XZDDF Bootstrapping in Fully Homomorphic Encryption

Simon Ljungbeck
`si5126lj-s@student.lu.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Qian Guo

Examiner: Thomas Johansson

January 2024

© 2024
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Despite the vast research on the topic in recent years, fully homomorphic encryption schemes remain time-inefficient. The main bottleneck is the so-called bootstrapping, whose purpose is to reduce noise that has accumulated after having performed homomorphic operations on a ciphertext. This thesis is about bootstrapping, and how to do it more efficiently. More specifically, a new algorithm, here called XZDDF, is analyzed. The thesis contains a solution to a problem in the original XZDDF algorithm that was encountered during the project. The new algorithm was then implemented in the open-source library OpenFHE. Theoretically, the algorithm has lower time complexity than previous bootstrapping techniques, but the execution time of the implementation was not faster than other algorithms.

Keywords: Fully Homomorphic Encryption, Homomorphic Encryption, Bootstrapping, Blind Rotation, XZDDF.

Preface

This thesis is a master's thesis in the program Engineering Mathematics, written during the Autumn of 2023.

I would like to acknowledge my supervisor Qian Guo for his invaluable guidance and feedback throughout the whole project. Moreover, the topic he suggested suited me perfectly, containing a lot of beautiful mathematics, challenging problem-solving, and fun algorithm implementation.

At last, I also want to thank my family and friends for their encouragement and support during my whole time at LTH.

Lund, December 2023
Simon Ljungbeck

Table of Contents

List of Algorithms

List of Figures

List of Tables

List of Abbreviations

Term	**Meaning**	**Reference**
AP	Bootstrapping algorithm	Section 3.4.1
BGV	FHE scheme	Section A.2.2
BV	FHE scheme	Section A.2.1
CKKS	FHE scheme	Section A.4.1
CVP	Closest vector problem	Definition 2.9
DLWE	Decisional LWE problem	Definition 2.13
DRLWE	Decisional RLWE problem	Definition 2.15
FHE	Fully homomorphic encryption	Chapter 3
FHEW	FHE scheme	Section A.3.2
GINX	Bootstrapping algorithm	Section 3.4.2
GSW	FHE scheme	Section A.3.1
LHE	Levelled homomorphic encryption	Definition 3.3
LMKCDEY	Bootstrapping algorithm	Section 3.4
LWE	Learning with errors	Definition 2.12, 2.17
NTRU	Encryption algorithm	Section 4.1
RLWE	Ring learning with errors	Definition 2.14, 2.18
SHE	Somewhat homomorphic encryption	Definition 3.2
SIS	Shortest integer solution	Definition 2.10
SSSP	Sparse subset sum problem	Definition 2.11
SVP	Shortest vector problem	Definition 2.8
TFHE	FHE scheme	Appendix A.3.3
XZDDF	Bootstrapping algorithm	Chapter 4

Chapter 1

Introduction

Fully Homomorphic Encryption (FHE) was for a long time seen as the holy grail of cryptography. It was mentioned by Rivest et al. already in 1978 [2], but its capability of performing arbitrary computations on encrypted data, without the need to decrypt it first, seemed to be impossible for a long time. However, in 2009, Gentry proved the opposite when he published the first FHE scheme in his PhD thesis [3]. Since then, the research about FHE has exploded, and in 2022, the Gödel Prize was awarded to three FHE researchers for their work.

Almost all existing FHE schemes are noise-based, which means that each ciphertext contains some noise. The noise is an essential part of the encryption since this is what makes the scheme secure, but if the noise becomes too big, the decryption will fail. When performing computations on encrypted data homomorphically, the noise increases, and at some point, the ciphertext needs to be refreshed so that the decryption does not fail if further homomorphic operations are performed.

The technique that is used for doing this refreshing, i.e. reducing the noise, is both simple and elegant. Since FHE schemes can compute any function on a ciphertext, one just computes a second layer of encryption and then evaluates the decryption function homomorphically (see Figure 3.1 for an illustration of the process). This technique is called bootstrapping. Although a beautiful solution in theory, bootstrapping turns out to be quite inefficient in practice, at least for all known FHE schemes today.

In a recent paper, Xiang et al. [1] propose a new algorithm for doing bootstrapping more efficiently than previous techniques. We will call the new algorithm XZDDF, after the authors, and this thesis will be about the XZDDF bootstrapping. We will learn about the theory behind the algorithm and then implement it in a popular open-source FHE library. On the way, we will also learn about bootstrapping and fully homomorphic encryption in general.

1.1 Motivation

FHE does not only contain a lot of beautiful mathematics, but it also has several interesting real-life applications. One example is when letting a third party, such as a cloud service or a fog network, do computations on private data. With FHE, these computations can be performed without compromising the privacy of the data.

A practical example of this is when a machine learning model, e.g. a neural network, is trained on an external supercomputer. If using FHE, one can upload the training data in an encrypted form. Then, the third party can do computations as usual, i.e. it can train the machine learning model, but it does not what it is computing or what the training data is. In this way, sensitive data, such as patient data or bank credentials, can also be used, without revealing it to the third party.

Despite its transformative potential, FHE is still not used much in practice. The main reason for this is the inefficiency of the bootstrapping, resulting in too slow homomorphic computations. Finding an efficient bootstrapping algorithm is important to be able to use FHE more in practice. This is why bootstrapping was chosen as the main topic in this thesis.

1.2 Goals

The main goal of this thesis is to analyze and implement the XZDDF algorithm. The underlying mathematics will be studied and presented. In case of any flaws in the algorithm, the goal is to solve these so that the algorithm can be implemented. Then, we aim to test the efficiency of the implementation and compare it with the theoretical time complexity of the algorithm. On the way, FHE schemes and older bootstrapping techniques will be studied as well to get a deeper understanding of FHE in general.

1.3 Scope

This thesis primarily focuses on the theory of the XZDDF algorithm that is needed for implementing it. The thesis will therefore not focus on other aspects of the algorithm such as its security and how the noise grows when performing homomorphic operations. For these things, we refer to the original paper by Xiang et al. [1] instead.

1.4 Contributions

This thesis contains theory introducing the reader to the field of FHE and bootstrapping. It also contains new results about XZDDF bootstrapping. Firstly, it announces a problem with the rotation polynomial in the original XZDDF algorithm, and secondly, it proposes a solution to the problem in a special case, when doing Boolean operations with binary messages. At last, an implementation of the solution was programmed and integrated into the `OpenFHE` library. The implementation works well but is not as fast as expected from the theoretical results about the time complexity of the algorithm.

1.5 Structure of the Thesis

In Chapter 2, some cryptographic notations and terms are introduced. The next chapter is about FHE and bootstrapping in general. Then, in Chapter 4, the XZDDF bootstrapping is introduced. The following chapter contains a correction of an error in the original XZDDF algorithm while the last two chapters contain the results of the implementation and conclusions that can be drawn from the project. In the end, there are two appendices. The first one contains theory about a few common FHE schemes, and the second appendix shows figures from the benchmarking of the implementation.

Chapter 2
Preliminaries

2.1 Basic Notations

In this paper, the set of natural numbers $\mathbb{N}$ is defined as all non-negative numbers (including 0), while $\mathbb{N}^* := \mathbb{N}\backslash\{0\}$.

$\mathcal{R} := \mathbb{Z}[X]/(X^N+1)$ is a quotient ring of a polynomial ring $\mathbb{Z}[X]$ over the integers, modulo (X^N+1). If nothing else is specified, $N = 2^d$ for a $d \in \mathbb{N}^*$.

$\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R} = \mathbb{Z}_Q[X]/(X^N+1)$, where $\mathbb{Z}_Q := \mathbb{Z}/Q\mathbb{Z}$. To specify the degree N of the polynomial, $\mathcal{R}_{Q,N}$ is used.

All vectors are bold, while elements in polynomial rings are not. The scalar product of two vectors $\mathbf{u}$ and $\mathbf{v}$ is denoted as $\langle \mathbf{u}, \mathbf{v} \rangle$.

$\lfloor x \rfloor$, $\lceil x \rceil$ and $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$ denote the floor function, the ceiling function, and the rounding function, respectively.

$X \overset{s}{\leftarrow} \chi$ means that a random variable X is sampled from a distribution χ. For example, $X \overset{s}{\leftarrow} \mathcal{U}(S)$ denotes a sample X that is uniformly drawn from a set S.

2.2 Probability Theory

Marcolla et al. [4] define a negligible probability function in the following way.

Definition 2.1 (Negligible probability function)**.** *A probability function* $\operatorname{negl}(x) : \mathbb{Z} \to \mathbb{R}$ *is called negligible if, for any* $c \in \mathbb{Z}$*, there exists an* $N \in \mathbb{Z}$*, such that* $|\operatorname{negl}(x)| < 1/x^c$ *for all* $x > N$*.*

The opposite of a negligible probability function is an overwhelming probability function.

Definition 2.2 (Overwhelming probability function)**.** *A probability function* $\mathrm{overwhelm(x)} : \mathbb{Z} \to \mathbb{R}$ *is called an overwhelming probability function if and only if* $1 - \mathrm{overwhelm}(x)$ *is a negligible probability function.*

Now, B-boundedness can be defined similarly to the definition of Gentry et al. [5].

Definition 2.3 (B-boundedness)**.** *A distribution* χ*, supported over integers, is called* B*-bounded if*

$$\mathbb{P}_{X \overset{s}{\leftarrow} \chi}[|X| > B] = \mathrm{negl}(x),$$

where $\mathrm{negl}(x)$ *is a negligible function.*

In other words, a distribution χ is B-bounded if the probability of sampling a value $X \overset{s}{\leftarrow} \chi$ greater than B is negligibly small.

Next, the discrete Gaussian distribution [6] is defined.

Definition 2.4 (Discrete Gaussian distribution)**.** *A discrete distribution* $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$*, where* $\mu \in \mathbb{Z}$ *and* $\sigma \in \mathbb{R}$*, is called a discrete Gaussian distribution with center* μ *and scale* σ*, if it has the discrete probability function*

$$\mathbb{P}_{X \overset{s}{\leftarrow} \mathcal{N}_{\mathbb{Z}}(\mu,\sigma^2)}[X = x] = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sum_{y \in \mathbb{Z}} e^{-(y-\mu)^2/(2\sigma^2)}}.$$

Note the similarity between the discrete Gaussian distribution $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$ and the continous Gaussian (normal) distribution $\mathcal{N}(\mu, \sigma^2)$, where the later has the density function

$$\mathbb{P}_{X \overset{s}{\leftarrow} \mathcal{N}(\mu,\sigma^2)}[X = x] = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}.$$

2.3 Number Theory

To understand where the name fully homomorphic encryption comes from, a homomorphic function is defined.

Definition 2.5 (Homomorphic function)**.** *A function* $f : A \to B$ *for two algebraic structures* A *and* B *of the same type is called homomorphic under the operation* $\diamond$ *if*

$$f(x \diamond y) = f(x) \diamond f(y).$$

Moreover, the mathematical structure torus is defined as follows.

Definition 2.6 (The real torus)**.** *The mathematical structure consisting of the real numbers modulo 1 is denoted* $\mathbb{T} := \mathbb{R}/\mathbb{Z}$ *and is called the real torus.*

This means that the elements of the real torus can be represented by the interval $[0, 1)$. Just as the reals, the torus has the addition operation, but the multiplication operation is not defined, and hence, it is not a ring [7]. To see why multiplication is not defined, take for example the element $\left[\frac{1}{3}\right] = \left[\frac{4}{3}\right] \in \mathbb{T}$. Then

$$\left[\frac{1}{3}\right] \cdot \left[\frac{1}{3}\right] = \left[\frac{1}{9}\right] \neq \left[\frac{7}{9}\right] = \left[\frac{16}{9}\right] = \left[\frac{4}{3}\right] \cdot \left[\frac{4}{3}\right].$$

However, multiplication between elements of the torus and integers is well-defined.

2.4 Lattices

Most FHE schemes rely on hard lattice problems. This section contains some theory about lattices.

Definition 2.7 (Lattice)**.** *For any two positive integers $k, n \in \mathbb{N}^*$ where $k \leq n$, let $B = (\mathbf{b}_1, ..., \mathbf{b}_k)$ be k linearly independent vectors in $\mathbb{R}^n$. Then the k-dimensional lattice $\mathcal{L}$ generated by B is defined as*

$$\mathcal{L} = \mathcal{L}(B) = \left\{ \sum_{i=1}^{k} \gamma_i \mathbf{b}_i : \ \gamma_i \in \mathbb{Z} \right\}.$$

B is called the base of the lattice, and k is called the rank.

Each lattice $\mathcal{L}$ is associated to a problem called the shortest vector problem, which is defined below.

Definition 2.8 (Shortest vector problem)**.** *Given a lattice $\mathcal{L}$ and a norm $\|\cdot\|$ (usually the L^2 norm), the problem of finding*

$$\arg\min_{\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\mathbf{v}\|$$

is called the shortest vector problem (SVP).

The shortest vector problem can be generalized to the closest vector problem.

Definition 2.9 (Closest vector problem)**.** *Given a lattice $\mathcal{L}$, a norm $\|\cdot\|$ (usually the L^2 norm), and a vector $\mathbf{t} \in \mathbb{R}^n$, the problem of finding*

$$\arg\min_{\mathbf{v} \in \mathcal{L}} \|\mathbf{v} - \mathbf{t}\|$$

is called the closest vector problem (CVP). *The expression $\min_{\mathbf{v} \in \mathcal{L}} \|\mathbf{v} - \mathbf{t}\|$ is called the distance betwen $\mathbf{t}$ and $\mathcal{L}$, and is denoted by $\operatorname{dist}(\mathbf{t}, \mathcal{L}) := \min_{\mathbf{v} \in \mathcal{L}} \|\mathbf{v} - \mathbf{t}\|$.*

There is also a problem called the short integer solution.

Definition 2.10 (Short integer solution)**.** *Let $q, m, n \in \mathbb{N}^*$ be three positive integers, and take a matrix $A \in \mathbb{Z}_q^{m\times n}$. Moreover, let $\beta \in \mathbb{R}$ be a real number such that $\beta < q$, and let $\|\cdot\|$ be some norm. Then the problem of finding a non-zero vector $\mathbf{x} \in \mathbb{Z}^n \backslash \{\mathbf{0}\}$ such that*

$$\begin{aligned} &\textit{(i)} && \|\mathbf{x}\| \leq \beta \\ &\textit{(ii)} && A\mathbf{x} \equiv \mathbf{0} \mod q, \end{aligned}$$

is called the short integer solution problem and is abbreviated $\mathsf{SIS}_{n,m,q,\beta}$.

At last, a problem that partly relies on a version of the SVP problem is defined. See Marcolla et al. [4] for more details.

Definition 2.11 (Sparse subset sum problem)**.** *Let $S = \{a_1, \ldots, a_n\} \subseteq \mathbb{Z}$ be a set of integers. Then the problem of finding a subset $A \subseteq S$ such that*

$$\sum_{x\in A} x = 0$$

is called the sparse subset sum problem (SSSP).

2.5 Lattice-Based Cryptography

This section gives an introduction to lattice-based cryptography.

2.5.1 Hard Problems

The modern lattice-based cryptosystems are usually based on the learning with errors (LWE) problem. The LWE problem was introduced by Regev in 2005 [8], and is defined below.

Definition 2.12 (Learning with errors problem)**.** *For two positive integers $q, n \in \mathbb{N}^*$, let $\mathbf{a} \in \mathbb{Z}_q^n$, and $b \in \mathbb{Z}_q$. Then, the problem of finding a vector $\mathbf{s} \in \mathbb{Z}_q^n$ such that*

$$b = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q,$$

for some sample $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathbb{Z}$, is called the learning with errors problem (LWE).

There is also a decisional version of the LWE problem.

Definition 2.13 (Decisional learning with errors problem)**.** *For two positive integers $q, n \in \mathbb{N}^*$, and an error sample $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathbb{Z}$, the problem of distinguishing the tuple $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ from a tuple uniformly chosen from $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is called the decisional learning with errors problem* (DLWE).

The LWE and the DLWE problems are reducible to each other [9], and the LWE problem can be reduced to the worst-case hardness of SVP [10].

The LWE problem also has a ring-version.

Definition 2.14 (Ring learning with errors problem). *For two positive integers $Q, N \in \mathbb{N}^*$, let $\mathcal{R}_Q := \mathbb{Z}_Q[X]/(X^N + 1)$. Moreover, let $a, b \in \mathcal{R}_Q$. Then, the problem of finding a ring element $s \in \mathcal{R}_Q$ such that*

$$b = a \cdot s + e,$$

for some sample $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathcal{R}_Q$, is called the ring learning with errors problem (RLWE).

Definition 2.15 (Decisional ring learning with errors problem). *For two positive integers $Q, N \in \mathbb{N}^*$, a ring $\mathcal{R}_Q := \mathbb{Z}_Q[X]/(X^N + 1)$, and an error sample $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathcal{R}_Q$, the problem of distinguishing the tuple $(a, b = a \cdot s + e) \in \mathcal{R}_Q \times \mathcal{R}_Q$ from a tuple uniformly chosen from $\mathcal{R}_Q \times \mathcal{R}_Q$ is called the decisional ring learning with errors problem* (DRLWE).

At last, we will also define the decisional NTRU problem.

Definition 2.16 (Decisional NTRU problem). *For two positive integers $Q, N \in \mathbb{N}^*$, a ring $\mathcal{R}_Q := \mathbb{Z}_Q[X]/(X^N+1)$, and an error sample $g \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathcal{R}_Q$, the problem of distinguishing $g/f \in \mathcal{R}_Q$ from a random polynomial uniformly drawn from $\mathcal{R}_Q$ is called the decisional* NTRU *problem.*

2.5.2 Encryption Algorithms

LWE-based encryption can be written on the form defined below.

Definition 2.17 (LWE ciphertext). *For two positive integers $q, n \in \mathbb{N}^*$, let $m \in \mathbb{Z}_q$ be a message and let $\mathbf{a}, \mathbf{s} \in \mathbb{Z}_q^n$ be a public vector and a private key, respectively. Then the LWE encryption of m is defined as*

$$\text{LWE}_{q,\mathbf{s}}(m) = (\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + \text{noised}(m)) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

where $\text{noised}(m)$ *is a noised encoding of m using some noise $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathbb{Z}_q$.*

Regev [8] uses $\text{noised}(m) = m \cdot \frac{q}{t} + e$, so that

$$\text{LWE}_{q,\mathbf{s}}^{\text{Regev}}(m) = \left(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + m \cdot \frac{q}{t} + e\right). \tag{2.1}$$

Note that knowing the private key $\mathbf{s}$, a ciphertext $\text{LWE}_{q,\mathbf{s}}(m) = (\mathbf{a}, b)$ is easily decrypted:

$$\text{noised}(m) = b - \langle \mathbf{a}, \mathbf{s} \rangle.$$

It is also possible to encrypt using the RLWE problem.

Definition 2.18 (RLWE ciphertext)**.** *For two positive integers $Q, N \in \mathbb{N}^*$, let $m, a, s \in \mathcal{R}_Q$, be a message, a public value, and a private key, respectively. Then the RLWE encryption of m is defined as*

$$\mathrm{RLWE}_{Q,s}(m) = (a, b = a \cdot s + \mathrm{noised}(m)),$$

where $\mathrm{noised}(m)$ *is a noised encoding of m using some noise $e \overset{s}{\leftarrow} \chi$ drawn from an error distribution χ over $\mathcal{R}_Q$.*

Encryption using the NTRU problem is defined in Chapter 4.

Chapter 3

Fully Homomorphic Encryption

3.1 Cryptographic Notations and Definitions

In this section, some notations and definitions related to fully homomorphic encryption are introduced.

First of all, let $\mathcal{E}$ denote an encryption scheme that can encrypt and decrypt messages and ciphertexts, respectively. The set of messages (plaintexts) that the cryptosystem can encrypt is denoted by $\mathcal{M}$. Similarly, denote the set of ciphertexts that the system can decrypt by $\mathcal{C}$.

Also, for a given encryption scheme $\mathcal{E}$, let the functions

$$\begin{aligned}\mathrm{Enc}:\ &\mathcal{M}\to\mathcal{C}\\ \mathrm{Dec}:\ &\mathcal{C}\to\mathcal{M}\end{aligned}$$

denote the encryption function and the decryption function, respectively. This means that $\mathrm{Dec}(\mathrm{Enc}(x)) = x$ for any message $x \in \mathcal{M}$.

In the case of symmetric-key cryptography, one can use the notations Enc_k and Dec_k to emphasize the need for the secret key k to encrypt and decrypt. Similarly, we write $\mathrm{Enc}_{\mathsf{pk}}$ and $\mathrm{Dec}_{\mathsf{sk}}$ in public-key cryptography, where sk is the private key and pk is the public key.

Now, fully homomorphic encryption can be defined.

Definition 3.1 (Fully homomorphic encryption)**.** *An encryption scheme $\mathcal{E}$ is called a fully homomorphic encryption (FHE) scheme if its encryption function* $\mathrm{Enc}:\ \mathcal{M}\to\mathcal{C}$ *preserves the two operations addition and multiplication, i.e.*

$$\begin{aligned}\mathrm{Enc}(x+y) &= \mathrm{Enc}(x)\oplus\mathrm{Enc}(y)\\ \mathrm{Enc}(x\cdot y) &= \mathrm{Enc}(x)\odot\mathrm{Enc}(y),\end{aligned}$$

for some operations $\oplus$ and $\odot$ on the set of ciphertexts.

This means that in fully homomorphic encryption schemes, computations with encrypted messages can be performed without the need of first decrypting them.

Being able to perform both addition and multiplication means that we can also compute XOR and AND operations. Since these two operators form a functionally complete set of Boolean operators, i.e. they can express all possible truth tables, any Boolean circuit can be computed in a fully homomorphic encryption scheme [4].

Now, the following notations are introduced

$$\mathscr{M} = (\mathcal{M}, +, \times)$$
$$\mathscr{C} = (\mathcal{C}, \oplus, \otimes).$$

$\mathscr{M}$ is a mathematical structure consisting of the plaintext set $\mathcal{M}$ associated with the addition and multiplication operation. Similarly, $\mathscr{C}$ is the cipher text space, consisting of the corresponding two operators $\oplus$ and $\otimes$, operating on ciphertexts in $\mathcal{C}$. In practical implementations, we usually have that $\oplus = +$ and $\otimes = \times$.

We also define some weaker forms of homomorphic encryption.

Definition 3.2 (Somewhat homomorphic encryption)**.** *An encryption scheme $\mathcal{E}$ is called a somewhat homomorphic encryption (SHE) scheme if it is homomorphic only for a limited class of circuits.*

Definition 3.3 (Levelled homomorphic encryption)**.** *An encryption scheme $\mathcal{E}$ is called a levelled homomorphic encryption (LHE) scheme if it can evaluate any circuits of bounded depth, i.e. a depth lower than a predetermined value.*

Note that the sets of SHE schemes and LHE schemes are not disjoint.

At last, the security parameter is defined.

Definition 3.4 (Security parameter)**.** *The security parameter λ in an encryption scheme $\mathcal{E}$ is a parameter in bits that decides the sizes of the other parameters in $\mathcal{E}$ so that the time complexity for breaking the system becomes $\mathcal{O}(2^\lambda)$.*

3.2 Structure of Fully Homomorphic Encryption Schemes

Usually, FHE schemes are based on the LWE problem. This means that each ciphertext has some noise associated with it, and the noise will increase each time an operation is performed. If too many operations are computed, the cumulative noise will become so big that the decryption fails. Luckily, there are solutions to avoid this by decreasing the noise before it becomes too large. We say that a ciphertext is *refreshed* when we decrease its noise.

Without refreshing, the encryption scheme is homomorphic only for a limited number of operations, i.e. it is just an SHE scheme.

Gentry [3] shows that any SHE scheme can be modified to become fully homomorphic. To refresh the ciphertexts, he introduces a technique called *bootstrapping.*

The general FHE scheme Gentry [3] suggests consists of the four algorithms

$$\mathcal{E} = (\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval}),$$

where all components are probabilistic polynomial-time (PPT) algorithms [4]. As before, Enc and Dec denote the encryption and the decryption functions, respectively.

KeyGen is a function that takes a security parameter λ as input and outputs

$$\text{KeyGen} : \ \lambda \mapsto (\mathsf{sk}, \mathsf{pk}, \mathsf{evk}),$$

where sk is a private key, pk is a public key, and evk is a public evaluation key. The latter is used to evaluate functions on ciphertexts homomorphically.

Eval is a function that takes the following input:

- An evaluation key evk
- A function $f : \mathcal{M}^t \to \mathcal{M}$ taking t inputs $x_1, ..., x_t \in \mathcal{M}$
- t ciphertexts $(c_1, ..., c_t)$.

It then, with overwhelming probability, outputs

$$\text{Eval}_{\mathsf{evk}} : \ (f, (c_1, ..., c_t)) \mapsto \text{Enc}_{\mathsf{pk}}\big(f(\text{Dec}_{\mathsf{sk}}(c_1), ..., \text{Dec}_{\mathsf{sk}}(c_t))\big).$$

This means that Eval is a function that takes a given number of ciphertexts as input and outputs the encrypted value that f would output if inputting the corresponding plaintexts to it. Note that Eval does this without any knowledge about the private key sk.

Appendix A contains theory about some of the common FHE schemes.

3.3 Introduction to Bootstrapping

The bootstrapping technique that Gentry [3] proposes for reducing the noise is illustrated in Figure 3.1 below. Algorithm 1 shows more specifically how the method works. Note that the algorithm does not contain any secret data, and hence, it may be performed by a third party.

At the last line of Algorithm 1, the new ciphertext c' encrypts the same message as the inputted ciphertext c. This can be shown by expanding the expression

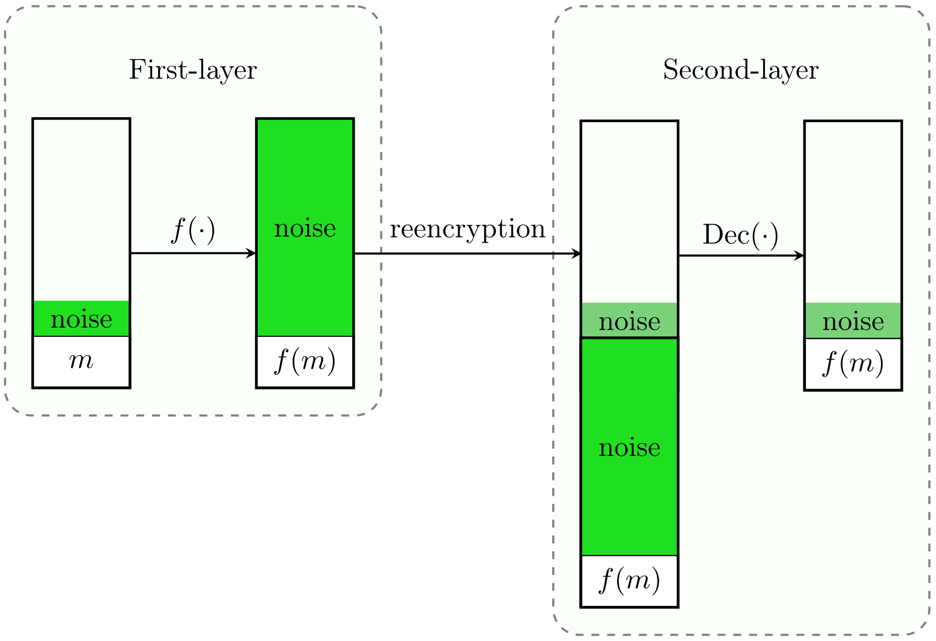


Figure 3.1: Illustration showing how bootstrapping works.

assigned to c':

$$\begin{aligned} c' &= \text{Eval}_{\mathsf{evk}}(\text{Dec}, \bar{c}, \overline{\mathsf{sk}}) \\ &= \text{Eval}_{\mathsf{evk}}(\text{Dec}, \text{Enc}_{\mathsf{pk}}(c), \text{Enc}_{\mathsf{pk}}(\mathsf{sk})) \\ &= \text{Enc}_{\mathsf{pk}}(\text{Dec}_{\mathsf{sk}}(c)) \\ &= \text{Enc}_{\mathsf{pk}}(m) \end{aligned}$$

In other words,

$$\text{Dec}_{\mathsf{sk}}(c') = \text{Dec}_{\mathsf{sk}}(\text{Enc}_{\mathsf{pk}}(m)) = m = \text{Dec}_{\mathsf{sk}}(\text{Enc}_{\mathsf{pk}}(m)) = \text{Dec}_{\mathsf{sk}}(c).$$

This means that c' and c are decrypted to the same message, but since many homomorphic computations have been performed on c, while c' is new, c' contains

Algorithm 1 Naive bootstrapping

```
Require:
  $\overline{\mathsf{sk}} = \text{Enc}_{\mathsf{pk}}(\mathsf{sk})$        // encryption of the secret key
  $\mathsf{pk}$
  $\mathsf{evk}$
  $c = \text{Enc}_{\mathsf{pk}}(m)$        // encrypted message to refresh
Ensure: $c' = \text{Enc}_{\mathsf{pk}}(m)$        // $c'$ has smaller noise than $c$
  $\bar{c} \leftarrow \text{Enc}_{\mathsf{pk}}(c)$
  $c' \leftarrow \text{Eval}_{\mathsf{evk}}(\text{Dec}, \bar{c}, \overline{\mathsf{sk}})$
```

less noise. Note, however, that one operation has already been performed, namely the homomorphic decryption (last line of Algorithm 1), so it does not contain as little noise as if m was encrypted completely from scratch.

One last note is that bootstrapping, unfortunately, is quite computationally heavy and requires much memory. Therefore, there is huge interest in today's FHE research to increase the efficiency of bootstrapping.

3.4 Bootstrapping

As seen above, bootstrapping refreshes FHE ciphertexts by evaluating their decryption algorithm homomorphically. In Appendix A, we also see that most of the existing FHE schemes decrypt by computing a function

$$g(\langle \mathbf{c}, \mathbf{s} \rangle \mod q) = g(\sum_{i=1}^{n} c_i s_i \mod q),$$

where $\mathbf{c} \in \mathbb{Z}_q^n$ is the ciphertext, $\mathbf{s} \in \mathbb{Z}_q^n$ is the private key, and g is a function that in some way uses the result of the scalar product to decrypt the ciphertext.

The naive approach of doing bootstrapping is to simply compute g homomorphically as in Algorithm 1. However, the modulo operation is computationally expensive [1]. Therefore, in practice, one needs to modify the bootstrapping so that the execution time is reduced. There are several ways of doing so, and one is based on so-called blind rotation. This section will give an overview of how this way of doing bootstrapping usually works. In the next chapter, we introduce a new, faster blind rotation algorithm developed by Xiang et al. [1].

The main idea of doing bootstrapping with blind rotation is to transform the noisy ciphertext to a ciphertext in another scheme, where the modulo operations can be computed cheaper. More specifically, one wants to transform the ciphertext to a ring-based encryption system, where the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ is constructed so that $q = 2N$. In this way, the order of $X \in \mathcal{R}$ is $\text{ord}_{\mathcal{R}}(X) = q$, which means that that the computations modulo q can be computed easily in the exponent:

$$X^{\sum_i c_i s_i} = X^{(\sum_i c_i s_i \bmod q) + kq} = X^{\sum_i c_i s_i \bmod q} \cdot (X^q)^k = X^{\sum_i c_i s_i \bmod q}.$$

Then, it is just to read the exponent of $X^{\sum_i c_i s_i}$ to get the result of the modulo computation.

Usually, bootstrapping using blind rotation consists of three steps:

1. **Blind rotation**
   Transforms the first-layer ciphertext to a ring-based ciphertext.
2. **Extraction**
   Transforms the ring-based ciphertext back to the first-layer encryption again.

3. **Key switching**
   Switches the new first-layer key back to the original key.

There are two common bootstrapping algorithms that are usually used, and they differ mainly in how the blind rotation is performed [11]. One performs blind rotation using the AP algorithm [12], while the other uses the GINX algorithm [13].

Section 3.4.1 and 3.4.2 below give a short introduction to these algorithms. We refer to [12] and [13] for more details about how AP and GINX work.

Usually, AP needs a larger evaluation key than GINX. GINX is faster for binary and ternary secrets, but slower for larger secrets [11].

In Chapter 4, we present a new algorithm from Xiang et al. [1], which both has a small evaluation key and is fast for large secrets. Another algorithm, that also beats AP and GINX in memory and time complexity, is called LMKCDEY [14].

3.4.1 AP Blind Rotation

The AP method for doing blind rotation relies on decompositions of $a_i = \sum_j a_{i,j} B^j$ in a base B. All possible values of $a_{i,j} s_i \in \mathbb{Z}_q$, where $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ is the private key, are pre-computed and encrypted, and then stored in the blind rotation evaluation key BRK^{AP}.

Algorithm 2 shows how the evaluation key is generated, while Algorithm 3 shows how the blind rotation is performed, where the operator $\otimes$ is the external product between RLWE ciphertexts and RGSW ciphertexts, defined by for example Alperin-Sheriff and Peikert [12]. The output ACC is essentially an RLWE ciphertext of the decryption of the inputted LWE ciphertext $c = (\mathbf{a}, b)$.

Algorithm 2 AP.BRKGen

```
Require:
  $\mathbf{s} \in \mathbb{Z}_q^n$                                   // secret key
  $B \in \mathbb{Z}_q$                                        // basis
Ensure: $BRK^{\text{AP}}$                                 // evaluation key
  for $i = 0 .. (n-1)$ do
    for $j = 0 .. (\log_B(q) - 1)$ do
      for $v = 0 .. B$ do
        $BRK^{\text{AP}}_{i,j,v} \leftarrow \text{RGSW}_{\mathbf{z}}(X^{vB^j s_i})$     // RGSW encryption
      end for
    end for
  end for
```

Algorithm 3 AP.BREval

```
Require:
  $f \in \mathcal{R}_Q$
  $c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$                // LWE ciphertext
  $BRK^{\mathrm{AP}}$
  $B \in \mathbb{Z}_q$                                   // basis
Ensure: ACC                                       // accumulator
  $Y \leftarrow X^{\frac{2N}{q}} \in \mathcal{R}_Q$                       // $\implies \mathrm{ord}_{\mathcal{R}_Q}(Y) = q$
  ACC $\leftarrow (0, Y^{-b} \cdot f) \in \mathcal{R}_Q \times \mathcal{R}_Q$
  for $i = 0 .. (n-1)$ do
    for $j = 0 .. (\log_B(q) - 1)$ do
      $a_{i,j} \leftarrow \lfloor a_i / B^j \rfloor \mod B$
      ACC $\leftarrow$ ACC $\otimes BRK^{\mathrm{AP}}_{i,j,a_{i,j}}$              // $\otimes$ is the external product
    end for
  end for
```

3.4.2 GINX Blind Rotation

In the GINX method for doing blind rotation, the elements of the private key $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_q^n$ are decomposed as $s_i = \sum_{j=0}^{|U|-1} u_j \cdot s_{i,j}$ where $s_{i,j} \in \{0,1\}$ and $U = \{u_0, \ldots, u_{|U|-1}\}$ is a public set. All $s_{i,j}$ are then encrypted and stored in the blind rotation evaluation key BRK^{GINX}.

Algorithm 4 shows how the evaluation key is generated, while Algorithm 5 shows how the blind rotation is performed. Just as for the AP blind rotation, the operator $\otimes$ is the external product between RLWE ciphertexts and RGSW ciphertexts.

Algorithm 4 GINX.BRKGen

```
Require:
  $\mathbf{s} \in \mathbb{Z}_q^n$                                       // secret key
  $U \subset \mathbb{Z}_q$
Ensure: $BRK^{\mathrm{GINX}}$                               // evaluation key
  for $i = 0 .. (n-1)$ do
    $s_{i,j} \leftarrow \mathrm{Solve}\left(s_i = \sum_{j=0}^{|U|-1} u_j \cdot s_{i,j} \quad \text{s.t. } s_{i,j} \in \{0,1\}\right)$
    for $j = 0 .. (|U| - 1)$ do
      $BRK^{\mathrm{GINX}}_{i,j} \leftarrow \mathrm{RGSW}_{\mathbf{z}}(s_{i,j})$                // RGSW encryption
    end for
  end for
```

Algorithm 5 GINX.BREval

```
Require:
  $f \in \mathcal{R}_Q$
  $c = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$                // LWE ciphertext
  $BRK^{\text{GINX}}$
Ensure: ACC                                            // accumulator
  $Y \leftarrow X^{\frac{2N}{q}} \in \mathcal{R}_Q$                          // $\implies \quad \text{ord}_{\mathcal{R}_Q}(Y) = q$
  ACC $\leftarrow (0, Y^{-b} \cdot f) \in \mathcal{R}_Q \times \mathcal{R}_Q$
  for $i = 0 .. (n-1)$ do
    for $j = 0 .. (|U| - 1)$ do
      ACC $\leftarrow$ ACC $+ (Y^{a_i u_j} - 1) \cdot (\text{ACC} \otimes BRK_{i,j}^{\text{GINX}})$
    end for
  end for
```

3.5 Security

As can be seen in Algorithm 1, an encryption of the private key $\overline{\mathsf{sk}} = \text{Enc}_{\mathsf{pk}}(\mathsf{sk})$ is required when doing bootstrapping. All existing FHE schemes today require this in one way or another [4]. Therefore, one assumption for FHE schemes to be secure is that it is safe to decrypt a private key under its public key. This assumption is called the *circular security assumption.*

For additional information about the security of FHE schemes, beyond what is necessary for understanding this thesis, we refer to Chapter VI in [4].

Chapter 4

XZDDF Bootstrapping

Xiang et al. [1] propose a new way of doing blind rotation, using NTRU-based encryption, instead of RLWE-based encryption, as in AP [12], GINX [13] and LMKCDEY [14]. The result turns out to be both faster and more memory efficient than the other three methods. We will now present how this fast blind rotation works. In this thesis, it will be called XZDDF bootstrapping.

First, let us assume that the first-layer encryption, which is to be refreshed, has the form $(\mathbf{a}, b = \sum_{i=0}^{n-1} a_i s_i - \text{noised}(m))$, where $\mathbf{a} = (a_0, ..., a_{n-1}) \in \mathbb{Z}_q^n$ is a public random vector, $\mathbf{s} = (s_0, ..., s_{n-1}) \in \mathbb{Z}_q^n$ is the private key, and noised(m) is an encoding of the plaintext with some noise $e \in \mathbb{Z}_q$.

As before, we construct a ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$ where N is a power of 2 and $q = 2N$ so that the order of X is q and the modulo q operations can be computed for free in the exponent. We then want to compute

$$X^{\text{noised}(m)} = X^{\sum_{i=0}^{n-1} a_i s_i - b \mod q} = X^{-b} X^{\sum_{i=0}^{n-1} a_i s_i},$$

and extract the exponent noised(m) by multiplying the rotation polynomial $r(X) = \sum_{i=0}^{q-1} iX^{-i}$, and read the coefficient of the constant term:

$$\text{noised}(m) = \text{coeff}_0 \left(r(X) \cdot X^{-b} X^{\sum_{i=0}^{n-1} a_i s_i} \right).$$

Xiang et al. [1] now makes this method more general by introducing looser constraints on q and N, allowing us to compute noised(m) with any pair of (N, q), where $q|2N$, by just multiplying $2N/q$ in the exponent. In this way, we get

$$\left(X^{\frac{2N}{q}} \right)^q = X^{\frac{2N}{q} q} = X^{2N} = (X^N - 1)(X^N + 1) + 1 \equiv 1 \mod (X^N + 1).$$

This means, that we once again can do modulo q computations for free in the exponent. We can then easily compute

$$\text{noised}(m) = \text{coeff}_0 \left(r(X^{\frac{2N}{q}}) \cdot X^{-\frac{2N}{q} b} X^{\frac{2N}{q} \sum_{i=0}^{n-1} a_i s_i} \right).$$

Since $\mathbf{s}$ is private, we do only have access to encryptions $c_i(X) \in \mathcal{R}_Q$ of X^{s_i} under a private key $f(X) \in \mathcal{R}_Q$. To get a ciphertext that encrypts $X^{a_i s_i}$, Xiang et al. [1] applies the automorphism $X \mapsto X^{a_i}$ so that the ciphertext $c_i(X^{a_i})$ encrypts $X^{a_i s_i}$.

Firstly, note that the new ciphertext $c_i(X^{a_i})$ is not under the original private key $f(X)$, but under a key $f(X^{a_i})$. Since our goal is to homomorphically compute the product $r(X^{\frac{2N}{q}}) \cdot X^{-\frac{2N}{q}b} X^{\frac{2N}{q}\sum_{i=0}^{n-1} a_i s_i}$, we need all encrypted terms to be under the same private key, and therefore, the private keys $f(X^{a_i})$ to $X^{a_i s_i}$ need to be switched back to $f(X)$.

Secondly, note that the automorphism works only if a_i is coprime to $2N$, which is not true when a_i is even. The proposed solution from Xiang et al. [1] is to require that $q|N$ instead of $q|2N$. In this way, one can compute

$$X^{\frac{2N}{q}a_i s_i} = X^{(\frac{2N}{q}a_i+1)s_i - s_i} = X^{w_i s_i} X^{-s_i}$$

instead, where $w_i = \frac{2N}{q}a_i + 1$. Since $q|N$, w_i must be odd, so that it is coprime to $2N$ for all a_i. Now, what we want to compute has become

$$\text{noised}(m) = \text{coeff}_0 \left(r(X^{\frac{2N}{q}}) \cdot X^{-\frac{2N}{q}b} X^{\sum_{i=0}^{n-1} w_i s_i} X^{-\sum_{i=0}^{n-1} s_i} \right) \tag{4.1}$$

To solve the key switching problem, Xiang et al. [1] constructs an NTRU-based encryption scheme for the second-layer encryption, designed to switch keys efficiently. The next section describes how this scheme works.

4.1 NTRU-Based Encryption

Let us begin by defining two parameters (τ, Δ):

$$(\tau, \Delta) := \begin{cases} \left(1, \left\lfloor \frac{Q}{t} \right\rceil\right), & \text{if noised}(m) = e + \left\lfloor \frac{q}{t} \right\rceil \cdot m \\ (t, 1), & \text{if noised}(m) = t \cdot e + m \\ (1, 1), & \text{if noised}(m) = e + m. \end{cases}$$

This means that which parameters to use depends on how noised(m) is encoded in the first-layer encryption.

Xiang et al. [1] then presents two versions of NTRU encryption – one scalar version and one vector version. Definition 4.1 and 4.2 show how these encryptions work.

Definition 4.1 (Scalar NTRU encryption)**.** *Given the two parameters (τ, Δ), the scalar NTRU encryption of a message $u \in \mathcal{R}_Q$ under an invertible private key $f \in \mathcal{R}_Q$, is defined as*

$$\text{NTRU}_{Q,f,\tau,\Delta}(u) := \tau \cdot g/f + \Delta \cdot u/f \in \mathcal{R}_Q,$$

where $f, g \in \mathcal{R}_Q$ are polynomials with small coefficients.

Definition 4.2 (Vector NTRU encryption). *Given the parameter τ and an integer $B \in \mathbb{N}^*$, the vector NTRU encryption of a message $v \in \mathcal{R}_Q$ under an invertible private key $f \in \mathcal{R}_Q$, is defined as*

$$\mathrm{NTRU}'_{Q,f,\tau}(v) := (\tau \cdot g_0/f + B^0 \cdot v, \ldots, \tau \cdot g_{d-1}/f + B^{d-1} \cdot v) \in \mathcal{R}_Q^d,$$

where $f, g_0, \ldots, g_{d-1} \in \mathcal{R}_Q$ are polynomials with small coefficients and $d = \lceil \log_B Q \rceil$.

We also define a bit decomposition function $\mathrm{BitDecom}_B(\cdot)$ like Xiang et al. [1].

Definition 4.3 (Bit decomposition). *Assume that an element $a \in \mathcal{R}_Q$ can be written as $a = \sum_{i=0}^{d-1} a_i \cdot B^i$ in a base $B \in \mathbb{N}^*$, where $d = \lceil \log_B Q \rceil$. Then the bit decomposition of a in base B is defined as*

$$\mathrm{BitDecom}_B(a) := (a_0, \ldots, a_{d-1}) \in \mathcal{R}_B^d.$$

Xiang et al. [1] now defines an external binary operation.

Definition 4.4 ($\odot$ product). *The external binary operator $\odot$ takes in a polynomial $c \in \mathcal{R}_Q$ and a vector NTRU ciphertext $\mathbf{c}' = (c_0, \ldots, c_{d-1}) = \mathrm{NTRU}'_{Q,f,\tau}(v) \in \mathcal{R}_Q^d$, and outputs the scalar product*

$$c \odot \mathbf{c}' := \langle \mathrm{BitDecom}_B(c), \mathbf{c}' \rangle = \sum_{i=0}^{d-1} c_i c'_i = \tau \cdot \sum_{i=0}^{d-1} c_i g_i/f + cv \in \mathcal{R}_Q.$$

Like Xiang et al. [1] show, the $\odot$ operator has a homomorphic property.

Lemma 4.1 (Homomorphic multiplication). *Assume that $c = \mathrm{NTRU}_{Q,f,\tau,\Delta}(u)$ and $\mathbf{c}' = \mathrm{NTRU}'_{Q,f,\tau}(v)$. Then $\hat{c} = c \odot \mathbf{c}'$ is a scalar NTRU ciphertext of uv.*

Proof. By the definitions of scalar and vector NTRU ciphertexts above, we can write $c = \tau \cdot g/f + \Delta \cdot u/f$ and $\mathbf{c}' = (\tau \cdot g_0/f + B^0 \cdot v, \ldots, \tau \cdot g_{d-1}/f + B^{d-1} \cdot v)$. Let us assume that $\mathrm{BitDecom}_B(c) = (c_0, \ldots, c_{d-1})$. Then, we get

$$\begin{aligned}
\hat{c} &= c \odot \mathbf{c}' \\
&= \langle \mathrm{BitDecom}_B(c), \mathbf{c}' \rangle \\
&= \sum_{i=0}^{d-1} c_i(\tau \cdot g_i/f + B^i \cdot v) \\
&= \tau \left(\sum_{i=0}^{d-1} c_i g_i \right) /f + cv \\
&= \tau \left(\left(\sum_{i=0}^{d-1} c_i g_i \right) + gv \right) /f + \Delta \cdot uv/f \\
&= \tau \cdot \hat{g}/f + \Delta \cdot uv/f,
\end{aligned}$$

where $\hat{g} := \left(\sum_{i=1}^{d-1} c_i g_i \right) + gv$. We see that the product has the form of a scalar NTRU encryption of uv. □

4.1.1 Key Switching for Scalar NTRU Ciphertexts

To switch the key of a scalar NTRU ciphertext, Xiang et al. [1] propose two algorithms ($\mathsf{NTRU.KSKGen}, \mathsf{NTRU.KS}$), defined in Algorithm 6 and 7 below. Note that $\mathbf{ksk}_{f_1,f_2}$ in Algorithm 6 has the same form as a vector NTRU encryption of f_1/f_2 under the private key f_2.

Algorithm 6 NTRU.KSKGen

```
Require:
  $f_1 \in \mathcal{R}_Q$                    // invertible key to switch from
  $f_2 \in \mathcal{R}_Q$                    // invertible key to switch to
Ensure: $\mathbf{ksk}_{f_1,f_2}$              // key switching key
  $\mathbf{ksk}_{f_1,f_2} \leftarrow (\tau \cdot g_0/f_2 + B^0 \cdot f_1/f_2, \dots, \tau \cdot g_{d-1}/f_2 + B^{d-1} \cdot f_1/f_2)$
```

Algorithm 7 NTRU.KS

```
Require:
  $c$                                       // scalar NTRU ciphertext
  $\mathbf{ksk}_{f_1,f_2}$
Ensure: $\hat{c}$                           // ciphertext under the new key
  $\hat{c} \leftarrow c \odot \mathbf{ksk}_{f_1,f_2}$
```

Now, let us prove that Algorithm 7 works for key switching.

Lemma 4.2 (NTRU key switching)**.** *The outputted ciphertext $\hat{c}$ in Algorithm 7 is a scalar NTRU encryption of the same message as the original ciphertext c but under the new private key $f_2 \in \mathcal{R}_Q$.*

Proof. Let us assume that c encrypts a message $u \in \mathcal{R}_Q$, so that we can write $c = \tau \cdot g/f_1 + \Delta \cdot u/f_1$, and let $\mathrm{BitDecom}_B(c) = (c_0, \dots, c_{d-1})$. Then, since $\mathbf{ksk}_{f_1,f_2} = \mathrm{NTRU}'_{Q,f_2,\tau}(f_1/f_2)$, we get

$$\begin{aligned}\hat{c} &= c \odot \mathbf{ksk}_{f_1,f_2} \\ &= \sum_{i=0}^{d-1} c_i(\tau \cdot g_i/f_2 + B^i \cdot f_1/f_2) \\ &= \tau\left(\sum_{i=0}^{d-1} c_i g_i\right)/f_2 + c \cdot f_1/f_2 \\ &= \tau\left(\left(\sum_{i=0}^{d-1} c_i g_i\right) + g\right)/f_2 + \Delta \cdot u/f_2 \\ &= \tau \cdot \hat{g}/f_2 + \Delta \cdot u/f_2\end{aligned}$$

where $\hat{g} := \left(\sum_{i=1}^{d-1} c_i g_i\right) + g$. We see that the result is a scalar NTRU ciphertext of u under f_2. □

Having access to the key-switching algorithms, we can easily do the needed key-switching after having applied the automorphisms mentioned earlier. One simply applies the automorphism

$$\begin{aligned}\psi_t :\ & \mathcal{R}_Q \to \mathcal{R}_Q \\ & c(X) \mapsto c(X^t)\end{aligned}$$

on a scalar NTRU ciphertext $c(X)$ under a key $f(X)$, and then switch the key $f(X^t)$ of the new ciphertext $c(X^t)$ back to $f(x)$. The procedure can be divided into two steps, defined in Algorithm 8 and 9 below.

Algorithm 8 NTRU.AutoKGen

```
Require:
  $t \in \mathbb{Z}_{2N} \backslash 2\mathbb{Z}_{2N}$            // $t$ must be odd
  $f \in \mathcal{R}_Q$                                            // private scalar NTRU encryption key
Ensure: $\mathbf{ksk}_t$                                           // ksk for the automorphism
  $\mathbf{ksk}_t \leftarrow \mathsf{KSKGen}(f(X^t), f(X))$
```

Algorithm 9 NTRU.EvalAuto

```
Require:
  $t \in \mathbb{Z}_{2N} \backslash 2\mathbb{Z}_{2N}$            // $t$ must be odd
  $c$                                                              // scalar NTRU ciphertext
  $\mathbf{ksk}_t$
Ensure: $\hat{c}$                                                  // automorphic transformation
  $\hat{c} \leftarrow \mathsf{NTRU.KS}(c(X^t), \mathbf{ksk}_t)$
```

4.2 Fast Blind Rotation Using the NTRU Setting

In this section, we put the previous results together and show how Xiang et al. [1] perform a fast blind rotation on an LWE-based first-layer ciphertext $\mathsf{LWE}_{q,\mathbf{s}}(m) = (\mathbf{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$. If the first-layer encryption is RLWE-based instead, one first needs to transform the RLWE ciphertext to N LWE ciphertexts, and then, after the bootstrapping, transform all LWE ciphertexts back to an RLWE ciphertext again. For more details about these steps, we refer to the paper by Xiang et al. [1].

The goal is to output a second-layer scalar NTRU encryption of $r(Y) \cdot Y^{\langle \mathbf{a}, \mathbf{s} \rangle - b}$, where $Y \in \mathcal{R}_Q$ is a monomial of order q. To do so, we introduce two algorithms $(\mathsf{XZDDF.BRKGen}, \mathsf{XZDDF.BREval})$ as Xiang et al. [1]. They are defined in Algorithm 10 and 11. Note that $q|N$ is assumed.

Algorithm 10 XZDDF.BRKGen

```
Require:
    $q, n \in \mathbb{N}^*$                                  // first-layer parameters
    $\mathbf{s} \in \mathbb{Z}_q^n$                          // first-layer private key
    $Q, N, \tau, \Delta \in \mathbb{N}^*$                    // second-layer parameters
    $f \in \mathcal{R}_Q$                                    // second-layer private key
Ensure: $\mathbf{EVK}_{\tau,\Delta}$                         // blind rotation evaluation keys
    $\mathbf{evk}_0 \leftarrow \mathrm{NTRU}'_{Q,f,\tau}(X^{s_0}/f)$
    for $i = 1 .. (n-1)$ do
        $\mathbf{evk}_i \leftarrow \mathrm{NTRU}'_{Q,f,\tau}(X^{s_i})$
    end for
    $\mathbf{evk}_n \leftarrow \mathrm{NTRU}'_{Q,f,\tau}(X^{-\sum_{i=0}^{n-1} s_i})$
    $S \leftarrow \left\{\frac{2N}{q} i + 1\right\}_{i=1}^{q-1}$        // all elements $j \in S$ are odd
    for $j \in S$ do
        $\mathbf{ksk}_j \leftarrow \mathsf{NTRU.AutoKGen}(j, f)$
    end for
    $\mathbf{EVK}_{\tau,\Delta} \leftarrow (\mathbf{evk}_0, \ldots, \mathbf{evk}_n, \{\mathbf{ksk}_j\}_{j \in S})$
```

4.3 Switching Back to First-Layer Encryption

When having performed the blind rotation with XZDDF.BREval on an LWE ciphertext, one gets an NTRU ciphertext which needs to be transformed back to an LWE ciphertext. We will now show how to do this by using the extraction algorithm from Kim et al. [15], which is recommended by Xiang et al. [1].

After the blind rotation, we get a ciphertext

$$c = \mathrm{NTRU}_{Q,f,\tau,\Delta}(u) := \tau \cdot g/f + \Delta \cdot u/f \in \mathcal{R}_Q$$

where

$$u = r(X^{\frac{2N}{q}}) \cdot X^{-\frac{2N}{q}b} X^{\sum_{i=0}^{n-1} w_i s_i} X^{-\sum_{i=0}^{n-1} s_i}.$$

We want to turn the constant term $\mathrm{coeff}_0(u) = \mathrm{noised}(m)$ of u back to an LWE ciphertext.

Start by defining the coefficient vectors of c, f, and g as

$$\mathbf{c} := (c_0, \ldots, c_{N-1}) \in \mathbb{Z}_Q^N$$
$$\mathbf{f} := (f_0, \ldots, f_{N-1}) \in \mathbb{Z}_Q^N$$
$$\mathbf{g} := (g_0, \ldots, g_{N-1}) \in \mathbb{Z}_Q^N.$$

Now, let us investigate the polynomial $fc \in \mathcal{R}_Q$. By the definition of scalar NTRU encryption, we get that

$$\mathrm{coeff}_0(fc) = \tau \cdot g_0 + \Delta \cdot \mathrm{noised}(m) \in \mathbb{Z}_Q.$$

Algorithm 11 XZDDF.BREval

```
Require:
    $(\mathbf{a}, b) = \mathsf{LWE}_{\mathbf{s},q}(m) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$
    $r(X) \in \mathcal{R}_Q$                                   // rotation polynomial
    $\mathbf{EVK}_{\tau,\Delta} = (\mathbf{evk}_0, \dots, \mathbf{evk}_n, \{\mathbf{ksk}_j\}_{j \in S})$
Ensure: $\mathsf{ACC} = \mathrm{NTRU}_{Q,f,\tau,\Delta}\left(r(X^{\frac{2N}{q}}) \cdot X^{\frac{2N}{q}(-b+\sum_{i=0}^{n-1} a_i s_i)}\right)$
    for $i = 1 \,..\, (n-1)$ do
        $w_i \leftarrow \frac{2N}{q} a_i + 1$
        $w_i' \leftarrow w_i^{-1} \mod 2N$
    end for
    $w_n' \leftarrow 1$
    $\mathsf{ACC} \leftarrow \Delta \cdot r(X^{\frac{2N}{q} w_0'}) \cdot X^{-\frac{2N}{q} b w_0'}$
    for $i = 1 \,..\, (n-1)$ do
        $\mathsf{ACC} \leftarrow \mathsf{ACC} \odot \mathbf{evk}_i$
        if $w_i w_{i+1}' \neq 1$ then
            $\mathsf{ACC} \leftarrow \mathsf{NTRU.EvalAuto}(\mathsf{ACC}, \mathbf{ksk}_{w_i w_{i+1}'})$
        endif
    end for
    $\mathsf{ACC} \leftarrow \mathsf{ACC} \odot \mathbf{evk}_n$
```

We can also compute the constant term in the product with the binomial theorem, resulting in another expression for the constant term

$$\mathrm{coeff}_0(fc) = f_0 c_0 - \sum_{i=1}^{N-1} c_i f_{N-i},$$

where we used that

$$aX^N = a(X^N + 1) - a \equiv -a \mod (X^N + 1).$$

Let $\hat{\mathbf{c}} = (c_0, -c_{N-1}, \dots, -c_1) \in \mathbb{Z}_Q^N$. Then, putting the two expressions for the constant term together, we get

$$0 = \langle \hat{\mathbf{c}}, \mathbf{f} \rangle - (\tau \cdot g_0 + \Delta \cdot \mathrm{noised}(m)) \in \mathbb{Z}_Q.$$

Looking at $(\hat{\mathbf{c}}, 0) \in \mathbb{Z}_Q^N \times \mathbb{Z}_Q$, we observe that this can be seen as an LWE ciphertext of m under the private key $\mathbf{f}$. Then, what remains is to switch the key $\mathbf{f}$ back to $\mathbf{s}$, and then switch the modulus Q back to q. This is relatively easy, and we refer to the paper by Xiang et al. [1] to see how it is done.

Chapter 5

Modification of XZDDF Bootstrapping

There is a problem with the XZDDF blind rotation from Xiang et al. [1]. In this chapter, we will first explain what goes wrong in the algorithm, and then propose a solution to the problem when working with Boolean operations.

5.1 The Problem

The problem with the blind rotation from Xiang et al. [1] is related to the rotation polynomial $r(X^{\frac{2N}{q}}) = \sum_{i=0}^{q-1} iX^{-\frac{2N}{q}\cdot i}$. Since we are working in the ring $\mathcal{R}_Q := \mathbb{Z}_Q[X]/(X^N+1)$, we have that $X^N \equiv -1$ and $X^{2N} \equiv 1$, so

$$X^{-i} = \begin{cases} 1, & \text{if } i = 0 \\ -X^{N-i}, & \text{if } 1 \leq i \leq N \\ X^{2N-i}, & \text{if } N+1 \leq i \leq 2N-1. \end{cases}$$

To make things simpler, let us assume that $q = 2N$. Then

$$\begin{aligned} r(X) &= \sum_{i=0}^{q-1} iX^{-i} \\ &= -1\cdot X^{N-1} - 2\cdot X^{N-2} - \cdots - N + \\ &\quad + (N+1)\cdot X^{N-1} + (N+2)\cdot X^{N-2} + \cdots + (2N-1)\cdot X \\ &= -N + N\cdot X + N\cdot X^2 + \cdots + N\cdot X^{N-1}. \end{aligned}$$

This means, that when computing the product $r(X)\cdot X^{\text{noised}(m)}$, in general, the coefficient term

$$\text{coeff}_0\left(r(X)\cdot X^{\text{noised}(m)}\right) \neq \text{noised}(m).$$

The problem is that the second half of the terms in the sum $r(X) = \sum_{i=0}^{q-1} iX^{-i}$ wraps around, and is added to the first half of the terms. This means, that the problem arises not only for $q = 2N$ that we assumed, but also for any $q|2N$ and $r(X^{\frac{2N}{q}})$.

An obvious way to avoid the problem with the wrap-around is to just skip the second half of the terms, i.e. defining the rotation polynomial as

$$r(X^{\frac{2N}{q}}) := \sum_{i=0}^{q/2-1} iX^{-\frac{2N}{q}\cdot i}.$$

However, then for messages $\text{noised}(m) \geq \frac{q}{2}$, we still have that

$$\text{coeff}_0\left(r(X^{\frac{2N}{q}}) \cdot X^{\frac{2N}{q}\cdot \text{noised}(m)}\right) \neq \text{noised}(m).$$

We will now explain how this rotation polynomial can be slightly modified so that we can retrieve all binary plaintext messages.

5.2 A Suggestion of Solution

In this section, a proposal for a solution to the problem above will be suggested. The solution is, however, constrained to the case when working with Boolean operations and binary messages.

Assume that the message is $m \in \mathbb{Z}_2$, and that we are using a Regev-like encryption

$$c = \text{LWE}_{q,\mathbf{s}}(m) = \left(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s}\rangle + m \cdot \frac{q}{t} + e\right),$$

where $t = 4$ (this choice will be explained below).

Let $\diamond$ denote a binary operation that we want to compute on two ciphertexts $c_0 = (\mathbf{a}_0, b_0)$ and $c_1 = (\mathbf{a}_1, b_1)$, i.e. we want to compute $c_0 \diamond c_1$. For example, $\diamond$ can be OR or AND.

The algorithm then starts by computing

$$c = c_0 + c_1 = (\mathbf{a}_0 + \mathbf{a}_1, b_0 + b_1) =: (\mathbf{a}, b).$$

Since LWE is homomorphic, we now have that

$$\text{Dec}(c) = \begin{cases} 0, & \text{if } (m_0, m_1) = (0,0) \\ 1, & \text{if } (m_0, m_1) = (0,1) \text{ or } (1,0) \\ 2, & \text{if } (m_0, m_1) = (1,1). \end{cases}$$

This is why we set $t = 4$ – we want to be able to handle twos and threes so that we keep the information about whether the sum was an addition of two zeros or two ones.

Now, we will do a modified version of XZDDF bootstrapping on c.

Let us denote noised$(m) := b - \langle \mathbf{a}, \mathbf{s} \rangle = m \cdot \frac{q}{t} + e$, and define t intervals

$$I_i := \left[i \cdot \frac{q}{t} - \frac{q}{2t}, i \cdot \frac{q}{t} + \frac{q}{2t} \right) \subset \mathbb{Z}_q,$$

for $i \in \{0, 1, \ldots, t-1\}$. In our case, we have the $t = 4$ intervals

$$\begin{aligned} I_0 &= \left[-\frac{q}{8} = \frac{7q}{8}, \frac{q}{8} \right), \\ I_1 &= \left[\frac{q}{8}, \frac{3q}{8} \right), \\ I_2 &= \left[\frac{3q}{8}, \frac{5q}{8} \right), \\ I_3 &= \left[\frac{5q}{8}, \frac{7q}{8} \right). \end{aligned}$$

If for example $\diamond =$ OR, we now want a function f_{OR} that maps

$$f_{\text{OR}} : \quad \left(X^{\frac{2N}{q}} \right)^{\text{noised}(m)} \mapsto \begin{cases} 0, & \text{if noised}(m) \in I_0 \\ 1, & \text{if noised}(m) \in I_1 \\ 1, & \text{if noised}(m) \in I_2 \\ 0, & \text{if noised}(m) \in I_3. \end{cases}$$

Similarly, if $\diamond =$ AND, we want a function f_{AND} that maps

$$f_{\text{AND}} : \quad \left(X^{\frac{2N}{q}} \right)^{\text{noised}(m)} \mapsto \begin{cases} 0, & \text{if noised}(m) \in I_0 \\ 0, & \text{if noised}(m) \in I_1 \\ 1, & \text{if noised}(m) \in I_2 \\ 1, & \text{if noised}(m) \in I_3. \end{cases}$$

Working cyclically, modulo q, the interval I_0 can be seen as the following interval to I_3. We then see that for both operators OR and AND, the intervals that are mapped to 0 and 1 consist of two following intervals

$$\begin{aligned} I^0 &= I_k \cup I_{(k+1 \bmod 4)} \\ I^1 &= I_{(k+2 \bmod 4)} \cup I_{(k+3 \bmod 4)}, \end{aligned}$$

where $k = 3$ for OR and $k = 0$ for AND. In fact, as can be seen in the `binfhe/` directory of `OpenFHE` [16], one can for any binary operation find two intervals $I^0 = [q_0, q_1)$ and $I^1 = [q_1, q_0)$, where $q_1 = q_0 + q/2$ (sometimes the binary operation needs to be computed as a composition of other binary operations).

Working in R_Q, we know that X^i wraps around negacyclically at $i = N$, so that $aX^i = -aX^{i+N} \mod (X^N + 1)$. This property was the cause of failure when using the original rotation polynomial, but it can also be used to solve the problem by forming a new rotation polynomial

$$\begin{aligned} r(X^{\frac{2N}{q}}) :&= -\sum_{i=0}^{q/4-1}\left(X^{-\frac{2N}{q}}\right)^i + \sum_{i=q/4}^{q/2-1}\left(X^{-\frac{2N}{q}}\right)^i \\ &= -1\cdot\left(X^{-\frac{2N}{q}}\right)^0 - 1\cdot\left(X^{-\frac{2N}{q}}\right)^1 - \cdots - 1\cdot\left(X^{-\frac{2N}{q}}\right)^{\frac{q}{4}-1} + \\ &\quad + 1\cdot\left(X^{-\frac{2N}{q}}\right)^{\frac{q}{4}} + 1\cdot\left(X^{-\frac{2N}{q}}\right)^{\frac{q}{4}+1} + \cdots + 1\cdot\left(X^{-\frac{2N}{q}}\right)^{\frac{q}{2}-1} \end{aligned}$$

In this way,

$$m' := \mathrm{coeff}_0\left(r(X^{\frac{2N}{q}})\cdot\left(X^{\frac{2N}{q}(\mathrm{noised}(m)+(\frac{q}{4}-q_1))}\right)\right) = \begin{cases} -1, & \text{if noised}(m)\in[q_0,q_1) \\ 1, & \text{if noised}(m)\in[q_1,q_0). \end{cases}$$

We compute this constant term just as in the original XZDDF bootstrapping. The final problem to overcome with this algorithm is to find a way to map the constant term back to binary values:

$$m' \mapsto \begin{cases} 0, & \text{if } m' = -1 \\ 1, & \text{if } m' = 1. \end{cases}$$

This is easily achieved by first choosing $\Delta = \frac{Q}{4}\cdot\frac{1}{2} = \frac{Q}{8}$ in the NTRU encryption. The output from the XZDDF blind rotation is an LWE ciphertext on the form

$$c' = \mathrm{LWE}_{Q,\mathbf{f}}(m') = (\mathbf{a}, b' = \langle\mathbf{a},\mathbf{s}\rangle + \Delta\cdot m' + e)$$

in modulo Q and under the secret key $\mathbf{f}$. We know that $m'\in\{-1,1\}$, and with $\Delta = \frac{Q}{8}$, we get

$$c' = \mathrm{LWE}_{Q,\mathbf{f}}(m') = \begin{cases} (\mathbf{a}, b' = \langle\mathbf{a},\mathbf{s}\rangle - \frac{Q}{8} + e), & \text{if } m' = -1 \\ (\mathbf{a}, b' = \langle\mathbf{a},\mathbf{s}\rangle + \frac{Q}{8} + e), & \text{if } m' = 1. \end{cases}$$

Finally, we compute a ciphertext

$$c = (\mathbf{a}, b) = \left(\mathbf{a}, b' + \frac{Q}{8}\right).$$

In this way,

$$c = \mathrm{LWE}_{Q,\mathbf{f}}(m') = \begin{cases} (\mathbf{a}, b = \langle\mathbf{a},\mathbf{s}\rangle + e), & \text{if } m' = -1 \\ (\mathbf{a}, b = \langle\mathbf{a},\mathbf{s}\rangle + \frac{Q}{4} + e), & \text{if } m' = 1, \end{cases}$$

so that

$$m = \mathrm{Dec}(c) = \begin{cases} 0, & \text{if } m' = -1 \\ 1, & \text{if } m' = 1, \end{cases}$$

which is just the map we wanted. Then, it is just to do the modulo switch and the key switch back to q and $\mathbf{s}$ as usual. Now, the problem has been solved.

5.2.1 Summary of Solution

In summary, we have found a modification of the XZDDF algorithm that works when doing Boolean operations with binary messages, and when the first-layer encryption is Regev LWE encryption. The modified XZDDF bootstrapping is performed by the following steps.

1. Set $\Delta = Q/8$.
2. Add $(q/4 - q_1)$ to the last element b_{in} of the inputted LWE ciphertext $c_{\text{in}} = (\mathbf{a}_{\text{in}}, b_{\text{in}})$, where q_1 is depends on the binary operation and can be computed by for example `OpenFHE`.
$$c'_{\text{in}} = (\mathbf{a}_{\text{in}}, b_{\text{in}} + q/4 - q_1)\,.$$
3. Use the rotation polynomial
$$r(X^{\frac{2N}{q}}) = -\sum_{i=0}^{q/4-1}\left(X^{-\frac{2N}{q}}\right)^i + \sum_{i=q/4}^{q/2-1}\left(X^{-\frac{2N}{q}}\right)^i.$$
4. Perform XZDDF bootstrapping as usual and extract the LWE ciphertext.
5. Add $Q/8$ to the last element b'_{out} of the outputted LWE ciphertext $c'_{\text{out}} = (\mathbf{a}_{\text{out}}, b'_{\text{out}})$:
$$c_{\text{out}} = \left(\mathbf{a}_{\text{out}}, b'_{\text{out}} + \frac{Q}{8}\right).$$

Chapter 6

Efficiency Tests

6.1 Implementation

A part of this thesis work was about implementing the new blind rotation algorithm. I chose to start from the open-source FHE library `OpenFHE` and then add support for the XZDDF blind rotation algorithm to it. In this way, others can hopefully make use of the implementation as well. The implementation is available at

`https://github.com/SL2000s/masters_thesis_xzddf`

The core of `OpenFHE` consists of two different directories: `binfhe/` and `pke/`. The first one only handles fully homomorphic encryption for binary messages and Boolean operations, while the second increases the message space and the number of functions that can be evaluated. Since there was a problem with the original XZDDF algorithm, and the solution suggested in Chapter 5 just works for the Boolean operations, the support for XZDDF was just added to the `binfhe/` directory.

One thing to note about the implementation is that the paper by Xiang et al. [1] assumes that the LWE encryption is on the form

$$c = \mathrm{LWE}_{q,\mathbf{s}}(m) = (\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle - \mathrm{noised}(m)),$$

so that the decryption algorithm becomes

$$\mathrm{Dec}(c) = \langle \mathbf{a}, \mathbf{s} \rangle - b,$$

but the LWE encryption in `OpenFHE` is implemented as

$$c = \mathrm{LWE}_{q,\mathbf{s}}(m) = \left(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + m \cdot \frac{q}{t} + e\right),$$

with the decryption algorithm

$$\mathrm{Dec}(c) = b - \langle \mathbf{a}, \mathbf{s} \rangle.$$

This was easily solved by modifying (4.1) to

$$\mathrm{noised}(m) = \mathrm{coeff}_0 \left(r(X^{\frac{2N}{q}}) \cdot X^{\frac{2N}{q}b} X^{\sum_{i=0}^{n-1} -w_i s_i} X^{\sum_{i=0}^{n-1} s_i} \right),$$

where $w_i = \frac{2N}{q} a_i + 1$ as before.

6.2 Testing Methods

After having implemented the algorithm, the efficiency of it was tested. The efficiency tests can be found at the following link.

`https://github.com/SL2000s/masters_thesis_xzddf/tree/main/benchmark`

Using Regev LWE encryption as the first-layer encryption algorithm, the tests timed a few different actions a user potentially would like to do when using FHE encryption. We divide the tests into two categories: simple tests (Table 6.1), and batch tests (Table 6.2).

Table 6.1: Description of some simple tests that were performed.

Test	Description
S1:	Generate a bootstrapping key.
S2:	Perform a single bootstrapping.
S3:	Perform an OR operation on two ciphertexts c_0 and c_1.
S4:	Perform an AND operation on two ciphertexts c_0 and c_1.

In the batch tests, where a ciphertext c_1 was updated x times in a row by the following sequence of homomorphic operations:

$$c_1 \leftarrow \mathrm{NOT}(\mathrm{AND}(c_0, \mathrm{OR}(c_0, c_1))) =: f(c_0, c_1).$$

In all tests, the ciphertexts were initially $(c_0, c_1) = (1, 0)$. For the simple tests S1–S4, and for the batch tests B1–B2, the actions were executed 100 times each, and then the average execution time was computed. For Test B3 and B4, the average execution time was instead computed over 10 tests and 3 tests, respectively, to decrease the total execution time for the test.

All of the blind rotation algorithms AP, GINX, LMKCDEY, and XZDDF were tested with the same setup. Each algorithm was tested with one parameter set

Table 6.2: Description of the batch tests that were performed.

Test	Description
B1:	Update c_1 with the value of $f(c_0, c_1)$ 1 time.
B2:	Update c_1 with the value of $f(c_0, c_1)$ 10 times.
B3:	Update c_1 with the value of $f(c_0, c_1)$ 100 times.
B4:	Update c_1 with the value of $f(c_0, c_1)$ 1000 times.

corresponding to 128-bit security, and one parameter set corresponding to 192-bit security. Table 6.3 shows the parameter values in each parameter set. The key distribution for each parameter set was either ternary, i.e. $\mathcal{U}(\{-1, 0, 1\})$, or discrete Gaussian with mean 0 and standard deviation 3.19, i.e. $\mathcal{N}_{\mathbb{Z}}(0, 3.19^2)$. For XZDDF, the four optimized parameter sets (P128T, P128G, P192T, and P192G) from Xiang et al. [1] were tested as well.

Table 6.3: The parameter sets for bootstrapping in `OpenFHE`. P128T, P128G, P192T, and P192G are designed for XZDDF [1]. STD128L (called STD128_LMKCDEY in `OpenFHE`) is designed for LMKCDEY.

Set	Dist.	n	q	N	Q	B	Q_{ks}	B_{ks}
STD128	Tern.	503	1024	1024	134215681	2^9	2^{14}	2^5
STD128L	Gauss.	446	1024	1024	268369921	2^{10}	2^{13}	2^5
P128T	Tern.	512	1024	1024	995329	2^4	2^{14}	2^7
P128G	Gauss.	465	1024	1024	995329	2^4	2^{14}	2^7
STD192	Tern.	805	1024	2048	137438822401	2^{13}	2^{15}	2^5
P192T	Tern.	1024	1024	2048	44421121	2^9	2^{19}	28
P192G	Gauss.	870	1024	2048	44421121	2^9	2^{17}	28

At last, the tests S3 and S4 were also performed with the original rotation polynomial in the XZDDF paper [1]. This implementation can be found in the branch `XZDDF_original` of the `GitHub` repository.

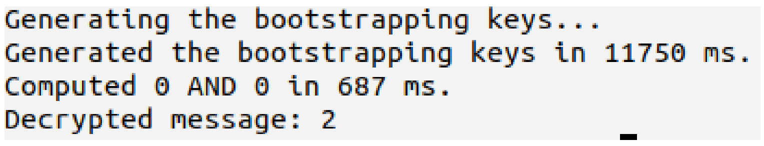
The tests were conducted on a laptop equipped with an Intel Core i7-6600U CPU running at 2.60GHz (4 cores). To decrease noise in the result caused by background activities as much as possible, all other programs on the computer were shut down before the test, and the internet connection was also turned off.

6.3 Results

The results when doing the simple tests can be seen in Table 6.4. In Appendix B, Figure B.1–B.10 show the distribution of the execution times.

Table 6.4: The execution time for different bootstrapping algorithms and different security levels λ (in bits), when performing the four simple tests S1–S4 described in Table 6.1.

Algorithm	**Param.**	**S1** (ms)	**S2** (ms)	**S3** (ms)	**S4** (ms)
AP	STD128	10541	182	175	175
GINX	STD128	2583	153	145	145
LMKCDEY	STD128L	2121	120	132	134
XZDDF	STD128	2438	174	184	185
XZDDF	P128T	6386	214	216	216
XZDDF	P128G	5820	194	195	195
AP	STD192	38489	651	662	645
GINX	STD192	8546	467	467	468
LMKCDEY	STD192	8833	493	512	435
XZDDF	STD192	8391	626	622	626
XZDDF	P192T	11808	700	699	699
XZDDF	P192G	9989	592	592	592



```
Generating the bootstrapping keys...
Generated the bootstrapping keys in 11750 ms.
Computed 0 AND 0 in 687 ms.
Decrypted message: 2
```

Figure 6.1: Decryption failure when using the original rotation polynomial. The AND operation is performed on two zeros, so the result should be 0.

The results when doing the batch tests can be seen in Table 6.5. In Appendix B, Figure B.11–B.12 show log-log plots of the execution times for each bootstrapping algorithm and each parameter set.

Figure 6.1 shows an example of a decryption failure when using the original XZDDF algorithm.

6.4 Discussion

Looking at Table 6.4, we see that when using STD128, XZDDF is among the fastest when generating the evaluation key (Test S1), beating all other algorithms except 128-bit secure LMKCDEY. This aligns quite well with the results of Xiang et al. [1], where XZDDF was faster in all tests. However, theoretically, the key generation should be even faster than in Table 6.4 – at least twice the speed of LMKCDEY and even faster when compared with AP and GINX.

Table 6.5: The execution time for different bootstrapping algorithms and different security levels λ (in bits), when performing the batch tests B1–B4 described in Table 6.2.

Algorithm	**Param.**	**B1** (ms)	**B2** (ms)	**B3** (ms)	**B4** (ms)
AP	STD128	350	3476	35364	355842
GINX	STD128	293	2898	29325	315418
LMKCDEY	STD128L	259	2475	28362	280270
XZDDF	STD128	330	3687	33481	331717
XZDDF	P128T	432	4316	42416	426885
XZDDF	P128G	392	3925	38949	391517
AP	STD192	1146	13052	115382	1213410
GINX	STD192	949	9417	96319	1005410
LMKCDEY	STD192	854	8172	86082	822024
XZDDF	STD192	1254	12439	114381	1163360
XZDDF	P192T	1397	13913	139535	1387043
XZDDF	P192G	1183	11799	117632	1175503

The measured execution time of 128-bit secure XZDDF might be slower than 128-bit LMKCDEY due to noise since 192-bit XZDDF is faster than 192-bit LMKCDEY, but it might also be slower because `OpenFHE` has an optimized security parameter set for LMKCDEY when using 128-bit security (STD192L), but not for 192-bit security.

When using the parameter sets P128T, P128G, P192T, and P192G from Xiang et al., designed specifically for XZDDF, we see that the key generation (Test S1) is significantly slower for these than when using STD128. This is probably due to the larger n, resulting in a longer vector of values to compute in the evaluation key. However, the performance of the homomorphic operations is better, although still slower than when using STD128 for XZDDF.

Just as the results of Xiang et al. [1], there seems to be a small efficiency gain when using a Gaussian distribution instead of a uniformly ternary distribution (compare P128T vs. P128G, and P192T vs. P192G in Table 6.4).

When looking at the results of Test S2–S4 in Table 6.4, one can first note that the Boolean operations (which include one bootstrapping) take about the same time as just performing a single bootstrapping. This is as expected because bootstrapping is the main bottleneck in FHE. In `OpenFHE`, Boolean operations involve one addition and one bootstrapping, with the addition being performed basically instantly compared to the bootstrapping.

Next, one can note that the XZDDF implementation does not seem to perform bootstrapping better than GINX and LMKCDEY, and just marginally better than AP. This differs from the results of Xiang et al. [1], where the XZDDF bootstrapping was more than two times faster than GINX and even faster compared to AP (see e.g. Table 5 in [1]).

One possible explanation for this is that the other implementations have existed for a longer time in `OpenFHE`, so a lot of good programmers have had time to read the code and optimize it. The XZDDF implementation in this project likely has parts that can be optimized more.

There are two different ways to represent polynomials in the `OpenFHE` library – `Format::COEFFICIENT` and `Format::EVALUATION`. The first representation is probably what we are used to – a vector of coefficients – while the second format is a format that makes polynomial multiplications more efficient. The XZDDF implementation switches polynomials back and forth between these representations a few times. Some switches are necessary, but each switch seems to be significantly inefficient. Therefore, if one aims to optimize the XZDDF implementation further, reducing the number of format switches might be a good starting point.

In Figure B.1–B.10, we see that the distributions of the measured execution times usually seem to have a high peak around one value, and then a smaller number of measured times around that peak. There are some exceptions, e.g. 128-bit secure GINX in Figure B.7 that has two peaks, which could be interesting to investigate further, but a possible explanation for these figures is that they arose due to noise, e.g. some background activity on the computer.

In Table 6.5 and in Figure B.11–B.12, we see that the execution time of multiple Boolean operations in a row seems to be linear in the number of operations. This is also logical since the part of the `OpenFHE` library that was used for testing did not support any batching of operations, so the Boolean operations were always followed by a bootstrapping operation.

Chapter 7

Conclusions and Future Work

Although the XZDDF algorithm is faster in theory, the implementation in this project did not perform better than GINX and LMKCDEY when doing the bootstrapping. The XZDDF key generation, on the other hand, performed better, beating all algorithms except the optimized 128-bit secure LMKCDEY. One conclusion that can be drawn from this project is that the implementation of the XZDDF algorithm most likely can be optimized more so that it beats all algorithms in bootstrapping, just as it theoretically should.

Future work on this topic would be to optimize the XZDDF implementation available on `GitHub` and to verify that the implementation is indeed secure to use. Moreover, the problem related to the rotational polynomial, explained in Chapter 5, needs to be solved for the general case, and not only for Boolean operations on binary messages.

Another area, that was not investigated in this thesis, is the theoretical security of the XZDDF algorithm. Since XZDDF is a new algorithm, based on the NTRU problem instead of the RLWE problem, it could be interesting in the future to see if any new kinds of attacks can be made on it.

From a broader perspective, a conclusion that can be drawn from this project is that, although FHE schemes contain a lot of beautiful mathematics, all existing algorithms remain inefficient. Even if the XZDDF implementation can be optimized so that the bootstrapping time is reduced, the execution time for Boolean operations will still be in the magnitude of around 100 milliseconds on today's personal computers, which is far too slow to be of practical use. The bootstrapping in FHE needs to be further developed.

I believe that bootstrapping will continue to be a hot topic in the research about FHE, having interesting applications in other hot research areas such as AI and machine learning. Considering the fact that Gentry's first implementation for bootstrapping took 30 minutes [17], and we already can do the bootstrapping in about 100 milliseconds, I think that bootstrapping still has the potential to become even more efficient. Hopefully, we can soon see FHE being used in practice by cloud services and other third parties.

References

[1] B. Xiang, J. Zhang, Y. Deng, Y. Dai, and D. Feng, "Fast blind rotation for bootstrapping fhes," in *Advances in Cryptology – CRYPTO 2023*, H. Handschuh and A. Lysyanskaya, Eds. Cham: Springer Nature Switzerland, 2023, pp. 3–36.

[2] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations of secure computation*, vol. 4, no. 11, 1978, pp. 169–180.

[3] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2009.

[4] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," Cryptology ePrint Archive, Paper 2022/1602, 2022. [Online]. Available: https://eprint.iacr.org/2022/1602

[5] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 75–92.

[6] C. L. Canonne, G. Kamath, and T. Steinke, "The discrete gaussian for differential privacy," *CoRR*, vol. abs/2004.00010, 2020. [Online]. Available: https://arxiv.org/abs/2004.00010

[7] M. Joye, "Guide to fully homomorphic encryption over the [discretized] torus," Cryptology ePrint Archive, Paper 2021/1402, 2021. [Online]. Available: https://eprint.iacr.org/2021/1402

[8] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Proceedings of the thirty-seventh annual ACM*, pp. 84–93, 2005.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Paper 2018/421, 2018. [Online]. Available: https://eprint.iacr.org/2018/421

[10] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," Cryptology ePrint Archive, Paper 2011/344, 2011. [Online]. Available: https://eprint.iacr.org/2011/344

[11] R. Leluc, E. Chedemail, A. Kouande, Q. Nguyen, and N. Andriamandratomanana, "Fully homomorphic encryption and bootstrapping," 2022. [Online]. Available: https://hal.science/hal-03676650

[12] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," Cryptology ePrint Archive, Paper 2014/094, 2014. [Online]. Available: https://eprint.iacr.org/2014/094

[13] N. Gama, M. Izabachene, P. Q. Nguyen, and X. Xie, "Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems," Cryptology ePrint Archive, Paper 2014/283, 2014. [Online]. Available: https://eprint.iacr.org/2014/283

[14] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo, "Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption," Cryptology ePrint Archive, Paper 2022/198, 2022. [Online]. Available: https://eprint.iacr.org/2022/198

[15] A. Kim, M. Deryabin, J. Eom, R. Choi, Y. Lee, W. Ghang, and D. Yoo, "General bootstrapping approach for rlwe-based homomorphic encryption," Cryptology ePrint Archive, Paper 2021/691, 2021. [Online]. Available: https://eprint.iacr.org/2021/691

[16] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022. [Online]. Available: https://eprint.iacr.org/2022/915

[17] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," Cryptology ePrint Archive, Paper 2010/520, 2010. [Online]. Available: https://eprint.iacr.org/2010/520

[18] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, P. Rogaway, Ed. Springer Berlin Heidelberg, 2011, pp. 505–524.

[19] S. Halevi and V. Shoup, "Helib," https://github.com/homenc/HElib.

[20] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014. [Online]. Available: https://doi.org/10.1137/120868669

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

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

[23] A. Kim, Y. Polyakov, and V. Zucca, “Revisiting homomorphic encryption schemes for finite fields,” in *Advances in Cryptology – ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings, Part 3*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), M. Tibouchi and H. Wang, Eds. Germany: Springer Science and Business Media Deutschland GmbH, 2021, pp. 608–639.

[24] L. Ducas and D. Micciancio, “Fhew: Bootstrapping homomorphic encryption in less than a second,” Cryptology ePrint Archive, Paper 2014/816, 2014. [Online]. Available: https://eprint.iacr.org/2014/816

[25] D. Micciancio and Y. Polyakov, “Bootstrapping in fhew-like cryptosystems,” Cryptology ePrint Archive, Paper 2020/086, 2020. [Online]. Available: https://eprint.iacr.org/2020/086

[26] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” *Advances in Cryptology — ASIACRYPT 2017*, pp. 409–437, 2017.

[27] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for approximate homomorphic encryption,” Cryptology ePrint Archive, Paper 2018/153, 2018. [Online]. Available: https://eprint.iacr.org/2018/153

[28] B. Li and D. Micciancio, “On the security of homomorphic encryption on approximate numbers,” Cryptology ePrint Archive, Paper 2020/1533, 2020. [Online]. Available: https://eprint.iacr.org/2020/1533

Appendix A

Fully Homomorphic Encryption Schemes

Fully homomorphic encryption schemes are usually divided into four generations. The first generation started in 2009 when Gentry [3] proposed the first FHE scheme. Since then, a bunch of other schemes have been invented, and this chapter will present some of the more common ones.

A.1 First Generation

The first generation of FHE schemes can be further divided into two categories:

- Ideal lattice-based
- Approximate greatest common divisor problem (AGCD) based

All first-generation schemes rely on the assumption that the SSSP problem is a hard problem. The second category also relies on another problem called the AGCD problem [4].

First-generation schemes are quite inefficient, and today they have been replaced by more efficient FHE schemes. For example, the implementation of Gentry's first scheme in [3], which is ideal lattice-based, needed 30 minutes to bootstrap [17].

A.2 Second Generation

The second-generation FHE schemes are based on the LWE problem and the RLWE problem. The development of schemes in this generation was, according to Marcolla et al. [4], started by Brakerski and Vaikuntanathan, who published some initial papers [10] and [18] about FHE with LWE and RLWE, respectively. The subsequent research and development of these schemes finally led to the so-called

BGV and B/FV schemes, which are nowadays implemented by many open-source FHE libraries, such as HElib [19] and OpenFHE [16].

The initial paper [10] about LWE-based FHE was extended to a new version [20] in 2014. This section will start by presenting some of the main ideas from that paper before the actual BGV scheme is described. Finally, some notes about the B/FV scheme will be made.

A.2.1 BV Scheme

The LWE-based FHE scheme presented by Brakerski and Vaikuntanathan [20] is called the BV scheme. We will now describe how the encryption and decryption work in the BV scheme.

To get the encryption c of a message $m \in \mathbb{Z}_2$, one computes

$$c = (\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + 2e + m) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

where $\mathbf{a}$ is a public vector, $\mathbf{s}$ is a secret vector and e is a random noise from an error distribution χ.

The ciphertext c can then be decrypted as

$$m' = (b - \langle \mathbf{a}, \mathbf{s} \rangle \mod q) \mod 2.$$

If the error term is small enough, we can assume that $e < q/2$, and then we see that the decryption succeeds because

$$\begin{aligned} m' &= (b - \langle \mathbf{a}, \mathbf{s} \rangle \mod q) \mod 2 \\ &= (2e + m \mod q) \mod 2 \\ &= 2e + m \mod 2 \\ &= m. \end{aligned}$$

One advantage of the BV scheme is that it, in contrast to the FHE schemes in the first generation, does not rely on the SSSP problem [4]. In fact, it only relies on the LWE problem, which, as we have seen, can be reduced to the shortest vector problem (SVP) on arbitrary lattices [10].

There is also an RLWE-based[1] version of the BV scheme, presented by Brakerski and Vaikuntanathan [18].

[1]To be precise, it is based on the so-called PLWE problem, but this problem relies on RLWE problem, i.e. if RLWE can be solved, PLWE can also be solved [18].

A.2.2 BGV Scheme

In 2014, Brakerski, Gentry, and Vaikuntanathan [21] introduced a new levelled fully homomorphic encryption scheme. It is called the BGV scheme, and the level of computations it can handle is set by the user and depends on the purpose, i.e. how deep the circuits to evaluate will be. The purpose of this structure is to avoid bootstrapping. However, there are also bootstrapping techniques for BGV, so that the scheme becomes fully homomorphic. Brakerski et al. [21] present one bootstrapping technique for the scheme. In Chapter 4 and 5, we present another bootstrapping technique, based on a technique from Xiang et al. [1], that can be used for BGV.

There are two versions of the BGV scheme – one based on the LWE problem and one based on the RLWE problem. The RLWE-based is more efficient and usually the one implemented in open-source FHE libraries [4].

We will now present a simplified version of the original BGV scheme in Brakerski et al. [21]. The simplified scheme will consist of the algorithms

$$\mathcal{E}_{BGV} = (\mathsf{BGV.Setup}, \mathsf{BGV.KeyGen}, \mathsf{BGV.Enc}, \mathsf{BGV.Dec}, \mathsf{BGV.Eval}),$$

and our notations and procedures are based on the BGV presentations in Brakerski et al. [21] and Marcolla et al. [4]. See Algorithm 12 – 16 for the pseudocode.

BGV.Setup is described in Algorithm 12. L is the level of the scheme, i.e. the maximum depth of an arithmetic circuit that the scheme can evaluate without bootstrapping.

The output of BGV.Setup is a list, or a ladder, of parameter sets – one for each level in the arithmetic circuit. The main idea of Brakerski et al. [21] to achieve a levelled FHE scheme is to decrease the modulo Q_j between each homomorphic operation. In this way, the size of the error also decreases, so that it does not escalate when performing many multiplications. For each modulo, there is a parameter set for encrypting and decrypting. Note that in our simplified version, $\mathcal{R}, N, n$ and χ are the same at all levels of the ladder, but this is not necessary.

In Algorithm 13, the procedure of generating a public and a private key is presented. The output is a list of key pairs – one for each circuit level. Algorithm 14 and 15 then show how to use these keys to encrypt and decrypt.

Algorithm 12 BGV.Setup

```
Require:
  $\lambda$                                              // security parameter
  $L$                                                    // number of levels
Ensure: $params = \{params_j\}_{j=0}^{L}$                // a ladder of parameters
  $N \leftarrow N(\lambda)$                              // degree of the ring
  $\mathcal{R} \leftarrow \mathbb{Z}[x]/(X^N + 1)$
  $n \leftarrow n(\lambda)$                              // dimension of the ring
  $\chi \leftarrow \chi(\lambda)$                        // error distribution over $\mathcal{R}$
  for $j = L$ .. 0 do
    $Q_j \leftarrow Q(\lambda, j, L)$                    // modulo at level $j$
    $M_j \leftarrow M(\lambda, j, L) = n \cdot \text{polylog}(Q_j)$
    $params_j \leftarrow (\mathcal{R}, n, \chi, Q_j, M_j)$
  end for
```

The correctness of the decryption algorithm can be shown in the following way:

$$\begin{aligned}
&\mathsf{BGV.Dec}(params, sk, \mathbf{c}, j)\\
&\quad = (\langle \mathbf{c}, \mathbf{s}_j \rangle \mod Q_j) \mod 2\\
&\quad = (\langle \mathbf{m} + \mathbf{r}^T \mathbf{A}_j, \mathbf{s}_j \rangle \mod Q_j) \mod 2\\
&\quad = (\langle \mathbf{m}, \mathbf{s}_j \rangle + \langle \mathbf{r}^T \mathbf{A}_j, \mathbf{s}_j \rangle \mod Q_j) \mod 2\\
&\quad = (m + \mathbf{r}^T \mathbf{A}_j \mathbf{s}_j \mod Q_j) \mod 2\\
&\quad = (m + \mathbf{r}^T (\mathbf{b}_j \cdot 1 - \mathbf{B}_j \mathbf{t}_j) \mod Q_j) \mod 2\\
&\quad = (m + 2\mathbf{r}^T \mathbf{e}_j \mod Q_j) \mod 2\\
&\quad = [\mathbf{r}^T \mathbf{e}_j \text{ is small so } (m + 2\mathbf{r}^T \mathbf{e}_j) \text{ does not wrap around } Q_j]\\
&\quad = m + 2\mathbf{r}^T \mathbf{e}_j \mod 2\\
&\quad = m
\end{aligned}$$

To evaluate a function f on some encrypted data, Algorithm 16 can be used. Without loss of generality, the algorithm assumes that f is represented by an arithmetic circuit. The additions and multiplications are performed one at a time, and between each operation, the result is refreshed by moving to the next step of the parameter ladder. The refreshing consists of two steps, where a new ciphertext (still encrypting the same plaintext message) is computed in each step. First, it switches the key pair to the next key pair, and then it decreases the modulo under which, the message is encrypted.

One final note about the BGV scheme is that Brakerski et al. [21] also present a batching technique, that can be used when having many blocks of encrypted data that should be evaluated with the same function f. Batching increases the

Algorithm 13 BGV.KeyGen

```
Require:
  ${params_j\}_{j=0}^{L}$
Ensure: $(sk, pk)$                          // private and public key pair
  for $j = L$ .. 0 do
    $\mathbf{t}_j \xleftarrow{s} \chi^n$
    $\mathbf{s}_j \leftarrow (1, \mathbf{t}_j[0], ..., \mathbf{t}_j[n-1]) \in \mathcal{R}_{Q_j}^{n+1}$
    $\mathbf{B}_j \xleftarrow{s} \mathcal{U}(\mathcal{R}_{Q_j}^{M_j \times n})$
    $\mathbf{e}_j \xleftarrow{s} \chi^{M_j}$
    $\mathbf{b}_j \leftarrow \mathbf{B}_j \mathbf{t}_j + 2\mathbf{e}_j$
    $\mathbf{A}_j \leftarrow (\mathbf{b}_j \,||\, -\mathbf{B}_j) \in \mathcal{R}_{Q_j}^{M_j \times (n+1)}$
  end for
  $sk \leftarrow \{\mathbf{s}_j\}_{j=0}^{L}$
  $pk \leftarrow \{\mathbf{A}_j\}_{j=0}^{L}$
```

Algorithm 14 BGV.Enc

```
Require:
  $params$
  $pk$
  $m \in \mathcal{R}_2$
Ensure: $\mathbf{c}$
  $\mathbf{m} \leftarrow (m, 0, ..., 0) \in \mathcal{R}_{Q_L}^{n+1}$
  $\mathbf{r} \xleftarrow{s} \mathcal{U}\left(\mathcal{R}_2^{M_L}\right)$
  $\mathbf{c} \leftarrow \mathbf{m} + \mathbf{r}^T \mathbf{A}_L \in \mathcal{R}_{Q_L}^{n+1}$
```

performance a lot for some types of functions f. One example, given by Brakerski et al. [21], is when deciding whether a word is present or not in a text. If batching the words to one text block, instead of having one block for each word in the text, we do not have to do a lot of OR operations. To batch messages, Brakerski et al. [21] suggest replacing the plaintext space $\mathcal{R}_2$ with a ring $\mathcal{R}_p$, where p is a prime. The BGV scheme then needs some other modifications as well, see the paper by Brakerski et al. [21] for further details.

A.2.3 B/FV Scheme

The BGV scheme was further developed by Fan and Vercauteren to the so-called B/FV scheme [22], and this scheme is also implemented in many open-source FHE libraries. We refer to their paper for more details about how this scheme works.

Algorithm 15 BGV.Dec

```
Require:
    $params$
    $sk$
    $\mathbf{c}$
    $j$                                  // level of $\mathbf{c}$
Ensure: $m$
    $m \leftarrow (\langle \mathbf{c}, \mathbf{s}_j \rangle \mod Q_j) \mod 2$
```

Kim et al. [23] compare optimized versions of the BGV and the B/FV schemes. The conclusions are that the noise grows slower in B/FV and that B/FV is faster for small plaintexts, but that BGV is faster for medium and large plaintexts.

A.3 Third Generation

The third generation of FHE schemes started when the GSW scheme [5] was published in 2013 [4]. Just as the schemes in the second generation, third-generation schemes are based on the LWE and RLWE problems. However, an important difference is that the GSW scheme has a new approach for performing homomorphic operations, using a method called *approximate eigenvector method* instead [4].

In this section, we will first briefly present how the GSW scheme encrypts and decrypts since all schemes in the third generation more or less are built on the basics of these techniques. Then we will describe the FHEW and the TFHE schemes, which also belong to the third-generation schemes but have taken care of some of the drawbacks of the GSW scheme.

A.3.1 GSW Scheme

A brief presentation of the GSW scheme, based on the simplified GSW scheme from Marcolla et al. [4], can be found below.

Firstly, choose a random private key $\mathbf{s} = (1, s_1, ..., s_{n-1}) \in \mathbb{Z}_q^n$, and let $A \in \mathbb{Z}_q^{n\times n}$ be the public key, chosen so that $A \cdot \mathbf{s} = \mathbf{e} \approx 0$.

The encryption C of a message $m \in \mathbb{Z}_q$ is then computed as $C = mI_n + RA$, where I_n is the identity matrix and $R \in \mathbb{Z}_2^{n\times n}$ is randomly chosen matrix.

To decrypt, one computes $C\mathbf{s} = mI_n\mathbf{s} + RA\mathbf{s} = mI_n\mathbf{s} + R\mathbf{e} \approx mI_n\mathbf{s}$, because both $R \in \mathbb{Z}_2^{n\times n}$ and $\mathbf{e}$ are small. Since, $mI_n\mathbf{s} = (ms_0, ..., ms_{n-1}) = (m, ms_1, ..., ms_{n-1})$, one simply outputs the first element of the vector $C\mathbf{s}$ as the decryption.

Algorithm 16 BGV.Eval

```
Require:
  $params$
  $pk$
  $f$                                          // circuit with add. and mult. gates
  $(\mathbf{c}_0, ..., \mathbf{c}_{l-1})$
Ensure: $\mathbf{c}' = \mathsf{Enc}(f(\mathsf{Dec}(\mathbf{c}_0), ..., \mathsf{Dec}(\mathbf{c}_{l-1})))$
  Function add$(pk, \mathbf{c}_0, \mathbf{c}_1)$
    $\mathbf{c}_2 \leftarrow \mathbf{c}_0 + \mathbf{c}_1 \mod Q_j$
    Return $\mathsf{BGV.Eval.Refresh}(\mathbf{c}_2, Q_j, Q_{j-1})$
  End Function
  Function mult$(pk, \mathbf{c}_0, \mathbf{c}_1)$
    $\mathbf{c}_2 \leftarrow$ coefficient vector of $\langle \mathbf{c}_0 \otimes \mathbf{c}_1, \mathbf{x} \otimes \mathbf{x} \rangle$    // $\otimes$ is the tensor product
    Return $\mathsf{BGV.Eval.Refresh}(\mathbf{c}_2, Q_j, Q_{j-1})$
  End Function
  Function refresh$(\mathbf{c}, Q_j, Q_{j-1})$
    $\mathbf{c}_0 \leftarrow \mathsf{SwitchKey}(\mathbf{c}, Q_j)$            // switch key to $\mathbf{s}_{j-1}$
    $\mathbf{c}_1 \leftarrow \mathsf{SwitchMod}(\mathbf{c}_0, Q_j, Q_{j-1})$     // switch mod to $Q_{j-1}$
  End Function
  Use add() and mult() to compute the circuit $f$ on $\mathbf{c}_0, ..., \mathbf{c}_{l-1}$ and output
  the result to $\mathbf{c}'$.
```

The error when performing homomorphic computations grows slower in the GSW scheme than in second-generation schemes [4]. However, the ciphertexts are large compared to the plaintext, leading to high communication costs, and the time complexity of homomorphic operations is quite high.

A.3.2 FHEW Scheme

Ducas and Micciancio [24] propose some improvements to the GSW scheme, calling the new scheme FHEW. It is provided with some new techniques to achieve fast bootstrapping. One is that it provides a method for homomorphically computing NAND with a very low noise growth. NAND is functionally complete, and therefore, any function can be represented as a circuit of NAND gates. In FHEW, a small refresh is performed on each output of a gate. We will now briefly describe how the NAND operation is performed on ciphertexts.

The FHEW scheme from Ducas and Micciancio [24] encrypts with standard Regev LWE encryption (see (2.1) and Definition 2.17). Let $\mathsf{LWE}^{t/q}_{\mathbf{s}}(m, E) \subset \mathbb{Z}_q^{n+1}$ denote the set of LWE encryptions $c = (\mathbf{a}, b)$ of the message $m \in \{0, 1\}$ under the private key $\mathbf{s} \in \mathbb{Z}_q^n$ such that the absolute value of the noise of ciphertext c is less than E. Then, the homomorphic NAND operation described by Ducas and Micciancio [24] is defined as:

$$\begin{aligned}\text{HomNAND}: \ \mathsf{LWE}_{\mathbf{s}}^{4/q}(m_0, q/16) \times \mathsf{LWE}_{\mathbf{s}}^{4/q}(m_1, q/16) &\to \mathsf{LWE}_{\mathbf{s}}^{2/q}(m_0 \barwedge m_1, q/4) \\ ((\mathbf{a_0}, b_0), (\mathbf{a_1}, b_1)) &\mapsto \left(-\mathbf{a_0} - \mathbf{a_1}, \frac{5q}{8} - b_0 - b_1\right),\end{aligned}$$

where $\barwedge$ denotes the NAND operator. We see that one then needs to transform the output $c \in \mathsf{LWE}_{\mathbf{s}}^{2/q}(m, q/4)$ to a ciphertext $c' \in \mathsf{LWE}_{\mathbf{s}}^{4/q}(m, q/16)$ again. Using Gentry's bootstrapping technique [3] as usual, one can do so by homomorphically decrypting c under an encryption corresponding to $\mathsf{LWE}_{\mathbf{s}}^{4/q}(m, q/16)$.

As noted by Ducas and Micciancio [24], the NAND operation itself is very fast – what takes time is the bootstrapping afterwards.

A.3.3 TFHE Scheme

Chillotti, Gama, Georgieva, and Izabachène [9] improve the FHEW scheme further by using the real torus $\mathbb{T}$ in different ways. They call the new technique TFHE, where 'T' stands for torus. In the paper, three versions of the TFHE scheme are proposed – the TLWE scheme, based on a generalization of the LWE problem for the torus; the TRLWE scheme, which is the ring version of TLWE; and the TRGSW scheme, which is an improvement of the GSW scheme, based on rings and a torus.

In Algorithm 17 – 21, we present how the TRLWE scheme works. One can simply switch between TRLWE and TLWE by just changing $\mathcal{T}$ to the real torus $\mathbb{T}$, changing R to $\mathbb{Z}$ and letting χ be $\{0,1\}$-bounded instead of B-bounded [4].

Algorithm 17 TRLWE.Setup

```
Require:
  $\lambda$                                  // security parameter
Ensure: $params$                             // a tuple of parameters
  $k \leftarrow k(\lambda) \in \mathbb{N}^*$
  $N \leftarrow 2^k$                          // degree of ring
  $\mathcal{R} \leftarrow \mathbb{Z}[X]/(X^N+1)$
  $\mathcal{T} \leftarrow \mathbb{T}[X]/(X^N+1)$        // $= \mathbb{R}[X]/(X^N+1) \mod 1$
  $\mathcal{R}_2 \leftarrow \mathbb{Z}_2[X]/(X^N+1)$
  $n \leftarrow n(\lambda)$                   // dimension of ring
  $M \leftarrow M(\lambda)$
  $\chi \leftarrow \chi(\lambda)$             // $B$-bounded distribution over $\mathcal{T}$
  $params \leftarrow (\mathcal{R}, \mathcal{T}, \mathcal{R}_2, n, M, \chi)$
```

TRLWE.Setup in Algorithm 17 describes how to set up the parameters used in the scheme. Algorithm 18 then shows how to generate a private and a public key for

Algorithm 18 TRLWE.KeyGen

```
Require:
  $params$
Ensure: $(sk, pk)$                          // private and public key pair
  $\mathbf{s} \overset{s}{\leftarrow} \mathcal{U}(\mathcal{R}_2^n)$                   // choose a random private key
  $A \overset{s}{\leftarrow} \mathcal{U}(T^{M \times n})$
  $\mathbf{e} \overset{s}{\leftarrow} \chi^M$
  $D \leftarrow (A \mid\mid A\mathbf{s} + \mathbf{e}) \in \mathcal{T}^{M \times (n+1)}$
  $sk \leftarrow \mathbf{s}$
  $pk \leftarrow D$
```

Algorithm 19 TRLWE.Enc

```
Require:
  $params$
  $pk$
  $m \in \mathcal{M} \subseteq \mathcal{T}$                      // $\mathcal{M}$ is the message space
Ensure: $\mathbf{c} \in \mathcal{T}^{n+1}$
  $D \leftarrow pk$
  $\mathbf{r} \overset{s}{\leftarrow} \mathcal{U}\left(\mathcal{R}_2^M\right)$
  $\mathbf{m} \leftarrow (0, ..., 0, m) \in \mathcal{T}^{n+1}$
  $\mathbf{c} \leftarrow \mathbf{r}^T D + \mathbf{m} \in \mathcal{T}^{n+1}$
```

the scheme. Similar to the BGV scheme, a matrix with M rows is generated as the public key, and then the encryption in Algorithm 19 chooses some random rows of this matrix to form a ciphertext.

We will now explain why the decryption in Algorithm 20 works. In the key generation (Algorithm 18), we compute

$$D = (A \mid\mid A\mathbf{s} + \mathbf{e}),$$

and in the encryption (Algorithm 19), we compute

$$\mathbf{c} = \mathbf{r}^T D + (0, ..., 0, m).$$

Therefore, for $(\mathbf{a}, b) = \mathbf{c}$ in the decryption (Algorithm 20), we get

$$\begin{aligned} \mathbf{a} &= \mathbf{r}^T A \\ b &= \mathbf{r}^T (A\mathbf{s} + \mathbf{e}) + m. \end{aligned}$$

Algorithm 20 TRLWE.Dec

```
Require:
    $params$
    $sk = \mathbf{s} \in \mathcal{R}_2^n$
    $\mathbf{c} \in \mathcal{T}^{n+1}$
Ensure: $m$
    $\mathcal{T}^n \times \mathcal{T} \ni (\mathbf{a}, b) \leftarrow \mathbf{c}$
    $m \leftarrow \text{round}(b - \langle \mathbf{a}, \mathbf{s} \rangle)$        // round to nearest point in $\mathcal{M} \subseteq \mathcal{T}$
```

Algorithm 21 TRLWE.Eval_lincomb

```
Require:
    $params$
    $\mathbf{c}_0, ..., \mathbf{c}_{p-1} \in \mathcal{T}^{n+1}$
    $f_0, ..., f_{p-1} \in \mathcal{R}$        // coefficients for linear combination
Ensure: $c = \text{Enc}\left(\sum_{i=0}^{p-1} f_i \cdot (\text{Dec}(\mathbf{c}_i)\right)$
    $c \leftarrow \sum_{i=0}^{p-1} f_i \cdot \mathbf{c}_i$
```

This means, that the decryption algorithm outputs

$$\begin{aligned} &b - \langle \mathbf{a}, \mathbf{s} \rangle \\ &= \mathbf{r}^T(A\mathbf{s} + \mathbf{e}) + m - \mathbf{r}^T A\mathbf{s} \\ &= \mathbf{r}^T \mathbf{e} + m \\ &\approx m, \end{aligned}$$

which is the plaintext message.

The evaluation function in Algorithm 21 is simple but can only handle linear combinations of messages. This is a drawback of the TRLWE scheme (and the TLWE scheme). To evaluate a non-linear function on encrypted data, the TRGSW algorithm can be used instead [4].

One advantage of the TFHE scheme is that when bootstrapping, univariate functions can be evaluated at the same time. This is called programmable bootstrapping (PBS), and it means that with just one algorithm, we can both decrease the noise and evaluate a function of the ciphertext. Note that normal bootstrapping can also be seen as programmable bootstrapping with the identity function [7], but usually, this is the only function that can be evaluated while refreshing.

Micciancio and Polyakov [25] compare the FHEW scheme with the TFHE scheme, and the conclusion they draw is that the main performance difference between the schemes mainly is due to the different bootstrapping techniques. FHEW uses AP bootstrapping, while TFHE uses GINX (these are explained more in Section 3.4). This results in TFHE being faster for binary and ternary messages, while FHEW is better for larger secrets. On the other hand, TFHE has a smaller bootstrapping key than FHEW [4].

A.4 Fourth Generation

The youngest generation of the FHE schemes is the fourth one. It was started by Cheon, Kim, Kim, and Song in 2017 [26] when they published a new kind of FHE scheme, nowadays called the CKKS scheme. Since then, a lot of improvements have been suggested, but the basics of the scheme are still the same. We will now describe how it works.

A.4.1 CKKS Scheme

The original CKKS scheme, from Cheon et al. [26], is a levelled fully homomorphic encryption scheme, but a bootstrapping technique was presented later by Cheon et al. [27]. The difference from previous schemes is that it only computes approximate results, allowing some errors in the last decimal places when evaluating functions. We will now describe how the scheme from Cheon et al. [26] works, using the presentation of it from Marcolla et al. [4].

Algorithm 22 – 26 show pseudocode for the encryption scheme, consisting of $\mathcal{E}_{CKKS} = (\mathsf{CKKS.Setup}, \mathsf{CKKS.KeyGen}, \mathsf{CKKS.Enc}, \mathsf{CKKS.Dec}, \mathsf{CKKS.Eval})$.

$\mathcal{U}(S)$ is the uniform distribution of a set S, while $\mathcal{N}_{\mathbb{Z}}^{N}(0, \sigma^2)$ denotes a multi-dimensional discrete Gaussian distribution over $\mathbb{Z}^N$, where each component is sampled from independent discrete Gaussian distributions with variance σ^2.

$\mathcal{Z}O_\rho$ is also a distribution, but over $\{-1, 0, 1\}$, and such that $\mathbb{P}[0] = 1 - \rho$ and $\mathbb{P}[1] = \mathbb{P}[-1] = \rho/2$, where $0 < \rho < 1$.

$\mathrm{HWT}(h, N)$ is a function that returns the set of signed binary vectors in $\{0, \pm 1\}^N$ that has Hamming weight h, i.e. h non-zero elements.

We will now explain why the decryption in Algorithm 25 works. In the key generation (Algorithm 23), we set

$$b = -as + e,$$

so when encrypting (Algorithm 24), the outputted ciphertext is

$$\begin{aligned}\mathbf{c} &= (vb + m + e_0, va + e_1)\\ &= (-vas + ve + m + e_0, va + e_1).\end{aligned}$$

Then, since $\mathbf{s} = (1, s)$, the decryption algorithm outputs

$$\begin{aligned}
&\langle \mathbf{c}, \mathbf{s} \rangle \mod Q_j \\
&\quad = \langle (-vas + ve + m + e_0, va + e_1), (1, s) \rangle \mod Q_j \\
&\quad = -vas + ve + m + e_0 + vas + e_1 s \mod Q_j \\
&\quad = m + ve + e_0 + e_1 s \mod Q_j \\
&\quad \approx m,
\end{aligned}$$

where the approximation at the last line can be made since the error terms are small.

Algorithm 22 CKKS.Setup

```
Require:
  $\lambda$                                        // security parameter
  $L$                                              // number of levels
Ensure: $params$                                   // a tuple of parameters
  $p \leftarrow p(\lambda)$
  $Q_0 \leftarrow Q_0(\lambda)$
  for $j = 1 .. L$ do
      $Q_j \leftarrow p^j Q_0$                     // level($\mathbf{c}$) $= j \implies \mathbf{c} \in \mathcal{R}^2_{Q_j}$
  end for
  $k \leftarrow k(\lambda, Q_L) \in \mathbb{N}^*$
  $N \leftarrow 2^k$
  $\mathcal{R} \leftarrow \mathbb{Z}[X]/(X^N + 1)$
  $t \leftarrow t(\lambda, Q_L)$                   // $t \in \mathbb{Z}$ is for KeyGen
  $h \leftarrow h(\lambda, Q_L)$                   // Hamming weight for private key
  $\sigma \leftarrow \sigma(\lambda, Q_L)$         // variance for sampled errors
  $params \leftarrow (\{Q_j\}_{j=0}^{L}, \mathcal{R}, N, t, h, \sigma)$
```

One last note about CKKS is that there are some attacks on it, for example from Li and Micciancio [28]. As Marcolla et al. [4] mention, it is possible to extract the private key by just knowing a ciphertext and its corresponding plaintext. Since the error is a linear combination of the components of the key, one can compute the key with just some basic linear algebra.

A.5 Comparison

Marcolla et al. [4] conclude that BGV and B/FV are good choices if working with finite fields and exact modular arithmetic. However, if bootstrapping will be needed, or if non-linear functions need to be evaluated, third and fourth-generation schemes are better instead. TFHE and other third-generation schemes are usually suitable when performing bit-wise operations or evaluating Boolean circuits,

Algorithm 23 CKKS.KeyGen

```
Require:
  $params$
Ensure: $(sk, pk, evk)$                          // private, public, and eval. key
  $\mathcal{R} \ni s \overset{s}{\leftarrow} \mathcal{U}(\mathrm{HWT}(h, N))$       // random vector is coefficients
  $a \overset{s}{\leftarrow} \mathcal{U}(\mathcal{R}_{Q_L})$
  $\mathcal{R} \ni e \overset{s}{\leftarrow} \mathcal{N}_{\mathbb{Z}}^{N}(0, \sigma^2)$          // random vector is coefficients
  $b \leftarrow -as + e \mod Q_L$
  $a' \overset{s}{\leftarrow} \mathcal{U}(\mathcal{R}_{t \cdot Q_L})$
  $\mathcal{R} \ni e' \overset{s}{\leftarrow} \mathcal{N}_{\mathbb{Z}}^{N}(0, \sigma^2)$         // random vector is coefficients
  $b' \leftarrow -a's + e' + ts^2 \mod (t \cdot Q_L)$
  $sk \leftarrow (1, s)$                         // $sk \in \mathcal{R}^2$
  $pk \leftarrow (b, a)$                         // $pk \in \mathcal{R}_{Q_L}^2$
  $evk \leftarrow (b', a')$                      // $evk \in \mathcal{R}_{t \cdot Q_L}^2$
```

Algorithm 24 CKKS.Enc

```
Require:
  $params$
  $pk$
  $m \in \mathcal{R}$
Ensure: $\mathbf{c} \in \mathcal{R}_{Q_L}^2$
  $v \overset{s}{\leftarrow} \mathcal{ZO}(1/2)$
  $\mathcal{R} \ni e_0, e_1 \overset{s}{\leftarrow} \mathcal{N}_{\mathbb{Z}}^{N}(0, \sigma^2)$     // random vector is coefficients
  $(b, a) \leftarrow pk \in \mathcal{R}_{Q_L}^2$
  $\mathbf{c} \leftarrow (vb + m + e_0, va + e_1)$
```

while CKKS is a good choice when doing real number arithmetic. At last, second and fourth-generation schemes are usually good when doing vector or matrix computations, since these schemes are provided with packing techniques.

Algorithm 25 CKKS.Dec

```
Require:
    $params$
    $sk = \mathbf{s} \in \mathcal{R}^2$
    $j$                                   // level of $\mathbf{c}$
    $\mathbf{c} \in \mathcal{R}_{Q_j}^2$
Ensure: $m$
    $m \leftarrow \langle \mathbf{c}, \mathbf{s} \rangle \mod Q_j$        // $m \in \mathcal{R}$
```

Algorithm 26 CKKS.Eval

```
Require:
  $params$
  $evk$
  $\mathbf{c}_0, ..., \mathbf{c}_{p-1}$
  $f$                                   // function to evaluate
Ensure: $\mathbf{c}' = \mathsf{Enc}(f(\mathsf{Dec}(\mathbf{c}_0), ..., \mathsf{Dec}(\mathbf{c}_{p-1})))$

  Function rescale($\mathbf{c}, l, l'$)
    $\mathbf{c}' \leftarrow \left\lfloor \frac{Q_{l'}}{Q_l}\mathbf{c} \right\rceil \mod Q_{l'}$
    Return $\mathbf{c}'$
  End Function

  Function same_level($\mathbf{c}_0, \mathbf{c}_1$)
    $l_0 \leftarrow \text{level}(\mathbf{c}_0)$
    $l_1 \leftarrow \text{level}(\mathbf{c}_1)$
    if $l_0 < l_1$ then
      $\mathbf{c}_0 \leftarrow \text{rescale}(\mathbf{c}_0, l_0, l_1)$
      $l_0 \leftarrow l_1$
    else if $l_0 > l_1$ then
      $\mathbf{c}_1 \leftarrow \text{rescale}(\mathbf{c}_1, l_1, l_0)$
    endif
    Return $(\mathbf{c}_0, \mathbf{c}_1, l_0)$
  End Function

  Function add($\mathbf{c}_0, \mathbf{c}_1, params$)
    $(\mathbf{c}_0, \mathbf{c}_1, l) \leftarrow \text{same\_level}(\mathbf{c}_0, \mathbf{c}_1)$     // $\implies \text{level}(\mathbf{c}_0) = \text{level}(\mathbf{c}_1)$
    $\mathbf{c}_{\text{add}} \leftarrow \mathbf{c}_0 + \mathbf{c}_1 \mod Q_l$
    Return $\mathbf{c}_{\text{add}}$
  End Function

  Function mult($\mathbf{c}_0, \mathbf{c}_1, params, evk$)
    $(\mathbf{c}_0, \mathbf{c}_1, l) \leftarrow \text{same\_level}(\mathbf{c}_0, \mathbf{c}_1)$
    $(b_0, a_0) \leftarrow \mathbf{c}_0$
    $(b_1, a_1) \leftarrow \mathbf{c}_1$
    $(d_0, d_1, d_2) \leftarrow (b_0 b_1, a_0 b_1 + a_1 b_0, a_0 a_1)) \mod Q_l$
    $\mathbf{c}_{\text{mult}} = (d_0, d_1) + \lfloor t^{-1} \cdot d_1 \cdot evk \rceil \mod Q_l$
    Return $\mathbf{c}_{\text{mult}}$
  End Function

  Use add() and mult() to compute the circuit $f$ on $\mathbf{c}_0, ..., \mathbf{c}_{p-1}$ and output
  the result to $\mathbf{c}'$.
```

Appendix B
Figures

On the following pages of this appendix, there are plots of the results in Chapter 6.

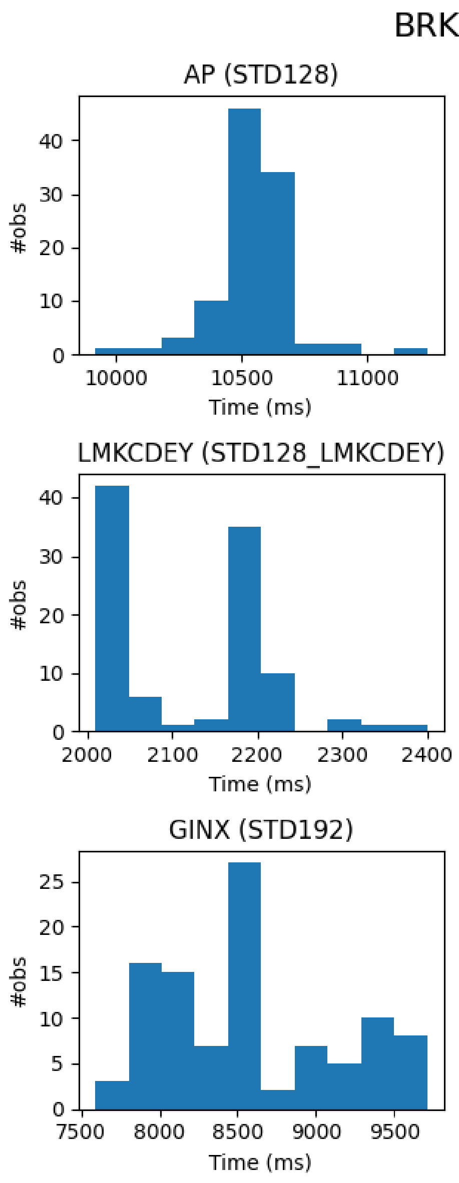


Figure B.1: Time distributions for different `OpenFHE` algorithms and different parameter sets when doing Test S1 (key generation) 100 times (see Chapter 6).

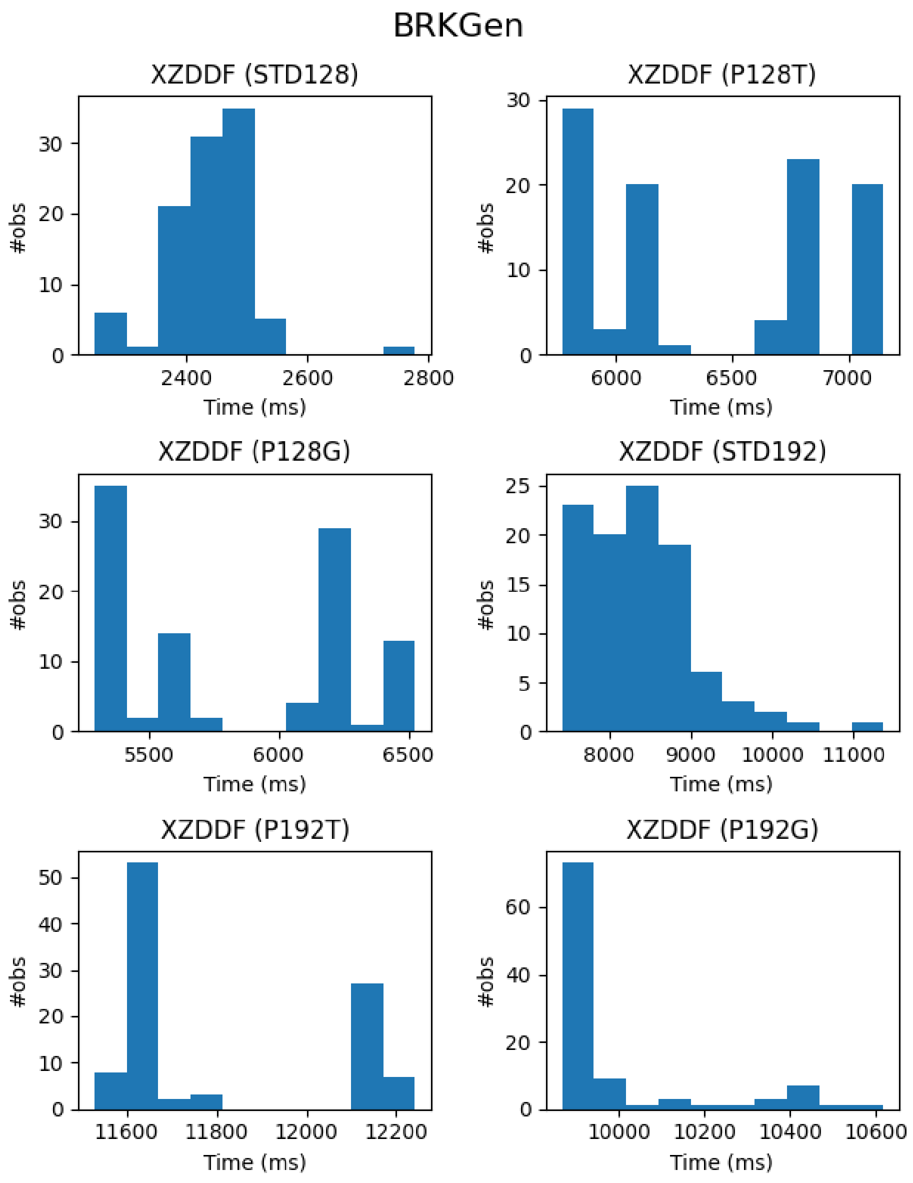


Figure B.2: Time distributions for XZDDF with different parameter sets when doing Test S1 (key generation) 100 times (see Chapter 6).

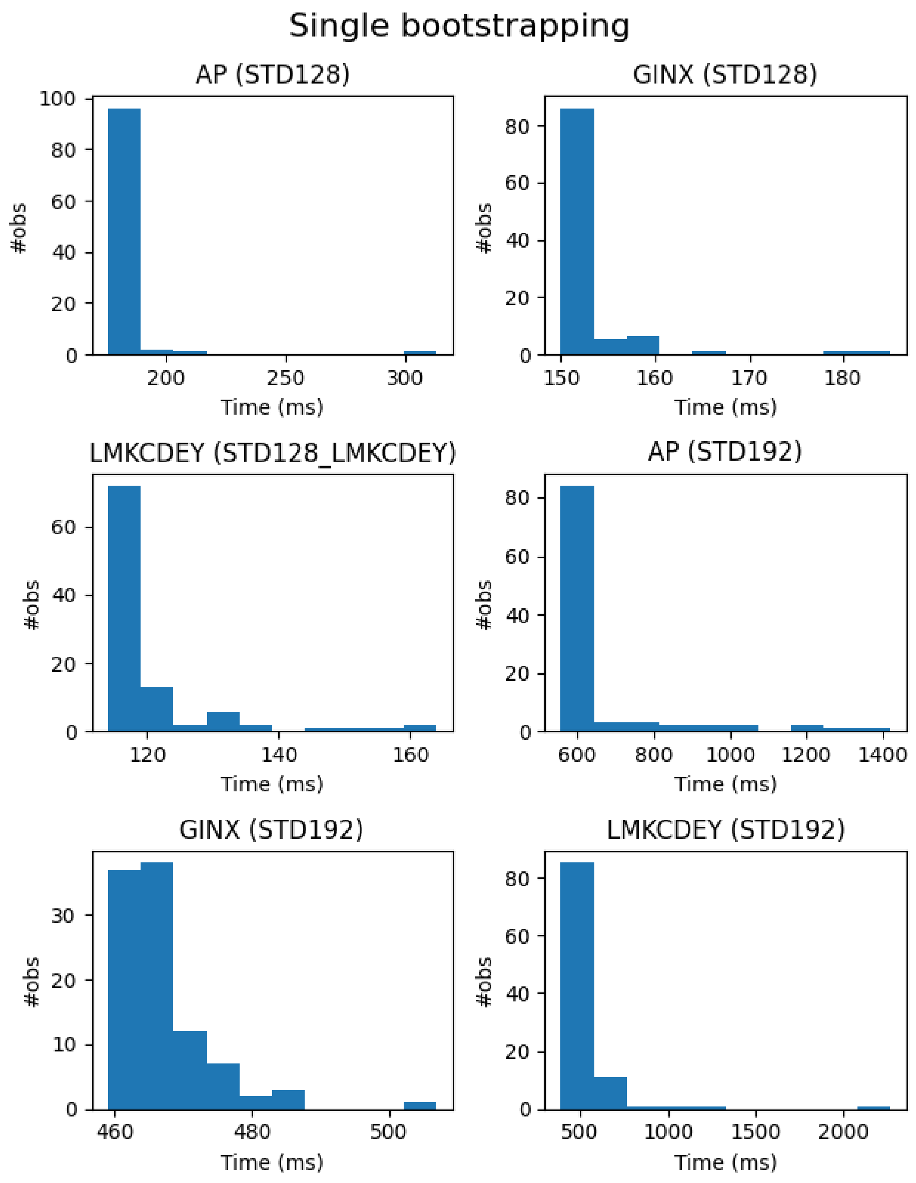


Figure B.3: Time distributions for different `OpenFHE` algorithms and different parameter sets when doing Test S2 (single bootstrapping) 100 times (see Chapter 6).

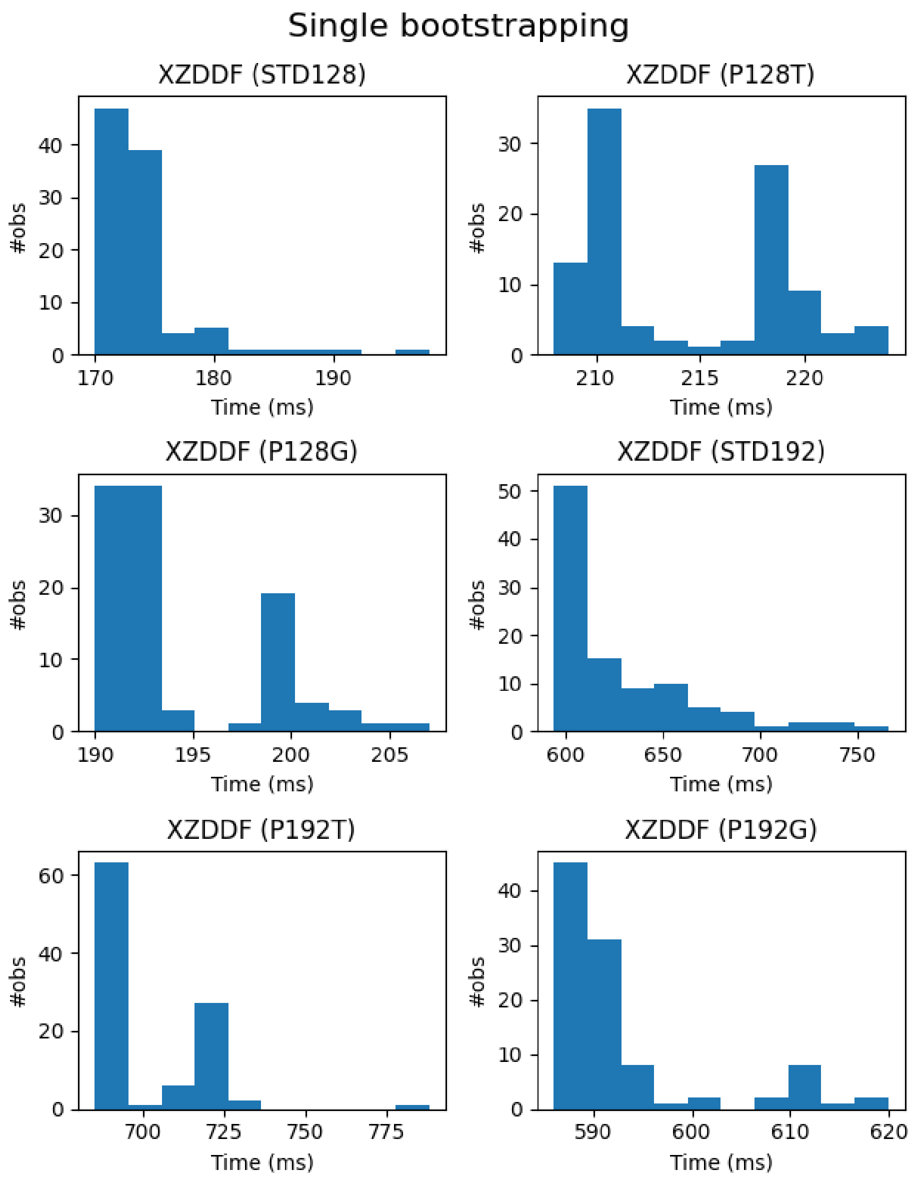


Figure B.4: Time distributions for XZDDF with different parameter sets when doing Test S2 (single bootstrapping) 100 times (see Chapter 6).

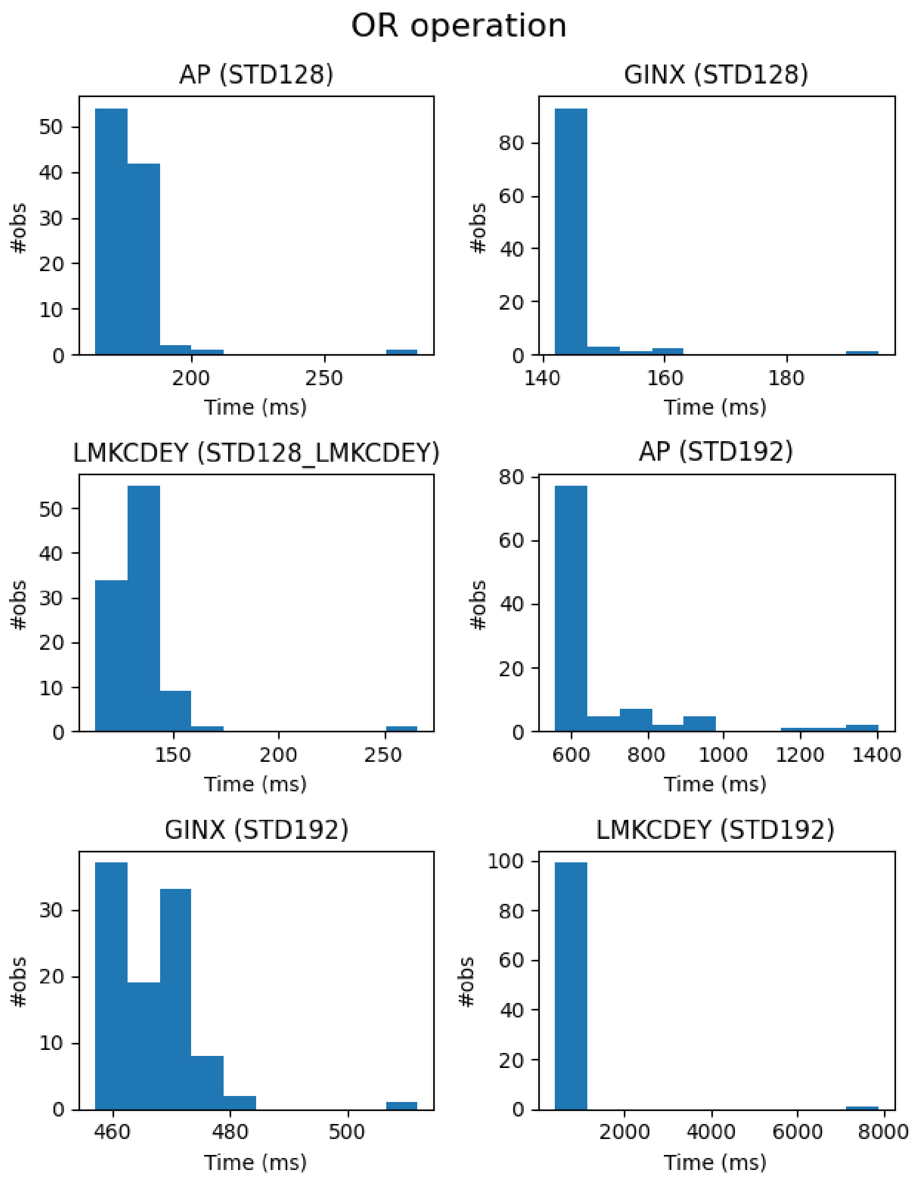


Figure B.5: Time distributions for different OpenFHE algorithms and different parameter sets when doing Test S3 (OR operation) 100 times (see Chapter 6).

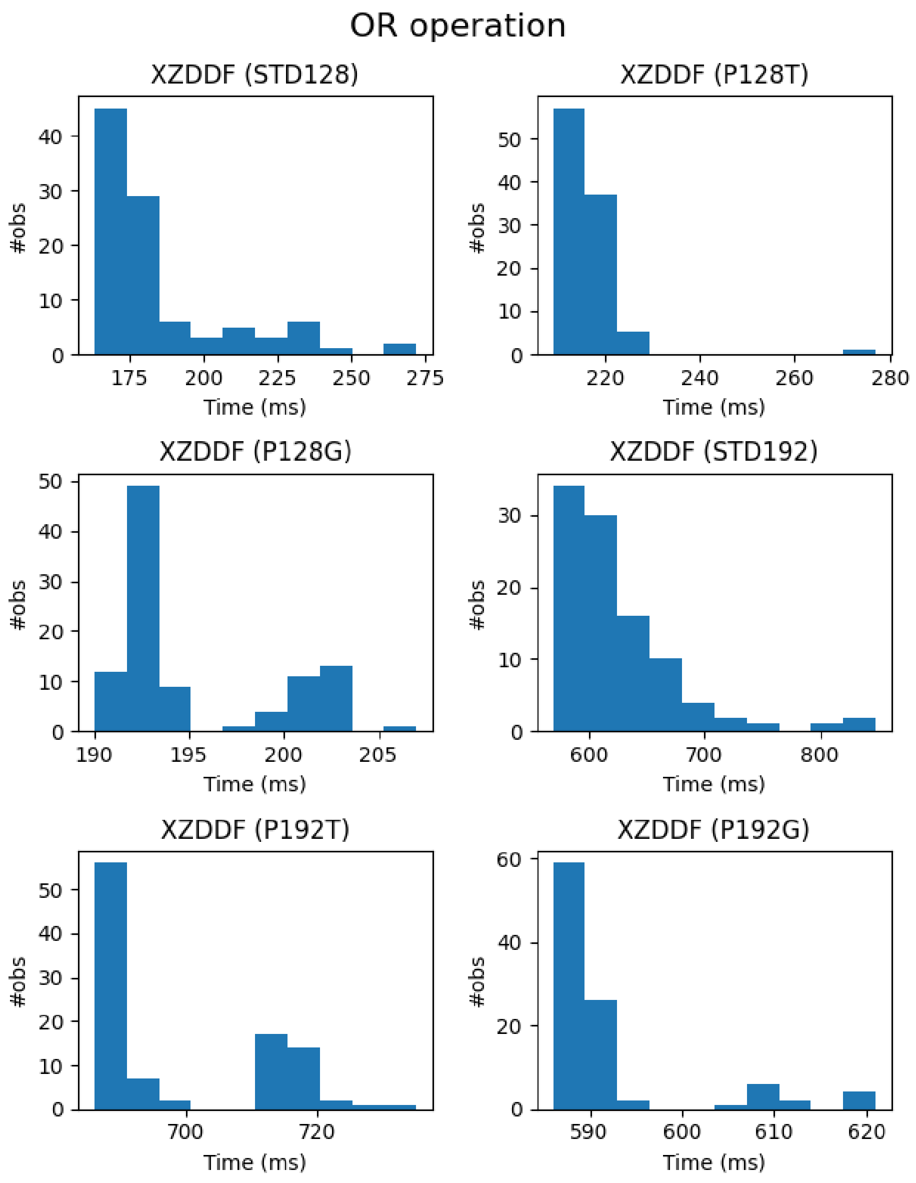


Figure B.6: Time distributions for XZDDF with different parameter sets when doing Test S3 (OR operation) 100 times (see Chapter 6).

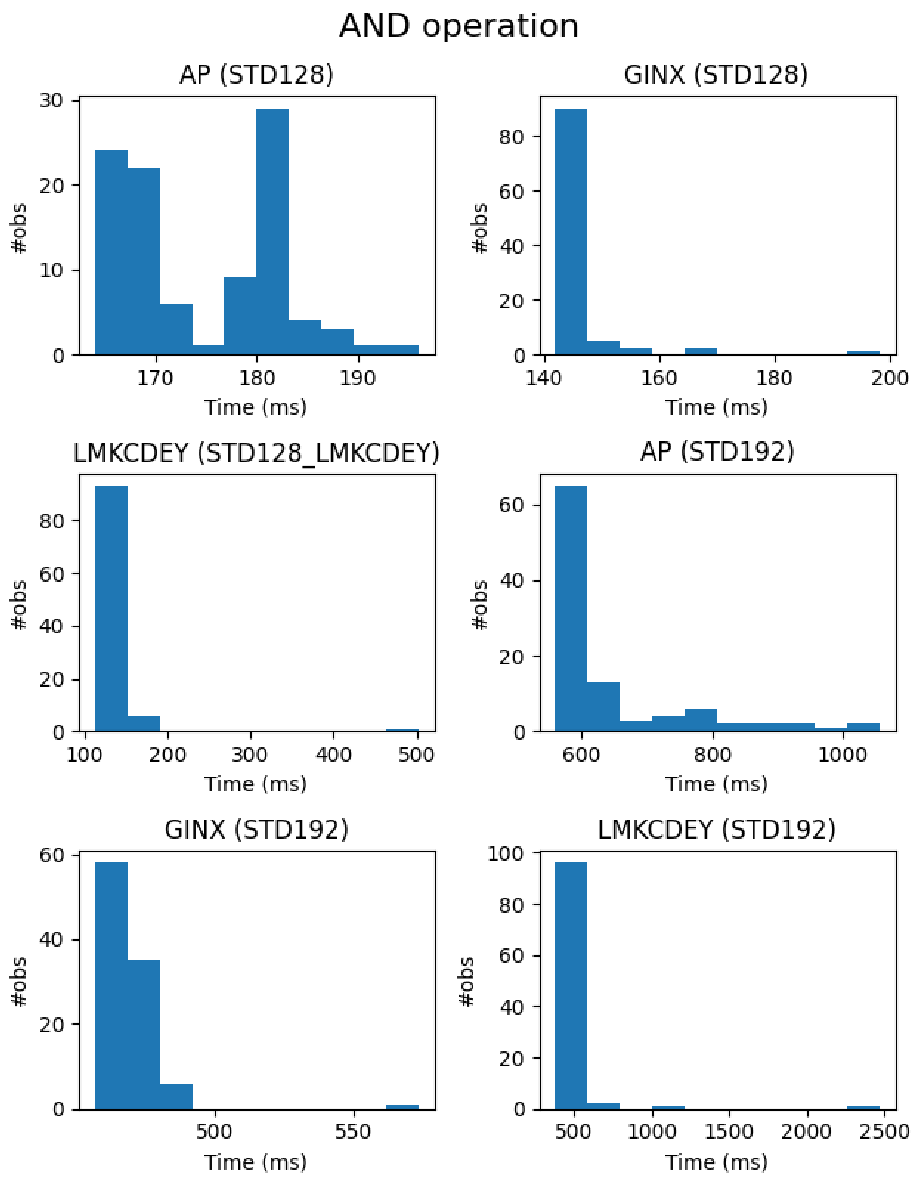


Figure B.7: Time distributions for different `OpenFHE` algorithms and different parameter sets when doing Test S4 (AND operation) 100 times (see Chapter 6).

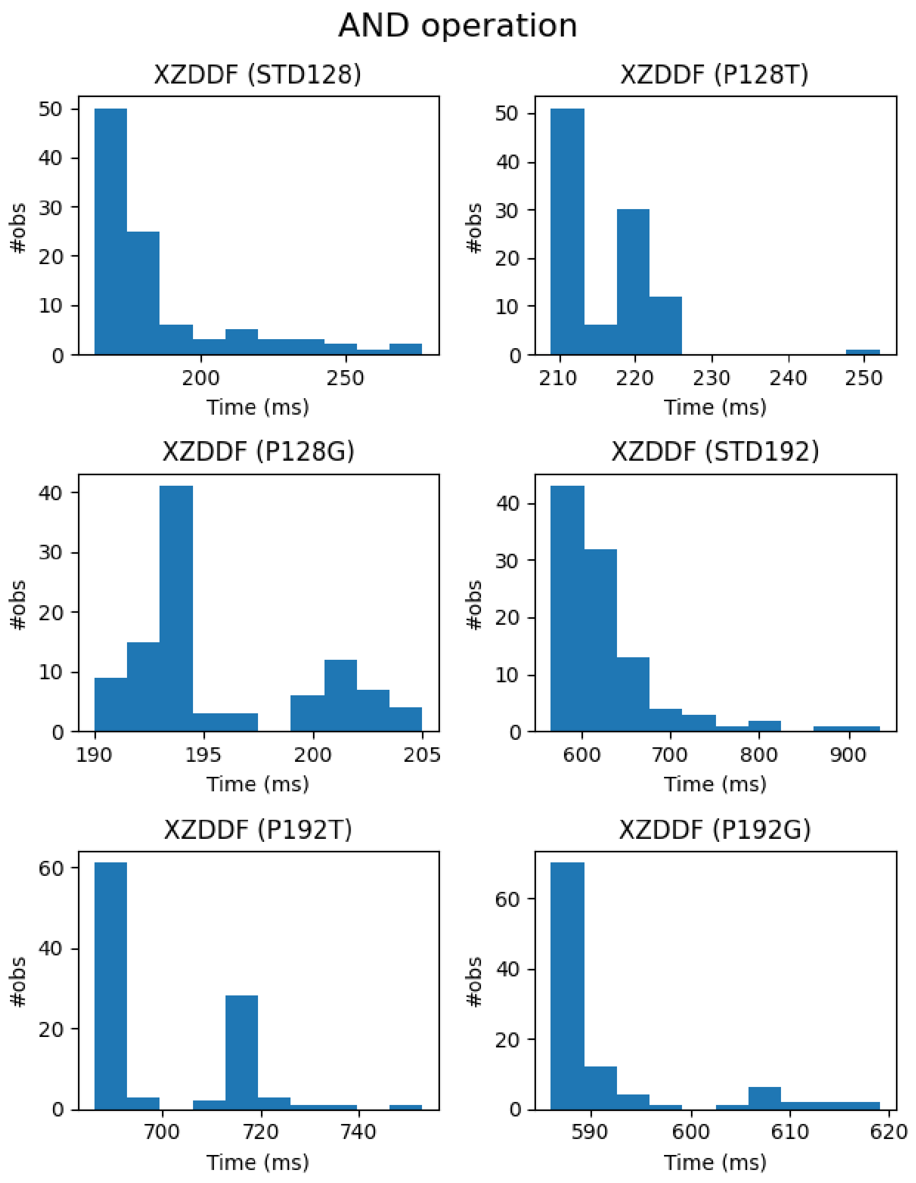


Figure B.8: Time distributions for XZDDF with different parameter sets when doing Test S4 (AND operation) 100 times (see Chapter 6).

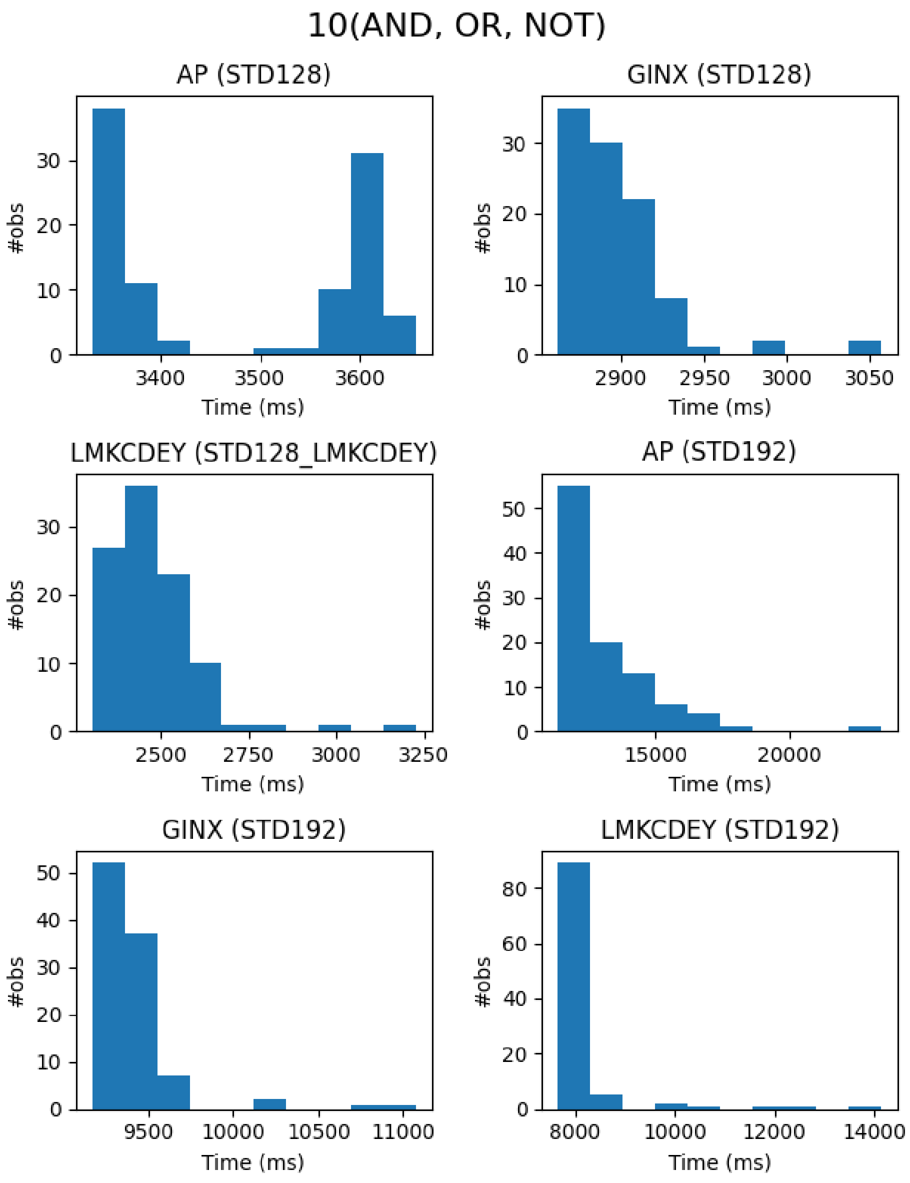


Figure B.9: Time distributions for different `OpenFHE` algorithms and different parameter sets when doing Test B2 (10 AND, OR, and NOT operations) 100 times (see Chapter 6).

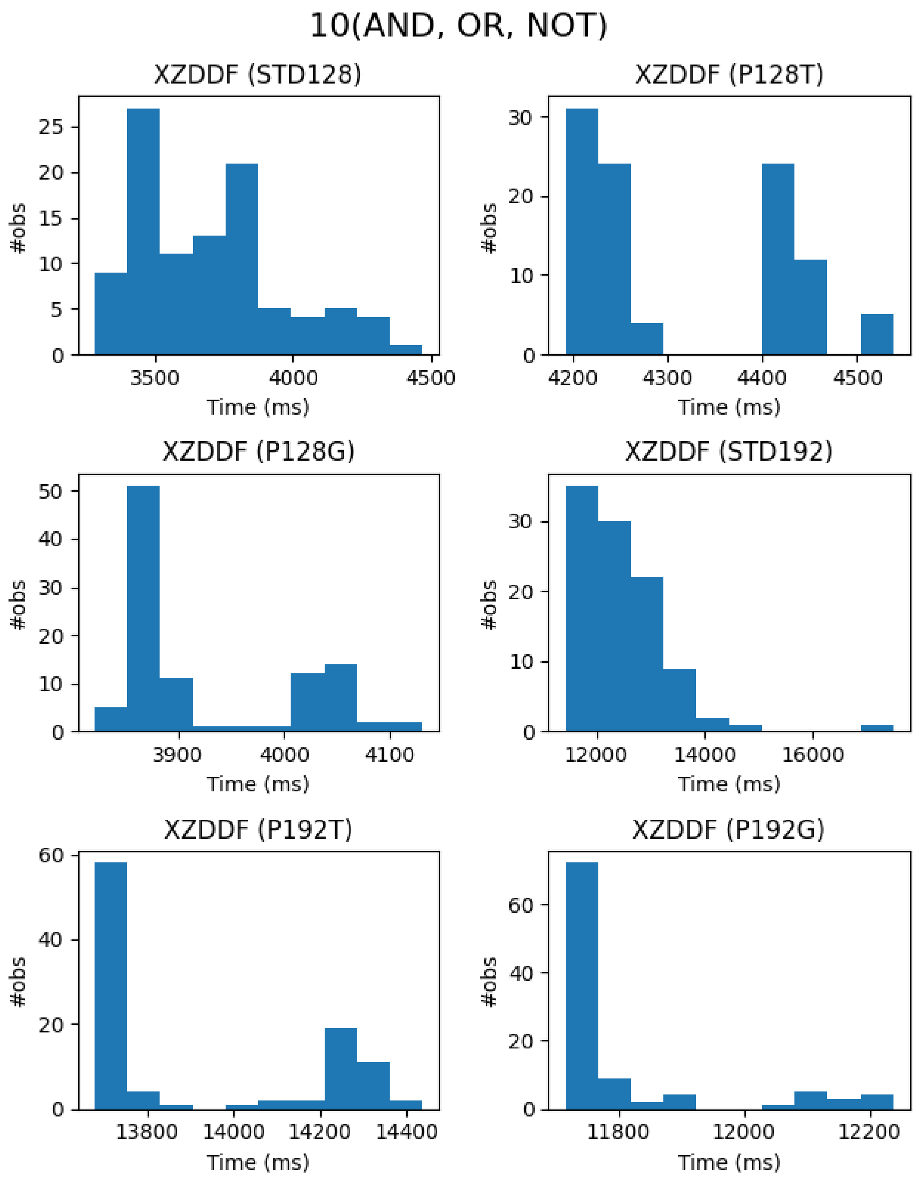


Figure B.10: Time distributions for XZDDF with different parameter sets when doing Test B2 (10 AND, OR, and NOT operations) 100 times (see Chapter 6).

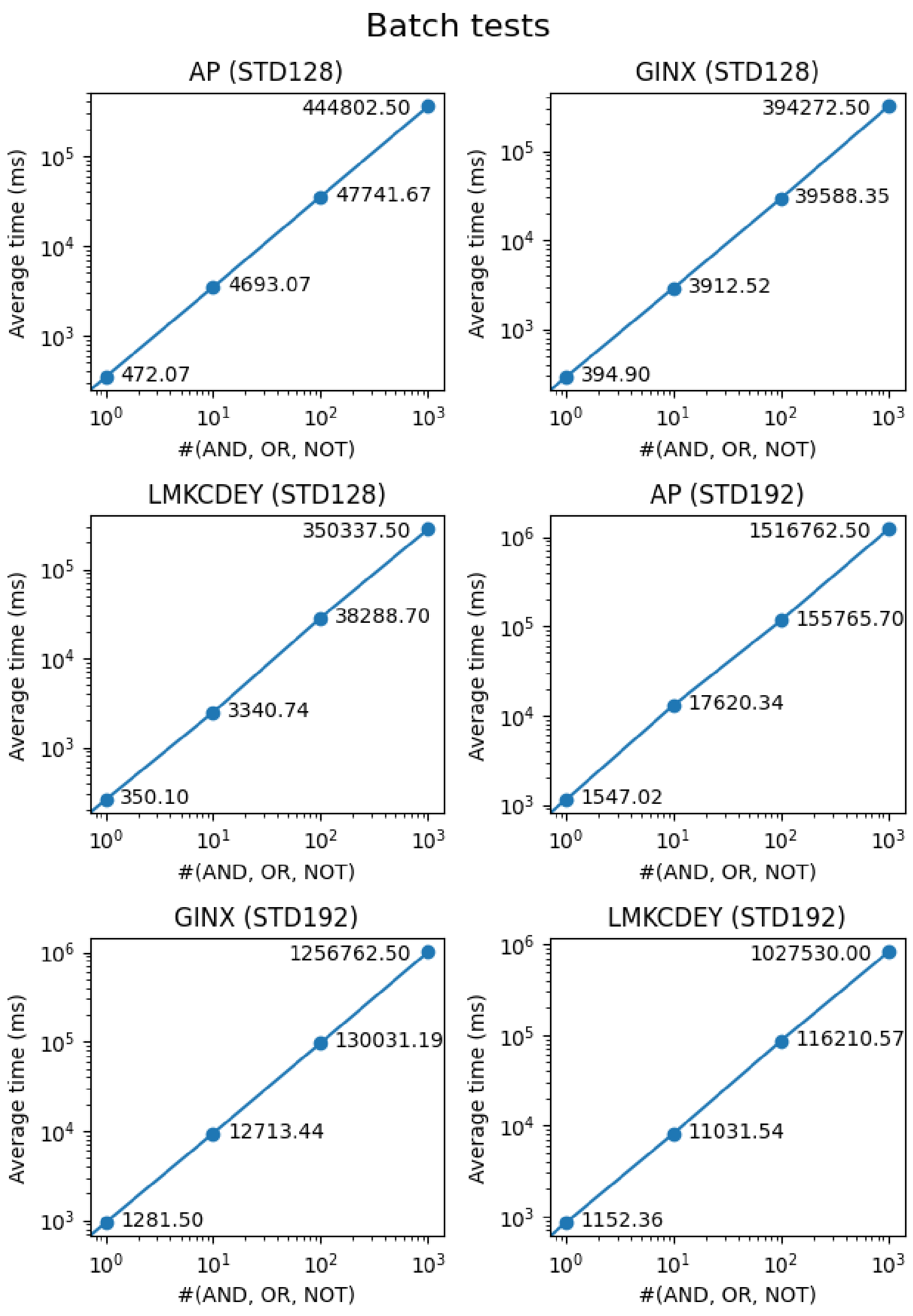


Figure B.11: Log-log plots of the average execution times for different `OpenFHE` algorithms and different parameter sets when doing the batch tests B1–B4 in Chapter 6, consisting of x AND, OR, and NOT operations.

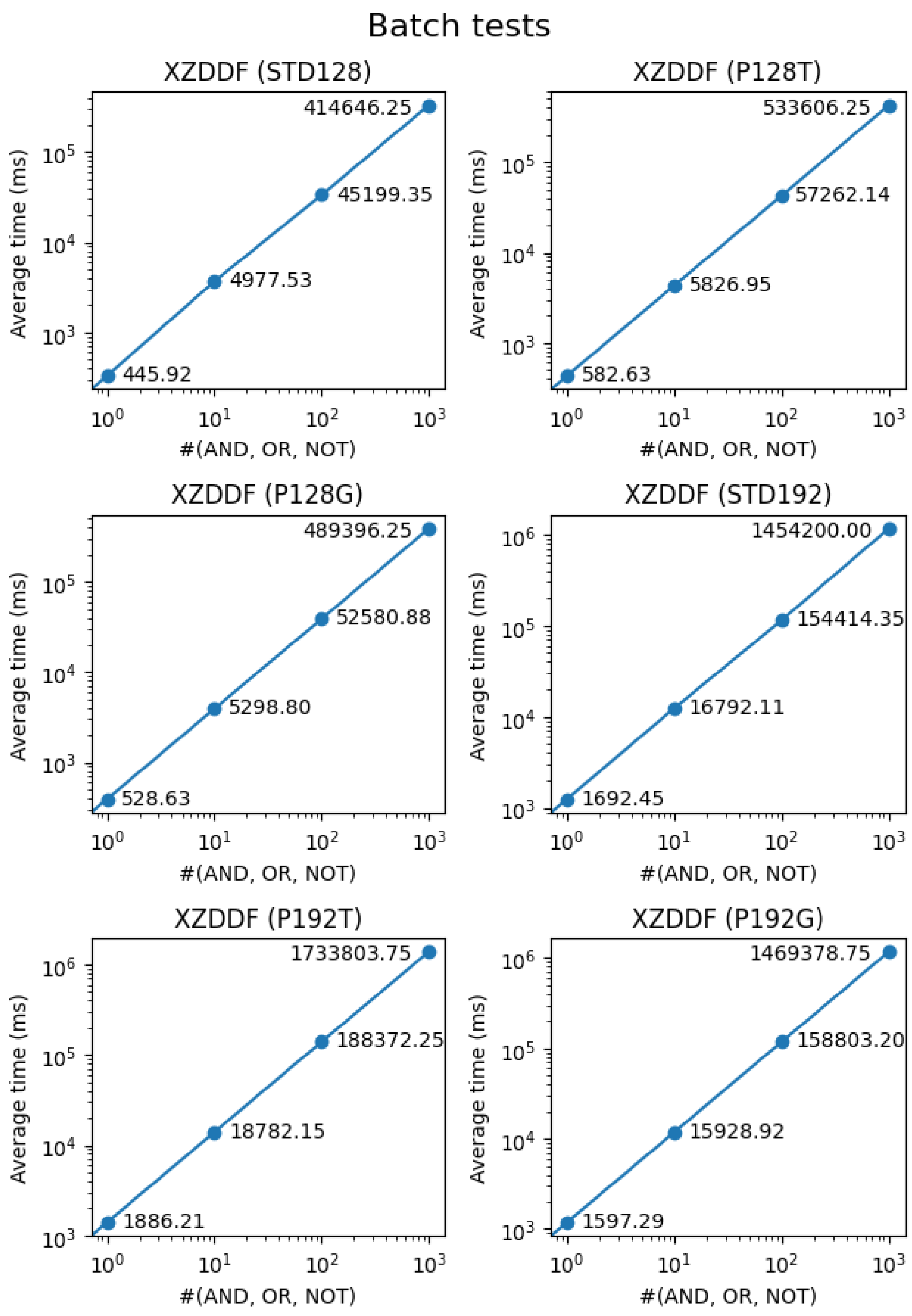


Figure B.12: Log-log plots of the average execution times for XZDDF with different parameter sets when doing the batch tests B1–B4 in Chapter 6, consisting of x AND, OR, and NOT operations.