP reduces to CVP, so they indeed have the same complexity.

Solving SVP/CVP for lattices with large enough dimensions gets pretty hard. In practice we would use from $n = 500$ to $n = 1000$, that is because $n \approx 180$ is

already breakable in months [8]. Solving the approximate versions are also hard when the dimension is large and the approximation factor is small. SVP/CVP are known to be NP-hard problems thus are hard classically and quantumly as well (one does not expect that quantum computer can solve NP-hard problems) [1][6]. That is if the dimension is large, when its small, for example in 2 dimensions, the problem is solvable in linear time with Gauß's algorithm. We will take a look at this in subsection 2.4. For cryptography, the problems we use have to be hard on the average case, while SVP/CVP are hard for the worst case. We will look at two such problems, The Shortest Integer Solution (SIS) and the Learning with Error problem (LWE).

Definition 2.12. $SIS_{n,m,\beta,q}$

Given $\beta, q \in \mathbb{Z}$, let $A \leftarrow \mathbb{Z}_q^{n \times m}$. Find $0 \neq x \in \mathbb{Z}^m$ such that:

$$\|x\| \leq \beta$$

and

$$Ax = 0 \pmod{q}$$

where $\beta < q$, $m \geq n \log q$

Lattice based cryptography began with SIS by works from Miklós Ajtai [1]. As we previously said, SIS is hard on the average case, so solving a random instance of SIS is bounded by 2 SVP instances, one of dimension n and one of m . This can be shown by reducing SIS into γ -SIVP [1]. . SIS also has a Ring analog, called Ring-SIS [21].

Definition 2.13. *LWE*

Given $q, B \in \mathbb{Z}$ so that $1 \leq q \ll B$, let $A \leftarrow \mathbb{Z}_q^{n \times m}$ and $s, e \leftarrow \{-B, \dots, B\}^n$.

Given A and $b := As + e \pmod{q}$, find s or e .

LWE can also be reduced to γ -SIVP so its also good for cryptography.

These problems are hard in the average case so we can build cryptographic primitives on them but first we will look at reduction algorithms which we can use to solve SVP/CVP.

2.4 Reduction algorithms

Given a bad basis of a lattice we can try to get a basis which is shorter and more orthogonal. We will look at the 2 dimensional example with Gauß's algorithm and LLL, for higher dimensions and approximations. Other algorithms also exist such as BKZ.

Gauß's algorithm for lattices in 2 dimensions

Definition 2.14. *Gaussian Lattice Reduction*

Let $\mathcal{L} \subset \mathbb{R}^2$ be a lattice with basis v_1, v_2 .

Algorithm Definition 2.14 works the same way the Gaussian greatest common divisor algorithms. It provably terminates in polynomial time and v_1 will be the shortest vector in $\mathcal{L}$. Actually we can solve SVP up to 4 dimensions [22].

Algorithm 1: Lagrange-Gaußreduction

input: v_1, v_2 a basis of $\mathcal{L}$ and $\|\cdot\|_*$

```

1 while  $m \neq 0$  do
2   if  $\|v_2\|_* < \|v_1\|_*$  then
3      $\text{swap } v_1 \text{ and } v_2$ 
4   end
5    $m := \lfloor v_1 \cdot v_2 / \|v_1\|_*^2 \rfloor$ 
6    $v_2 := v_2 - mv_1$ 
7 end
8 return  $v_1, v_2$ 
```

LLL The natural way to generalize lattice reduction to higher dimensions is to take two vectors at a time and reduce them using Gauß's algorithm. The issue with this approach is that although it terminates with the shortest vectors, it is unbearably slow even for small dimensions. The idea of LLL [19] is to only focus on b_i and b_{i+1} at a time and only reduce them if they are somehow far from reduced in the two dimensional sense, i.e., when $\|\overline{b_i}\| > (\gamma_2 + \epsilon)\|\overline{b_{i+1}}\|$ where γ_2 is the 2-dimensional Hermite constant and ϵ is some fixed small number chosen by the algorithm (typically it chosen in a way that $\gamma_2 + \epsilon = 2$). The choice of ϵ provides a trade-off between running time and output quality. The key observation of [19] was that by introducing this small ϵ there as an associated quantity (called the potential) that decreases by a factor $\frac{\gamma_2 + \epsilon}{\gamma_2}$ every iteration thus guaranteeing that the algorithm terminates in polynomial time. Next we describe the steps of the algorithm.

Definition 2.15. *LLL Lattice Reduction* [16]

Algorithm 2:

input: $B = \{b_1, b_2, \dots, b_n\}$ a basis of $\mathcal{L}$, $\|\cdot\|_*$, $\delta \in (\frac{1}{4}, 1)$

- 1 Compute into B^* the Gram-Schmidt orthogonalization of B (without normalization)
- 2 $\mu_{i,j} \leftarrow (b_i \cdot b_j^*) / (b_j^* \cdot b_j^*)$
- 3 $k \leftarrow 2$
- 4 **while** $k \leq n$ **do**
- 5 **for** j **from** $k-1$ **to** 1 **do**
- 6 **if** $|\mu_{k,j}| > 1/2$ **then**
- 7 $b_k \leftarrow b_k - \lfloor \mu_{k,j} \rfloor b_j$
- 8 Update B^* and the $\mu_{i,j}$'s
- 9 Compute into B^* the Gram-Schmidt orthogonalization of B
- 10 **end**
- 11 **end**
- 12 **if** $b_k^* \cdot b_k^* > (\delta - \mu_{k,j-1}^2) b_{k-1}^* \cdot b_{k-1}^*$ **then**
- 13 $k \leftarrow k + 1$
- 14 **else**
- 15 Swap b_k and b_{k-1}
- 16 Update B^* and the $\mu_{i,j}$'s
- 17 $k \leftarrow \max\{k-1, 2\}$
- 18 **end**
- 19 **end**
- 20 **return** B

For δ the most commonly used value is $3/4$. LLL solves γ -SVP, so the vectors are not the successive minima, but are upper bounded by constants $c_i > 1$, see Theorem 17.2.12 in [10].

We note that there are also other algorithms for SVP/CVP such as the Sieving algorithm which solves exact SVP in n dimension [18].

2.5 LWE, RLWE schemes

Many cryptographic schemes can be built on top of lattice problems such as public key encryption, trapdoors, signatures. There are also specific schemes that we can only build from lattices. We will look at how we can build public key encryption on LWE [20].

Let $B \in \mathbb{Z}$, $\chi_B := \{-B, \dots, B\}$. We have as parameters $n, q \in \mathbb{Z} > 0$ and χ_B .

KeyGen: $A \leftarrow \mathbb{Z}_q^{n \times m}$ and $s, e \leftarrow \chi_B^n$. Then the public key $pk = (A, b := As + e \bmod q)$ and the secret key $sk = s$.

Encrypt: Given a message bit $m \in \{0, 1\}$, $\tilde{s}, \tilde{e} \leftarrow \chi_B^n$ and $e' \leftarrow \chi_B$. Then the encrypted message $c = (\tilde{s}^T A + \tilde{e}^T \bmod q, \tilde{s}^T b + e' + m \cdot \lfloor q/2 \rfloor \bmod q)$.

Decrypt: Given the message as $c = (c_1, c_2)$ compute $x = c_1^T s - c_2 \bmod q$. If $x \in [q/4, 3q/4]$ then return 1 else 0.

If $q \geq 8n \cdot B^2 + 4B$ then the scheme is correct [13]. IND-CPA security follows from the hardness of decisional LWE. .

3 FHE

In this section we explore what FHE is, and look at Gentry's idea of bootstrapping.

3.1 Homomorphism in cryptography

The idea of fully homomorphic encryption was first proposed by Rivest, Adleman, and Dertouzos in [24]. Homomorphic encryption allows us to perform computations on encrypted data. This means if we have a public value m and the encryption $Enc()$ takes m to same $Enc(m)$ then for any efficiently computable function f we have that $Enc(f(m)) = f(Enc(m))$. Of course we can take f to be multivariate. FHE was long sought-after without any major progress until in 2009 Gentry showed a FHE scheme built on Ideal lattices [11]. Today FHE is a growing field with many different efficient instantiations. For a great survey on the history of FHE schemes the reader is referred to [27] and [15].

In general we call a public key encryption scheme $\mathcal{E}$ with procedures:

KeyGen: Given some security parameter λ (there can be additional parameters) generates (sk, pk) pair.

Encrypt: Given a message m and the public key pk , outputs a ciphertext c .

Decrypt: Given a ciphertext c and the private key sk outputs the plaintext.

Evaluate: Given the public key, a circuit Π and a vector of ciphertexts $\vec{c}$ with as many elements as many inputs Π has, outputs $\Pi(\vec{c})$.

fully homomorphic if its 'correct' for all Boolean circuits. In layman's terms correctness means that we can correctly decrypt every ciphertext, freshly encrypted ciphertexts and even ones that have been output by Evaluate. If the scheme is only correct for circuits up to some depth τ then we are talking about a Leveled or Somewhat homomorphic scheme.

3.2 Properties of homomorphic schemes

There are several properties we want a FHE scheme to have so that it can be useful and to rule out trivial constructions. One such trivial construction can be that Evaluate returns a ciphertext which trivially encodes (for example in a ABS) what operations have been computed on a ciphertext, meaning that each Evaluate appends what it should evaluate on the ciphertext. Then Decrypt decrypts the values encoded in the ciphertext and then calculates the operations on the plain values. This is bad for several reasons and now we will look at properties that prohibit this and many other behaviors we don't want, and enforce ones we would like.

Strong homomorphism: It requires that the fresh ciphertext should not be distinguishable from the ones that Evaluate puts out.

Compactness: Compactness requires that the size of ciphertexts output by Evaluate should not depend on the depth of the circuit that was evaluated, but only on some polynomial function of the security parameter λ .

Circuit privacy: It means that the ciphertext output by Evaluate should not reveal any information about the circuit that was evaluated.

There are also other properties a FHE can have, for example we can consider if we don't have the strong homomorphism property, then how many times can we evaluate on a given ciphertext. This idea is explored for example in [12].

3.3 Gentry's original scheme and bootstrapping

In Gentry's PhD Thesis [11], he lays out a somewhat homomorphic scheme based on ideal lattices, then shows how such a scheme can be turned into a fully homomorphic one. The idea is to have Decrypt be a shallow circuit that can itself be homomorphically evaluated by Evaluate. This is called bootstrapping, its basically a way to reduce noise with Evaluate so that the ciphertexts remain correct and can be again computed upon. The concept of bootstrapping in practice is explored in subsection 4.4 through the TFHE scheme.

Evaluate has to take the function it wants to evaluate as a circuit, this comes from the fact that we don't want any information to leak by evaluation, this means that we for example cannot branch on an encrypted value. In practice this makes efficient FHE very hard. Nowadays many optimizations have been found and implemented so that works such as ours, can be done, but major companies, such as Google, Microsoft and ZAMA are still working on bringing out the full potential of FHE. The current 'dream' is to have a compiler that the end developer can use without any cryptographic knowledge.

4 TFHE

The TFHE scheme was introduced in [4].

The TFHE scheme uses both LWE and RLWE ciphertexts, so we introduce a generalization of the two, which will be useful for defining operations on them at the same time. Throughout this section $\mathcal{R}$ will denote:

$$\mathbb{Z}[X]/(X^N + 1) \text{ where } N \text{ is a power of two}$$

Definition 4.1. Let $\mathbf{S} = (S_0, S_1, \dots, S_{k-1}) \in \mathcal{R}^k$ denote the secret key. Let $0 < p \leq q$ the message and ciphertext modulus respectively and $\frac{p}{q} = \Delta$ be the scaling factor. The GLWE encryption of a message $M \in \mathcal{R}_p$ is a vector $C = (A_0, A_1, \dots, A_{k-1}, B) \in \mathcal{R}_q^{k+1}$, where A_i is sampled uniformly random from $\mathcal{R}_q$ and $B = \sum_{i=0}^k S_i A_i + \Delta M + E$, where E is a small error drawn from the Gaussian distribution χ_σ .

We call ΔM the encoding of M , and denote the set of GLWE encryptions of ΔM under the secret key $\mathbf{S}$ with standard deviation σ by $GLWE_{\mathbf{S}, \sigma}(\Delta M)$. The encoding of M puts it into the $\log(\Delta)$ most significant bits (MSB) of the message, while the error is in the least significant bits (LSB).

Remark 4.2. p and q are usually powers of two, if not then the scaling factor becomes $\left\lfloor \frac{p}{q} \right\rfloor$.

Remark 4.3. While the secret keys could be sampled from a number of distributions, this scheme uses uniform binary distributions.

This generalization allows us to talk about both of them at the same time, in practice only the special cases are used. Notice that we get an LWE ciphertext if we set $N = 1$ and $\mathcal{R}$ becomes $\mathbb{Z}$ and we get a RLWE ciphertext if we set $k = 1$. There will also be two other types of ciphertext introduced later.

4.1 Decryption of a ciphertext

Having both the secret key $\mathbf{S}$, and a ciphertext $C = (A_0, A_1, \dots, A_{k-1}, B) \in GLWE_{\mathbf{S}, \sigma}(\Delta M_1)$ one can decrypt the ciphertext in the following way: First we calculate $B - \sum_{i=0}^{k-1} S_i A_i$. This will be equal to $\Delta M + E$. After this we can calculate M by:

$$\left\lfloor \frac{\Delta M + E}{\Delta} \right\rfloor = \left\lfloor \frac{B - \sum_{i=0}^{k-1} S_i A_i}{\Delta} \right\rfloor$$

The rounding is basically discarding the LSB-s, only keeping the $\log(\Delta)$ MSB of the encoding, which is M .

4.2 Homomorphic operations

Addition: Let

$$C_1 = (A_0, A_1, \dots, A_{k-1}, B) \in GLWE_{\mathbf{S}, \sigma}(\Delta M_1)$$

and

$$C_2 = (A'_0, A'_1, \dots, A'_{k-1}, B') \in GLWE_{\mathbf{S}, \sigma}(\Delta M_2)$$

If we want to calculate the ciphertext encrypting $M_1 + M_2$ we just have sum C_1 and C_2 elementwise:

$$C_1 + C_2 = (A_0 + A'_0, A_1 + A'_1, \dots, A_k + A'_k, B + B') \in GLWE_{\mathbf{S}, \sigma'}(\Delta(M_1 + M_2))$$

This is because when we decrypt $C_1 + C_2$ we get:

$$M_1 + M_2 = \left\lfloor \frac{\Delta(M_1 + M_2) + E_1 + E_2}{\Delta} \right\rfloor = \left\lfloor \frac{B + B' - \sum_{i=0}^{k-1} S_i(A_i + A'_i)}{\Delta} \right\rfloor$$

Notice that by the addition of two ciphertexts the resulting ciphertexts noise level/error grows by 1 bit.

Multiplication with a small scalar: This operation is similarly simple as addition was. Let

$$C = (A_0, A_1, \dots, A_{k-1}, B) \in GLWE_{\mathbf{S}, \sigma}(\Delta M)$$

and $\lambda \in \mathcal{R}$ a small polynomial, small meaning here that its coefficients are small, the degree of λ can be anything. We want to calculate a ciphertext which encrypts λM . Such a ciphertext will be

$$C' = (\lambda A_0, \lambda A_1, \dots, \lambda A_{k-1}, \lambda B) \in GLWE_{\mathbf{S}, \sigma'}(\Delta M)$$

When we decrypt C' we get

$$\lambda M = \left\lfloor \frac{\Delta \lambda M + \lambda E}{\Delta} \right\rfloor = \left\lfloor \frac{\lambda(B - \sum_{i=0}^{k-1} S_i A_i)}{\Delta} \right\rfloor$$

The reason this is only possible with "small" polynomials is because with this operation the noise grows proportional to the size of λ .

More complex operations Here we will see how the more complicated operations are realized in TFHE.

Continuing the previous section, we will first see how to multiply by a large polynomial. But in order to do that, we first have to introduce a new type of ciphertext, the GLev. A GLev ciphertext is composed of GLWE ciphertexts encrypting M with different scaling factors.

Definition 4.4. Let $\beta = 2^{m_\beta}$ the base, and $l \in \mathbb{N}$. The set of GLev ciphertexts encrypting M using secret key $\mathbf{S}$, base β and level l is

$$\left(GLWE_{\mathbf{S}, \sigma} \left(\frac{q}{\beta^1} M \right) \times \dots \times GLWE_{\mathbf{S}, \sigma} \left(\frac{q}{\beta^l} M \right) \right) = GLev_{\mathbf{S}, \sigma}^{\beta, l}(M)$$

Remark 4.5. Just like with GLWE we can get a Lev or an RLev ciphertext by the same special cases.

Remark 4.6. The same addition and small polynomial multiplication which we introduced for GLWE are possible with GLev as well, one just has to do it for every GLWE ciphertext from which the GLev ciphertext is composed of.

The key to multiplication with a large polynomial is to decompose the polynomial into several smaller polynomials, which is done in the following way: Let β be a base such that $\beta^l = q$ and $\lambda \in \mathcal{R}_q$. Then

$$\lambda = \lambda_1 \frac{q}{\beta^1} + \lambda_2 \frac{q}{\beta^2} + \dots + \lambda_l \frac{q}{\beta^l}$$

Where $\forall i \in \{1, 2, \dots, l\}$ $\lambda_i \in \mathcal{R}_\beta$. This is basically λ 's representation in base β . We will denote it by $\text{Decomp}^{\beta, l}(\lambda) = (\lambda_1, \lambda_2, \dots, \lambda_l)$

With everything necessary defined, we can now perform multiplication between a (GLev) ciphertext C and a large polynomial λ :

$$\lambda \cdot C = \langle \text{Decomp}(\lambda), C \rangle = \sum_{i=1}^l \lambda_i C_i$$

This will produce a GLWE ciphertext encrypting λM . This is because

$$C_i \in \text{GLWE}_{\mathbf{S}, \sigma} \left(\frac{q}{\beta^i} M \right)$$

From which

$$\lambda_i C_i \in \text{GLWE}_{\mathbf{S}, \sigma'} \left(\frac{q}{\beta^i} \lambda_i M \right)$$

for all $1 \leq i \leq l$. Now it's easy to see that

$$\sum_{i=1}^l \lambda_i C_i \in \text{GLWE}_{\mathbf{S}, \sigma''} \left(\sum_{i=1}^l \lambda_i \left(\frac{q}{\beta^i} M \right) \right) = \text{GLWE}_{\mathbf{S}, \sigma''} (\lambda M)$$

Notice that while the resulting ciphertext encrypts what we want, it does so without the scaling factor Δ . This makes the operation not that useful in itself, but it is rarely used alone in practice. We will shortly see it used in ciphertext-ciphertext multiplication.

Let's say we have 2 ciphertexts C_1 and C_2 encrypting message ΔM_1 and ΔM_2 . We would like to calculate a $M_1 \cdot M_2$. If $C_1 = (A_0, A_1, \dots, A_k, B)$ then

$$\Delta M_1 \cdot M_2 \approx \Delta \left(B - \sum_{i=0}^{k-1} A_i S_i \right) \cdot M_2 = \Delta B M_2 - \sum_{i=0}^{k-1} \Delta A_i S_i M_2$$

Notice that A_i, B are just large polynomials. So we can already calculate $B M_2$. Now if we would have a ciphertext which $\forall 0 \leq i \leq k$ encrypts $S_i M_2$ we could also calculate $A_i S_i M_2$. This is exactly what the final type of ciphertext in TFHE enables.

Definition 4.7. Let $\mathbf{S} = (S_0, S_1, \dots, S_{k-1}) \in \mathcal{R}^k$ the secret key and $M \in \mathcal{R}_p$ the message we want to encrypt. The GGSW encryption of M under $\mathbf{S}$ with base β and level l is $(C_0, C_1, \dots, C_k) \in \text{GGSW}_{\mathbf{S}, \sigma}^{\beta, l}(M)$, where $C_k \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(M)$ and $\forall i < k$ $C_i \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(-S_i M)$

Remark 4.8. Just like with the previous two ciphertexts, one can get a GSW or an RGSW ciphertext with the same special cases.

Remark 4.9. The same addition and small polynomial multiplication which we introduced for GLWE are possible with GGSW as well, one just has to do it for every GLWE ciphertext from which the GGSW ciphertext is composed of.

Now we can define the external product of a GLWE and a GGSW ciphertext.

Definition 4.10. Let $C = (A_0, A_1, \dots, A_{k-1}, B) \in \text{GLWE}_{\mathbf{S}, \sigma}(\Delta M_1)$ and $C' = (C_0, C_1, \dots, C_k) \in \text{GGSW}_{\mathbf{S}, \sigma}^{\beta, l}(M_2)$, with $C_i \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(-S_i M_2)$ where $0 \leq i < k$ and $C_k \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(M_2)$. Then

$$C \boxtimes C' = (\text{Decomp}(B), C_k) + \sum_{i=0}^{k-1} (\text{Decomp}(A_i), C_i) \in \text{GLWE}_{\mathbf{S}, \sigma'}(\Delta M_1 M_2)$$

From this its easy to construct an internal product between 2 GGSW ciphertexts:

Definition 4.11. Let $C = (C_0, C_1, \dots, C_k) \in \text{GGSW}_{\mathbf{S}, \sigma}^{\beta, l}(M_1)$ with

$$C_i = (C_{i,0}, C_{i,1}, \dots, C_{i,l}) \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(-S_i M_1) \text{ for } 0 \leq i < k$$

and

$$C_k = (C_{k,0}, C_{k,1}, \dots, C_{k,l}) \in \text{GLew}_{\mathbf{S}, \sigma}^{\beta, l}(M_1)$$

Let $C' \in \text{GGSW}_{\mathbf{S}, \sigma}^{\beta, l}(M_2)$. Then the internal product between C and C' is defined as:

$$C \boxtimes C' = (C_{0,1} \boxtimes C', \dots, C_{0,l} \boxtimes C', C_{1,1} \boxtimes C', \dots, C_{k,l} \boxtimes C') \in \text{GGSW}_{\mathbf{S}, \sigma'}^{\beta, l}(M_1 M_2)$$

Now we have addition and multiplication between ciphertexts. These together with bootstrapping to manage noise are what make TFHE fully homomorphic.

4.3 The building blocks of bootstrapping

Now we will show how TFHE implements bootstrapping. But first we have to define few more operations, which are the building blocks of bootstrapping.

The CMux operation is a homomorphic multiplexer gate. That is, we have two inputs d_0, d_1 and a selector bit $b \in \{0, 1\}$.

$$\text{Mux}(d_0, d_1, b) = \begin{cases} d_0 & \text{if } b = 0 \\ d_1 & \text{otherwise} \end{cases}$$

It can be calculated with $(d_0 - d_1) \cdot b + d_1$. Now that we can add and multiply ciphertexts this will be just as simple to do homomorphically: we have two GLWE ciphertexts encrypting d_0 and d_1 , and a GGSW ciphertext encrypting b .

Modulus switching is a simple operation, that switches the ciphertext modulus q to a new modulus $q' \leq p$. It is usually also a power of two and smaller than q . Each coefficient in the ciphertext's polynomials becomes $A' = \frac{q'A}{q} \in \mathcal{R}_{q'}$ (Which for powers of two is basically shifting the bits by $\log(q') - \log(q)$).

Key switching is an operation, which takes an input GLWE ciphertext $C = (A_0, A_1, \dots, A_{k-1}, B) \in \text{GLWE}_{\mathbf{S}, \sigma}(\Delta M)$ and a key switching key and outputs a new GLWE ciphertext C' encrypting the same message under a different secret key $\mathbf{S}'$.

Definition 4.12. *The key switching key from $\mathbf{S} = (S_0, S_1, \dots, S_{k-1})$ to $\mathbf{S}'$ is a list of GLWE ciphertexts encrypting $\mathbf{S}$ under $\mathbf{S}'$*

$$KSK_i = \text{GLWE}_{\mathbf{S}', \sigma}^{\beta, l}(S_i) \quad 0 \leq i < k$$

The operation is done by calculating $B - \sum_{i=0}^{k-1} A_i S_i$ homomorphically under $\mathbf{S}'$

$$C' = (0, 0, \dots, 0, B) - \sum_{i=0}^{k-1} \langle \text{Decomp}(A_i), KSK_i \rangle \in \text{GLWE}_{\mathbf{S}', \sigma'}(\Delta M)$$

The ciphertext $(0, 0, \dots, 0, B)$ is called a trivial ciphertext. It of course isn't encrypting B , just makes it a ciphertext, so we can add the encryptions of $A_i S_i$ to it.

Sample extraction is an operation which takes as input a GLWE ciphertext encrypting M and outputs an LWE ciphertext encrypting one of the coefficient of M .

Let the message $M = \sum_{j=0}^{N-1} m_j X^j \in \mathcal{R}_p$ and the secret key $\mathbf{S} = (S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j, \dots, S_{k-1} = \sum_{j=0}^{N-1} s_{k-1,j} X^j)$. The ciphertext encrypting M under $\mathbf{S}$ is

$$C = \left(A_0 = \sum_{j=0}^{N-1} a_{0,j} X^j, \dots, A_{k-1} = \sum_{j=0}^{N-1} a_{k-1,j} X^j, B = \sum_{j=0}^{N-1} b_j X^j \right) \in \text{GLWE}_{\mathbf{S}, \sigma}(\Delta M)$$

Let m_h be the message coefficient which we want to extract with $0 \leq h < N$. The new secret key, under which m_h will be encrypted, will be

$$\mathbf{s} = (s_{0,0}, \dots, s_{0,N-1}, s_{1,0}, \dots, s_{k-1,N-1})$$

The ciphertext encrypting m_h will be

$$c = (a_0, \dots, a_{N \cdot i+j}, b_h) \in LWE_{\mathbf{s}, \sigma}(m_h) \subseteq \mathbb{Z}_q^{kN}$$

with

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

This can be seen by looking at the

$$\Delta M + E = B - \sum_{i=0}^{k-1} S_i A_i$$

polynomial's h -th coefficient. The negation comes in, because

$$X^N \equiv -1 \pmod{(X^N + 1)}$$

Blind rotation is an operation which takes a GLWE ciphertext encrypting a polynomial $M \in \mathcal{R}_p$ and an encrypted rotation factor $0 \leq \pi \leq N$ and outputs a ciphertext which encrypts $M \cdot X^{-\pi}$.

If we would have to do this in the clear, it would be easy enough. If M is encrypted, say

$$C = (A_0, A_1, \dots, A_{k-1}, B) \in GLWE_{\mathbf{s}, \sigma}(\Delta M)$$

then X^π would be a small polynomial and we could just perform a scalar-ciphertext multiplication.

However when both M and π is encrypted we have to do a series of CMux gates. Let

$$\pi = \sum_{i=0}^{\delta} \pi_i 2^i$$

with $\pi_i \in \{0, 1\}$ and $\delta = \log(N)$.

Let $C_0 = \text{CMux}(C, C \cdot X^{-2^0}, \pi_0)$ and $C_i = \text{CMux}(C_{i-1}, C_{i-1} \cdot X^{-2^i}, \pi_i)$ for $0 < i \leq \delta$. The result of these will be $C_\delta = GLWE_{\mathbf{s}, \sigma'}(\Delta M \cdot X^{-\pi})$.

4.4 (Programmable) Bootstrapping

In the TFHE scheme bootstrapping can be performed on LWE ciphertexts. The high level picture of bootstrapping is this: We have a look-up table (LUT), which for every $\Delta m + e \in \mathbb{Z}_q$ maps the value $m \in \mathbb{Z}_p$. This LUT is stored as the coefficients of a polynomial on which we will perform a blind rotation by $-(\Delta m + e)$

which will result in having m as the constant coefficient of the polynomial. From here we can perform a sample extraction resulting in a ciphertext encrypting m .

Input: $c = (a_1, a_2, \dots, a_{k-1}, b) \in LWE_{\mathbf{s}, \sigma}(\Delta m)$, the GGSW encryptions of the secret key $\mathbf{s} = (s_0, \dots, s_{k-1})$ under a secret key

$$\mathbf{S}' = \left(S'_0 = \sum_{j=0}^{N-1} s'_{0,j} X^j, \dots, S'_{k'-1} = \sum_{j=0}^{N-1} s'_{k'-1,j} X^j \right)$$

which we will denote by $BSK_i \in GGSW_{\mathbf{S}', \sigma}^{\beta, l}(s_i)$.

Let $\mathbf{s}' = (s'_{0,0}, \dots, s'_{0,N-1}, s'_{1,0}, \dots, s'_{k'-1,N-1})$.

Let $KSK_i = GLew_{\mathbf{s}, \sigma}^{\beta, l}(s_i)$ for $0 \leq i < k'N$.

Denote $\frac{N}{p}$ by $\tilde{\Delta}$ and let

$$L = \sum_{j=0}^N \Delta \left\lfloor \frac{j}{\tilde{\Delta}} \right\rfloor X^j \in \mathcal{R}_q$$

Notice that in $L X^{\tilde{\Delta}m+e}$'s coefficient will be Δm .

Performing bootstrapping on c : Perform a modulus switch on c from q to N and denote its result by

$$\tilde{c} = (\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_{k-1}, \tilde{b}) \in LWE_{\mathbf{s}, \sigma}(\tilde{\Delta}m)$$

This step is required because we have to put $\Delta m + e$ into the exponent of a monomial, which can only be done if $\Delta m + e < N$.

Let V be a trivial encryption of L under $\mathbf{S}'$. Let $V_0 = V \cdot X^{-\tilde{b}}$. Let

$$V_i = \text{CMux}(V_{i-1}, V_{i-1} \cdot X^{\tilde{a}_{i-1}}, BSK_{i-1})$$

for $0 < i \leq k$. This will mean that $V_i = V \cdot X^{-\tilde{b} + \sum_{j=0}^{i-1} \tilde{a}_j s_j}$ and $V_k = V \cdot X^{-\tilde{\Delta}m + \tilde{e}} \in GLWE_{\mathbf{S}', \sigma'}(L \cdot X^{-\tilde{\Delta}m + \tilde{e}})$.

Now we can do a sample extraction, extracting the constant coefficient from V_k into the ciphertext $c' \in LWE_{\mathbf{S}', \sigma'}(\Delta m)$.

The last step is to perform a key switching from $\mathbf{s}'$ to $\mathbf{s}$ which can be done using the key switching keys. Finally we can get the result:

$$c'' \in LWE_{\mathbf{s}, \sigma''}(\Delta m)$$

It is called "programmable" bootstrapping, because we can swap m for any function of m . This enables us to evaluate an arbitrary function while reducing the noise of our ciphertext.

4.5 Public and server key

So far we only talked about encrypting with a private key. This is not unusual, since the main use-case for FHE is cloud-computing services, where one only needs to encrypt his own data. However the question naturally arises, whether we have a public key form of TFHE as well. This is possible, and the public key will be encryptions of 0-s. One can encrypt a message by adding some randomly selected encryptions of 0-s to the trivial encryption of the message. For more details, see [25].

In addition to public and private key, TFHE also has server key, which is the key required for doing the homomorphic operations. It consists of the bootstrapping and the key switching keys, which are necessary for bootstrapping.

4.6 Practicalities

FHE is usually thought of as an encryption scheme, where homomorphic addition and multiplication is possible. And while ciphertext-ciphertext is possible, only between GGSW-GGSW ciphertexts and that is really inefficient. So in practice TFHE is fully homomorphic in a slightly different way: through addition and programmable bootstrapping. It can be used not only for univariate, but for bivariate function evaluation as well by packing two messages into the same ciphertext. This way multiplication and of course any other function is possible.

5 MPC, Volume match algorithm

Secure multi party computation, also called multi party computation (MPC) is a field of cryptography which deals with creating protocols that allow a group to jointly compute a shared function, on their respective data without revealing their data. Work on MPC protocols started in the 1970s. Many of its problems have been solved but performance is still an issue. In MPC f is usually expressed as a circuit made up of additions and multiplications.

Algorithm 3 is taken from [5] where they implement it using MPC and achieve very good runtimes.

They demonstrated running Algorithm 3 implemented in MPC. With $n, m = 500$ and $l = 100$ they achieved ≈ 20 seconds of running with full threshold. Our goal was to reach this running time with TFHE-rs.

Algorithm 3: Volume match algorithm from [5]

required : n, m the number of sell/buy orders are public
input : $\langle s_1 \rangle, \langle s_2 \rangle, \dots, \langle s_n \rangle$ the sell orders and $\langle b_1 \rangle, \langle b_2 \rangle, \dots, \langle b_m \rangle$ the buy orders. Where $s_i, b_i \in [1, l]$
output: Unencrypted versions of the final $\langle s_i \rangle$ and $\langle b_j \rangle$

```

1  $\langle S \rangle \leftarrow \sum_{i=1}^n \langle s_i \rangle$ 
2  $\langle B \rangle \leftarrow \sum_{i=1}^m \langle b_i \rangle$ 
3  $\langle f \rangle \leftarrow \langle S \rangle > \langle B \rangle$ 
4  $\langle T \rangle \leftarrow \langle f \rangle \cdot (\langle B \rangle - \langle S \rangle) + \langle S \rangle$ 
5  $\langle L \rangle \leftarrow \langle T \rangle$ 
6 for  $i \leftarrow 1$  to  $n$  do
7    $\langle z_1 \rangle \leftarrow \langle L \rangle \leq 0$ 
8    $\langle z_2 \rangle \leftarrow \langle L \rangle < \langle s_i \rangle$ 
9    $\langle s_i \rangle \leftarrow ((\langle L \rangle - \langle s_i \rangle) \cdot \langle z_2 \rangle + \langle s_i \rangle) \cdot (1 - \langle z_1 \rangle)$ 
10   $\langle L \rangle \leftarrow \langle L \rangle - \langle s_i \rangle$ 
11 end
12  $\langle L \rangle \leftarrow \langle T \rangle$ 
13 for  $i \leftarrow 1$  to  $m$  do
14    $\langle z_1 \rangle \leftarrow \langle L \rangle \leq 0$ 
15    $\langle z_2 \rangle \leftarrow \langle L \rangle < \langle b_i \rangle$ 
16    $\langle b_i \rangle \leftarrow ((\langle L \rangle - \langle b_i \rangle) \cdot \langle z_2 \rangle + \langle b_i \rangle) \cdot (1 - \langle z_1 \rangle)$ 
17    $\langle L \rangle \leftarrow \langle L \rangle - \langle b_i \rangle$ 
18 end
19 return Decrypt  $\langle s_i \rangle$  and  $\langle b_j \rangle$  for all  $i, j$ 
```

The notation $\langle x \rangle$ means that x must be 'secret'. In the MPC case this means that the MPC engine must not reveal information about x , in the FHE case it means that x must be encrypted. This way Algorithm 3 securely matches m buy orders and n sell orders in the time priority of order number.

6 Implementation

TFHE-rs is a Rust library developed by the company ZAMA (see here). It is a pure Rust implementation of TFHE for Boolean and integer arithmetics over encrypted data. In this section, we will explain how we implemented Algorithm 3 using the TFHE-rs library and what optimizations we used.

But in order to do that, we first have to talk about how the TFHE-rs library is implemented.

6.1 TFHE-rs

In TFHE-rs there are 4 high-level modules which can be used for creating applications using FHE. These are boolean, shortint, integer and high-level api, but we did not use the first two, so we will not talk about them here.

In the high-level api after minimal configuration and the encryption of the number, one can manipulate them using the same syntax as one would use for unencrypted numbers. The supported operations are addition, subtraction, multiplication, negation, bitwise operation, comparisons and min/max.

In the integer module encrypted integers are represented by several encrypted "short integers", usually with 4 bit messages: 2 bits of message and 2 bits of padding ("carry bits"). This basically means that integers are stored base 4-ish.

Let's say we have two 6 bit integers encrypted 013_4 and 003_4 which would be stored as $(0,1,3)$ and $(0,0,3)$ where every number in the tuple would be an encrypted shortint (in reality the order of the list is reversed, but for readability we leave it in this natural order). If we add them together, we don't get $(0,2,2)$, but instead $(0,1,6)$. This is because first only the shortint ciphertexts are added together and the overflow from the addition $3+3$ gets stored in the carry bits of the resulting shortint. But of course to prevent the carry overflowing we have to "propagate" the carry to the next shortint sometime. However we don't actually know what value is stored in the shortints, so each ciphertext stores with itself what the largest value stored in it can be, which is called its degree. In the example above the degrees of the ciphertexts would be (if they are freshly encrypted) $(3,3,3)$ and $(3,3,3)$ respectively, and the degrees of the sum of them would be $(6,6,6)$. So the most important operation is full-propagate, which from the least to most significant bits adds the carry to the next shortint, then sets it to 0. It does so by bootstrapping, hence also managing noise levels.

Each operation comes with 4 flavors: unchecked, checked, smart, default. Unchecked flavored operations don't check whether the carry will overflow, checked flavored will check and throw an error if the operation is not possible, smart operations will propagate the ciphertexts if needed, while default operations will always propagate both before and after the operation.

6.2 Bounty

The bounty proposed by Zama was to implement Algorithm 3 using TFHE-rs or concrete-python. The runtime goal set was 20 seconds. The constraints for the input was $1 \leq s_i, b_j \leq 100$, and $N, M \leq 500$.

The full description of the bounty can be seen [here](#)

6.3 Trivial implementation with high level API

First we implemented Algorithm 3 using the libraries high-api module without any major modification to the algorithm. We used the built-in **FheUint16** type. As can be seen in table Figure 4 the runtime of this was slowest. This was the baseline that we started improving upon.

The one tweak we used was to simplify the two main loops in lines 6-10 and lines 12-18. Calculating $(L - s_i) \cdot z_2$ is basically calculating $\min(s_i, L)$, so we used the libraries built-in method for min instead of $(L - s_i) \cdot z_2$, because it was faster. Calculating z_1 was unnecessary because it's easy to see by induction that $L \geq 0$ at all times and when $L = 0$ then $(L - s_i) \cdot z_2$ will become also 0, so we also left that out.

6.4 Improvements and optimizations

Moving forward instead of the high-level api we used the integer module, which gave us more control over the operations.

One major bottleneck was using the same sized integers everywhere. This is because operations' runtime scale roughly with the size of the numbers. So operations on 16bit integers take roughly twice as much time as operations on 8bit integers. This was unfortunate, because most of the operations we wanted to do, were operations between 8 and 16 bit integers. But the libraries are built in such a way that when generating client and server keys one has to specify the size of the integers one wants to encrypt. So what we tried to do, was to implement our own operations between 8 and 16 bit integers, namely addition, subtraction, and min.

We represented a 16 bit integer a as two 8 bit integers a_1, a_2 such that $a = a_1 + 256a_2$. The concrete addition which we needed was: $a := a + b$, where a was 16bit, b was 8bit integer. This was done with the following calculation (with $a = a_1 + 256a_2$):

$$\begin{aligned} \langle s \rangle &\leftarrow \langle a_1 \rangle \\ \langle a_1 \rangle &\leftarrow \langle a_1 \rangle + \langle b \rangle \\ \langle z \rangle &\leftarrow \langle s \rangle > \langle a_1 \rangle \\ \langle a_2 \rangle &\leftarrow \langle a_2 \rangle + \langle z_1 \rangle \end{aligned}$$

This works because calculations are done modulo 256, so if $a_1 + b < a_1 \bmod 256$ then $a_1 + b$ overflowed and we have to carry that over to a_2 .

Subtraction is based on the same principle:

$$\begin{aligned}\langle s \rangle &\leftarrow \langle a_1 \rangle \\ \langle a_1 \rangle &\leftarrow \langle a_1 \rangle + \langle b \rangle \\ \langle z \rangle &\leftarrow \langle s \rangle < \langle a_1 \rangle \\ \langle a_2 \rangle &\leftarrow \langle a_2 \rangle - \langle z_1 \rangle\end{aligned}$$

The minimum we needed to implement was a min between an 8 and a 16 bit integers, where an 8 bit integer is returned.

$$\begin{aligned}\langle z_1 \rangle &\leftarrow \langle a_2 \rangle > \langle 0 \rangle \\ \langle z_2 \rangle &\leftarrow \langle a_1 \rangle > \langle b \rangle \\ \langle z_1 \rangle &\leftarrow \langle z_1 \rangle \cdot \langle z_2 \rangle \\ \langle d \rangle &\leftarrow \langle b \rangle - \langle a_1 \rangle \\ \langle e \rangle &\leftarrow \langle d \rangle \cdot \langle z_1 \rangle \\ \langle \text{min} \rangle &\leftarrow \langle e \rangle \cdot \langle a_1 \rangle\end{aligned}$$

Where min was the result of the operation.

One other optimization which we found was that by most of the operations, the smart version checks whether the operation can be performed without overflowing. If it cannot it calls full-propagation on both inputs. But in our case we knew that s_i 's and b_i 's degree were as small as possible, hence it was unnecessary to call propagation on them. This proved to be a significant improvement.

We also parallelized both the summation and the two main loops of the algorithm. This was not possible in the high-level api, due to limitations in the module. With enough threads available this meant a roughly 2x time improvement.

This implementation (called `integer_paral`) provided some improvement but having to use comparisons in addition and subtraction was slowing it down heavily, because these use bootstrapping internally.

After inspecting the inner workings of the TFHE-rs library closer, we realized that there is a lot of functionality not mentioned in the documentation. Here is what we found and could utilize:

- While the client/server key stores the size of the integers they were created for, it is not used most of the time. This meant that we didn't have to stay with a fixed size ciphertext.
- Addition and subtraction was implemented in a way, that it could be used (with slight modifications) with different sized ciphertexts.

- It is possible to create an integer ciphertext from a list of shortint ciphertexts. This meant that it was possible to create one $2n$ bit integer from two n bit integers or the other way around.

With all of this, we improved the runtime of the algorithm, by only using 16bit integers where necessary. Namely, we used addition and subtraction between 8 and 16 bit integers, and minimum between two 16 bit integers, because we could not improve on the speed of the built-in min function. This became our fastest implementation on machines with few thread available (few meaning less than 16).

6.5 Fully parallelizing the algorithm

We were also able to modify the algorithm to allow for full parallelization of the two main loops.

If we not only calculate the sum of s_i but all the partial sums as well, we can calculate s_i in a parallelizable way. Let

$$S_i = \sum_{j < i} s_j, \quad L = \min(S_n, B_m)$$

and let $L_i = L$ in the i -th iteration of the algorithm. What we have to calculate is: $s_i = \min(L_i, s_i)$ and $L_{i+1} = L_i - s_i$ (And of course the same with b_i) But if we calculate L_i the following way:

$$L_i = L - \min(L, S_i)$$

we can calculate s_i as

$$s_i = \min(s_i, L - \min(L, S_i))$$

This allows the parallelization of the main loop in the algorithm, because s_i is not dependent on all of the previous s_j . Due to this method having more calculation in a single iteration of the loop, it's performance is worse until it has enough threads to work with. But with enough threads, this was by far our fastest implemetation.

7 Results

In this section we present what runtimes we could reach with the different implementations. Figure 4 shows the times for a single run in seconds. It was run in out university computer which had 512 GB ram and an AMD EPYC 7763 64-core processor. Figure 5 is the statistics, of our fastest implementations from about 100 runs, it was run on the same machine. The final Figure 6 shows the statistics of the Fully parallelized version, these were run on an AWS m6i.24xlarge.

In Table Figure 4 'summing' refers to calculating S, B by summing up s, b vectors. 'Transaction volume' is the part where we calculate the minimum of S and B so the calculation of L . 'loop' is the two for loops where we calculate the new s, b vectors and 'total' is the total runtime.

Implementation	Summing	Transaction volume	Main loop	Total
integer_u16_parallel	76.4967s	0.4549980s	456.1669s	532.6642s
high_api	310.0140s	0.2707371s	616.4213s	926.4357s
integer_padded_parallel	66.6170s	0.5247487s	440.1115s	506.7287s
integer_parallel	132.6795s	1.3659s	629.4350s	762.1148s

Fig. 4. Single run of the different implementations broken down by parts of the algorithm.

	integer_u16_parallel	integer_padded_parallel
Avarage	445,2426	374,4662
Minimum	412,2715	343,9027
Maximum	528,9544	502,2297
Standard deviation	20,6528	35,7

Fig. 5. Statistics of the improved implementations introduced in subsection 6.4. (Values in seconds)

	full_parallel
Avarage	110,577
Minimum	105,423
Maximum	157,426
Standart deviation	9,704

Fig. 6. Statistics of the fully parallelized algorithm presented in subsection 6.5. (Values in seconds)

Bibliography

- [1] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.
- [2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
- [3] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3–7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.
- [4] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
- [5] Mariana Botelho da Gama, John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. All for one and one for all: Fully decentralised privacy-preserving dark pool trading using multi-party computation. *Cryptology ePrint Archive*, 2022.
- [6] I Dinur, G Kindler, R Raz, and S Safra. Approximating cvp to within almost-polynomial factors is np-hard. *Combinatorica*, 23:205–243, 2003.
- [7] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26–30, 2015, Proceedings, Part I 34*, pages 617–640. Springer, 2015.
- [8] Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on gpus, with tensor cores. *Cryptology ePrint Archive*, Paper 2021/141, 2021. <https://eprint.iacr.org/2021/141>.
- [9] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.
- [10] Steven D Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, 2012.
- [11] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
- [12] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15–19, 2010. Proceedings 30*, pages 155–172. Springer, 2010.
- [13] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 197–206, 2008.

- [14] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology—CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.
- [15] Shai Halevi. Homomorphic encryption. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 219–276. Springer, 2017.
- [16] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. *An introduction to mathematical cryptography*, volume 1. Springer, 2008.
- [17] Thijs Laarhoven, Joop van de Pol, and Benne de Weger. Solving hard lattice problems and the security of lattice-based cryptosystems. Cryptology ePrint Archive, Paper 2012/533, 2012. <https://eprint.iacr.org/2012/533>.
- [18] Lenstra, Lenstra Jr, Manasse, and Pollard. The number field sieve. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 564–572, 1990.
- [19] Arjen K Lenstra. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.
- [20] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology—CT-RSA 2011: The Cryptographers’ Track at the RSA Conference 2011, San Francisco, CA, USA, February 14–18, 2011. Proceedings*, pages 319–339. Springer, 2011.
- [21] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007.
- [22] Phong Q Nguyen and Damien Stehlé. Low-dimensional lattice basis reduction revisited. *ACM Transactions on algorithms (TALG)*, 5(4):1–48, 2009.
- [23] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT’99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pages 223–238. Springer, 1999.
- [24] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.
- [25] Ron Rothblum. Homomorphic encryption: From private-key to public-key. In *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28–30, 2011. Proceedings 8*, pages 219–234. Springer, 2011.
- [26] Noah Stephens-Davidowitz. Dimension-preserving reductions between lattice problems. <http://www.noahsd.com/latticeproblems.pdf>, 2016.
- [27] Vinod Vaikuntanathan. Computing blindfolded: New developments in fully homomorphic encryption. In *2011 IEEE 52nd annual symposium on foundations of computer science*, pages 5–16. IEEE, 2011.