FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Secure machine learning via homomorphic encryption

Bernardo Ramalho

Mestrado em Engenharia Informática e Computação

Supervisor: Manuel Barbosa

Second Supervisor: Alberto Pedrouzo

July 24, 2023

Secure machine learning via homomorphic encryption

Bernardo Ramalho

Mestrado em Engenharia Informática e Computação

July 24, 2023

Abstract

Cloud computing [15] is becoming more popular than ever with the growth of the internet because it lowers costs by dynamically scaling computing resources on demand. As such, ensuring that privacy is maintained while the data is being processed and used is a crucial enabler for moving many security-critical applications to the Cloud. They may be the key to unlocking new socially-relevant applications and business opportunities.
Homomorphic encryption [1] (HE) is a form of encryption that enables operations over encrypted data and, therefore, ensures privacy at every step. The main problem being it is still not efficient enough, and it does not allow to efficiently use all types of inputs. However, in very specific cases, homomorphic encryption can be used effectively as a subcomponent in other cryptographic protocols, such as electronic voting, secure multiparty computation, etc.
Currently, there is a lack of investigation done in how smaller components and tasks can be optimized. By tackling this efficiency problem, it will be possible to convert new and more powerful algorithms to use HE.
To do that, this dissertation aims to explore how existing homomorphic encryption schemes (in particular, BFV [2]) can be optimized to make simpler machine-learning tasks as fast as possible. We will begin by choosing some concrete machine-learning primitives that have already been explored with HE but have not been fully optimized. Those will then be optimized either by changing the algorithm or by manipulating the scheme's parameters. In the end, we will have a set of optimized encrypted machine-learning primitives, time-wise and resource-wise, than what is currently available. This will allow more complex implementations to be created and used.

Acknowledgments

I want to thank all those who were essential to these last six months and those that were a big part of the last six years.

Manuel and Alberto for their confidence in me and all their help. I would have never learned so much if it wasn't for their patience and dedication.

Rosário for being with me in my best (and worst) projects. This degree wouldn't have been half as unique without you.

Leonor, Clara, and Chica for being the amazing girl bosses they are. I'm so thankful for the opportunity to watch you grow as you did.

João Carlos and Rita Mafalda for all the countless 50 minutes breaks that made the days easier. You made my last semester the most wonderful of them all, and for that, I'll be forever grateful.

Guedes and Cátia for always making me feel comfortable and welcomed. All the moments we had taught me that, with you, it will always be worth it.

My roommates, Tuxa and Maria, for giving me the best home I could have asked for. Without it, I would have never become the person I am today.

My best friend, the most wonderful person FEUP gave me, Neiva, for always being there for me. It would be impossible to have made it this far without our "Varanda Sessions".

And lastly, my family. For the sacrifices and hard work, they had to go through to give me all the conditions I needed and more. I am (and always will be) grateful for all they did; this work is dedicated to them.

Bernardo Ramalho

"A vida é dura para quem é mole"

Afonso Eurico Santos

Contents

List of Tables

Chapter 1

Introduction

1.1 Context

Machine learning has become a significant part of most digital systems. These systems require enormous amounts of data to train their models, which can become a problem when the data is supposed to be private. This is aggravated by the fact that sometimes the data must be outsourced or used by non-fully trusted parties, as is the case with cloud computing.

So, it is imperative that data is protected from malicious actors in all process stages, be it during transit or processing.

Conventional cryptographic techniques are already the go-to method to protect data during transit. Unfortunately, for this data to be used by the machine learning algorithms, it has to be decrypted. This produces a point of weakness that can be exploited.

Homomorphic encryption (HE) appears as a very promising solution to this problem since it enables operations over encrypted data. The idea of homomorphic encryption started in 1978 [18], but the first plausible construction was only made in 2009 by Craig Gentry [9]. This construction was based on lattice-based cryptography and supported both additions and multiplication on ciphertexts.

The main principle of lattice-based HE schemes is the introduction of noise in the ciphertexts. This noise is incremented each time a homomorphic operation is made over the ciphertext.

This creates a threshold on how many homomorphic operations can be made over the same ciphertext since the decryption will not be correct if there is too much noise.

To combat this, FHE (Fully Homomorphic Encryption) schemes implement a bootstrapping technique. Bootstrapping is based on homomorphically evaluating the decryption circuit of the encryption scheme. This makes it possible to refresh the noise inside ciphertexts, which enables them to perform an unlimited number of consecutive operations.

While Gentry showed that FHE schemes were theoretically possible, there was no practical implementation of a Fully Homomorphic Encryption scheme at that time. Since the release of that paper, a series of works have focused on continuously improving and optimizing the efficiency of these schemes

In 2011, the first second-generation FHE scheme was created, the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [3]. Second-generation FHE schemes are based on the hardness of the Ring Learning With Errors (RLWE) problem. They are faster and have a much slower noise growth than first-generation schemes. These schemes were fast enough to be used by a real-world application without needing bootstrapping. Besides BGV, the other notable scheme of this generation is the Brakerski/Fan-Vercauteren (BFV) scheme [2].

In 2013, Craig Gentry, Amit Sahai, and Brent Waters [10] proposed a new cryptosystem with a slower noise growth rate and better efficiency. This also enabled a more efficient bootstrapping technique. This system was then converted into an efficient ring variant, the FHEW scheme. The most notable scheme of this third generation is the TFHE scheme, which is an improvement over FHEW.

We are currently in the fourth generation of fully homomorphic encryption schemas. This generation began with the proposal of Cheon, Kim, Kim and Song (CKKS) in 2016 [4]. The main difference between this scheme from other generations is that CKKS uses approximate arithmetics instead of exact arithmetics.

1.2 Motivation and Objectives

The main problem with homomorphic encryption is that, despite all its improvements, it is still not efficient enough to be used in a large-scale application.

Most research in FHE has focused on showing that it can be used for the execution of some very complex ML computations (e.g., deep neural networks). On the contrary, here we try to answer a slightly different question and wonder how efficient FHE can be when implementing a representative set of widely used but simpler statistical primitives, like the mean or variance, that are the base of many ML algorithms.

By doing this, we hope that homomorphic encryption will become not only viable enough but also highly efficient to be used in large-scale applications, making them truly secure and practical for the end user.

Chapter 2

State of the Art

2.1 Introduction

This chapter outlines the essential homomorphic encryption schemas and the most common libraries that implement them.

The objective is to compare all the existing possibilities and choose the most appropriate library and schema for the problem in question. For that, we also look into some implementations of the most representative machine learning in those libraries.

2.2 Schemas

2.2.1 Technical Background

2.2.1.1 Learning with Errors

Instead of directly working with lattices, the most recent HE schemes base their security on other hardness assumptions related to lattice problems, which are more amenable for practical implementations. One of these examples is the case of the Learning with Errors (LWE) problem [16]. It is proposed as a method to combat the fact that quantum computers pose a threat to existing public-key methods, as they can break existing encryption techniques in a couple of days.

To describe LWE, we first need to define a size parameter n, such that $n \geq 1$, a modulus q, such that $q \geq 2$, a "noise" Probability distribution χ over $\mathbb{Z}_q$ and a secret $\boldsymbol{s} \in \mathbb{Z}_q^n$.

Given a random vector $\boldsymbol{a} \in \mathbb{Z}_q^n$ and a noise $e \in \mathbb{Z}_q$ according to χ, we output $\mathsf{O} = (\boldsymbol{a}, \boldsymbol{a} \cdot \boldsymbol{s} + e)$. The hardness of LWE is described by how hard it is for an algorithm to find the secret $\boldsymbol{s}$ given O. In particular, SearchLWE allows the adversary to see multiple samples, and the goal is to find $\boldsymbol{s}$ given these samples. DecisionLWE is the decision version of LWE where we only have to distinguish between true LWE samples and uniformly random values.

As of today, there is no efficient way to solve this problem.

2.2.1.2 Ring Learning with Errors

Ring Learning with Errors (RLWE) problem [12] is the LWE problem applied to Polynomial Rings. This makes RLWE constructions have a smaller size than LWE constructions.

More formally, a Ring is an algebraic structure with two binary operations. These operations have properties analogous to the properties of addition and multiplication on integers. Still, the rings used for RLWE do not require the existence of a multiplicative inverse.

All the schemes use two rings, the plaintext ring (denoted P in the rest of this dissertation) to represent the space for all messages and the ciphertext ring (denoted C in the rest of this dissertation) to represent the space for all encrypted messages.

2.2.1.3 Some first examples for Plaintext Encoding and Decoding

Since we are using a polynomial ring to represent the plaintext space, we need to encode the message so it is represented over the ring before encrypting.

There are plenty of ways to do this. We will only present two more straightforward ways to give a general overview of how it can work.

The more naive approach is to represent the message as a constant polynomial with the value of the message as the constant term. That is, if we have a integer message $\boldsymbol{m}$ we convert it to a polynomial, denoted as $\boldsymbol{M}(x) = m \cdot x^0 + 0 \cdot x + 0 \cdot x^2 + ... + 0 \cdot x^{n-1}$. Although this method is very simple, it is inefficient since the plaintext coefficients grow as operations are performed on ciphertexts. Decoding for this method is very straightforward. It is only necessary to read the coefficient of the first term of the polynomial that is returned by the decryption.

The second method is known as the integer encoding scheme. The first step for this method is to represent the message in binary representation. We would use the values of the binary representation as the coefficient of the polynomial. Zero would be used for bits that are not used. To decode, we take the coefficient from the polynomial returned by the decryption and convert it from binary to plaintext.

2.2.2 Brakerski-Fan-Vercauteren (BFV)

The Brakerski-Fan-Vercauteren scheme [2] is a second-generation FHE scheme based on the Ring-Learning with Errors (RLWE) problem.

We need to define three parameters to create the P and the C. Those parameters are $n, q, t \in \mathbb{Z}$. q and t are, respectively, the ciphertext and plaintext coefficients and n specifies the ring dimension.

P is defined as $\mathbb{Z}_t[x]/(x^n+1)$ and C is defined as $R_q \times R_q$ where $R_q = \mathbb{Z}_q[x]/(x^n+1)$. P and C can be seen as the set of all polynomials with integer coefficients modulo both t or q, respectively, and (x^n+1) which introduces the maximum degree for the polynomials $n-1$. Since, in real-world applications, q is much larger than t, C will also be larger in size than P. This means that an element in P can be mapped to multiple valid elements in C.

The encryption of a message is based on Regev's public key encryption scheme [17]. In that scheme, the result of encrypting a message $\boldsymbol{m} \in \{0,1\}$ is an integer vector $\boldsymbol{c}$ such that the decryption circuit will be $\langle \boldsymbol{c}, \boldsymbol{s} \rangle = \lfloor q/2 \rceil \cdot m + e + q \cdot I$, where $|e| \leq E$, with $E < q/4$, and $\boldsymbol{I} \in \mathbb{Z}$.

Here, instead of using C, the authors use the fractional ciphertext $\tilde{\mathbf{c}} = \mathbf{c}/q$. Due to this change, the resulting ciphertext will be $\langle \tilde{\mathbf{c}}, \boldsymbol{s} \rangle = m/2 + \tilde{e} + I$, where $|\tilde{e}| \leq E/q$ and $\boldsymbol{I} \in \mathbb{Z}$. As we can see, the plaintext message is put on the Most Significant Bits (MSB), which lets the modulus take any form.

Homomorphic addition is easily derived from the definition of encryption. If we encrypt m_1 and m_2 as c_1 and c_2, respectively, we have that $c_{add} = c_1 + c_2$ encrypts $[m_1 + m_2]_2$.

Homomorphic multiplication has to be done by tensoring the input ciphertexts. Tensoring is a binary operation that concatenates two vectors into a single bigger vector. The representation of the resulting ciphertext is $c_{mult} = 2 \cdot c_1 \otimes c_2$. The decryption of this is done using a tensored secret key since $\langle 2 \cdot c_1 \otimes c_2, \boldsymbol{s} \otimes \boldsymbol{s} \rangle = 2 \cdot \langle c_1, \boldsymbol{s} \rangle \cdot \langle c_2, \boldsymbol{s} \rangle$. In-depth proof of this is provided in the original paper.

Two important properties of BFV are that BFV is a scale-invariant scheme since the size of the modulus is constant, and it does not require any modulus switching (it works with a single modulus for ciphertexts, contrary to BGV).

The main improvement over previous schemes is that BFV boosts low noise growth. With every multiplication, the noise grows linearly instead of quadratically.

2.2.3 Brakerski-Gentry-Vaikuntanathan (BGV)

The Brakerski-Gentry-Vaikuntanathan schema [3], like BFV, is a second-generation FHE schema based on the Ring-Learning with Errors (RLWE) problem.

One of the main differences between BGV and BFV is that BGV is not scale-invariant. This means that there is not only one modulus, but a small set of moduli. Each modulus of the set is associated with a level. As computations are made, the ciphertext moves from a higher to a lower level. The total number of levels is defined by $L \in \mathbb{Z}$, and each encryption level is defined as l, such that, $0 \leq l \leq L$.

Due to this, BGV has a slight difference in the parameters that it needs to create for the P and the C. While n and t continue to be the ring dimension and the plaintext coefficient, $\boldsymbol{q}$ is now, instead of a single value, a set of size L. These parameters are defined as: $n, t \in \mathbb{Z}$ and $\boldsymbol{q} \in \mathbb{Z}^L$.

In BGV, P is defined as $\mathbb{Z}_t[x]/(x^n + 1)$, just like in BFV. C is defined as $R_{q_l} \times R_{q_l}$ where $R_{q_l} = \mathbb{Z}_{q_l}[x]/(x^n + 1)$, where $q_l \in \mathbb{Z}$ represents the modulus for level l. Just like in BFV, $\boldsymbol{q}$ is usually much larger than t, so C will also be larger in size than P.

The other difference is that BGV puts the plaintext in the Least Significant Bits (LSB) while BGV puts it in the MSB.

2.2.4 Cheon-Kim-Kim-Song (CKKS)

As the previous schemas, CKKS [4] is also based on the Ring-Learning with Errors (RLWE) problem. We won't go into much detail about this and the next scheme because their main characteristics and features are not interesting for our use case.

Compared to BGV and BFV, the main difference is that this schema uses approximate rather than exact arithmetic. This means a small error is added to the plaintext message to guarantee the security of hardness assumptions. This is possible because, if e is small enough, it will not have a meaningful impact on the final output. For example, depending on the chosen parameters, this could mean that if we encrypt 4 and 5, then add the ciphertext and decrypt the result, we would obtain 8,99 or 9,01 instead of 9, which BGV and BFV would return.

Since we are looking to make computations that require exact arithmetics, this scheme is not useful for us.

2.2.5 TFHE

As all previous schemas, TFHE [6], or CGGI, is based on the Ring-Learning with Errors (RLWE) problem, and it was proposed as an improvement for the FHEW schema [7]. TFHE is a homomorphic encryption scheme defined over a torus, which is the set of real numbers modulo one, unlike other schemes that are defined over a ring. This makes TFHE more efficient for non-polynomial operations compared to BFV, BGV, and CKKS.

The main difference between a ring and a torus is that the multiplication of two elements defined in the torus does not return a result defined over the torus. But, if we multiply an element defined on the torus with an integer, the result will be defined over the torus.

TFHE has two versions: the leveled version [5], which is faster for small-depth circuits, and the bootstrapped version [6], which was released earlier.

A relevant property of BFV, BGV, and CKKS which is not present in TFHE, is that they enable "slot packing" or encrypted Single Instruction/Multiple Data (SIMD) computations. This means that they can operate over many plaintext values at the same time. So, while TFHE can be very fast with one non-polynomial operation over a value, BFV/BGV/CKKS can be very slow over only a value but have a much faster total amortized runtime, because they can work over many plaintexts at the same time.

2.3 Libraries

2.3.1 Zama

Zama is a set of 4 libraries implemented in Rust:

- Concrete Core – Library that contains a set of low-level primitives which are used to implement FHE programs;
- Concrete Lib – Library that abstracts away the FHE details;

- Concrete ML – Library with machine learning models converted into FHE equivalent;
- Concrete Numpy – Turns Numpy function into FHE equivalent.

Concrete Core is the base library where all primitive homomorphic operations are implemented. This library gives complete freedom over all the parameters and how the operations are performed. It enables us to create our implementation of cryptographic features supported by Concrete on a given hardware.

Concrete Lib is just an abstraction layer of Core, and its target audience is people unfamiliar with HE, which makes it easier to use.

Concrete Numpy is an adaptation of the mathematics python library, Numpy, to HE. It has a massive role in the implementation of Concrete ML.

Concrete ML is a library where various machine learning algorithms are implemented, based on sklearn, a machine learning python library. It has three types of models: Linear Models, Tree-based Models, and Neural Networks.

Linear Models have both regression and classification models. The most interesting ones are Linear/Logistic regression and Support Vector Regression/Classification. In terms of Tree-based models, it has both classification and regression for decision trees and random forests. For Neural Networks, it also has classification and regression, which are based on Torch models.

All these libraries use the bootstrap version of the TFHE scheme, which is open-sourced.

2.3.2 OpenFHE

OpenFHE is a C++ library consisting of several library objects. These objects contain functions, classes, variables, and other data structures that can be use to create lattice cryptography applications.

It has two programming modes, user-friendly and compiler-friendly. The main feature of the user-friendly mode is that the library automatically calls maintenance operations. The compiler-friendly mode, on the other hand, allows for an external compiler to make those decisions.

OpenFHE implements all the encryption schemes explored in chapter 2.2 plus the Ducas-Micciancio scheme [7]. It uses standard LWE ciphertexts modulo q and has adapted the CGGI bootstrapping procedure to work directly on DM ciphertexts. The library also supports RNS variants for improved efficiency, but it does not include noise estimation.

When using OpenFHE, the user must specify the multiplicative depth and a maximum number of additions/key-switching operations. Based on these inputs, the library will select the necessary parameters, such as the number of bits needed for each multiplicative level. However, it is the user's responsibility to ensure that the specified bounds are respected.

The last relevant feature of OpenFHE is that it supports hardware acceleration technologies.

2.3.3 HElib

HElib is an open-source C++ library that wants to serve as the "assembly language for HE."

To do this, HElib supports low-level routines like set, add, multiply, and shift. It also has sophisticated automatic noise management and implements the BGV and CKKS schema. It is worth noting that the BGV implementation has improved bootstrapping.

There is none that we could find in terms of already implemented machine learning algorithms, but we did find work on optimizing PCA and Linear Regression through Hypercubes [14].

2.3.4 Microsoft SEAL

Microsoft SEAL is comprised of several open-source C++ libraries developed by Microsoft. The library implements the BFV, BGV, and CKKS encryption schemes.

In terms of algorithms implemented with the library, we could find an implementation of linear regression, but it uses sklearn, a python library.

2.4 Machine Learning Primitives

For this work, we are only looking at Machine Learning primitives that are both useful in real-world applications and are, at the same time, simpler. By simpler, we mean that they do not have very "deep" circuits in terms of multiplications.

In particular, we will explore the implementations of the mean, the inner product, and the variance. David J. Wu and Jacob Haven have already explored the implementation of these algorithms using FHE in their article [19].

For all the following examples, $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ are arrays of elements with size n. $\boldsymbol{x_i}$ and $\boldsymbol{y_i}$ represent the ith element in the array.

2.4.1 Mean

The mean is the most simple task we will look at and is represented by the symbol μ. Calculation of the mean can be described as:

$$\mu = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

The mean is used in other statistical computations like Variance and Covariance.

2.4.2 Inner Product

The inner product is not a statistical index itself, but it is an operation used in many algorithms. It corresponds to a multiplication between vectors, in which the multiplication result is a scalar value. The calculation of the inner product can be described as follows:

$$\langle x, y \rangle = \sum_{i=0}^{n-1} (x_i \cdot y_i)$$

2.4.3 Variance

The variance, together with the mean, is used to describe a population of a certain community. It is represented by σ^2. The calculation of the variance can be described as follows:

$$\sigma^2 = \frac{\sum_{i=0}^{n-1}(x_i - \mu)^2}{n}$$

This value tells us how spread the data is or, in other words, how far the values are from the mean. A high variance tells us that the values are far away from the mean, while the closer to 0, the closer the values are to the mean.

Chapter 3

Methodology

3.1 Problem Description

This dissertation will focus on the possible ways to optimize the computations of basic statistics methods and small-scale machine learning primitives over encrypted data using homomorphic encryption.

The problem we are studying is to look at parts of the design space that have not been looked at for optimization (for example, tailoring HE schemes to small-depth circuits).

We choose the algorithms described in chapter 2.4 because they are widely used in practical applications and have small multiplicative depths. These classes of algorithms have not been explored in terms of aggressive optimization. Most research in this field focuses on the feasibility of large and complex algorithms.

We hope to identify the main bottlenecks and trade-offs in terms of performance and benefits gained from different optimization solutions. With this, we aim to also contribute to the optimization of more complex algorithms.

3.2 Strategy

We will be using the BFV scheme. Although CKKS and TFHE would be adequate for our use case, they have some particular characteristics that do not align perfectly with what we want to do.

CKKS uses approximate arithmetic while, as we see by the algorithms we choose, we want to do exact arithmetics. TFHE, on the other hand, doesn't allow us to use SIMD computations which, as we want to go as fast as possible, is something we want to experiment with.

The library we will use will be OpenFHE because it lets us use the scheme we want to use, it is low-level enough to have freedom over everything we need, and it has the most active community. It also allows us to test two different types of packing for each implementation, slot packing or coefficient packing.

So, first, we will start by finding out which implementations work best for each algorithm and for each packing type.

When we have the best implementation based on the packing type, we will optimize each of them by fine-tuning the scheme parameters.

Chapter 4

Packing and Implementations

4.1 Type of Packing

OpenFHE allows us to use two types of packing: coefficient packing and slot packing.

4.1.1 Slot Packing

4.1.1.1 Introduction

Slot packing is the default packing used by OpenFHE. It allows the computation of three vector operations homomorphically under the underlying plaintext: addition, multiplication, and rotation. Addition and multiplication can be done as ciphertext/ciphertext, ciphertext/plaintext, or ciphertexts/constants.

Operations are done value-wise, which means the operation is done between vector values with the same index.

Given a vector with n values, a ciphertext can be described as the encryption of the values x_0^m to x_{n-1}^m, such that, $c_m = [[x_0^m, x_1^m, ..., x_{n-1}^m]]$.

4.1.1.2 Addition

The addition is simple and can be described as $c_0 + c_1 = [x_0^0 + x_0^1, x_1^0 + x_1^1, ..., x_{n-1}^0 + x_{n-1}^1]$.

4.1.1.3 Multiplication

The multiplication is also simple and can be described as $c_0 \cdot c_1 = [x_0^0 \cdot x_0^1, x_1^0 \cdot x_1^1, ..., x_n^0 \cdot x_0^1]$.

4.1.1.4 Rotation

Rotations can only be done to the left, and for each rotation index, an evaluation key needs to be generated before performing the rotation. This is done by passing a vector with all the indices we will use into an OpenFHE function.

If we have ciphertexts with elements in $\mathbb{Z}_q[x]/(x^n+1)$, each rotation applies a transformation $x \to x^i$ into the ciphertexts, where here $x \to y$ indicates that the element in the left, x, is replaced by the one in the right, y. There are a total of n different transformations:

- The first corresponds to the identity $x \to x$, which doesn't change the ciphertext.
- There are $n/2-1$ transformations like $x \to x^i$, where i is an odd number smaller than n; that is $i = 3,5,...,n-1$. In these cases, the ciphertext is divided into two halves, and each half is shifted by $(i-1)/2$ positions.
- Another one is $x \to -x$ which exchanges the two mentioned halves.
- There are $n/2-1$ transformations like $x \to -x^i$, where i is an odd number smaller than n; that is $i = 3,5,...,n-1$. They exchange the two halves and introduce a shift in each half of $(i-1)/2$ positions.

4.1.2 Coefficient Packing

4.1.2.1 Introduction

Coefficient packing is the only other option available for packing in OpenFHE. Instead of working with a vector, this packing method encrypts the values as coefficients of a polynomial. This means that homomorphic operations are done as if we were working directly with addition and multiplication of polynomials. As such, rotations are not natively implemented in OpenFHE with coefficient packing, but we can still effectively execute them by means of multiplication operations.

A ciphertext with n values can be described as $C_m = [[c_0^m, c_1^m, ..., c_{n-1}^m]]$, where the i-th element represents the coefficient of the variable of the polynomial that is raised to i. That is $c_0^m + c_1^m x + \ldots + c_{n-1}^m x^{n-1}$.

4.1.2.2 Addition

The addition is as simple as in the slot packing and can be described as $C_0 + C_1 = [c_0^0 + c_0^1, c_1^0 + c_1^1, ..., c_n^0 + c_0^1]$.

4.1.2.3 Multiplication

The multiplication is done as a polynomial multiplication, as such, can be described as

$$C_0 \cdot C_1 = \{c_k \mid c_k = \sum_{j=0}^{n-1}\sum_{i=0}^{n-1}(c_j^0 \cdot c_i^1) - \sum_{l=0}^{n-1}\sum_{m=0}^{n-1}(c_l^0 \cdot c_m^1) \mid i+j = k \wedge i+j < n \wedge l+m = k \bmod n \wedge l+m \geq n\}$$

4.1.2.4 Rotation

Although rotation is not implemented for Coefficient Packing in OpenFHE, we can use multiplication to rotate the ciphertext elements.

If we multiply a ciphertext with a plaintext filled with 0's except for one index i with the value one, we will rotate the ciphertext by i positions.

If we have a ciphertext c_0 and multiply it by a plaintext $p = [0,0,1,...,0]$, each element of c_0 will be multiplied by x^2. This makes that each element c_i^m (which represents the coefficient of the polynomial that is multiplied with the variable x^i) gets shifted to represent the coefficient that is multiplied with the variable x^{i+2}.

4.2 Implementations

4.2.1 Assumptions

To ease the explanation and comparison of algorithms, we assume that we are encrypting n values. The value of n is equal to the ring dimension which is actually the maximum number of elements that can be encrypted in a ciphertext.

4.2.2 Sum

4.2.2.1 Description

The goal of this algorithm is to calculate the addition of all elements. This is the equivalent of the mean algorithm. Note that, since we cannot do real division homomorphically, it is assumed to be done directly over the plaintext after decryption. Because of that, we only look at the sum part of the mean algorithm.

We implemented four methods: three of them were implemented under both Slot packing and Coefficient packing, while the last version is only implemented with coefficient packing.

4.2.2.2 One element per ciphertext

In this implementation we created n ciphertexts, each with one of the elements in the first index. Then, to calculate the sum of all elements we need to sum all the ciphertexts together and extract the first element.

Since we can parallelize ciphertext additions, if additions were done using a binary tree with perfect parallelization, we would only do $\log_2(n)$ additions.

The main drawback of this implementation is the need to encrypt n ciphertexts.

4.2.2.3 One ciphertext with all the elements

For this implementation, we encrypt all n elements into a single ciphertext. Since there is no way to add elements encrypted under the same ciphertext in OpenFHE, we have to use rotations.

Algorithm 1 Non-optimized Sum

$C_0, C_1 \leftarrow c$
for $i = 0$ to n **do**
    $C_1 \leftarrow$ Rotate(C_0, 1)
    $C_0 \leftarrow$ Add(C_0, C_1)
end for

This was the first approach that was implemented. In this case, c corresponds to a ciphertext encrypting all the elements, and we need to do $O(n)$ rotations and additions to add everything homomorphically. For more details see Algorithm 1.

If we use slot packing, we end up with the sum value in all positions of the ciphertext. If we use Coefficient packing, the sum value will only be in the first element.

4.2.2.4 Optimization

This implementation is based on the previous one as there was still room for one more optimization.

In particular, if n is a power of 2, then we can only do $\log_2(n)$ rotations and additions. Even if n is not a power of 2, we can create a ciphertext with a size equal to the smallest value that is bigger than n and a power of 2 and fill the remaining elements with 0. For more details, see Algorithm 2.

Algorithm 2 Optimized Sum

Require: $x \geq 1, n = 2^x$
$C_0, C_1 \leftarrow c$
for $i = 0$ to $\log_2(n)$ **do**
    $C_1 \leftarrow$ Rotate(C_0, 2^i)
    $C_0 \leftarrow$ Add(C_0, C_1)
end for

If we use slot packing, we end up with the sum value in all positions of the ciphertext. If we use Coefficient packing, the sum value will only be in the first element.

4.2.2.5 Multiplying

This method is only applicable to Coefficient packing. Due to how multiplication works when using this packing, we get the sum of all elements in a ciphertext if we multiply that ciphertext with a polynomial plaintext filled with all 1's. The sum result will only appear in the last element of the ciphertext.

For example, if we have a ciphertext c, such that $c = [[x_0, x_1, ..., x_{n-1}]]$, and we multiply it with a plaintext consisting of only 1's we get that the last element $c[n-1] = x_0 \cdot 1 + x_1 \cdot 1 + ... + x_{n-1} \cdot 1 = x_0 + x_1 + ... + x_{n-1}$.

4.2.3 Inner Product

4.2.3.1 Slot Packing

Using Slot packing, the calculation of the inner product is trivial since we can multiply both ciphertexts and then use the most efficient algorithm to sum all the elements in the resulting ciphertext.

4.2.3.2 Coefficient Packing

In order to calculate the inner product using coefficient packing, we have to invert the coefficient ordering of one of the ciphertexts. If we then multiply one of the ciphertexts with the "inversion" of the other one, we get the Inner Product value in the last coefficient of the ciphertext.

By inverting Y we get $Y^t = [y_{n-1}, ..., y_1, y_0]$. If we then multiply Y^t with X, we get that the last element z_{n-1} will be $z_{n-1} = x_0 \cdot y_0 + x_1 \cdot y_1 + ... + x_{n-1} \cdot y_{n-1}$, which is the definition of the inner product. For more details, see Algorithm 3.

Algorithm 3 Inner Product using Coefficient Packing

$C_0 \leftarrow c$
$C_1 \leftarrow \text{Invert}(c)$
$C_1 \leftarrow \text{Mult}(C_0, C_1)$
for $i = 0$ to $\log_2(n)$ **do**
    $C_1 \leftarrow \text{Rotate}(C_0, 2^i)$
    $C_0 \leftarrow \text{Add}(C_0, C_1)$
end for

4.2.4 Variance

4.2.4.1 Description

Since with BFV, we cannot calculate division homomorphically, we can't use the value of the mean, so we have to adapt the formula shown in section 2.4.3.

4.2.4.2 Deduced Method

The first approach, that we will call the deduced method, consists of expanding $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$ inside the variance expression. With this in mind, we can deduce that

$$\sigma^2 = \frac{1}{n} \cdot \sum_{i=1}^{n} (x_i - \frac{\sum_{i=1}^{n} x_i}{n})^2 = \frac{1}{n^3} \cdot \sum_{i=1}^{n} (n \cdot x_i - \sum_{i=1}^{n} x_i)^2.$$

4.2.4.3 Wu and Haven Method

The second method can be found in the paper [19], where it is shown that the variance can be calculated as

$$\sigma^2 = \frac{1}{n^2} \cdot (n \cdot X \cdot X^t - (n \cdot \mu) \cdot (n \cdot \mu)^t).$$

However, since we cannot homomorphically calculate the total value of the mean, we need to expand it as we did with the first approach. With that in mind we get that the variance can be calculated as

$$\sigma^2 = \frac{1}{n^2} \cdot (n \cdot X \cdot X^t - (n \cdot \frac{\sum_{i=1}^{n} x_i}{n}) \cdot (n \cdot \frac{\sum_{i=1}^{n} x_i}{n})^t) = \frac{1}{n^2} \cdot (n \cdot X \cdot X^t - (\sum_{i=1}^{n} x_i)^2).$$

4.2.4.4 Coefficient Packing Challenges

Due to how coefficient packing works, implementing both methods had its challenges.

For the deduced method, we had to find a way to get the sum in all elements of the ciphertext in order to do the subtraction correctly. This is something that cannot be done without some extra pre-processing of the plaintext elements before homomorphically doing the operations.

The pre-processing we applied was based on the proof made in section 4.A of the work [13]. This pre-processing is an "element-wise" multiplication in the plaintext space, and by applying it, we are able to change the homomorphic operations in the plaintext ring from $\mathbb{Z}_p[x]/(1+x^n)$ to $\mathbb{Z}_p[x]/(1-x^n)$.

The benefit of applying this pre-processing is that all ciphertext elements get the summation value.

We remark that for the Wu and Haven method, the main problem is calculating the square of the sum. Before realizing we could use this pre-processing, our approach to compute the square of the sum using Coefficient packing was based on dividing the original ciphertext into two ciphertexts with size $n/2$ (each with a different half of the elements). If we multiply first these ciphertexts together and, afterwards, we sum all the elements, we will get the value of the square of the sum.

Contrarily, after implementing the pre-processing successfully we discovered that, after calculating the summation value and having it in all the elements, if we multiply that ciphertext with itself, we get a ciphertext that has $n \cdot (\sum_{i=1}^{n} x_i)^2$ in all its elements.

For this to work, we have to change the algorithm by multiplying the formula by $\frac{n}{n}$, which gives us the following expression:

$$\sigma^2 = \frac{1}{n^2} \cdot (n \cdot X \cdot X^t - (\sum_{i=1}^{n} x_i)^2) = \frac{1}{n^3} \cdot (n^2 \cdot X \cdot X^t - n \cdot (\sum_{i=1}^{n} x_i)^2)$$

So, in our testing, we have two implementations of the Wu and Haven method for coefficient packing, one with pre-processing and one without.

Chapter 5

Experimental Results

5.1 Introduction

For the BFV scheme, five parameters can be changed: ring dimension, ciphertext modulus, plaintext modulus, error distribution, and secret key distribution.

It is not advisable to change the error distribution or the secret key distribution as that can have practical consequences that make the implementation unusable. An error distribution that has a standard deviation that is too low makes it insecure, but we could go for higher until we reach a point where decryption no longer holds. However, we chose not to modify the noise parameter, as this would only be interesting for smaller ring dimensions, because, for large numbers of inputs in the computations, it required us to use standard deviations that are too large.

The default secret key distribution is a uniform ternary distribution with values from $\{-1, 0, 1\}$, and the default error distribution is a Gaussian distribution with a standard deviation of 3.2.

For the ciphertext modulus, we could not find a way of modifying it without explicitly changing the code by hand, so we always use the modulus that OpenFHE calculates based on the parameters we give.

Regarding the plaintext modulus, OpenFHE only allows using a prime smaller than 64 bits for slot packing and 32 bits for coefficient packing. Also, by default, OpenFHE uses a ring dimension of 8192.

For all the implementations, we will measure the total run time and the run time for the setup, encryption, and homomorphic operations phases:

- In the setup phase, we set the parameters, create the cryptographic context, generate the necessary evaluation keys, and, in the case of coefficient packing, do the pre-processing and initialize the plaintexts we need to do rotations.
- In the encryption phase, we pack the values into plaintexts and then encrypt those plaintexts into ciphertexts.
- In the homomorphic operations phase, we execute all the specific operations required by each algorithm. These operations include addition, multiplication, and rotations.

5.2 Coefficient vs. Slot Packing

5.2.1 Setup

The main difference in the setup comes from the fact of having to use or not slot rotations in our implementations. Depending on this choice, we need to generate a different type of auxiliary information.

- Slot packing: generate the rotation evaluation keys.
- Coefficient packing: generate the plaintexts with which we will multiply the original ciphertext.

Generating evaluation keys is more time-consuming than generating the equivalent plaintexts, as the corresponding evaluation keys require generating a set of BFV-type encryptions whose size depends on the number and type of rotations being considered.

There are also cases where we have to do the pre-processing before encrypting. This pre-processing is still faster than generating the evaluation keys, as the pre-processing only requires a sequence of cleartext multiplications.

5.2.2 Encrypt

For this case we see that OpenFHE has optimized the encryption process for slot packing, but not for coefficient packing.

Theoretically, slot packing should have a slightly slower encryption time than coefficient packing since it requires one more operation. When using coefficient packing, the plaintext is already a polynomial, but when using slot packing, we need to do an encoding operation beforehand to transform the slot-packed plaintext into a polynomial.

The optimizations done in OpenFHE are described in a paper [11] published by the developers, and it shows how by changing the scaling factor representation, they reduce the noise growth rate, which allows them to have a faster encryption time. In the library implementation, this optimization only applies to the slot packing, as we can see from the results of our experiments.

5.2.3 Addition

In terms of addition, in both cases, the addition is done point-wise, so the time it takes to make additions is the same for both packing types.

5.2.4 Multiplication

For multiplication, we see that the performance for coefficient packing depends on the size of each coefficient. This is, the bigger the coefficient gets, the slower the multiplication.

This is shown in table A.1, which shows the times for each phase when running the exact same implementation with coefficient and slot packing. The implementation in question is the sum multiplication implementation, described in 4.2.2.5.

As we increase the number of elements, we see that the coefficient packing starts having a worse performance while slot packing continues taking the same time. We don't know for certain why this happens. The best hypothesis is that OpenFHE's implementation of the coefficient packing uses the classical version of BFV, which has a slightly bigger noise than the proposed optimization they discussed in the paper [11]. The noise would then become more evident for a larger plaintext modulus and could explain why the multiplication becomes slower as we increase the elements. Since when we increase the number of elements, we must also increase the plaintext modulus to store the result of the computation correctly.

5.2.5 Rotation

The rotation operations are very different depending on whether we work with coefficient or slot packing.

With coefficient packing, each rotation is computed through one ciphertext-cleartext multiplication. With slot packing, each rotation requires computing a switching key operation using the rotation evaluation keys generated during the setup phase.

One slot rotation requires evaluating a higher number of ciphertext multiplications/additions than for the case of coefficient packing. The concrete number of multiplications depends on the specific implementation of key switching (see [11]).

5.3 Implementation Comparison

5.3.1 Sum

Every implementation in this section was tested with the default parameters given by OpenFHE, the minimum possible plaintext modulus, and a total number of elements equal to 8192. This corresponds to the values for the key distributions and the ring dimension specified in section 5.1. For the plaintext modulus, we use a value of 65537.

The results can be seen in the following graphs, where: "simple_coef" and "simple_slot" refer to the implementation explained in chapter 4.2.2.2, "rotation_coef" and "rotation_slot" refer to the implementation explained in chapter 4.2.2.3, "optimized_coef" and "optimized_slot" refer to the implementation explained in chapter 4.2.2.4 and "pre_processed refers to the implementation explained in chapter 4.2.2.5.

The only value not showing in the graphs is the value for the setup of rotation method using coefficient packing. That value is too big and couldn't be put into the graph. It has, on average, a setup duration of 53067 ms.

Total run time:

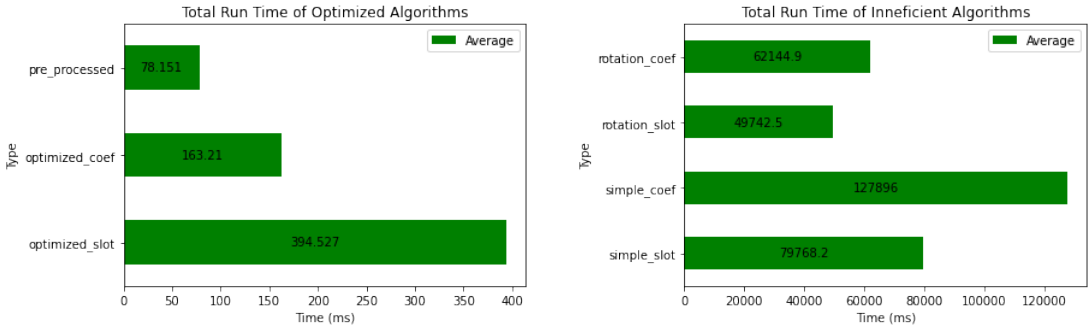

Figure 5.1: Average total run time

Setup phase:

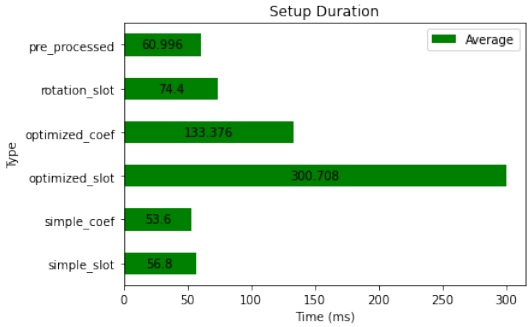

Figure 5.2: Average times for the setup phase for a single vector of size 8192.

Encryption phase:

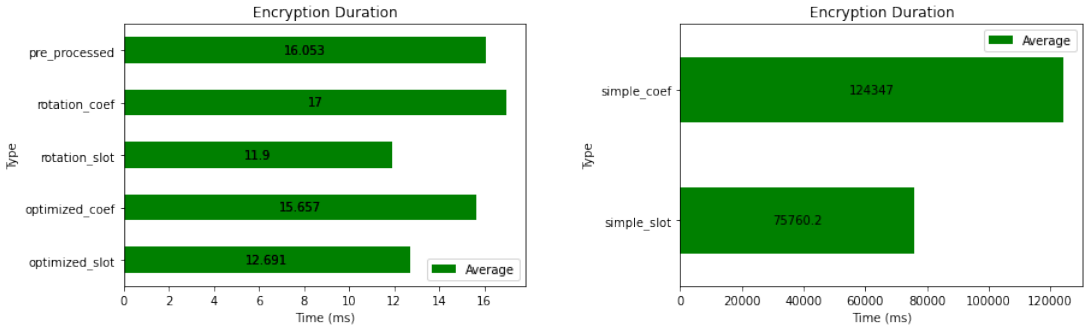

Figure 5.3: Average times for the encryption phase

Homomorphic operations phase:

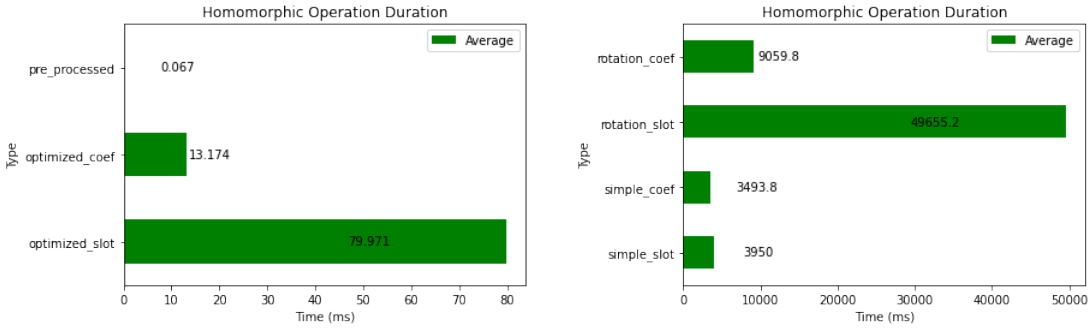

Figure 5.4: Average times for the homomorphic operation phase

5.3.1.1 One element per ciphertext (simple_coef, simple_slot)

We did one implementation per packing type for this algorithm (which is explained in 4.2.2.2). For this algorithm, we encrypt each element into a single ciphertext and then sum all the ciphertexts together. The logic was to check if the time we gained in the homomorphic operations phase (by removing the rotations) would compensate for the time lost in the encryption phase.

By comparing graph 5.4 with 5.3, we see that, although we get a faster homomorphic phase compared to the method with all the elements in a ciphertext, the time lost in encryption does not compensate.

As such, this is by far the worse performer. The only phase in which it performs better corresponds to the setup, as it does not need to generate evaluation rotation keys/plaintexts, nor does it have to do any pre-processing.

5.3.1.2 One ciphertext with all the elements (rotation_coef, rotation_slot)

For this algorithm, we also did an implementation for each packing type. The algorithm is explained in section 4.2.2.3, and it involves putting all elements into a ciphertext, applying then n rotations and additions, where n is the ring dimension, to finally get the total sum value.

This approach performed substantially faster than the first algorithm as it gained a lot of time in the encryption phase since it only needed to do one encryption (refer to graph 5.3).

The coefficient implementation has the worst setup phase of all algorithms since packing 8192 plaintexts for the rotation is far more expensive than generating the one rotation evaluation key needed for the slot alternative. This is due to the number of operations done.

This experiment also showed that slot packing rotations are far more expensive than doing them with coefficient packing. Rotations with slot packing involve many multiplications, while with coefficient packing, we only need to do one multiplication. As such, in graph 5.4, we see an improvement of 81.80% in the duration of the homomorphic operations when using coefficient packing.

5.3.1.3 Optimized Rotations (optimized_coef, optimized_slot)

This algorithm is an optimization of the previous one and is explained in section 4.2.2.4. By using ciphertexts that have a size that is a power of two, we can reduce the number of rotations from n to $\log_2(n)$ by doing rotations in powers of two.

This approach makes the setup phase take longer for slot packing as we have to generate more rotations evaluation keys. On the contrary, it is faster for the coefficient packing as, in turn, we reduce the number of plaintexts used for rotations from n to $\log_2(n)$.

This is compensated by the boost in performance compared to the non-optimized version, as we see the number of operations reduced from $2 \cdot n$ to $2 \cdot \log_2(n)$.

5.3.1.4 Plaintext Multiplication with preprocessing (pre_processed)

This algorithm is only implemented using coefficient packing because it is only possible due to the multiplication properties of this type of packing.

The algorithm is explained in section 4.2.2.5. For this implementation, we put all the elements into a single ciphertext and then multiply it with a plaintext composed of only ones.

Even when using pre-processing, we see, in graph 5.1, that this algorithm is the fastest. Since it only requires one multiplication, the duration of the homomorphic operations is less than $1ms$. Although the pre-processing makes the setup phase slower, as seen in graph 5.2, than the one element per ciphertext method, it is faster than generating all the evaluation rotation keys for the slot packing or packing all the plaintexts for the coefficient packing. This is because the pre-processing is just a sequence of cleartext multiplications while generating keys and packing plaintexts require more complex operations.

5.3.2 Inner Product

Every implementation in this section was tested with the default parameters given by OpenFHE, the minimum possible plaintext modulus, and a total number of elements equal to 8192. This means using the values for the key distributions and the ring dimension specified in the section 5.1. For the plaintext modulus, we use a value of 65537. The results of this experiment can be seen in the following graphs.

Setup phase:

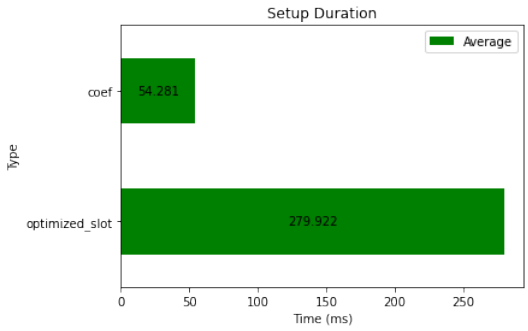

Figure 5.5: Average times for the setup phase.

Encryption phase:

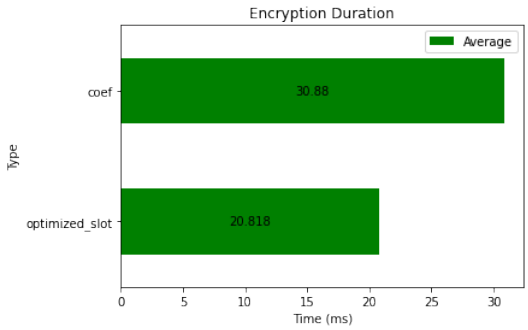

Figure 5.6: Average times for the encryption phase.

Homomorphic operations phase:

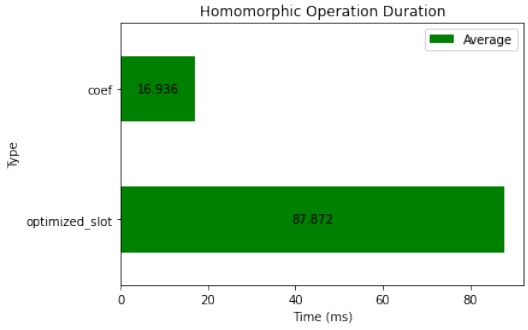

Figure 5.7: Average times for the homomorphic operation phase.

For the inner product, we only did one implementation per packing. The implementation for the coefficient packing is described in 4.2.3.2, and it requires us to invert the coefficient order of one of the vectors before encrypting it and multiplying it with the other ciphertext. The implementation for the slot packing is described in 4.2.3.1, and it requires us to multiply the ciphertexts together and then use the optimized rotation method to get the total sum value. In the graphs, "coef" refers to the coefficient packing implementation and "optimized_slot" to the slot packing implementation.

As the slot packing implementation uses rotations and, as such, needs to generate the evaluation rotation keys, we see, in graph 5.5, that its setup time is slower than the coefficient implementation (that does not have to pack plaintexts, nor does it require pre-processing).

In terms of encryption time, we see, in graph 5.6 that coefficient packing takes more time, the reason being the same as we talked about in section 5.2.2.

Regarding homomorphic operations, as seen in graph 5.7, the coefficient packing implementation is faster as it only requires one multiplication. Which is way faster than doing $\log_2(n)$ rotations and additions due to the number of operations done.

5.3.3 Variance

Every implementation in this section was tested with the default parameters given by OpenFHE, the minimum possible plaintext modulus, and a total number of elements equal to 8192. This means using the values for the key distributions and the ring dimension specified in the section 5.1. For the plaintext modulus, we use a value of 1832743075841. The results can be seen in the following graphs, which we will refer to in the following sections.

Total run time:

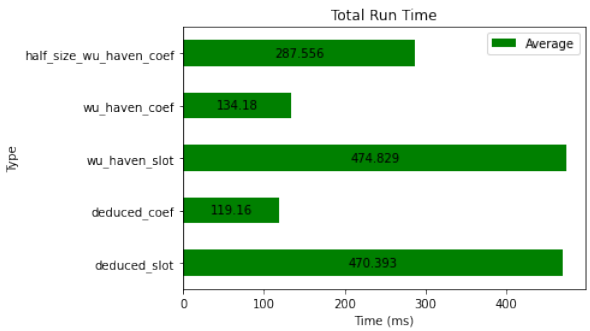

Figure 5.8: Average of total run time.

Setup phase:

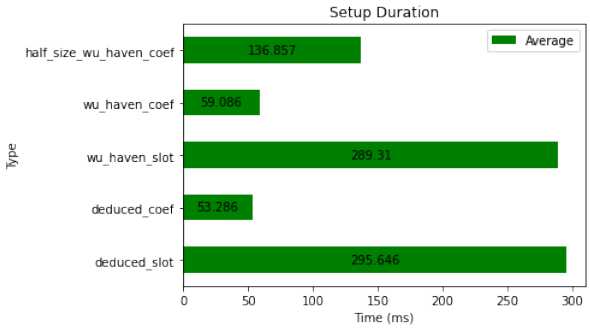

Figure 5.9: Average times for the setup phase.

Encryption phase:

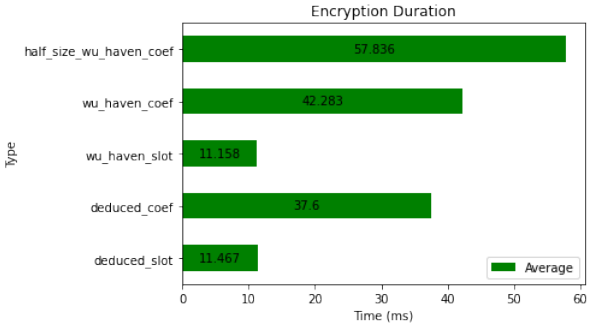

Figure 5.10: Average times for the encryption phase.

Homomorphic operations phase:

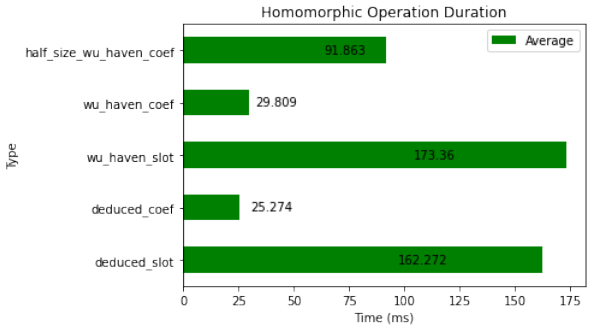

Figure 5.11: Average times for the homomorphic operation phase.

5.3.3.1 Wu and Haven Method

For this method, explained in section 4.2.4.3, we implemented a version with slot packing and two with coefficient packing. The main values we need to calculate in both cases are the inner product and the square of the sum.

For the **slot packing implementation**, we:

- Encrypt all the elements into a ciphertext;
- To calculate the inner product, we use the method described in the previous section that involves multiplying the vector with itself and then adding all the values together using rotations and addition;
- For the square of the sum, we again use rotation in addition to calculate the sum and then multiply the resulting ciphertext with itself;
- Before subtracting the square of the sum from the inner product, we have to multiply it with 8192, which is done by packing that value into a plaintext and multiplying that plaintext with the ciphertext that contains the value of the inner product.

For the coefficient packing, we have two approaches.

In the first one (we will call it the **half-sized approach**), which we implemented as we wanted to find a way not to use pre-processing, we:

- Encrypt four ciphertexts. One with the elements in their original position, the second with the position of the elements reversed, and two ciphertexts where each has a different half of the elements;
- To calculate the square of the sum, we use the ciphertexts that had half the elements by multiplying them with themselves and with each other, then summing all the resulting ciphertexts together, and then summing all the elements of the resulting ciphertext.
- To calculate the inner product, we used the other two ciphertexts by multiplying them with each other;
- Before subtracting the resulting ciphertexts, we multiply the inner product ciphertext with a plaintext containing a single coefficient of 8192;

In the second approach (we will call it the **pre-processed approach**) we:

- Before encryption, we pre-process the elements;
- We encrypt two ciphertexts. One with the pre-processed elements in their original position and the second with the position of the pre-processed elements reversed.
- To calculate the square of the sum, we need to multiply the original ciphertext with a plaintext with only one and then multiply the resulting ciphertext with itself. Although, due to how the pre-process works, we get the square of the sum multiplied by the size of the ciphertext.
- To calculate the inner product, we multiply the ciphertext with the reversed one.
- Before subtracting, instead of multiplying with a plaintext containing 8192, we multiply with a plaintext containing $8192 \cdot 8192$.

Regarding setup time, as seen in graph 5.9, coefficient implementation is always faster. This is, as we concluded many times in this chapter, because generating the evaluation rotations keys is far more expensive than the pre-processing operation.

In terms of encryption time, as seen in graph 5.10, coefficient implementations take longer since they encrypt more ciphertexts, up to 4x more ciphertexts in the half-sized approach.

Operations continue to be faster when using coefficient packing, and here we have confirmation that the pre-processing approach is far better than the half-sized one (refer to graph 5.11). This is because it does way fewer operations. So it does not only improve the algorithm space-wise, but it also improves it time-wise.

5.3.3.2 Deduced Method

For this method, which is explained in section 4.2.4.2, we implemented a version with slot packing and a version with coefficient packing. The main value we want to calculate is the sum of the squares.

To implement this algorithm with **slot packing** we:

- Encrypt one ciphertext;
- Calculate the value of the sum, using the optimized implementation that does $\log_2(n)$ rotations and additions;
- Multiply the original ciphertext with n (in this case, 8192) and subtract the value of the sum from the resulting ciphertext;
- Multiply the resulting ciphertext with itself to get the squared values;
- To get the sum of the squares, we sum all the values together together using $\log_2(n)$ rotations and additions.

For the **coefficient packing**, it is impossible to do this without the pre-processing, so we have to start by pre-processing all the values. To implement this we:

- Before encryption, pre-process the elements;
- Encrypt two ciphertexts. One with the pre-processed elements in their original position and the second with the position of the pre-processed elements reversed.
- Calculate the sum value by multiplying one of the ciphertexts with a plaintext full of ones.
- Multiply original ciphertexts by n (in this case, 8192) and subtract from them the value of the sum;
- To get the sum of the squares, we multiply the resulting ciphertexts together.

As in the previous method, we see, in graph 5.9, that setup time is faster when using the coefficient packing for the same reason. The coefficient packing does not need to generate the evaluation rotation keys.

Regarding encrypting, we also see, in graph 5.10, the same as in the Wu and Haven method because coefficient packing has to encrypt double the number of ciphertexts and, as such, has a bigger encryption time.

As seen in graph 5.8, using coefficient packing takes much less time to finish as it does fewer operations and does not require additional rotations and additions.

5.3.3.3 Wu and Haven vs. Deduced Method

Considering the comparisons between both methods, the results do not show a clear favorite, as seen in graph 5.8. They both have the same number of operations either in the setup phase or in the encryption phase, and, as such, the results show that they have the same duration.

The real difference comes from the type of operations being done:

- For the slot packing, we see that the only difference is that the Wu and Haven method does one extra ciphertext-ciphertext multiplication.
- For the coefficient packing, we see that the Wu and Haven method does one more ciphertext-ciphertext multiplication but it does one less ciphertext-ciphertext addition and two less ciphertext-plaintext multiplication.

Also, in both cases, the deduced method has a faster homomorphic operation time:

- In the slot packing, we can see that it is due to having one less multiplication.
- In the coefficient packing, the difference between the methods is less, but the ciphertext-ciphertext multiplication is still costlier than the addition and the ciphertext-plaintext multiplications. This is because the noise grows more in ciphertext-ciphertext multiplication than in any other operation.

So, in both packing types, the deduced method is the fastest but only by roughly 5ms.

5.4 Parameter Tuning

Every implementation we talked about in the previous section was tested with the default parameters given by OpenFHE and the minimum possible plaintext modulus. This equates to all of them having a huge level of security.

As we talked about in section 5.1, the parameters we can tune are the plaintext modulus and the ring dimension. We decided that it would be interesting to see the impact of the ring dimension on the performance in relation to the number of elements we needed to encrypt.

5.4.1 Test Cases

To test the impact of the ring dimension, we choose the best implementations (one using coefficient packing and one using slot packing) for each algorithm.

We then choose a range of the number of elements encrypted that would test both cases where we have (1) a low number of elements, translating into a practical example of voting in a small assembly, and (2) a big number of elements, translating to a presidential election in Portugal.

So we started with 16 elements and went up, in powers of two, to 8192 elements. From there, we went up until 8192000 elements in 7 steps: 81920 elements, 204800 elements, 409600 elements, 819200 elements, 2048000 elements, 4096000 elements, and 8192000 elements.

Regarding the plaintext modulus, we decided to fix it at the lowest prime we could use. To find the minimum modulus, we computed the maximum value in our computations and chose a prime that could handle that value.

During this computation, we realized that testing the variance implementations would not be possible for very large values since the plaintext modulus would need to be bigger than 42 bits. This is a problem because OpenFHE does not guarantee that the algorithms will work with such a big plaintext modulus. But even if it did, from 2048000 and beyond, we would need to use a plaintext bigger than 64 bits which is bigger than a machine word in C++. So we decided we would not be testing the variance implementations.

For the ring dimension, we have two cases:

- When the number of elements is below 8192, we always start with a ring dimension equal to the number of elements and then go down three steps in powers of two. If we don't achieve 128-bit security with any of the ring dimensions, we go up in powers of two until we get a ring dimension that puts the security at 128 bits or above.
- For the cases where the number of elements is 8192 or above, we test with ring dimension starting at 8192 and going down until 512, in powers of two.

All the results can be seen in the tables found in the appendix A. In these tables, we only include the rows that provide at least 30 bits of security.

5.4.2 Slot Packing

For slot packing, we chose the optimized algorithm for the sum, which involves summing all ciphertexts together and then combining rotations and addition to get the final value. For the inner product, we only had one implementation, which involved multiplying all ciphertexts with themselves, adding them all together, and then using rotations and addition to get the final value.

When we look at the results we got with a small number of elements (from 16 elements to 2048 elements), we see that we cannot use a ring dimension lower than 512. Going any lower than that does not allow us to have 128-bit security. This can be seen in the appendix in tables A.19 and A.18.

In terms of performance, in those cases, it all comes down to setup time, as operations, with the precision we are working with, have a total time of 0 to 1 ms. Although using a ring dimension 512 does not give us the best performance, the difference between the best performers is only 7-8ms (which is negligible).

The following discussion will be based on the results that can be seen in tables A.2 to A.9. With total elements from 4096 up until 8192000, we see that a ring dimension of 512 is not enough to guarantee 128-bit security. But it is the ring dimension that gives the best performance up to 819200 elements for both the sum and the inner product implementations.

For the sum, the ring dimension that gives the best performance, without counting 512, is 1024. This is true until 8192000 elements, in which case 2048 shows a better performance but by only 12ms, as can be seen in table A.9. In all other cases, 2048 is the second-best performer.

For the inner product, we see that the same is true, but only up to 819200. From this point on, 2048 shows the best performance, improving as the number of elements increases.

This clearly shows the trade-off between encrypting more but smaller ciphertexts and encrypting less but longer ciphertexts. This also influences the homomorphic operation times as we will have to do more operations but with fewer elements in each operation.

A ring dimension of 1024 or 2048 for both implementations always has the best encryption time. The difference between them is negligible. This is because, although we have to encrypt more ciphertexts than when using a ring dimension of 4096 and above, the fact that the ciphertexts are smaller compensates for the time lost in encrypting more ciphertexts.

In terms of the homomorphic operations, we see that as the number of elements grows, and in turn, the number of ciphertexts we encrypt and use increases, the implementations start favoring bigger ring dimensions. Slot packing never favors extreme cases. With this, we mean the worst results always have a ring dimension of 512 or 8192. This is because 512 has so many ciphertexts (especially for a huge number of elements) that the speed gained in each operation is not worth it compared to the amount of extra operations we have to do in encryption. For 8192, we see the opposite problem. We have way fewer ciphertexts, so we do fewer operations, but the fact that the ciphertexts are bigger makes each operation take longer. So, although we gain time by having fewer operations, that time is less than the time lost in all combined operations. The best ring dimensions are the ones that have the best trade-off between the number of operations done and the time it takes to do each operation.

From 8192 to 819200, the ring dimensions with the most efficient operations are 1024 and 2048. But, from 204800 onward, we see that using a ring dimension of 4096 is more efficient than using 1024.

Since we do more operations in the inner product than in the sum, we see that this factor starts to become more relevant. When using 4096000 and 8192000 elements, using a ring dimension of 4096 is better than using 1024, which does not happen when using fewer elements (compare tables A.8 and A.9 to, for example, A.7). This is also why the inner product implementation favors a ring dimension of 2048 sooner than the sum implementation.

5.4.3 Coefficient Packing

We chose the multiplication method for the coefficient packing, which involves summing all the ciphertexts and multiplying the result with a plaintext containing only ones. We only have one implementation for the inner product, which involves multiplying each ciphertext with its inverse and then summing all the resulting ciphertexts together.

When we look at the results in tables A.21 and A.20 we can conclude the same as in the slot packing. The only thing that matters in these cases is the setup phase, and the minimum plaintext moduli used are the same. So we also see that the lowest ring dimension we can use is 512 to guarantee 128-bit security.

We also see that from 4096 onward, we cannot use a ring dimension of 512 because it is insufficient to have 128-bit security. So we will not consider these values when talking about the best performers.

The following discussion will be based on the results that can be seen in tables A.10 to A.17. For the sum implementation, the ring dimension with the best performance for elements between 4096 and 409600 is 1024, followed by 2048. Although, with that ring dimension, the operations time is slightly longer, it gains an advantage in the encryption phase.

For the inner product, this only happens up until 81920 elements (refer to table A.11). This is because the inner product has a lot more operations than the sum, so the impact of the ring dimension on the operation time is more significant.

For the sum, homomorphic operations are more efficient as the ring dimension increases. Since the main time-consuming operation is the addition of all the ciphertexts together, we can conclude that when we have a lot of elements, it's better to have fewer but bigger ciphertexts. This is because, as additions are made coefficient-wise, the cost of having more elements in each operation is very small compared to doing more operations.

This pattern does not appear in the inner product implementation because the inner product has ciphertext-ciphertext multiplications. Since for multiplication, we have to multiply each coefficient with all other coefficients, the number of coefficients in each operation will have a more significant impact than when making additions. From the results, we can see that these types of operations work best with a ring dimension between 1024 and 4096, depending on the number of elements. These ring dimensions have the best trade-off between the time lost in doing more operations and the time gained by having fewer elements in each operation.

From 819200 elements and above, the sum implementation prefers the use of a ring dimension of 2048, except when we have 4096000 elements. In any case, the difference between the implementation with 2048 and 4096 is only 56 ms, so this effect could be dismissed as an artifact of not repeating the experiment enough times (refer to table A.16).

From 204800 elements and above, the inner product implementation prefers using a ring dimension of 2048. As the number of elements grows, the implementation starts having better performance with a ring dimension of 4096 compared to using 1024.

Eventually, when using 8192000 elements, the inner product implementation has better performance when using a ring dimension of 4096.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The first objective we had was to find out the most efficient way of implementing each of the proposed algorithms.

We started by exploring the implementation of the mean, which translated to exploring the most efficient way of implementing the sum of all the elements. For this, we divided it into two paths, one ciphertext for all the elements and one ciphertext per element. The latter didn't have much space where we could optimize, so we focused on the first approach. We successfully optimized the first approach, either by using properties of the type of packing used or by optimizing the algorithm itself.

When we started implementing the inner product, we already took what we had learned from the mean since, in both cases, we ended up having to sum up all the elements. This means that the approach we started with was the most optimized.

For the variance, we had two different algorithms that we wanted to explore. These algorithms posed a challenge when implementing them with coefficient packing since the properties of the operations with this packing made the calculations very inefficient. To fight this, we found a pre-processing technique that allowed us to use those properties in our favor. When comparing both approaches, we didn't see any clear benefit of using one of the approaches over the other.

With all the algorithms implemented successfully, we focused on tailoring the parameters to see how they affect performance. We decided to focus on the ring dimension, as it is not recommended to change the other parameters (with the exception of the plaintext and ciphertext modulus) and fix the plaintext modulus to be the minimum possible prime. Using the ring dimension also allowed us to see how the size of the ciphertexts and the number of ciphertexts affects both encryption time and the time it takes to do the operations.

To test this, we saw suitable to use a wide range of values, starting from sixteen elements up to eight million elements. For this, we calculated the minimum possible plaintext for the range of elements, and we found that we couldn't test the variance for a very large number of elements. This is because they would need a plaintext modulus bigger than 64 bits, which is the maximum size

the library accepts. Even with plaintext modulus smaller than 64 bits, we found some problems as the library, for coefficient packing, didn't allow using plaintexts bigger than 32-bit. We went into the library code and changed this, but the developers couldn't confirm that it would always work, so we decided only to explore the possibilities for the mean and the inner product.

In both cases, we could see that when the computation has a small plaintext modulus(smaller than forty thousand), it is not worth using ring dimensions lower than 512, as it does not have 128-bit security. For plaintext modulus bigger than sixty-five thousand, it is also not recommended to use ring dimensions smaller than 1024, as it also doesn't have 128-bit security.

Looking at the experiments with a total number of elements bigger than 8192, we see that both slot packing and coefficient packing prefer ring dimensions on the lower side, as increasing the ring dimension negatively impacts the time it takes to encrypt and do the operations.

We also wanted to compare the implementations using coefficient and slot packing to see which is the most efficient for each algorithm. This was not possible as the library applies important optimizations to the slot packing, which made it unfair to compare to coefficient packing.

6.2 Future Work

As for future work, we see benefits in testing these same algorithms, but instead of increasing the number of elements, increase the value of each element. All the tests had ciphertexts filled with elements that would either be 0's, 1's, or 2's. This allowed us to study how increasing the number of elements affects each algorithm, but we think it is also important to test how the size of each element affects the efficiency of each implementation.

We also see a way to combat the lack of support for a plaintext modulus bigger than 64 bits. If we decompose the plaintext modulus in two or three smaller primes, we can do the operations using those moduli and then use the Chinese Remainder Theorem [8] to reconstruct all the results to be in the original modulus. This would also allow us to parallelize operations, which could considerably improve performance.

Lastly, to compare the performance of slot packing versus coefficient packing, we have two recommendations. One is re-implementing these algorithms but using another library with either optimization for both packings or the classical BFV method. The other is going into the library and working out why the unexpected behavior is happening and implementing the same optimization for coefficient packing.

References

[1] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Paper 2015/1192, 2015. https://eprint.iacr.org/2015/1192.

[2] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *SIAM Journal on Computing*, 43(5):1541–1563, 2014.

[3] Zvika Brakerski, Vinod Vaikuntanathan, and Craig Gentry. Fully homomorphic encryption without bootstrapping. *IACR Cryptology ePrint Archive*, 2011(277):1–16, 2011.

[4] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

[5] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Improving tfhe: faster packed homomorphic operations and efficient circuit bootstrapping (2017), 2017.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[7] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*, pages 617–640. Springer, 2015.

[8] Carl Friedrich Gauss. *Disquisitiones arithmeticae*, volume 157. Yale University Press, 1966.

[9] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[10] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.

[11] Andrey Kim, Yuriy Polyakov, and Vincent 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, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 608–639. Springer, 2021.

[12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *SIAM Journal on Computing*, 39(6):2181–2213, 2010.

[13] Alberto Pedrouzo-Ulloa, Juan Ramón Troncoso-Pastoriza, and Fernando Pérez-González. Number theoretic transforms for secure signal processing. *IEEE Trans. Inf. Forensics Secur.*, 12(5):1125–1140, 2017.

[14] Deevashwer Rathee, Pradeep Kumar Mishra, and Masaya Yasuda. Faster pca and linear regression through hypercubes in helib. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, pages 42–53, 2018.

[15] Partha Pratim Ray. An introduction to dew computing: Definition, concept and implications. *IEEE Access*, 6:723–737, 2018.

[16] Oded Regev. The learning with errors problem. *Journal of Cryptology*, 18(4):267–302, 2005.

[17] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[18] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[19] David Wu and Jacob Haven. Using homomorphic encryption for large scale statistical analysis. *FHE-SI-Report, Univ. Stanford, Tech. Rep. TR-dwu4*, 2012.

Appendix A

Appendix

Total Elements	Packing	Setup (ms)	Encryption (ms)	Multiplication (ms)
8192	Slot	20.03	15.09	0.0
	Coefficent	17.93	8.33	0.00
81920	Slot	17.53	75.37	0.00
	Coefficent	18.51	134.02	0.0
204800	Slot	17.45	189.44	0.00
	Coefficent	18.68	337.86	0.0
409600	Slot	17.68	378.57	0.00
	Coefficent	18.47	669.72	0.0
819200	Slot	17.47	1942.44	1.00
	Coefficent	18.25	1338.89	0.00
2048000	Slot	17.91	189.06	0.00
	Coefficent	18.59	3499.11	05.09
4096000	Slot	17.78	3775.31	0.00
	Coefficent	18.87	6717.89	9.0
8192000	Slot	17.68	7562.50	0.00
	Coefficent	18.87	13538.16	17.0

Table A.1: Table of results comparing Multiplication implementation using coefficient and slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	10.31	6.01	11.15	27.47
	1024	18.03	6.04	12.07	36.14
	2048	33.61	7.00	15.58	56.19
	4096	135.76	9.87	44.95	190.58
	8192	288.11	11.02	84.84	384.97
Sum	512	10.00	6.05	1.03	17.08
	1024	17.08	6.11	3.03	26.22
	2048	31.50	7.01	7.17	45.68
	4096	127.10	9.89	30.41	167.4
	8192	271.76	11.06	69.47	353.29

Table A.2: Table of results for 8192 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	19.05	77.45	147.42	243.92
	1024	33.94	74.65	137.26	245.85
	2048	64.22	75.49	137.83	277.54
	4096	141.68	83.24	176.07	401.01
	8192	282.89	81.95	206.22	572.06
Sum	512	17.97	77.3	7.07	102.34
	1024	31.86	74.6	8.18	114.64
	2048	59.53	75.13	12.01	146.67
	4096	124.83	77.66	21.33	223.82
	8192	265.36	81.93	63.92	412.21

Table A.3: Table of results for 81920 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	18.98	193.35	364.27	576.60
	1024	33.77	185.60	333.49	552.86
	2048	64.17	186.76	328.39	579.32
	4096	138.10	201.79	375.45	715.34
	8192	283.16	200.48	399.22	883.86
Sum	512	17.92	192.74	15.07	225.73
	1024	31.77	185.74	14.89	232.40
	2048	59.43	186.91	18.13	264.47
	4096	125.06	191.63	27.03	343.72
	8192	264.99	200.02	69.42	535.43

Table A.4: Table of results for 204800 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	19.07	389.49	734.31	1142.87
	1024	34.29	379.57	683.24	1097.10
	2048	65.43	381.80	675.08	1122.31
	4096	136.13	394.29	697.92	1228.34
	8192	289.20	410.52	750.41	1451.13
Sum	512	17.96	388.36	28.71	435.03
	1024	32.45	383.10	28.10	443.65
	2048	60.75	383.31	29.94	474.00
	4096	126.44	392.14	38.89	557.47
	8192	269.13	407.67	61.80	739.60

Table A.5: Table of results for 409600 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	19.04	777.36	1465.09	2261.49
	1024	34.30	760.94	1366.42	2161.66
	2048	64.98	762.22	1328.81	2156.01
	4096	135.09	781.13	1351.90	2268.12
	8192	287.00	810.33	1421.04	2519.37
Sum	512	18.08	778.21	55.95	852.24
	1024	32.25	770.63	51.64	854.52
	2048	60.77	761.29	50.54	872.60
	4096	126.54	783.21	58.37	968.12
	8192	270.61	817.24	81.00	1169.85

Table A.6: Table of results for 819200 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	19.00	1954.31	3725.34	5698.65
	1024	34.22	1905.22	3432.38	5371.82
	2048	64.51	1897.34	3307.49	5269.34
	4096	135.01	1944.16	3346.10	5425.27
	8192	286.46	2019.67	3472.71	5779.84
Sum	512	18.02	1951.86	138.44	2108.32
	1024	31.93	1913.35	118.11	2063.39
	2048	60.33	1902.78	110.63	2073.74
	4096	126.31	1940.19	113.35	2179.85
	8192	268.37	2014.56	133.84	2417.77

Table A.7: Table of results for 2048000 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	19.08	3904.10	7477.17	11400.35
	1024	34.91	3900.77	7196.75	11132.43
	2048	65.52	3893.97	6868.75	10828.24
	4096	136.42	3921.60	6792.72	10850.74
	8192	286.59	4018.86	6921.48	11227.93
Sum	512	18.00	3895.15	271.42	4184.57
	1024	32.10	3793.31	230.49	4055.90
	2048	61.30	3878.68	214.51	4154.49
	4096	128.47	3952.01	214.28	4294.77
	8192	286.89	4258.16	242.68	4788.73

Table A.8: Table of results for 4096000 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	21.22	7944.55	15292.24	23258.01
	1024	33.55	7529.41	13633.27	21196.23
	2048	64.55	7507.62	13116.26	20688.43
	4096	134.48	7697.79	13242.39	21074.66
	8192	284.67	7983.18	13659.07	21927.92
Sum	512	18.15	8061.37	560.55	8640.07
	1024	31.95	7524.21	456.65	8012.81
	2048	60.15	7528.70	410.79	7999.64
	4096	125.61	7691.40	397.37	8214.38
	8192	266.02	7970.05	409.41	8646.48

Table A.9: Table of results for 8192000 elements, using slot packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	5.48	22.18	9.96	37.62
	1024	7.03	21.73	8.34	37.10
	2048	10.11	22.03	8.08	40.22
	4096	29.62	28.99	14.07	72.68
	8192	53.12	29.52	15.67	99.31
Sum	512	6.16	11.00	0.00	17.16
	1024	8.37	11.05	0.00	19.42
	2048	14.69	13.44	0.14	28.27
	4096	33.10	14.67	0.04	47.81
	8192	60.60	15.15	0.03	76.78

Table A.10: Table of results for 8192 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	8.98	272.95	144.45	426.38
	1024	12.26	265.70	131.60	409.56
	2048	16.54	265.20	128.18	409.92
	4096	29.67	268.38	129.86	427.91
	8192	52.14	274.89	147.49	475.52
Sum	512	10.45	139.64	5.04	155.13
	1024	13.31	133.48	4.02	150.81
	2048	17.82	133.52	3.15	154.49
	4096	32.65	134.27	3.00	169.92
	8192	58.38	140.63	2.00	202.01

Table A.11: Table of results for 81920 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	9.00	684.88	364.72	1058.60
	1024	12.30	666.49	330.15	1008.94
	2048	16.89	663.38	319.68	999.95
	4096	29.63	670.84	326.79	1027.26
	8192	52.16	680.29	345.27	1078.72
Sum	512	9.10	344.76	13.23	367.09
	1024	13.06	333.42	10.25	356.73
	2048	18.24	331.86	9.72	359.82
	4096	32.87	335.73	8.94	377.54
	8192	58.42	341.09	08.04	408.55

Table A.12: Table of results for 204800 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	09.01	1375.67	740.49	2125.17
	1024	12.22	1379.82	697.18	2089.22
	2048	16.85	1361.51	668.56	2046.92
	4096	29.35	1367.59	667.36	2064.30
	8192	52.71	1382.38	696.22	2132.31
Sum	512	09.03	683.97	26.45	719.45
	1024	12.94	675.55	23.28	711.77
	2048	18.80	675.17	20.68	714.65
	4096	32.42	683.62	20.41	736.45
	8192	58.87	693.65	19.07	772.59

Table A.13: Table of results for 409600 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	9.00	2752.64	1480.87	4242.51
	1024	12.48	2728.30	1377.27	4118.05
	2048	17.02	2703.40	1326.78	4047.20
	4096	29.18	2730.89	1340.00	4100.07
	8192	52.65	2766.05	1386.78	4206.48
Sum	512	9.18	1366.41	54.12	1429.71
	1024	13.28	1371.32	47.32	1431.92
	2048	18.48	1356.20	41.28	1415.96
	4096	32.06	1369.64	40.91	1442.61
	8192	59.35	1393.38	39.82	1493.55

Table A.14: Table of results for 819200 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	9.00	7011.65	3739.40	10760.07
	1024	12.21	6871.67	3458.34	10342.22
	2048	16.38	6774.39	3310.94	10101.71
	4096	28.73	6817.69	3346.44	10192.87
	8192	52.69	6900.48	3458.36	10412.53
Sum	512	9.16	3528.78	139.88	3677.82
	1024	13.08	3420.39	116.81	3550.38
	2048	18.08	3390.18	105.62	3513.98
	4096	32.52	3433.13	99.74	3565.50
	8192	58.77	3454.78	96.52	3611.07

Table A.15: Table of results for 2048000 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	09.01	14562.39	7509.45	22080.88
	1024	12.19	13951.89	6890.13	20854.30
	2048	16.94	13595.71	6669.86	20282.51
	4096	30.05	13896.89	6894.61	20821.55
	8192	52.73	13815.98	6922.15	20791.86
Sum	512	09.09	7170.85	269.74	7449.68
	1024	12.93	6863.75	230.03	7106.71
	2048	18.98	6875.24	211.53	7105.86
	4096	32.85	6821.48	195.60	7049.97
	8192	59.11	6900.99	191.71	7152.87

Table A.16: Table of results for 4096000 elements, using coefficient packing.

	Ring Dimension	Setup (ms)	Encryption (ms)	Homomorphic (ms)	Total (ms)
Inner Product	512	9.00	31718.65	15212.60	46941.38
	1024	12.08	28614.90	13769.17	42396.23
	2048	16.59	27513.82	13253.83	40784.24
	4096	29.35	27345.61	13371.38	40746.34
	8192	52.41	27438.09	13725.21	41216.73
Sum	512	9.10	15445.53	551.09	16005.72
	1024	12.84	14052.06	459.57	14524.47
	2048	18.40	13534.68	413.93	13967.01
	4096	33.12	13747.09	398.27	14178.53
	8192	58.36	13808.71	388.71	14256.84

Table A.17: Table of results for 8192000 elements, using coefficient packing.

Operation	Total Elements	Ring Dimension	Total Time (ms)	Bit Security
Inner Product	**16**	**32**	4.86	2^32
		64	5.41	2^43
		128	7.18	2^68
		256	8.68	2^74
		512	13.46	2^131
	32	**32**	5.21	2^32
		64	5.86	2^43
		128	7.32	2^68
		256	8.80	2^74
		512	13.49	2^131
	64	**32**	06.05	2^34
		64	5.58	2^43
		128	07.06	2^68
		256	08.18	2^74
		512	12.76	2^131
	128	**32**	5.44	2^33
		64	06.05	2^39
		128	5.58	2^57
		256	8.97	2^74
		512	13.23	2^131
	256	**64**	6.39	2^43
		128	7.40	2^42
		256	8.91	2^74
		512	13.36	2^131
	512	**64**	7.32	2^38
		128	8.39	2^40
		256	9.97	2^70
		512	13.20	2^131
	1024	**128**	10.48	2^40
		256	12.23	2^70
		512	15.66	2^131
		1024	24.15	2^264
	2048	**256**	13.98	2^70
		512	16.70	2^131
		1024	26.20	2^264
		2048	46.47	2^246
	4090	**512**	21.17	2^116
		1024	29.40	2^233
		2048	50.71	2^487
		4090	95.64	2^1024

Table A.18: Table of results of the Inner Product for less than 8192 elements, using slot packing.

Operation	Total Elements	Ring Dimension	Total Time (ms)	Bit Security
Sum	**16**	**32**	05.01	2^32
		64	05.04	2^43
		128	6.12	2^68
		256	forty	2^74
		512	11.47	2^131
	32	**32**	05.03	2^32
		64	5.10	2^43
		128	6.27	2^68
		256	8.28	2^74
		512	11.48	2^131
	64	**32**	5.20	2^34
		64	04.23	2^43
		128	06.03	2^68
		256	8.00	2^74
		512	11.01	2^131
	128	**32**	5.11	2^33
		64	5.20	2^39
		128	04.23	2^57
		256	8.14	2^74
		512	11.53	2^131
	256	**64**	5.10	2^43
		128	6.30	2^42
		256	08.29	2^74
		512	11.34	2^131
	512	**64**	5.20	2^38
		128	6.29	2^40
		256	8.21	2^70
		512	11.69	2^131
	1024	**128**	7.44	2^40
		256	9.29	2^70
		512	12.60	2^131
		1024	21.60	2^264
	2048	**256**	10.33	2^70
		512	13.36	2^131
		1024	22.50	2^264
		2048	42.29	2^246
	4090	**512**	14.74	2^116
		1024	23.91	2^233
		2048	43.94	2^487
		4090	86.67	2^1024

Table A.19: Table of results of the Sum for less than 8192 elements, using slot packing.

Operation	Total Elements	Ring Dimension	Total Time (ms)	Bit Security
Inner Product	**16**	**32**	04.09	2^32
		64	4.45	2^43
		128	05.05	2^68
		256	5.10	2^74
		512	6.97	2^131
	32	**32**	4.23	2^32
		64	04.15	2^43
		128	05.05	2^68
		256	05.22	2^74
		512	6.86	2^131
	64	**32**	4.16	2^34
		64	4.35	2^43
		128	5.00	2^68
		256	05.01	2^74
		512	6.82	2^131
	128	**32**	4.68	2^33
		64	4.16	2^39
		128	4.35	2^57
		256	5.25	2^74
		512	7.19	2^131
	256	**64**	6.64	2^43
		128	5.67	2^42
		256	5.18	2^74
		512	7.17	2^131
	512	**64**	08.04	2^38
		128	7.30	2^40
		256	6.99	2^70
		512	7.17	2^131
	1024	**128**	10.58	2^40
		256	9.46	2^70
		512	10.00	2^131
		1024	10.88	2^264
	2048	**256**	14.29	2^70
		512	14.03	2^131
		1024	14.72	2^264
		2048	17.63	2^246
	4090	**512**	22.30	2^116
		1024	22.62	2^233
		2048	25.35	2^487
		4090	32.81	2^1024

Table A.20: Table of results of the Inner Product for less than 8192 elements, using coefficient packing.

Operation	Total Elements	Ring Dimension	Total Time (ms)	Bit Security
Sum	**16**	**32**	4.45	2^32
		64	4.32	2^43
		128	05.07	2^68
		256	05.09	2^74
		512	6.10	2^131
	32	**32**	4.62	2^32
		64	4.78	2^43
		128	5.16	2^68
		256	5.55	2^74
		512	06.16	2^131
	64	**32**	4.30	2^34
		64	5.25	2^43
		128	8.34	2^68
		256	05.02	2^74
		512	6.8	2^131
	128	**32**	4.18	2^33
		64	4.30	2^39
		128	5.25	2^57
		256	5.60	2^74
		512	6.16	2^131
	256	**64**	4.67	2^43
		128	5.8	2^42
		256	5.33	2^74
		512	6.42	2^131
	512	**64**	5.66	2^38
		128	5.39	2^40
		256	5.36	2^70
		512	6.32	2^131
	1024	**128**	07.07	2^40
		256	6.62	2^70
		512	7.10	2^131
		1024	9.38	2^264
	2048	**256**	8.34	2^70
		512	8.60	2^131
		1024	10.69	2^264
		2048	14.60	2^246
	4090	**512**	11.66	2^116
		1024	13.44	2^233
		2048	17.07	2^487
		4090	25.83	2^1024

Table A.21: Table of results of the Sum for less than 8192 elements, using coefficient packing.