Jakob Feiner, BSc

Lightweight Communication for Secure Outsourced Homomorphic Computation

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Dipl.-Ing. Roman Walch, BSc

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications

Inffeldgasse 16a, 8010 Graz, Austria

Graz, January 2022

AFFIDAVIT

Abstract

Homomorphic Encryption (HE) is considered as one of the most advanced privacy preserving encryption methods concerning outsourced computation. It enables a client, on any kind of low powered edge device, to outsource expensive tasks on sensitive user data to a server in the cloud. In a traditional setting, this kind of outsourcing of the computation would lead to a huge privacy leak, although data is encrypted during transit. Personal user data has to be available in plain to the server in order to perform operations on them. However, using HE the server can perform any operation on the users personal data, without revealing its actual content. That is, because using HE, operations on the ciphertexts are equivalent to the same operations on the original plaintexts. After the computational intense operations on the ciphertexts, the server sends back the still homomorphically encrypted result to the client. The client then is the only one, who can decrypt it.

Although we would gain a huge privacy benefit by using HE, it's rarely used in practice. One downside is, that a lot of computational power is needed. A second one is, that HE suffers from ciphertext expansion, and the resulting communication overhead.

Nowadays, most HE schemes are based on Learning With Errors (LWE) and its ring variant, Ring Learning With Errors (RLWE). In particular, we will use the symmetric Brakerski/Fan-Vercauteren (BFV) HE scheme, which itself is based on LWE/RLWE.

This thesis builds on the findings of the recently published paper [Che+20]. It aims to accomplish a lightweight communication between the client and the server, i.e. mitigate the problem of ciphertext expansion. One can split the theoretical work in three main parts: Firstly, the introduction of lightweight LWE ciphertexts. Secondly, the description of efficient homomorphic conversion algorithms between LWE ciphertexts and RLWE ciphertexts. And finally, an efficient way to convert a resulting RLWE ciphertext into a HE ciphertext.

In the practical part of this thesis, we implemented these three steps to extend the Microsoft open source HE software library Simple Encrypted Arithmetic Library (SEAL). We not only performed benchmarks, and compared the results to the the original paper [Che+20], but also compared them to an alternative way of lightweight communication, namely Hybrid Homomorphic Encryption (HHE), presented in [Dob+21].

Keywords: Homomorphic Encryption, Privacy, Security, Simple Encrypted Arithmetic Library, Learning With Errors, Ring Learning With Errors, Brakerski/Fan-Vercauteren

Kurzfassung

Homomorphe Verschlüsselung (HE) ist eine der effektivsten Methoden, um sensible Daten während einer ausgelagerten Berechnung in der Cloud zu schützen. Will man auf leistungsschwachen Computern aufwendige Berechnungen mit persönlichen Daten durchführen, können diese auf einen Server auslagert werden. Für gewöhnlich stellt dies jedoch ein enormes Datenschutz Problem dar, da Berechnungen am Server nur mit dem unverschlüsselten Klartext durchgeführt werden können. HE löst dieses Problem, da es dem Server ermöglich Berechnungen mit verschlüsselten Daten durchzuführen. Der Grund dafür ist, dass bei der Verwendung von HE Operationen mit dem Chiffretext equivalent zu Operationen mit dem originalen Klartext sind. Nachdem der Server die aufwendigen Berechnungen durchgeführt hat, wird das homomorph verschlüsselte Resultat an den lokalen Rechner zurück geschickt, wo es wieder entschlüsselt werden kann.

Trotz dieses enormen Vorteils bezüglich des Datenschutzes, findet die HE in der Praxis kaum Verwendung. Zum einen ist HE sehr rechenintensiv, und zum anderen hat sie das Problem der Ciffretext Ausdehnung, die eine verlangsamte Kommunikation zur Folge hat.

Moderne HE Verfahren basieren häufig auf Lernen mit Fehlern (LWE), bzw. der zugehörigen Ring Variante Ringlernen mit Fehlern (RLWE). Im Speziellen werden wir uns mit dem symmetrischen BFV HE Verfahren beschäftigen, das auf eben diesen aufbaut.

Zum Großteil basierend auf einem kürzlich erschienenen Papiers [Che+20], beschäftigt sich diese Arbeit mit der Beschleunigung der Übertragung vom lokalen Rechner zum Server, d.h. mit der Lösung des Problems der Ciffretext Ausdehnung. Der theoretische Teil ist in drei Hauptpunkte gegliedert: Erstens, die Einführung eines platzsparenden LWE Chiffretextes. Zweitens, die Beschreibung einer effizienten Konvertierung zwischen LWE und RLWE Chiffretexten. Und schlussendlich, eine effiziente Methode um den resultierenden RLWE Chiffretext in einen HE Chiffretext zu konvertieren.

Im praktischen Teil werden diese drei Schritte in Microsofts quelloffene Software Bibliothek SEAL implementiert. Es werden Laufzeiten gemessen und mit den Ergebnissen des originalen Papiers [Che+20] verglichen. Zusätzlich werden die Ergebnisse auch mit einer alternativen Methode den Aufwand der Übertragung zu verringern, verglichen. Nämlich mit der Hybride Homomorphe Verschlüsselung (HHE) aus [Dob+21].

Schlagwörter: Homomorphe Verschlüsselung, Datenschutz, Sicherheit, Simple Encrypted Arithmetic Library, Lernen mit Fehlern, Ringlernen mit Fehlern, Brakerski/Fan-Vercauteren

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1.

Introduction

Homomorphic Encryption (HE), [RAD+78; Gen09], is considered as an optimal future solution to perform computations in the cloud without revealing any user data. In this section, we will give a short introduction to outsourced computation, HE, some drawbacks of HE and our contribution to reduce at least a few disadvantages of using HE already today.

1.1. Problem Description

Before we get into the details of this thesis, we will start with a short introduction, to give the reader some background and to motivate this work.

Especially, we will introduce the idea of Outsourced Computation and give a high level overview of HE and its special case of RLWE, [LPR10]. We will see, that in practice a full RLWEs encryption on the client side is mostly an overkill: Firstly, because RLWEs encryption is computationally expensive. And secondly, in general the client has to send the homomorphically encrypted data to a powerful server for further computation. This could be a problem, because if it only wants to send small datapoints repeatedly, the communication overhead explodes. This is, because modern HE schemes suffer from ciphertext expansion, which means, that the resulting ciphertext is significantly larger than the original plaintext.

Therefore, we will present the idea of lightweight communication, to solve the problem of the huge communication overhead. In particular, the client will only have to calculate a more efficient LWE symmetric encryption, [Bra12], instead of an RLWE symmetric encryption, [FV12], and will then send a much smaller, lightweight, ciphertext to the server.

In order for the server to proceed with its computation, we need, in addition to this lightweight communication, some conversion algorithms. These conversion algorithms were presented in a recently published paper, [Che+20], on which this thesis is heavily based on.

The server then will perform all the heavy tasks, as usual.

Outsourced Computation A client, on any kind of edge device, wishes to perform a couple of expensive computations on some personal data. Its computational power is insufficient, so it decides to outsource this computation to a performant server. Therefore the client sends the data to the server. The server operates on it, and sends the result back to the client.

This simple use case, however, introduces huge privacy problems: In order to perform the operations, the server has to work with the unencrypted data. So this setup possibly reveals the user's personal data to the server, as illustrated in Figure 1.1.

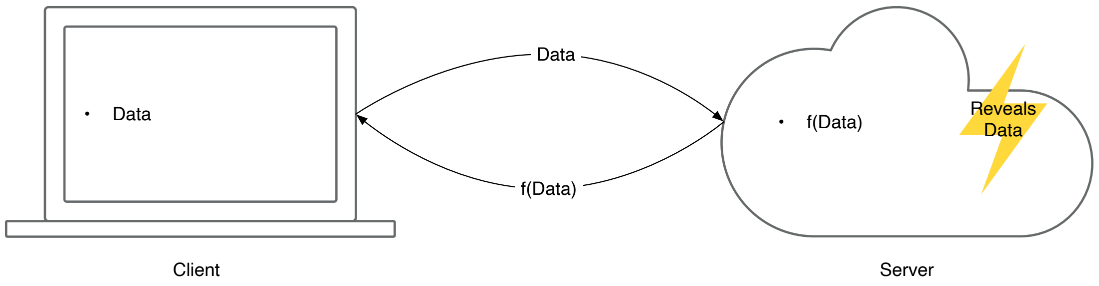


Figure 1.1.: Outsourced Computation Without Homomorphic Encryption (HE) (Incl. Privacy Problem Of Revealing Personal Data To The Server)

Simply encrypting the data is unfortunately not an option, since in general, the server then would not be able to perform computations on it. However, we still want the server to do the expensive operations on the user's data. But we would like to achieve a setup, where the server could not see the data it is operating on. Fortunately, there already exists a solution to this problem, called Homomorphic Encryption (HE).

Homomorphic Encryption (HE) Like the name suggests, HE is based on mathematical homomorphism. A very high-level description of HE is the following: For a map f and an operation $\oplus$, a homomorphism preserves the operations. This means it holds

$$f\left(x \oplus y\right) = f\left(x\right) \oplus f\left(y\right).$$

In case of encryption we could use this property in the following way: Given two messages m_1 and m_2. One can compute the operation $\oplus$, which could be for example, a multiplication or addition, of the encryptions of the messages (see right hand side of Equation (1.1)), and get the same result as if we encrypt the result of the operation $\oplus$ on the two messages (see left hand side of Equation (1.1)).

$$E\left(m_1 \oplus m_2\right) = E\left(m_1\right) \oplus E\left(m_2\right) \tag{1.1}$$

That means, we can operate on encrypted data and get the same result as if we operate on the plain data and encrypt it afterwards. Having an efficient HE scheme allows us to operate on the data without knowing it.

1.2. Background

We will continue with a further, but just sketchy, insight into HE and its usage in case of outsourced computation.

Properties of Homomorphic Encryption (HE) Modern HE schemes often rely on the LWE hardness assumption [Reg05], or its ring variant RLWE [LPR10].

In case of LWE, we encrypt a single integer value. In contrast, with an RLWE based encryption scheme, we can encrypt a polynomial with integer coefficients up to a certain degree.

In both cases, a newly encrypted ciphertext has a certain amount of noise budget, given in bits. Performing homomorphic calculations, for example an addition or a multiplication, on a ciphertext, reduces the noise budget by a certain value. If the noise budget is zero, then the decryption will fail. Therefore, noise consumption is in general the most limiting factor in HE schemes.

Additions do not consume much noise and are therefore often considered as free. However, multiplications consume a significant amount of noise, which is why the multiplicative depth is the most important performance metric of any HE use case.

Packing Packing allows us to encode a vector of integers into a single polynomial, such that polynomial addition and multiplication correspond to element-wise vector addition and multiplication. This allows us to perform HE in parallel on many integers, which will be useful later-on. In this thesis we refer to a ciphertext encrypting a packed polynomial as HE ciphertext.

Outsourced Computation with Homomorphic Encryption (HE) Using HE in the case of outsourced computation, is depicted in Figure 1.2: The setting is quite the same as in Figure 1.1, but instead of the raw data, we send the data homomorphically encrypted. HE now enables the server to compute on the encrypted data, but without revealing the possibly sensitive user data. After sending the homomorphically encrypted result back, the client can decrypt the results.

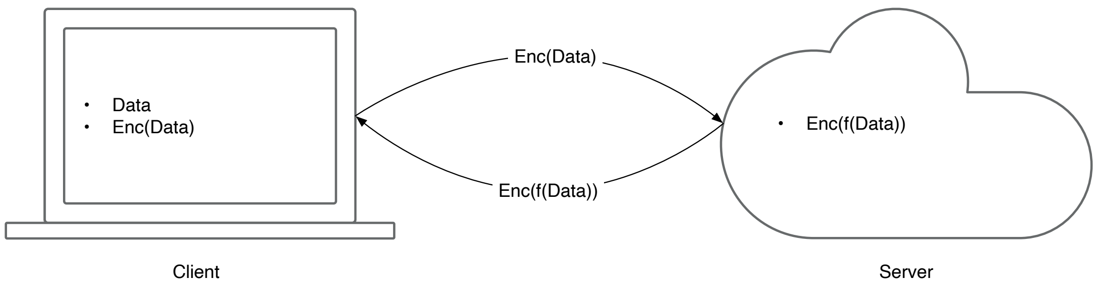


Figure 1.2.: Outsourced Computation With Homomorphic Encryption (HE)

Outsourcing the expensive operations to a server using HE, enables us to fully preserve the privacy of the data.

Drawbacks The question now arises, why HE, or in particular HE based on the RLWE hardness assumption, like the state-of-the-art BFV [Bra12; FV12] scheme, is not used everywhere.

One of the drawbacks is that currently HE is very expensive in terms of computational complexity.

Furthermore, the communication between the client and the server can be slow, because of the amount of data which has to be transmitted: A general downside of HE is, that it suffers from ciphertext expansion, which means, that the resulting ciphertext is significantly larger, than the original plaintext, [Dob+21, Sec. 1]. Especially when only a small amount of data needs to be transmitted.

1.3. Homomorphic Encryption (HE) With Outsourced Computation: Now & Then

In this section, we will describe the use case of HE with outsourced computation. Especially we will compare the current, traditional way of the client-server communication, against the way it is done using the idea of lightweight communication, i.e. a way to mitigate the common problem of ciphertext expansion when using HE.

1.3.1. Traditional Way

We will start by discussing the use case in the traditional setup.

Client The client starts with the plaintext, on which it wants the server to perform some, probably computationally expensive, operations on. This plaintext is a vector $\mathbf{m}$, which contains multiple, i.e. n, integers. I.e.,

$$\text{pt: } \mathbf{m} \in \mathbb{Z}_p^n.$$

Next, it packs, or merges, this vector of integers $\mathbf{m}$ into a single polynomial m. This step is called Packing, or Batching, and uses the inverse of a Number Theoretic Transform (NTT), [Har14]. I.e.,

$$\text{pack (using iNTT): } \mathbf{m} \to m \in \mathbb{Z}_p\left[X\right] / \left(X^N + 1\right).$$

Then, it homomorphically encrypts the resulting polynomial m with, for example, the BFV HE scheme, which depends on the RLWE hardness assumption. I.e.,

$$\text{enc: } ct = \text{Enc}_{\text{RLWE}}\left(m\right).$$

Note, in this thesis we refer to this type of ciphertext, i.e. a ciphertext encrypting a packed polynomial, as HE ciphertext.

Finally, the client sends the resulting RLWE ciphertext ct to the server. I.e.,

$$\text{Client} \xrightarrow{ct} \text{Server}.$$

Because HE suffers from ciphertext expansion [Dob+21, Sec. 1], the communication overhead in this step can be huge. This means, that the RLWE ciphertext ct, which the client sends to the server, is significantly larger than the original plaintext.

Server After receiving the RLWE ciphertext ct, the server first homomorphically performs the desired calculations on the ciphertext ct. I.e. $ct' = f(ct)$ Because we previously merged multiple integers into a single polynomial, i.e. performed Packing, the server now is able to perform its calculation efficiently on all integers in parallel.

Afterwards, it sends back the resulting RLWE ciphertext ct' to the client. I.e.,

$$\text{Server} \xrightarrow{ct'} \text{Client}.$$

Client After getting back the homomorphically encrypted result from the server, the client first has to decrypt the RLWE ciphertext ct'. I.e.,

$$\text{dec: } m' = \text{Dec}_{\text{RLWE}}\left(ct'\right).$$

The resulting plaintext m' is again a polynomial, i.e. $m' \in \mathbb{Z}_p[X] / \left(X^N + 1\right)$.

Finally, it unpacks the polynomial m', and gets the desired vector of the n modified plaintext integers. I.e.,

$$\text{unpack: } m' \to \mathbf{m}' \in \mathbb{Z}_p^n.$$

1.3.2. Using Lightweight Communication

We will now discuss the same use case as in the previous section, but this time applying the idea of lightweight communication, in order to mitigate the problem of ciphertext expansion.

Note: In this section, we give a quite general description of the overall use case. For the exact use case, which we will follow and also implement in the course of this thesis, see Section 6.1.

Client Like before, the client prepares its plaintext $\mathbf{m}$. I.e.,

$$\text{pt: } \mathbf{m} \in \mathbb{Z}_p^n.$$

In contrast to the traditional way, it now encrypts each integer $\mathbf{m}[i]$ individually with a lightweight LWE based HE scheme. This results in n ciphertexts c_i. I.e.,

$$\text{enc: } ct_i = \text{Enc}_{\text{LWE}}(\mathbf{m}[i]) \quad \forall i \in [n].$$

Finally, the client sends the n LWE ciphertexts ct_i to the server. I.e.,

$$\text{Client} \xrightarrow{ct_i} \text{Server} \quad \forall i \in [n].$$

By using these lightweight LWE ciphertexts, we mitigate the problem of HE ciphertext expansion, in other words, the huge communication overhead between the client and the server. This is, because the lightweight LWE ciphertexts are much smaller compared to the original RLWE ciphertext.

Server After receiving the n LWE ciphertexts ct_i, for $i \in [n]$, the server uses efficient conversion algorithms to merge these n LWE ciphertexts into a single RLWE ciphertext. I.e.,

$$\text{LWEs} \to \text{RLWE}.$$

After the conversion into an RLWE ciphertext, the values are stored in the coefficients of the polynomial. To be able to perform further computations on these values (in parallel), the server has to homomorphically evaluate the Packing algorithm, i.e. the inverse of the NTT, to efficiently pack the coefficients into slots. We say, that it converts the RLWE ciphertext into a HE ciphertext. I.e.,

$$\text{pack (i.e. coeff-to-slots, using hom. iNTT): RLWE} \to \text{HE}.$$

We call the resulting HE ciphertext ct.

The server is now able to homomorphically perform the desired calculations on the ciphertext ct, which results in the ciphertext $ct' = f\,(ct)$.

Finally, the server sends back the resulting RLWE ciphertext ct' to the client. I.e.,

$$\text{Server} \xrightarrow{ct'} \text{Client}.$$

Client After getting back the homomorphically encrypted result from the server, the client first has to decrypt the HE ciphertext ct'. I.e.,

$$\text{dec: } m' = \text{Dec}_{\text{RLWE}}\left(ct'\right).$$

The resulting plaintext m' is again a polynomial, i.e. $m' \in \mathbb{Z}_p\,[X]\,/\,\left(X^N + 1\right)$.

Like before, the client finally unpacks the polynomial m', and gets the desired vector of the n modified plaintext integers. I.e.,

$$\text{unpack: } m' \to \mathbf{m}' \in \mathbb{Z}_p^n.$$

1.4. Our Contribution

The goal of this thesis is to reduce the communication overhead, in other words, to solve the problem of the ciphertext expansion, which comes with HE, especially with RLWE based schemes. This will be done in three main steps, i.e.

- introduction of lightweight LWE ciphertexts, which mitigates the problem of ciphertext expansion,
- usage of efficient homomorphic conversion algorithms from [Che+20], and
- efficient coefficient-to-slots conversion.

The efficient homomorphic conversion between ciphertexts is the main contribution of [Che+20, Sec. 4.2]. But also the lightweight LWE ciphertexts and the final efficient coefficient-to-slots conversion are briefly mentioned there. However, no public implementation is available and a detailed description of the final efficient coefficient-to-slots transformation is missing.

We will discuss our exact use case, which is shown in Figure 6.1, in detail in Section 6.1. But before, we start by briefly introduce lightweight LWE ciphertexts.

Lightweight Learning With Errors (LWE) Ciphertext Instead of full RLWE ciphertexts, which consist of two polynomials of degree N, we will only send some lightweight symmetric LWE ciphertexts to the server. The idea of the lightweight LWE ciphertexts is mentioned in [Che+20, Sec. 4.2].

We note, that in a regular LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$ the vector $\mathbf{a}$ is purely random. That randomness of $\mathbf{a}$ allows us to modify the regular LWE encryption algorithm and build a much smaller, lightweight LWE ciphertext (b, se), with a sampled seed se. A Pseudo-Random Number Generator (PRNG) then uses this seed to calculate $\mathbf{a}$. The scalar b is calculated as in the original scheme. The client can now send much smaller ciphertexts to the server, i.e. we reduced the communication cost.

Additionally, we gain another advantage: If we want to create multtiple LWE ciphertexts (to send them to the server afterwards) we can reuse the seed se and calculate $\mathbf{a}_i$ with a counter i, for each ciphertext $(b_i, \mathbf{a}_i)$.

This is quite important, because assume we want to encrypt and send n integers, for $n < N$. Using RLWE, we have to send the encrypted polynomials, which always have degree N, independent of used slots n. But when using lightweight LWE ciphertexts, we only have to send n integers plus a seed. This means, we reduced the communication cost, respectively solved the problem of ciphertext expansion.

Efficient Homomorphic Conversion Algorithms In addition to the sending of these lightweight LWE ciphertexts, we need some conversion algorithms on the server-side to convert between LWE, RLWE and multiple LWEs ciphertexts. In the implementation, see Section 6, we will use the novel algorithms, which were introduced recently in [Che+20], i.e. LWE-to-LWE(), LWE-to-RLWE() and LWEs-to-LWE().

Efficient Coefficients-to-Slots Conversion The last step is, that the server will have to perform some kind of coefficient-to-slots conversion for further computations. Doing this, we transform the RLWE ciphertext into a HE ciphertext.

The advantage of a HE ciphertext is, that the polynomial encodes a vector of plaintexts. In this case the polynomial addition and polynomial multiplication is performed vector-element-wise. This means an HE additoin/multiplication is equivalent to a parallel integer addition/multiplication, which can result in a huge performance gain.

Implementation The goal is to extend Microsofts open source homomorphic library SEAL, [20], with all of these functionalities stated above. The implementation of the three steps is done in C++.

Benchmarks & Comparison Finally, we will perform benchmarks of the use case, see Section 6.1, and compare the results with [Che+20].

Additionally we will also compare our results with another way of reducing the communication overhead, i.e. using HHE, used in [Dob+21].

1.5. Related Work

When discussing related work, we will specially mention an alternative approach of achieving lightweight communication, see Section 1.5.1, and also state a current real world application of HE, see Section 1.5.2.

1.5.1. Hybrid Homomorphic Encryption (HHE) with Symmetric Ciphers

A different approach to reduce the communication overhead, has been presented in [NLV11], and was recently discussed in [Dob+21]. There, HHE with symmetric ciphers are used. We will briefly outline the idea of HHE [Dob+21]:

The client first encrypts sensitive data with a symmetric cipher and sends it to the server. Then, the client also homomorphically encrypts the previously used symmetric key, and sends it to the server. Then, the server, using this received homomorphically encrypted symmetric key, is able to transform the symmetric encrypted user data into an homomorphic ciphertext, by evaluating the symmetric decryption circuit. This means, the server decrypts the symmetric encrypted user data homomorphically, by using the homomorphically encrypted secret key.

In more detail: Generally, the decryption circuit of symmetric ciphers only consist of additions and multiplications. Therefore, the server is able to homomorphically evaluate the symmetric decryption circuit using, firstly, the symmetric encrypted user data and, secondly, the homomorphic encrypted symmetric key as inputs. This results in homomorphic encrypted user data.

After the conversion to a HE ciphertext, the server can, similar to our approach, perform further computations on it.

There exist multiple dedicated symmetric ciphers for HHE. In [Dob+21], the authors propose PASTA and extensively compare it to other solutions. It is shown, that PASTA outperforms their competitors for HHE.

We compare the runtimes of our implementation against the HHE approach of [Dob+21] in Section 7.4.

1.5.2. Privacy for Human Mobility Data Model

Human mobility plays an important role when spreading diseases. In the current COVID-19 pandemic, scientists, but also politicians, want to be informed of the mobility behaviour of humans, in particular sick ones, to mitigate further spreading. Mobile phone location data are therefore interesting and meaningful resources. In a simple approach, a mobile network operator (MNO), which is in possession of this sensitive data, aggregates them and creates a mobility model. The problem with this approach is, that it allows tracing a single person. Especially, if only sick humans, which of course are the most interesting ones, are considered. The MNO in this case would have to know users identities combined with their current health status. This, of course, leads to a huge privacy concern.

A team at the Institute of Applied Information Processing and Communications (IAIK) at the TU Graz recently published a solution to this problem, which preserves the privacy of mobile phone owners, [Bam+20]. They are able to connect the necessary health and mobility data in a way, which is compatible to the General Data Protection Regulation (GDPR) which applies in the European Union (EU). Especially, they used HE for expensive outsourced computations and other methods of applied cryptography to achieve the necessary privacy.

Because of the big amount of data which is needed in this use case, a way to reduce the communication could be a practical and therefore a welcome addition.

1.6. Outline

We want to give the reader a quick overview of the content and sections of this thesis.

Preliminaries We will start with some important preliminaries. I.e. in Section 2, we will give some mathematical background, an introduction to HE, respectively introduce the BFV HE scheme. Additionally, we will mention Microsofts open source software library SEAL and mention different useful techniques, like Residue Number System (RNS), NTT, and Batching.

Theoretical Work The theoretical part of this thesis is split into three main building blocks for achieving lightweight communication.

Firstly, in Section 3, we will introduce a lightweight variant of a LWE ciphertext. It allows us to minimise the data which has to be sent from the client to the server.

Secondly, in Section 4, we present the efficient homomorphic conversion algorithms, which the server uses to efficiently convert received ciphertexts. I.e. these conversion algorithms enable us to efficiently convert between LWE ciphertexts and RLWE ciphertexts.

Finally, in Section 5, we introduce an efficient conversion from coefficients into slots. After the conversions in Section 4, the original plaintexts are stored in the coefficients of the RLWE ciphertext, which is a polynomial. In order to perform further, parallel, computations on it, we have to convert this RLWE ciphertext into a HE ciphertext.

Practical Work In the practical part of this thesis, first we will introduce our use case and describe the implementation of the algorithms developed in Section 3, 4, and 5 into Microsofts open source HE software library SEAL, see Section 6.

Finally, in Section 7, we perform multiple benchmarks of our implementation. The results will not only be compared to the reference implementation of [Che+20], but also to a second approach of reducing the communication overhead using HHE with symmetric ciphers, i.e. we will compare to the PASTA implementation of [Dob+21].

Chapter 2.

Preliminaries

In this section we will give the reader some background information, such as notation, some mathematical building blocks, an introduction to HE, the BFV HE-scheme, the homomorphic encryption library SEAL, packing, etc., which will be needed during the whole thesis further on.

2.1. Notation

We write a vector in bold, i.e. $\mathbf{a}$, and the i-th entry of vector $\mathbf{a}$ is $\mathbf{a}[i]$. Similarly, for the polynomial $a(X)$, we write the i-th coefficient $a(X)$ as $a[i]$. Subscripts are used for numbering some arbitrary numbers, for example N integers c_i for $i \in [N]$. The inner product of two vectors $\mathbf{a}$ and $\mathbf{b}$ is defined as $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_i \mathbf{a}[i] \cdot \mathbf{b}[i]$ For a finite set S, we write the uniform distribution of S as $U(S)$. To represent the set $\mathbb{Z}_q$, we identify it by $\mathbb{Z} \cap \left[-\frac{q}{2}, \frac{q}{2}\right)$, i.e. $\mathbb{Z}_q = \mathbb{Z} \cap \left[-\frac{q}{2}, \frac{q}{2}\right)$. The index set is defined as $[N] = \{0, 1, \ldots, N-1\}$. For rounding a number a to the nearest integer, we write $\lfloor a \rceil$. Additionally for rounding up a number a we write $\lceil a \rceil$, and for rounding down $\lfloor a \rfloor$. The absolute value of a number a is denoted by $|a|$. Whereas the infinity norm of vector $\mathbf{a}$ is written as $\|\mathbf{a}\| = \|\mathbf{a}\|_\infty = \max_i |\mathbf{a}[i]|$. The modulus reduction of an integer a by modulus t is denoted by $a \mod t$. The modulus reduction of a polynomial $a(X)$ by an integer t is done by performing the modulus reduction to every integer coefficient of the polynomial separately. This modulus reduction is done into the interval $\left[-\frac{t}{2}, \frac{t}{2}\right)$. The logarithm to base a is denoted as $\log_a$. Instead of $\log_2$ we may simply write log. The ring of polynomials of degree less than N is defined as $R = \mathbb{Z}[X]/(X^N + 1)$. Additionally, the ring R with coefficients reduces modulo q is denoted by $R_q = R/qR = \mathbb{Z}_q[X]/(X^N + 1)$.

Further notations, respectively symbol explanations, are listed in the appendix on page 89.

2.2. Mathematical Building Blocks

In this section we briefly describe the general mathematical building blocks we will use later.

2.2.1. Learning With Errors (LWE) Hardness Assumption

All schemes we use in this thesis are based on the LWE hardness assumption, [Reg05], and its variant over polynomial rings, i.e. RLWE, [LPR10].

More precisely, the LWE encryption scheme in Section 2.5.1 is based on the LWE hardness assumption described in the current section, and the RLWE encryption scheme in Section 2.5.2 is based on the RLWE hardness assumption from Section 2.2.3.

We will start with the LWE hardness assumption, [Reg05], [Che+20, Sec. 2.2], [Bra12, Def. 2.1], [BGV12, Def. 2]:

Learning With Errors (LWE) Distribution Given some dimension N, the ciphertext modulus q, an error distribution ψ over $\mathbb{Z}$, and a key distribution χ over $\mathbb{Z}^N$.
We choose a secret key vector $\mathbf{s} \in \mathbb{Z}^N$, sample a vector $\mathbf{a} \leftarrow U\left(\mathbb{Z}_q^N\right)$, sample some scalar noise $e \leftarrow \psi$, and calculate the scalar $b = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q$.
The LWE distribution over $\mathbb{Z}_q^{N+1}$ is then defined as the scalar-vector tuple $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$.
Learning With Errors (LWE) Hardness Assumption The LWE hardness assumption, i.e. Decision LWE, is defined as follows:

- Given parameters (N, q, χ, ψ).
- It is computationally infeasible to distinguish between the LWE distribution $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$ for a secret key $\mathbf{s} \leftarrow \chi$ and the uniform distribution $(b, \mathbf{a}) \leftarrow U\left(\mathbb{Z}_q^{N+1}\right)$.

We will state the ring variant of the LWE hardness assumption, i.e. the RLWE hardness assumption in Section 2.2.3. But beforehand, we need a short introduction to rings, see Section 2.2.2.

2.2.2. Cyclotomic Field & Rings

Let $N = 2^k, k \in \mathbb{N}$ be a power-of-two integer. We define ζ as a $2N$-th root of unity. Furthermore we define the $2N$-th cyclotomic field $K = \mathbb{Q}(\zeta)$, and $R = \mathbb{Z}[\zeta]$ as the ring of integers of K.

Using the map $\zeta \mapsto X$, we can identify the field K with $\mathbb{Q}[X] / \left(X^N + 1\right)$. In the same way we identify the ring R with $\mathbb{Z}[X] / \left(X^N + 1\right)$. Finally we define the residue ring of R modulo an integer q as $R_q = R/qR$, [Che+20, Sec. 2.1], [Che+18b, Sec. 2.1].

In this thesis we will mostly work with elements of $R_q = R/qR = \mathbb{Z}_q[X] / \left(X^N + 1\right)$. This is the ring of polynomials of degree less than N, with coefficients reduced modulo q. I.e. this ring consists of all polynomials of the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}, \tag{2.1}$$

with $a_i \in \mathbb{Z}_q, \forall i \in [N]$.

2.2.3. Ring Learning With Errors (RLWE) Hardness Assumption

The RLWE encryption scheme in Section 2.5.2 is based on the RLWE hardness assumption from this Section. The RLWE hardness assumption is a translation of the LWE hardness assumption from Section 2.2.1 to polynomial rings, [LPR10], [Che+20, Se. 2.2], [BGV12, Def. 4], [FV12, Def. 1] and [Lai17, Def. 3]:

Ring Learning With Errors (RLWE) Distribution Given some dimension N, the ciphertext modulus q, a ring R, the residue ring of R modulo q, defined as R_q, an error distribution ψ over the ring R, and a key distribution χ over the ring R.
We choose a secret key ring element $s \in \chi$, sample a ring element $a \leftarrow U(R_q)$, sample some noise ring element $e \leftarrow \psi$, and calculate the ring element $b = a \cdot s + e \mod q$, where $a \cdot s$ is a ring-product.
The RLWE distribution over R_q^2 is then defined as the tuple of ring elements $(b, a) \in R_q^2$.
Ring Learning With Errors (RLWE) Hardness Assumption The RLWE hardness assumption, i.e. Decision RLWE, is defined as follows:

- Given parameters (N, q, χ, ψ).
- It is computationally infeasible to distinguish between the RLWE distribution $(b, a) \in R_q^2$ for a secret key $s \leftarrow \chi$ and the uniform distribution $(b, a) \leftarrow U\left(R_q^2\right)$.

Note, in this thesis we will use the ring $R = \mathbb{Z}[X] / \left(X^N + 1\right)$, and elements in $R_q = R/qR = \mathbb{Z}_q[X] / \left(X^N + 1\right)$. In our case a ring element is a polynomial of the form (2.1), see Section 2.2.2.

2.2.4. Chinese Remainder Theorem (CRT)

In this Section we will state the Chinese Remainder Theorem (CRT). Find details including the proof in [Cor+09, Sec. 31.5].

Let n be an integer, which is the product of k pairwise coprimes n_i, i.e. $n = \prod_{i=1}^{k} n_i$. The CRT gives us two main properties:

- The structure of $\mathbb{Z}_n$ is identical to $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$, where addition and multiplication are performed component-wise modulo n_i in the i-th component.
- CRT allows us to perform some operations in a more efficient way by doing this operation separately in all $\mathbb{Z}_{n_i}$, for $i \in \{1, \ldots, k\}$, than performing the same operation only once directly modulo n.

The following theorem, i.e. Theorem 2.1 is exactly stated in [Cor+09, Theorem 31.27].

Theorem 2.1 (Chinese Remainder Theorem (CRT) [Cor+09, Theorem 31.27])**.** *Let $n = n_1 \cdot n_2 \cdots n_k = \prod_{i=1}^{k} n_i$, where the n_i are pairwise coprime. Consider the correspondence*

$$a \leftrightarrow (a_1, a_2, \dots, a_k)\,, \tag{2.2}$$

where $a_i \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$, and

$$a_i = a \mod n_i$$

for $i = 1, 2, \dots, k$. Then, mapping (2.2) is a one-to-one correspondence (bijection) between $\mathbb{Z}_n$ and the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operations performed on the elements of $\mathbb{Z}_n$ can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$a \leftrightarrow (a_1, a_2, \dots, a_k)\,,$$

and

$$b \leftrightarrow (b_1, b_2, \dots, b_k)\,,$$

then

$$\begin{aligned}
(a+b) \mod n &\leftrightarrow ((a_1+b_1) \mod n_1, \dots, (a_k+b_k) \mod n_k)\,,\\
(a-b) \mod n &\leftrightarrow ((a_1-b_1) \mod n_1, \dots, (a_k-b_k) \mod n_k)\,,\\
(a\cdot b) \mod n &\leftrightarrow ((a_1\cdot b_1) \mod n_1, \dots, (a_k\cdot b_k) \mod n_k)\,.
\end{aligned}$$

Proof. See [Cor+09, Sec. 31.5]. □

We will use the CRT on two locations in this thesis: First, in Section 2.6.1 we use the CRT to split the ciphertext modulus q into a product of primes q_i, i.e. $q = \prod_i q_i$. And second, in Section 2.7, CRT is used to split the polynomial $\left(X^N + 1\right)$ into the product of $(X - \alpha_i)$, i.e.

$$\left(X^N + 1\right) = \prod_{i=0}^{N-1} (X - \alpha_i) \mod t,$$

for $\alpha_i = \zeta^{2i+1}$, see Equation (2.6).

2.3. Homomorphic Encryption (HE)

HE allows us to perform any computation on encrypted data, without knowing the secret encryption key. It is therefore often considered as the holy grail of cryptography. In general one can distinguish between multiple types of HE, see [Bam+20, Sec. 3.2] and [Dob+21, Sec. 2]:

- A Partially Homomorphic Encryption (PHE) scheme allows us to perform a limited set of operations on some encrypted data.
- A Somewhat Homomorphic Encryption (SWHE) scheme allows us to evaluate an arbitrary circuit over encrypted data, but only up to a certain depth.

- Finally, a Fully Homomorphic Encryption (FHE) scheme allows us to evaluate an arbitrary circuit on encrypted data, without any depth constraints.

The idea of HE was first introduced by Rivest et al. [RAD+78], but back in these days, there were only Partially Homomorphic Encryption (PHE) schemes. For example the Rivest–Shamir–Adleman (RSA) encryption scheme [RSA78] is only homomorphic for multiplication and Paillier's encryption scheme [Pai99] is only homomorphic for addition.

The big breakthrough came in 2009 with Gentry's work [Gen09], where the first Fully Homomorphic Encryption (FHE) encryption scheme was introduced. This scheme is based on ideal lattices, but was also considered as too impractical to use. However, it was the motivation for multiple other HE schemes, like [Bra12], [FV12], [BGV12], [Che+20].

Modern HE schemes are often based on the LWE hardness assumption [Reg05], or its polynomial ring variant the RLWE hardness assumption [LPR10], see Section 2.2.1 respecively Section 2.2.3. In a nutshell, an LWE or RLWE ciphertext initially contains some random noise. During operations on this ciphertext, for example performing additions or multiplications, the noise grows. In case of addition, the noise grows only a little and is therefore neglectable, but during multiplication the impact is significant. The decryption is only possibly as long as the noise doesn't reach a certain threshold, leading to a limited number of multiplications. Because of the restriction of the total number of allowed multiplications, we talk about a Somewhat Homomorphic Encryption (SWHE) scheme.

In [Gen09] Gentry introduced a technique to turn a SWHE into a FHE scheme, called bootstrapping. Bootstrapping allows to reduce or reset the noise of a homomorphic ciphertext. So there would not be an upper bound for the multiplicative depth of the circuit evaluation any more. The problem of this bootstrapping is, that it is operationally expensive. In practice, one often chooses to use a SWHE scheme with sufficiently large parameters to evaluate a given use case.

In this thesis we use the BFV SWHE scheme [Bra12], [FV12] for all homomorphic encryptions.

2.4. Learning With Errors (LWE) & Ring Learning With Errors (RLWE)

Before we deep dive into details of a specific HE scheme, we will give a short introduction to HE schemes which are based on the LWE or RLWE hardness assumptions, see Section 2.2.1 respectively Section 2.2.3.

Modern HE schemes often rely on the LWE, [Bra12, Def. 2.1], or its Ring variant RLWE, [LPR10], hardness assumption. We first choose some dimension N. In case of SEAL, N has to be a power-of-two, i.e. $N = 2^k$ for some integer $k \geq 12$, see Section 2.6.3. Depending on N we set a ciphertext modulus q, which is a product of some primes, i.e. $q = \prod q_i$ with $q_i \in \mathbb{P}$. The ciphertext modulus q typically has several hundred bits. Finally we set a plaintext modulus t, which is some integer, for example 1024.

In case of LWE the goal is to encrypt an integer $m \in \mathbb{Z}_t$. That means, the integer is reduced modulo the plaintext modulus t. The secret key $\mathbf{s}$ is a vector in $\mathbb{Z}^N$. The LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$ consists of a scalar $b \in \mathbb{Z}_q$ and a randomly chosen vector $\mathbf{a} \in \mathbb{Z}_q^N$. The phase $\mu \in \mathbb{Z}_q$ is defined as $\mu = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q$, and is an encoding of the plaintext, which also includes some noise. As long as this noise is small enough, the ciphertext can be decrypted correctly.

The ring variant, i.e. RLWE, is a translation of LWE to the polynomial ring setting: In particular we use the residue ring R modulo q: $R_q = R/qR = \mathbb{Z}_q[X] / \left(X^N + 1\right)$. This is the ring of polynomials of degree less than N, with coefficients reduced modulo q. I.e. it consists of polynomials of the form $a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}$. In contrast to the LWE encryption, where we encrypt an integer, here we encrypt a polynomial $m \in R_t$. R_t means, that the coefficients of the polynomial are reduced modulo the plaintext modulus t. The secret key s is an element of the ring R, that means it is a polynomial. The RLWE ciphertext $(b, a) \in R_q^2$ consists of two polynomials b and a, where a is again a randomly chosen polynomial. The phase $\mu \in R_q$ is defined as $\mu = b + as \mod q$. It is again an encoding of the plaintext, which includes some noise.

2.5. Brakerski/Fan-Vercauteren (BFV) Homomorphic Encryption (HE)-scheme

In this section, we will describe the BFV HE scheme [Bra12; FV12], which we will mostly use in this thesis.

More precisely, we introduce the symmetric encryption scheme, i.e. key generation, encryption and decryption formulas for the symmetric BFV HE scheme, for both LWE (Section 2.5.1) and RLWE (Section 2.5.2) variants, which are based on the the LWE and respectively RLWE hardness assumptions, see Section 2.2.1 and Section 2.2.3 respectively. This results in the introduction of LWE and RLWE ciphertexts, which we use later on.

Note: LWE/RLWE are hardness assumptions. Nevertheless, in this thesis we may refer to the basic schemes based on LWE/RLWE as LWE/RLWE encryption schemes.

2.5.1. Brakerski/Fan-Vercauteren (BFV) Homomorphic Encryption (HE)-scheme with Learning With Errors (LWE)

We start with the LWE variant, i.e. we describe the symmetric en-/decryption of the BFV HE-scheme in the LWE variant, [HPS19, Sec. 3.1], [Bra12], and [Reg05].

Preconditions Before we start with the actual encryption, we have to set some encryption parameters.

Preconditions

- choose: dimension N, plaintext modulus t, where $t > 1$, and ciphertext modulus q, where $q \gg t$

Learning With Errors (LWE) KeyGen After fixing all necessary parameters, we can generate the LWE key, which is a vector in case of LWE.

LWE KeyGen

- sample: secret key $\mathbf{s} \leftarrow \mathbb{Z}^N$, with $\|\mathbf{s}\| \ll \frac{q}{t}$

Learning With Errors (LWE) Encryption Using the just generated LWE secret key vector $\mathbf{s}$, we are now able to homomorphically and symmetrically encrypt an LWE plaintext m, which is an integer.

LWE Encryption, i.e. encrypt LWE plaintext $m \in \mathbb{Z}_t$ into an LWE ciphertext $\mathrm{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$:

- input: N, q, t, m, $\mathbf{s}$
- encode: LWE plaintext m into phase μ
  - calc: $\Delta = \lfloor \frac{q}{t} \rfloor$
  - sample: Gaussian noise e, with $|e| \ll \frac{q}{t}$
  - calc: $\mu = (\Delta m + e) \mod q \in \mathbb{Z}_q$
- sample: $\mathbf{a} \leftarrow U(\mathbb{Z}_q^N)$
- calc: $b = -\langle \mathbf{a}, \mathbf{s} \rangle + \mu \mod q \in \mathbb{Z}_q$
- return: LWE ciphertext $\mathrm{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$

Note, the phase μ can be seen as a randomised encoding of the plaintext m, [Che+20, Sec. 4.2]. Additionally note, using this HE scheme, all operations on the ciphertext are equivalent to operations on the plaintext in $\mathbb{Z}_t$.

Learning With Errors (LWE) Decryption The reverse direction, i.e. the symmetric LWE decryption, is done as follows. I.e. using the same secret key $\mathbf{s}$ as during the encryption process, we can recover the original LWE plaintext m.

LWE Decryption, i.e. decrypt LWE ciphertext $\mathrm{ct} = (b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$ into an LWE plaintext $m \in \mathbb{Z}_t$:

- input: q, t, $\mathrm{ct} = (b, \mathbf{a})$, $\mathbf{s}$
- calc: $\mu = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$
- decode: phase μ into LWE plaintext m
  - calc: $m = \left\lfloor \frac{t\mu}{q} \right\rceil \mod t$
- return: LWE plaintext $m \in \mathbb{Z}_t$

2.5.2. Brakerski/Fan-Vercauteren (BFV) Homomorphic Encryption (HE)-scheme with Ring Learning With Errors (RLWE)

After introducing the LWE encryption scheme in Section 2.5.1, we now derive the symmetric RLWE BFV encryption scheme, i.e. the ring variant. The RLWE encryption scheme is in principle equal to the LWE one, but translated to rings, [Lai17, Sec. 4], [FV12, Sec. 3.2], or [HPS19, Sec. 3.2 & 4.1]. Both, [Lai17, Sec. 4] and [HPS19, Sec. 3.2 & 4.1], state an asymmetric version of BFV. For a symmetric version, see [BV11, 1.1 The Basic Scheme & 3.1].

Note: There exist multiple versions of the BFV RLWE encryption scheme, which are all equivalent, in terms of security. However, the one we use in this section is a direct translation of the BFV LWE encryption scheme of Section 2.5.1 to rings. This particular HE scheme is also used in SEAL.

Preconditions Like for the LWE version in Section 2.5.1, we start with setting some encryption parameters.

Preconditions

- choose: dimension N, plaintext modulus t, where $t > 1$, and ciphertext modulus q, with $q \gg t$

Ring Learning With Errors (RLWE) KeyGen After setting all required encryption parameters in the previous step, we can sample an RLWE secret key s, which now is a ring element.

RLWE KeyGen

- sample: secret key $s \leftarrow U(R)$, with $\|s\| \ll q/t$

Note, the homomorphic encryption library SEAL, see Section 2.6, uniformly samples the secret key s from R_3, i.e. polynomials with coefficients in $\{-1, 0, 1\}$, [Lai17, Sec. 5.9].

Ring Learning With Errors (RLWE) Encryption Using the just generated RLWE secret key s, we can now homomorphically symmetrically encrypt an RLWE plaintext m.

Note, in textbook-BFV we use the plaintext space $R_t = \mathbb{Z}_t[X]/(X^N + 1)$, see Section 2.2.2. That means that the plaintext m is an element of R_t, i.e. $m \in R_t$, so it's a polynomial of the form (2.1), [Lai17, Sec. 4.1].

RLWE Encryption, i.e. encrypt RLWE plaintext $m \in R_t$ into an RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$:

- input: N, q, t, m, s
- encode: RLWE plaintext m into phase μ
  - calc: $\Delta = \lfloor \frac{q}{t} \rfloor$
  - sample: Gaussian noise e, with $\|e\| \ll q/t$
  - calc: $\mu = (\Delta m + e) \mod q \in R_q$
- sample: $a \leftarrow U(R_2)$
- calc: $b = -a \cdot s + \mu \mod q \in R_q$
- return: RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$

Ring Learning With Errors (RLWE) Decryption The inverse operation, i.e. the decryption, is defined as follows. Like in Section 2.5.1, we can recover a plaintext, in this case an RLWE plaintext $m \in R_t$, from the RLWE ciphertext ct by using this symmetric RLWE decryption with the same symmetric key s as during the symmetric RLWE encryption.

RLWE Decryption, i.e. decrypt RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$ into an RLWE plaintext $m \in R_t$:

- input: q, t, $\mathrm{ct} = (b, a)$, s
- calc: $\mu = b + a \cdot s \mod q \in R_q$
- decode: phase μ into RLWE plaintext m
  - calc: $m = \left\lfloor \frac{t\mu}{q} \right\rceil \mod t$
- return: RLWE plaintext $m \in R_t$

2.6. Simple Encrypted Arithmetic Library (SEAL)

For the implementation, see Section 6, we use the existing homomorphic encryption library Microsoft SEAL, [20], which is developed by the Cryptography and Privacy Research Group at Microsoft.

SEAL was introduced back in the year 2015 with the goal of being a homomorphic encryption library, without any external dependencies, to be easily used both by crypto experts as well as non-experts, [Lai17, Sec. 1]. It is written in C++ and runs in many different environments. We will use SEAL in Version 3.6.1, see Section 6. For Version 2.3.1, there exists a high level guide, [Lai17].

2.6.1. Residue Number System (RNS)

SEAL uses RNS, also called CRT representation, see [HPS19, Sec. 3.3], to represent the, very large, ciphertext modulus q as the product of some, smaller, primes q_i.

Residue Number System (RNS) Let the k moduli $q_1, \dots, q_k$ be some integers larger than 1, i.e. $q_i > 1$, $\forall i \in \{1, \dots, k\}$, which also are coprime. We define q as the product of all moduli q_i, i.e. $q = \prod_{i=1}^{k} q_i$. Additionally, we define

$$q_i^* = q/q_i \in \mathbb{Z} \tag{2.3}$$

and

$$\tilde{q}_i = q_i^{*-1} \mod q_i \in \mathbb{Z}_{q_i} \tag{2.4}$$

with the properties $\tilde{q}_i \in \left[-\frac{q_i}{2}, \frac{q_i}{2}\right)$ and $q_i^* \cdot \tilde{q}_i = 1 \mod q_i$ for all $i \in \{1, \dots, k\}$, [HPS19, Sec. 2].

With the CRT representation from [HPS19, Sec. 2.1 & 3.3], we now can represent an integer $x \in \mathbb{Z}_q$ relative to the CRT basis $\{q_1, \dots, q_k\}$ as k integers $x_i \in \mathbb{Z}_{q_i}$, where x_i is the reduction of x into the interval $\mathbb{Z}_{q_i} = \mathbb{Z} \cap \left[-\frac{q_i}{2}, \frac{q_i}{2}\right)$, [HPS19, Sec. 2]

So, using Equation (2.3), Equation (2.4) and the k reductions $x_i \in \mathbb{Z}_{q_i}$, we can write x as the following sum

$$x = \sum_{i=1}^{k} x_i \cdot \tilde{q}_i \cdot q_i^* \mod q.$$

Composite Ciphertext Modulus In case of our coefficient modulus q, [Lai17, Sec. 5.8], we define it as the product of multiple distinct small primes q_i. This means $q = \prod_{i=1}^{k} q_i$, for $q_i \in \mathbb{P}$, $i \in \{1, \dots, k\}$.

Again, using the CRT from Section 2.2.4 and RNS we have the ring isomorphism $R_q \equiv R_{q_1} \times R_{q_2} \times \dots, \times R_{q_k}$ where we can perform all ring operations, which all are modulo q, separately in the much smaller rings R_{q_i} in order to reduce the computation cost, [Lai17, Sec. 5.8], [Che+18a, Sec. 2.2].

Additionally, all moduli q_i in SEAL have less or equal than 60 bits, and it must hold $q_i = 1 \mod 2N$, for $i \in \{1, \dots, k\}$, [Lai17, Sec. 5.8 & 8.5]. The latter is a condition for the NTT.

2.6.2. Number Theoretic Transform (NTT)

The NTT is a Discrete Fourier Transform (DFT) over a finite field, [Har14, Sec. 1]: Let β be the machine word size, e.g $\beta = 2^{32}$ or $\beta = 2^{64}$. We use the field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ with p being a word-sized prime and $p = 1 \mod L$ for the transform length L, which is a power-of-two integer, i.e. $L = 2^{\ell}$. Under these assumptions there exists a primitive L-th root of unity of $\mathbb{F}_p$, and we define one as ζ.

The NTT is now defined as a map $(\mathbb{F}_p)^L \to (\mathbb{F}_p)^L$ with

$$\begin{gathered}(\mathbb{F}_p)^L \to (\mathbb{F}_p)^L \\ \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{L-1} \end{bmatrix} \mapsto \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{L-1} \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{L-1} \zeta^{i\cdot 0} a_i \\ \sum_{i=0}^{L-1} \zeta^{i\cdot 1} a_i \\ \sum_{i=0}^{L-1} \zeta^{i\cdot 2} a_i \\ \vdots \\ \sum_{i=0}^{L-1} \zeta^{i\cdot (L-1)} a_i \end{bmatrix},\end{gathered} \tag{2.5}$$

i.e. where $b_j = \sum_{i=0}^{L-1} \zeta^{i\cdot j} a_i$, for $0 \leq j < L$. It can also be seen as the map which evaluates the polynomial

$$a_0 + a_1 x + \cdots + a_{L-1} x^{L-1}$$

at the powers of the root of unity, [Har14, Sec. 1],

$$1, \zeta, \zeta^2, \ldots, \zeta^{L-1}.$$

Additionally, we could rewrite Equation (2.5) as a matrix vector multiplication

$$\begin{aligned}\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{L-1} \end{bmatrix} &= \begin{bmatrix} \zeta^{0\cdot 0} & \zeta^{1\cdot 0} & \zeta^{1\cdot 0} & \cdots & \zeta^{(L-1)\cdot 0} \\ \zeta^{0\cdot 1} & \zeta^{1\cdot 1} & \zeta^{1\cdot 2} & \cdots & \zeta^{(L-1)\cdot 1} \\ \zeta^{0\cdot 2} & \zeta^{1\cdot 2} & \zeta^{2\cdot 2} & \cdots & \zeta^{(L-1)\cdot 2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \zeta^{0\cdot (L-1)} & \zeta^{1\cdot (L-1)} & \zeta^{2\cdot (L-1)} & \cdots & \zeta^{(L-1)\cdot (L-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{L-1} \end{bmatrix} = \\ &= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \zeta^1 & \zeta^2 & \cdots & \zeta^{(L-1)} \\ 1 & \zeta^2 & \zeta^4 & \cdots & \zeta^{2(L-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta^{(L-1)} & \zeta^{2(L-1)} & \cdots & \zeta^{(L-1)(L-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{L-1} \end{bmatrix}.\end{aligned}$$

The NTT can be computed with the Fast Fourier transform (FFT) using $\mathcal{O}(L \log L)$ operations in $\mathbb{F}_p$. The in-place iterative radix-2 FFT from [Har14, Sec. 1] is shown in Algorithm 2.1.

Using the negacyclic polynomial $\Phi_{2N}(X) = X^N + 1$, where N is a power-of-two integer, as polynomial modulus, allows us to efficiently multiply polynomials using a negacyclic NTT, [Dob+21, Sec. 5.3.3].

Algorithm 2.1: FFT: Fast way to compute the DFT [Har14, Sec. 1, Algorithm 1]

```
Input  : $\zeta \in \mathbb{F}_p$,
         $\mathbf{x} = (x_0, \ldots, x_{L-1}) \in (\mathbb{F}_p)^L$, with $L = 2^\ell$.
Output : DFT of $\mathbf{x}$ with respect to $\zeta$, in bit-reversed order
1 for $i \leftarrow 1, 2, \ldots, \ell$ do
2     $\zeta \leftarrow \zeta^{2^{i-1}}$
3     $m \leftarrow 2^{i-1}$
4     for $0 \leq j < 2^{i-1}$ do
5         $t \leftarrow 2jm$
6         for $0 \leq k < m$ do
7             $\begin{bmatrix} x_{t+k} \\ x_{t+k+m} \end{bmatrix} \leftarrow \begin{bmatrix} x_{t+k} + x_{t+k+m} \\ \zeta^k (x_{t+k} - x_{t+k+m}) \end{bmatrix}$
```

In SEAL, [Lai17, Sec. 8.5], the NTT is computed using David Harvey's algorithm, which is an improvement of the FFT over $\mathbb{F}_p$, [Har14]. SEAL also uses a negacyclic variant of the NTT to compute the isomorphisms in the CRT Batching, [Lai17, Sec. 7.4]. This CRT Batching, or Packing, is used to encode multiple integers into a single polynomial, see Section 2.7.

2.6.3. Brakerski/Fan-Vercauteren (BFV) Encryption Parameters

We now briefly describe the parameters which have to be set in advance in SEAL, in order to perform an encryption, see [Lai17, Sec. 8] and [Bam+20, Sec. 8.5 & Appendix D], namely

- polynomial modulus $X^N + 1$, i.e. polynomial degree N, which is a power of 2,
- ciphertext modulus q, which is a product of some primes,
- plaintext modulus t.

Runtime, security, initial noise budget and noise consumption mostly depend on the three parameters polynomial degree N, ciphertext modulus q and the plaintext modulus t.

Every freshly encrypted ciphertext has a certain amount of initial noise budget. The noise budget decreases with every homomorphic operation, i.e. we consume noise. If the noise budget is becomes zero, we can not decrypt a ciphertext correctly anymore.

SEAL also provides some hard coded default pairs (N, q) for different levels of security, i.e. 128 bit, 192 bit and 256 bit, [Lai17, Table 3]. Nevertheless for our implementation, see Section 6, we will extend this list with further parameter combinations. In our implementation, we use a computational security level of 128 bit.

Plaintext Modulus t The plaintext modulus t could be an arbitrary integer with the constraint that t must be at least 2 and at most 60 bits long. If we want to use CRT batching, t additionally has to be a prime, i.e. $t \in \mathbb{P}$, and it must hold $t = 1 \mod 2N$, see Section 2.7. Because the plaintext modulus t defines the ring $\mathbb{Z}_t$ we operate in, the integer t must be big enough in order to mitigate possible overflows when performing all calculations. Using a larger t furthermore is often required for encoding larger integers, see Section 2.7. Contrariwise, the ciphertext noise gets bigger, for a bigger plaintext modulus t. Additionally, a bigger t negatively affects the performance.

Ciphertext Modulus q The ciphertext modulus, or coefficient modulus, q is the product of multiple, arbitrary, distinct small prime numbers $q_i \in \mathbb{P}$, i.e. $q = \prod_{i=1}^{k} q_i$.

It directly affects the noise budget of a new ciphertext and the security level. The positive effect is, that a larger q gives us a larger noise budget for the newly encrypted ciphertext. At the downside, a larger q lowers the security. A larger q, or more precisely, a larger amount of distinct small primes q_i also negatively affects the runtime of the homomorphic operations.

Theoretically, the user can choose any arbitrary primes q_i, as long as q is not too large to have a negative impact on the security level. But additionally in SEAL, the primes q_i must be at most 60 bit long and it must hold $q_i = 1 \mod 2N \quad \forall i \in \{1, 2, \dots, k\}$. The requirement $q_i = 1 \mod 2N$ is necessary in order to perform an efficient NTT, [Che+20, Sec. 3.4, Footnote 5]. Note, if the user does not need such a large q, for example if the available q in the SEAL default pairs (N, q), see [Lai17, Table 3], would be too large, one could always lower the ciphertext modulus q for gaining a higher security level and for a better performance.

Polynomial Modulus $X^N + 1$ The polynomial modulus, or reduction polynomial, is a polynomial of the form $X^N + 1$, where the polynomial degree N is a power of 2, i.e. $N = 2^\ell$ with $\ell \in \mathbb{N}$.

On the one side, if we use a larger N, we could, without lowering the security level, also use a larger ciphertext modulus q. Using this larger q gives us a larger noise budget and therefore allows us to perform more HE operations.

On the other side, using a lager polynomial degree N has a huge negative impact on the runtime of the encryption scheme.

2.7. Packing/Batching

Assume, instead of encrypting and calculating with polynomials we just want to operate with multiple single integers. Of course, we could simply interpret every integer as a constant polynomial, but in this case we would waste valuable computational power. A better solution is combining multiple integers and operate on them in parallel. This idea brings us to packing, and accordingly, encoders, described in [BGH13], [SV14], [Bam+20, Sec. 8.1], [Dob+21, Sec. 2.1 & Sec. 5.3.3 Inner Structure of HE ciphertexts], [Lai17, Sec. 7 & 7.4] and [Che+20, Sec. 2.5].

Packing Packing allows us, to encode a vector of integers into a single polynomial. This means, we could merge multiple integers into a single polynomial. Doing this, some operations on the polynomial, for example addition, correspond to element-wise vector operations, in this case element-wise vector addition. I.e. we say the operation is performed on each slot of the encrypted vector. Available operations are the already mentioned addition, but also subtraction, multiplication and slot rotation. This allows us to perform HE in parallel on many integers. We can therefore compare packing with Single-Instruction-Multiple-Data (SIMD) on modern Central Processing Units (CPUs).

Recalling the textbook-BFV described in Section 2.5.2, one can see that the plaintexts are elements of R_t. In many practical use cases one would like to encrypt some integers $\in \mathbb{Z}_t$ instead and not some elements in R_t. Therefore, so called encoders for encoding integers to some elements in R_t are used, [Lai17, Sec. 7].

Encoder: Chinese Remainder Theorem (CRT) Batching In this thesis, we use the CRT Batch Encoder, [Lai17, Sec. 7.4], [BGH13], [SV14], for packing, i.e. in order to encode n integers modulo the plaintext modulus t into a single plaintext polynomial $\in R_t$.

There are some assumptions, [Lai17, Sec. 7.4], for the parameters we have to make: The plaintext modulus t has to be a prime number, i.e. $t \in \mathbb{P}$, and additionally congruent to 1 mod $2N$, i.e. $t = 1 \mod 2N$. This also implies that $t > 2N$ holds. With these assumptions, there exists a subgroup of size $2N$ within the multiplicative group of integers modulo t. I.e. there exists a so called $2N$-th root of unity modulo t, which is defined as an integer $\zeta \in \mathbb{Z}_t$ to which the following two statements apply:

- $\zeta^{2N} = 1 \mod t$, and
- $\zeta^i \neq 1 \mod t \quad \forall 0 < i < 2N$.

The existence of such an primitive $2N$-th root of unity in $\mathbb{Z}_t$ implies, that the polynomial modulus $X^N + 1$ factors modulo t in the following form

$$\begin{aligned}\left(X^N + 1\right) &= (X - \zeta)\left(X - \zeta^3\right)\dots\left(X - \zeta^{2N-1}\right) = \\ &= \prod_{i=0}^{N-1}\left(X - \zeta^{2i+1}\right) \mod t.\end{aligned} \tag{2.6}$$

Further on, we can use Equation (2.6) and the CRT from Section 2.2.4 to factor the ring R_t as follows

$$\begin{aligned}R_t &= \frac{\mathbb{Z}_t[X]}{(X^N + 1)} = \\ &\overset{(2.6)}{=} \frac{\mathbb{Z}_t[X]}{\prod_{i=0}^{N-1}\left(X - \zeta^{2i+1}\right)} = \\ &\overset{\text{CRT}}{\cong} \prod_{i=0}^{N-1} \frac{\mathbb{Z}_t[X]}{\left(X - \zeta^{2i+1}\right)} \cong \prod_{i=0}^{N-1} \mathbb{Z}_t[\zeta^{2i+1}] \cong \prod_{i=0}^{N-1} \mathbb{Z}_t.\end{aligned} \tag{2.7}$$

Every used isomorphism in Equation (2.7) is a ring isomorphism, which therefore respects the multiplicative and the additive structure on each of the two sides. This brings us to the following, wanted property: A single addition or multiplication of a polynomial in R_t on the left hand side of Equation (2.7), equals N coefficient-wise additions or multiplications of integers modulo t on the right hand side of Equation (2.7).

Like in [Lai17, Sec. 7.4], we will briefly mention the easy isomorphism direction, i.e. Decode, but will omit the opposite, non trivial one, i.e. Encode. We first define $\alpha_i = \zeta^{2i+1}$. Decode is then defined as:

$$\begin{aligned} R_t &\overset{\cong}{\to} \prod_{i=0}^{N-1} \mathbb{Z}_t \\ a(X) &\mapsto \left[a\left(\alpha_0\right), a\left(\alpha_1\right), \ldots, a\left(\alpha_{N-1}\right)\right]. \end{aligned}$$

The opposite direction, i.e. Encode, is just defined as the inverse of Decode, to simplify matters. A negacyclic variant of the NTT from Section 2.6.2 is used for computing all these isomorphisms. This means, by evaluating a NTT or respectively the inverse NTT, we can decode or respectively encode a polynomial or vector, [Dob+21, Sec. 5.3.3].

Note, that $X^N + 1$ is a negacyclic cyclotomic reduction polynomial. This means, all roots are also roots of unity. Therefore we use a negacyclic NTT here.

As we discuss in the next section, i.e. Section 2.8, Galois automorphisms can be used to implement slot rotations on this encoded vectors.

2.8. Galois Automorphism

We will use Galois automorphisms of the cyclotomic extension $\mathbb{Q} \hookrightarrow \mathbb{Q}[X] / \left(X^N + 1\right)$, where $X^N + 1$ is the polynomial modulus. This extension can be generated by any primitive $M = 2N$-th root of unity. With ζ defined as the first primitive root, we get the other primitive roots, by calculating $\zeta^3, \zeta^5, \ldots, \zeta^{M-1}$. An Galois automorphism is now equal to the map $\zeta \mapsto \zeta^{2k-1}$, so to say a Galois automorphism permutes the roots of unity. In case of the cyclotomic extension ring it is the map for the polynomial $a(X) \mapsto a\left(X^{2k-1}\right)$, [Lai17, Sec. 5.6].

In other words, see [Che+20, Sec. 2.5], a Galois automorphism is a map

$$\tau_d : a(X) \to a\left(X^d\right), \tag{2.8}$$

where $d \in \mathbb{Z}_{2N}^{\times}$, which are the invertible residues modulo $2N$. Thereby, $\mathbb{Z}_{2N}^{\times}$ is defined as the multiplicative group of the integers modulus $2N$, i.e. all odd integers up to $2N$. This means Galois automorphisms can be implemented as permutations of the coefficients, i.e. a Galois automorphism τ_d shifts a coefficient of X to X^d: $\tau_d : a(X) \mapsto a\left(X^d\right)$. Together all Galois automorphisms form the Galois group $\mathrm{Gal}(K/\mathbb{Q})$.

During the implementation we will restrict to $R = \mathbb{Z}[X] / \left(X^N + 1\right)$ and also reduce all coefficients modulo the plaintext modulus t, [Lai17, Sec. 5.6].

Note: Galois automorphisms are per definition homomorphous for addition and multiplications. Therefore, we can evaluate them on HE ciphertexts.

Another important fact is, that the Galois group is isomorphic to $\mathbb{Z}_{N/2} \times \mathbb{Z}_2$, where the first factor, i.e. $\mathbb{Z}_{N/2}$, is generated by the Galois element 3 and the second, i.e. $\mathbb{Z}_2$, by the Galois element $M - 1 = 2N - 1$, [Lai17, Sec. 5.6]. I.e. it holds $\mathbb{Z}_{2N}^{\times} \cong \mathbb{Z}_{\frac{N}{2}} \times \mathbb{Z}_2$. We used this property in Section 2.7, where the primitive root ζ is an element of $\mathbb{Z}_t^{\times}$, i.e. $\zeta \in \mathbb{Z}_t^{\times}$: During batching, see Section 2.7, a plaintext can be viewed as a $2 \times \frac{N}{2}$ matrix.

If we arrange the rows of the inner matrix such that element j of the encoded vector is stored at the root ζ^{3^j}, then cyclic rotation of the vector by d steps can be implemented as the Galois automorphism τ_{3^d}. Similar for the second row, where the element j is stored at the root $\zeta^{(2N-1)\cdot 3^j}$. Then swapping of the rows is equal to τ_{2N-1}.

Because we repeatedly use these powers of the roots of unity, it is convenient to precompute them. In SEAL the powers are already stored in the order

$$\left[\underbrace{\zeta, \zeta^3, \zeta^{3^2}, \zeta^{3^3}, \dots, \zeta^{3^{\frac{N}{2}-1}}}_{\text{first vector}}, \underbrace{\zeta^{2N-1}, \zeta^{(2N-1)\cdot 3}, \zeta^{(2N-1)\cdot 3^2}, \dots, \zeta^{(2N-1)\cdot 3^{\frac{N}{2}-1}}}_{\text{second vector}}\right], \tag{2.9}$$

or more generally

$$\zeta^{(2N-1)^i \cdot 3^j},$$

for $i \in \mathbb{Z}_2 = [2]$ and $j \in \mathbb{Z}_{\frac{N}{2}} = \left[\frac{N}{2}\right]$. However, during implementation, see Section 6, we have to be aware, that this form differs from the natural order

$$\left[\zeta, \zeta^3, \zeta^5, \zeta^7, \dots, \zeta^{N-1}, \zeta^{N+1}, \zeta^{N+3} \dots, \zeta^{2N-1}\right],$$

or generally

$$\zeta^{2i+1},$$

for $i \in [N]$, which we have seen above.

This means, using the Galois automorphism (2.8), we can cyclically rotate every row of this $2 \times \frac{N}{2}$ matrix, and we can also rotate the columns of the matrix, i.e. we can swap the rows, [Dob+21, Sec. 5.3.3], as illustrated in the following:

$$\begin{bmatrix} \mathbf{x}_{\mathrm{L}} \\ \mathbf{x}_{\mathrm{R}} \end{bmatrix} \overset{\text{encode}}{\rightarrow} x \in R_t : \quad \begin{array}{l} \tau_{3^d}(x) \overset{\text{decode}}{\rightarrow} \begin{bmatrix} \mathrm{rot}_d(\mathbf{x}_{\mathrm{L}}) \\ \mathrm{rot}_d(\mathbf{x}_{\mathrm{R}}) \end{bmatrix} \\ \tau_{2N-1}(x) \overset{\text{decode}}{\rightarrow} \begin{bmatrix} \mathbf{x}_{\mathrm{R}} \\ \mathbf{x}_{\mathrm{L}} \end{bmatrix} \end{array} \tag{2.10}$$

In more detail, a rotation of the rows by an index i has the following form:

$$\tau_{3^i} : \zeta \mapsto \zeta^{3^i}.$$

In particular, for a certain power y of ζ, it follows:

$$\zeta^y \mapsto \zeta^{3^i \cdot y}.$$

Similarly, a rotation of the columns is defined as:

$$\tau_{2N-1} : \zeta \mapsto \zeta^{2N-1},$$

and likewise:

$$\zeta^{y} \mapsto \zeta^{(2N-1)\cdot y}.$$

To sum up: Lots of HE schemes which are based on RLWE, including the BFV scheme which we use in this thesis, see Section 2.5.2, use the DFT, i.e. the NTT, see Section 2.6.2, to encode multiple plaintexts, in our case integers, into a single polynomial, see Packing/Batching in Section 2.7. The Galois automorphism can then be used to permute the slots of the ciphertext, see [Che+20, Sec. 2.5].

Note: We will further cover how the Galois automorphism τ_d acts on the coefficients of some polynomial, in Section 4.1.

Chapter 3.

Lightweight Learning With Errors (LWE) Ciphertext

The goal of this thesis is to reduce the communication overhead between the client and the server, i.e. mitigate the problem of ciphertext expansion.

Ciphertext expansion means, that the encryption of a plaintext is much larger in size than the plaintext itself. Think of an RLWE encryption of a single integer. There the lightweight ciphertext is a full polynomial of degree N, whereas the plaintext was only a single integer.

So, in order to prevent ciphertext expansion, we want the client to send some smaller, lightweight, symmetric LWE ciphertexts, instead of the full RLWE ciphertexts, to the server. The idea of the lightweight LWE ciphertexts is discussed in [Che+20, Sec. 4.2].

Regular Learning With Errors (LWE) Ciphertext In Section 2.5.1 we introduced the regular LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$, with:

Regular LWE Ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$:

- Sample a vector $\mathbf{a} \leftarrow U(\mathbb{Z}_q^N)$.
- Calculate $b = -\langle \mathbf{a}, \mathbf{s} \rangle + \mu \mod q$ from the vector $\mathbf{a}$, the secret key $\mathbf{s}$, and the phase μ, which is a noisy encoding of the plaintext.

Lightweight Learning With Errors (LWE) Ciphertext We note that $\mathbf{a}$ is purely random and so, for minimising the communication cost, we modify the regular LWE encryption algorithm to use the, much smaller, lightweight LWE ciphertext (b, se) with:

Lightweight LWE Ciphertext (b, se):

- Sample a seed se.
- Use a PRNG $f : \{0,1\}^* \to \mathbb{Z}_q^N$ and the seed se to calculate $\mathbf{a} = f(\mathrm{se})$. Finally, calculate b like above

$$\begin{aligned} b &= -\langle \mathbf{a}, \mathbf{s} \rangle + \mu \mod q \\ &= -\langle f(\mathrm{se}), \mathbf{s} \rangle + \mu \mod q. \end{aligned}$$

I.e. we first sample a seed se. We then use this seed and a PRNG to calculate the vector $\mathbf{a}$ and use it to further calculate the scalar b as usual.

Advantage These lightweight LWE ciphetexts are much smaller than the regular ones, i.e. one q-bit integer plus a seed, instead of a vector of size $N + 1$.

Additionally there is another advantage when using this optimisation: Let's assume we want to create multiple LWE ciphertexts, to send them to the server afterwards. We now can even reuse the same seed se and calculate multiple vectors $\mathbf{a}_i = f(\mathrm{se}; i)$ with a counter i.

Lightweight Ring Learning With Errors (RLWE) Ciphertext The previous trick with the seed could, of course, also be applied to an RLWE ciphertext $(b, a) \in R_q^2$ from Section 2.5.2. This means, that instead of a regular RLWE ciphertext, which consists of two polynomials of degree N, a lightweight version of the RLWE ciphertext only consists of a single polynomial of degree N, plus a seed.

Ring Learning With Errors (RLWE) Ciphertext VS. Lightweight Learning With Errors (LWE) Ciphertext So let's compare an RLWE ciphertext to the introduced lightweight LWE ciphertext. Assume we have some Dimension N and the corresponding ciphertext modulus q.

On the one side, for the RLWE ciphertext (b, a) we have two polynomials of degree N, where all coefficients are in $\mathbb{Z}_q$. So in total we get $2 \cdot N \cdot q$ bits. On the other side, the lightweight LWE ciphertext (b, se) consists of a single element in $\mathbb{Z}_q$ and a seed. In total, we only have q bits (plus the seed). Additionally, like we have seen before, we also have to transmit the seed only once for multiple ciphertexts.

Also, when we recall the packing technique from Section 2.7, we notice, that it would be possible to encode up to N plaintexts into a single RLWE ciphertexts to reduce the communication cost. But assume we want to send n plaintexts: With lightweight LWE ciphertext, we have to send $n \cdot q$ bits. With an RLWE ciphertext we still have to send $2 \cdot N \cdot q$ bits, which can be reduced to $N \cdot q$ using the seed trick for RLWE ciphertexts, i.e. using a lightweight RLWE ciphertext.

Another drawback of sending RLWE ciphertexts would be, that we have to wait until we have enough integers to fill up an RLWE ciphertext. In contrast, when using a lightweight LWE ciphertext, we immediately can send every single encrypted integer.

In total we see, that with only sending the lightweight LWE ciphertexts from the client to the server, we can mitigate the problem of ciphertext expansion.

Chapter 4.

Efficient Homomorphic Conversion Algorithms

In [Che+20] they introduced some algorithms for efficient homomorphic conversion between (Ring) LWE ciphertexts. In this section, we give an overview, and we will use these algorithms during the implementation in Section 6. In particular, these operations are, [Che+20, Sec. 1],

1. improving Key Switching (KS) between LWE ciphertexts,

2. conversion of an LWE ciphertext into an RLWE ciphertext, and

3. packing of multiple LWE ciphertexts into a single RLWE ciphertext.

The authors of this paper claim, [Che+20, Sec. 1], that their new algorithms are almost optimal. That means, that, with respect of the size of the input ciphertext(s), the complexities of their algorithms are all quasi-linear. This whole section is based on the findings in [Che+20].

4.1. Technical Overview

We will start with some background based on [Che+20] and present the resulting algorithms in Section 4.3.

Preconditions Again, we need some dimension N, which is a power-of-two integer, i.e. $N = 2^k$, with $k \in \mathbb{N}$. Additionally, there is the ciphertext modulus q. Note, in SEAL the ciphertext (coefficient) modulus q is the product of k primes $q_i \in \mathbb{P}$ where $q_i = 1 \mod 2N$, i.e $q = \prod_{i=1}^{k} q_i$, see Section 2.6.3.

Learning With Errors (LWE) Encryption We recall the LWE encryption from Section 2.5.1.

LWE Encryption:

- secret key: $\mathbf{s} \in \mathbb{Z}^N$
- LWE ciphertext: $(b, \mathbf{a}) \in \mathbb{Z}_q^{N+1}$
- phase: $\mu = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$

The phase μ is an encoding of the plaintext, which also includes some random noise. With each operation on the ciphertext, the noise in the phase μ increases. A correct decryption of the ciphertext is possible as long as the noise does not exceed a certain threshold, i.e. there is enough noise budget left.

Ring Learning With Errors (RLWE) Encryption And we also recall the RLWE encryption from Section 2.5.2. Here we use the ring $R = \mathbb{Z}[X] / \left(X^N + 1\right)$, in particular its residue ring $R_q = R/qR = \mathbb{Z}_q[X] / (X^N + 1)$, described in Section 2.2.2. This means, the elements in the ring are polynomials of the form (2.1).

RLWE Encryption:

- secret key: $s \in R$
- RLWE ciphertext: $(b, a) \in R_q^2$
- phase: $\mu = b + a \cdot s \mod q \in R_q$

Identifying Polynomial and Vector of its Coefficients We define the mapping ι such that it is an encoding of a vector $\mathbf{a} \in \mathbb{Z}_q^N$ into a polynomial $a \in R_q$, [Che+20, Sec. 2.1 & 3.2]:

$$\begin{aligned} \iota : \quad & \mathbb{Z}_q^N \to R_q \\ & \mathbf{a} \mapsto a = \sum_{i \in [N]} \mathbf{a}[i] \cdot X^i \end{aligned} \tag{4.1}$$

So to say, we rewrite a vector as a polynomial, by identifying the vector entry $\mathbf{a}[i]$ as the i-th coefficient of the polynomial a.

Additionally, we can define the inverse of ι, i.e. ι^{-1}. In this case it is an encoding of a polynomial $a \in R_q$ into a vector $\mathbf{a} \in \mathbb{Z}_q^N$: Let $a[i] \in \mathbb{Z}_q$, for $i \in [N]$, be coefficients of a polynomial $a = \sum_{i \in [N]} a[i] \cdot X^i \in R_q$

$$\begin{aligned} \iota^{-1} : \quad & R_q \to \mathbb{Z}_q^N \\ a = \sum_{i \in [N]} a[i] \cdot X^i \mapsto \mathbf{a} = & \begin{bmatrix} a[0] \\ a[1] \\ \vdots \\ a[N-1] \end{bmatrix} \end{aligned} \tag{4.2}$$

with

$$\mathbf{a}[i] = a\,[i] \quad \forall i \in [N]\,.$$

So to say, we rewrite a polynomial as a vector of its coefficients.

Integral Vector & Gadget Decomposition Let $\mathbf{g}$ be an integral vector, [Che+20, Sec. 2.3],

$$\mathbf{g} = \left(g\,[0]\,, g\,[1]\,, \ldots, g\,[L-1]\right). \tag{4.3}$$

We define a gadget decomposition $\mathbf{g}^{-1}$, [Che+20, Sec. 2.3],

$$\mathbf{g}^{-1} : \mathbb{Z}_q \to \mathbb{Z}^L$$

as a map, which satisfies

$$\left\langle \mathbf{g}^{-1}\,(a)\,, \mathbf{g}\right\rangle = a \mod q \quad \forall a \in \mathbb{Z}_q$$

for an integer q and the integral vector $\mathbf{g}$.

By extending the domain of the gadget decomposition to

$$\mathbf{g}^{-1} : R_q \to R^L,$$

we can define

$$\begin{aligned} \mathbf{g}^{-1} : \quad & R_q \to R^L \\ a = \sum_{i \in [N]} a\,[i] \cdot X^i \mapsto & \sum_{i \in [N]} \mathbf{g}^{-1}(a\,[i]) \cdot X^i. \end{aligned} \tag{4.4}$$

Examples for a gadget decomposition are the base (digit) decomposition, and the prime decomposition, [Che+20, Sec. 2.3].

Let us give a short explanation of the gadget decomposition degree L: The gadget decomposition $\mathbf{g}^{-1}$, see Equation (4.4), which we use in, for example KeySwitch(), i.e. Algorithm 4.2, transforms a scalar into a vector. The size of this vector depends on the decomposition. If we, for example, use bit-decomposition, a value gets transformed into its bit-representation. Then, L is the number of bits.

We will use the integral vector $\mathbf{g}$ during the generation of the KS key, i.e. Algorithm 4.1, and the gadget decomposition $\mathbf{g}^{-1}$ in the actual KS algorithm, i.e. Algorithm 4.2.

Galois Automorphism on Polynomial Coefficients We recall the Galois automorphism τ_d from Section 2.8, especially Equation (2.8), i.e.

$$\tau_d : a\,(X) \to a\left(X^d\right).$$

This means a Galois automorphism τ_d shifts a coefficient of X to X^d.

In this section we will use the Galois automorphism τ_d from Section 2.8 and further investigate how this τ_d acts on the coefficients of some polynomial, [Che+20, Sec. 3.1].

First, we define the following set

$$I_k = \begin{cases} \{i \in [N] : 2^k \parallel i\} & \text{for} \quad 0 \leq k < \log N \\ \{0\} & \text{for} \quad k = \log N \end{cases}. \tag{4.5}$$

The used expression $2^k \parallel i$ means that k ist the maximum power of 2 which divides i, i.e $2^k \mid i$ and $2^{k+1} \nmid i$, [Che+20, Sec. 3.1 Footnote].

Using these sets in (4.5), allows us to write the index set $[N]$ as a disjoint union

$$[N] = \bigcup_{k=0}^{\log N} I_k.$$

We want to further investigate the behaviour of the Galois automorphism τ_d on monomials, especially for $d = 2^\ell$, $1 \leq \ell \leq \log N$. Therefore, [Che+20, Sec. 3.1], we define the map

$$i \mapsto i \cdot d \mod N,$$

which is a signed permutation on the set I_k. That means: For some integer $i \in I_k$ it holds $\tau_d\left(X^i\right) = \pm X^j$ for some integer $j \in I_k$. Especially it holds

$$\tau_d\left(X^i\right) = \begin{cases} +X^i & \text{for} \quad i \in \bigcup_{k > \log N - \ell} I_k \\ -X^i & \text{for} \quad i \in I_{\log N - \ell} \end{cases}. \tag{4.6}$$

Equation (4.6) gives us the following main result on coefficients of some polynomial μ: When using the map

$$\mu \mapsto \mu + \tau_d(\mu),$$

the coefficients $\mu[i]$ of the polynomial μ get

- doubled, if $2^{\log N - \ell + 1} \mid i$, and
- zeroized, if $2^{\log N - \ell} \parallel i$.

This is an important result, which will be used for a property of the field trace later this section, i.e. Equation (4.9).

Secret Key Conversion Like stated in [Che+20, Sec. 3.2], we can convert an LWE secret key $\mathbf{s} \in \mathbb{Z}^N$ into an RLWE secret key $s \in R$, using the Galois automorphism τ_d, i.e. Equation (2.8) from Section 2.8, for $d = -1$, and Equation (4.1):

$$\begin{aligned} &\mathbb{Z}^N \to R \\ \mathbf{s} = \begin{bmatrix} s[0] \\ s[1] \\ \vdots \\ s[N-1] \end{bmatrix} &\mapsto s = \tau_{-1} \circ \iota(\mathbf{s}) \overset{(2.8),(4.1)}{=} \sum_{i \in [N]} \mathbf{s}[i] \cdot X^{-i} \end{aligned} \tag{4.7}$$

We can perform the opposite operation, by applying the inverse of Equation (4.7). I.e. we can convert an RLWE secret key $s \in R$ into an LWE secret key $\mathbf{s} \in \mathbb{Z}^N$. We do this by using Equation (4.2) and again Equation (2.8):

$$\begin{aligned} R &\to \mathbb{Z}^N \\ s &\mapsto \mathbf{s} \overset{\text{Inv. of (4.7)}}{=} \iota^{-1} \circ (\tau_{-1})^{-1}(s) = \iota^{-1} \circ \tau_{-1}(s) \end{aligned}$$

Note: When looking at Equation (2.8), we see, that the inverse of the Galois automorphism τ_{-1} is again τ_{-1}. I.e. we just change the sign of the exponent twice.

Field Trace We define the field trace $\mathrm{Tr}_{K/\mathbb{Q}}$ of the number field $K = \mathbb{Q}[X] / \left(X^N + 1\right)$ like in [Che+20, Sec. 3.3]:

$$\begin{aligned} \mathrm{Tr}_{K/\mathbb{Q}} : \quad & K \to \mathbb{Q} \\ & a \mapsto \sum_{\tau \in \mathrm{Gal}(K/\mathbb{Q})} \tau(a), \end{aligned} \tag{4.8}$$

and note its property

$$\mathrm{Tr}_{K/\mathbb{Q}}\left(X^i\right) = \begin{cases} N & \text{for} \quad i = 0 \quad (\text{Note: } X^0 = 1) \\ 0 & \text{for} \quad i \in [N], i \neq 0 \end{cases}, \tag{4.9}$$

which is derived from Equation (4.6).

Considering the tower of finite fields

$$K = K_N \geq K_{\frac{N}{2}} \geq \dots K_1 = \mathbb{Q}, \tag{4.10}$$

where each of the K_n is the $2n$-th cyclotomic field for a power-of-two integer n, i.e. $n = 2^\ell$, the field trace can be seen as a composition of $\log N$ field traces, i.e.

$$\mathrm{Tr}_{K/\mathbb{Q}} = \mathrm{Tr}_{K_2/K_1} \circ \dots \mathrm{Tr}_{K_N/K_{\frac{N}{2}}}.$$

Note: In the case of our Galois automorphisms, $\mathbb{Q}$ is equivalent to $\mathbb{Z}$. Therefore, we can use R and $\mathbb{Z}$, instead of K and $\mathbb{Q}$.

The fiel trace Tr_{K_N/K_n} zeroises all coefficients with index i for $\frac{N}{n} \nmid i$ and multiplies all others, i.e. for indices $\frac{N}{n} \mid i$, with the scalar N, following Equation (4.6).

In particular, in the special case of $n = 1$, we get Equation (4.9), because $\mathrm{Tr}_{K/\mathbb{Q}} \overset{(4.10)}{=} \mathrm{Tr}_{K_N/K_1}$. I.e. we get the polynomial $N \cdot a[0] \cdot X^0 = N \cdot a[0]$.

4.2. Helper Algorithms

Before we present the actual efficient conversion algorithms in Section 4.3, we present some corresponding helper algorithms which were also described in [Che+20].

4.2.1. Ring Learning With Errors (RLWE) Key Switching (KS)

The KS procedure, [Che+20, Sec. 2.4], changes the secret key of a ciphertext from the original one to an arbitrary new one. For example, in case of an RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$, which originally is decrypted under the RLWE secret key s, we convert the ciphertext ct into the RLWE ciphertext ct', such that ct' can be decrypted with a new RLWE secret key s'. During this KS procedure it is important to approximately preserve the phase μ of the ciphertext. Which implies, that the encrypted plaintext will not change.

We will state a common KS algorithm for an RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$, like in [Che+20, Sec. 2.4]:

Algorithm: KSKeyGen $(s \in R, s' \in R)$ In order to make the actual KS happen, we need some special key, called key switch key. Given are two RLWE secret keys: s, which is the original secret key of ct, and s', which should be the new key after the KS. The algorithm KSKeyGen $(s \in R, s' \in R)$, i.e. Algorithm 4.1, now gives us the needed key switch key $\mathbf{K}$.

Algorithm 4.1: KSKeyGen $(s \in R, s' \in R)$: Generation of a KS key [Che+20, Sec. 2.4]

```
  Input  : Old RLWE secret key $s \in R$,
           New RLWE secret key $s' \in R$
  Output : KS key $\mathbf{K} \in R_q^{L\times 2}$
1 Sample $\mathbf{k}_1 \leftarrow U(R_q^L) \in R_q^L$
2 Sample $\mathbf{e} \leftarrow \chi^L \in R_q^L$
3 $\mathbf{k}_0 = -s' \cdot \mathbf{k}_1 + s \cdot \mathbf{g} + \mathbf{e} \mod q \in R_q^L$          // For g see (4.3)
4 return $\mathbf{K} = [\mathbf{k}_0 \mid \mathbf{k}_1] \in R_q^{L\times 2}$
```

We will use these key switch keys as input for the methods KeySwitch(), i.e. Algorithm 4.2, AutoKeyGen(), i.e. Algorithm 4.3 and LWE-to-LWE(), i.e. Algorithm 4.6.

Note: The key switch keys generation is already implemented in SEAL for RLWE based ciphertexts.

Algorithm: KeySwitch $\left(\mathbf{ct} \in R_q^2; \mathbf{K} \in R_q^{L\times 2}\right)$ After the creation of the key switch key $\mathbf{K}$ we can perform the actual KS: Given are the RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$ and the just created key switch key $\mathbf{K}$, which belongs to the original secret key s and the new secret key s'. With these inputs, the algorithm KeySwitch $\left(\mathrm{ct} \in R_q^2; \mathbf{K} \in R_q^{L\times 2}\right)$, i.e. Algorithm 4.2, transforms the RLWE ciphertext ct into an RLWE ciphertext ct', with corresponding RLWE secret key s'.

We will use this KS in EvalAuto(), i.e. Algorithm 4.4, and LWE-to-LWE(), i.e. Algorithm 4.6.

Note: The KS procedure is already implemented in SEAL for RLWE based ciphertexts.

Algorithm 4.2: KeySwitch $\left(\mathrm{ct} \in R_q^2; \mathbf{K} \in R_q^{L\times 2}\right)$: Transformation of a RLWE ciphertext under an old RLWE secret key $s \in R$ into a RLWE ciphertext under a new RLWE secret key $s' \in R$ [Che+20, Sec. 2.4]

```
Input  : RLWE ciphertext $\mathrm{ct} = (c_0, c_1) \in R_q^2$ (under old RLWE secret key
         $s \in R$),
         KS key $\mathbf{K} \in R_q^{L\times 2}$
Output : RLWE ciphertext $\mathrm{ct}' \in R_q^2$ (under new RLWE secret key $s' \in R$)
1 $\mathrm{ct}' = (c_0, 0) + \mathbf{g}^{-1}(c_1) \cdot \mathbf{K} \mod q \in R_q^2$        // For $\mathbf{g}^{-1}(c_1)$ see (4.4)
2 return $\mathrm{ct}' \in R_q^2$
```

4.2.2. Evaluation of Galois Automorphisms

Like in [Che+20, Sec. 2.5] we will describe a common method to homomorphically evaluate an Galois automorphism of the form (2.8). This will result in the generation of an Galois automorphism key $\mathbf{A}_d$, see Algorithm 4.3 and the actual evaluation of the Galois automorphism on an RLWE ciphertext for a specific Galois element d, see Algorithm 4.4.

Algorithm: AutoKeyGen $\left(d \in \mathbb{Z}_{2N}^\times; s \in R\right)$ Before we can perform the actual evaluation of an Galois homomorphism, i.e. Algorithm 4.4, we have to create a needed special key, called the Galois automorphism key $\mathbf{A}_d$. Assuming we have an RLWE ciphertext $\mathrm{ct} \in R_q^2$ under a secret key $s \in R$. The algorithm for the generation of the automorphism key AutoKeyGen $\left(d \in \mathbb{Z}_{2N}^\times; s \in R\right)$, i.e Algorithm 4.3, is then in principle a common algorithm for the generation of a KS key, see Algorithm 4.1, from the (old) key $\tau_d(s)$ to the (new) key s.

Algorithm 4.3: AutoKeyGen $\left(d \in \mathbb{Z}_{2N}^\times; s \in R\right)$: Generation of an automorphism key $\mathbf{A}_d$ [Che+20, Sec. 2.5]

```
Input  : Galois element $d \in \mathbb{Z}_{2N}^\times$;
         RLWE secret key $s \in R$
Output : Automorphism key $\mathbf{A}_d \in R_q^{L\times 2}$
1 $\mathbf{A}_d \leftarrow \mathrm{KSKeyGen}(\tau_d(s), s) \in R_q^{L\times 2}$        // For $\tau_d(s)$ see (2.8)
2 return $\mathbf{A}_d \in R_q^{L\times 2}$
```

We will use this Galois automorphism key $\mathbf{A}_d$ as input for EvalAuto(), i.e. Algorithm 4.4. We have to mention, that a dedicated Galois automorphism key is needed for every single evaluation of an automorphism τ_d, i.e. we need an own key for every Galois element d. For example, if we have 2 Galois elements, we also have to generate 2 Galois automorphism keys.

Note: The generation of the Galois automorphism key is already implemented in SEAL.

Algorithm: EvalAuto $\left(\mathbf{ct} \in R_q^2, d \in \mathbb{Z}_{2N}^\times; \mathbf{A}_d \in R_q^{L\times 2}\right)$ We want to evaluate an Galois automorphism on the RLWE ciphertext $\mathrm{ct} = (c_0, c_1) \in R_q^2$. First we have to keep in mind, that the Galois automorphism τ_d is homomorphous. Looking at the BFV HE decryption scheme from Section 2.5.2, respectively the phase μ, we then follow a homomorphic partial decryption in order to apply the Galois automorphism on the RLWE ciphertext $\mathrm{ct} = (c_0, c_1)$, i.e.

$$\begin{aligned}\tau_d(\mu) &= \tau_d(c_0 + c_1 \cdot s) = \\ &\overset{\tau_d \text{ hom.}}{=} \tau_d(c_0) + \tau_d(c_1) \cdot \tau_d(s),\end{aligned}$$

and therefore get the Galois automorphism applied to every single part of the original RLWE ciphertext individually. I.e. we get a new RLWE ciphertext $(\tau_d(c_0), \tau_d(c_1))$. Finally, we have to perform a KS from the original RLWE secret key s to the new RLWE secret key $\tau_d(s)$, which corresponds to the new ciphertext $(\tau_d(c_0), \tau_d(c_1))$.

After creating the automorphism key $\mathbf{A}_d$ for a specific Galois element $d \in \mathbb{Z}_{2N}^\times$ using Algorithm 4.3, we can evaluate the actual Galois automorphism on an RLWE ciphertext $\mathrm{ct} = (c_0, c_1) \in R_q^2$ under the secret key $s \in R$.

The algorithm EvalAuto $\left(\mathrm{ct} \in R_q^2, d \in \mathbb{Z}_{2N}^\times; \mathbf{A}_d \in R_q^{L\times 2}\right)$, i.e. Algorithm 4.4, is in principle an application of the already known KS algorithm, i.e. Algorithm 4.2: We therefore apply the KS algorithm on the RLWE ciphertext $(\tau_d(c_0), \tau_d(c_1))$ using the Galois automorphism key $\mathbf{A}_d$ as KS key.

Algorithm 4.4: EvalAuto $\left(\mathrm{ct} \in R_q^2, d \in \mathbb{Z}_{2N}^\times; \mathbf{A}_d \in R_q^{L\times 2}\right)$: Homomorphic Evaluation of an Automorphism $\tau_d : a(X) \to a\left(X^d\right)$. [Che+20, Sec. 2.5]

```
Input  : RLWE ciphertext $\mathrm{ct} = (c_0, c_1) \in R_q^2$,
         Galois element $d \in \mathbb{Z}_{2N}^\times$;
         Automorphism key $\mathbf{A}_d \in R_q^{L\times 2}$ corresponding to $d$
Output : RLWE ciphertext $\mathrm{ct}' \in R_q^2$
1 $\mathrm{ct}' \leftarrow \mathrm{KeySwitch}((\tau_d(c_0), \tau_d(c_1)); \mathbf{A}_d) \in R_q^2$  // For $\tau_d(c_0)$, $\tau_d(c_1)$ see (2.8)
2 return $\mathrm{ct}' \in R_q^2$
```

We will use this evaluation of an Galois automorphism for $\mathrm{EvalTr}_{N/n}()$, i.e. Algorithm 4.5, and PackLWEs(), i.e. Algorithm 4.8.

Note: The evaluation of the Galois automorphism is already implemented in SEAL.

From now on we may use the short notation EvalAuto $\left(\mathrm{ct} \in R_q^2, d \in \mathbb{Z}_{2N}^\times\right)$ instead of EvalAuto $\left(\mathrm{ct} \in R_q^2, d \in \mathbb{Z}_{2N}^\times; \mathbf{A}_d \in R_q^{L\times 2}\right)$ and simply assume that there exists a proper automorphism key $\mathbf{A}_d \leftarrow$ AutoKeyGen $\left(d \in \mathbb{Z}_{2N}^\times; s \in R\right)$ already.

4.2.3. Evaluation of Trace Function

We will use the field trace function, i.e. Equation (4.8) to annihilate useless coefficients during the conversion of LWE ciphertext(s) into an RLWE ciphertext in Section 4.3.2 respectively Section 4.3.3.

Evaluating every automorphism $\tau(\cdot)$ and then adding up the resulting ciphertexts would be a simple possibility to implement the field trace. But it would need a total number of N KS operations. In [Che+20, Sec. 3.3] the authors came up with an even more performant algorithm, which is recursive and uses an algebraic structure of cyclotomic fields.

Algorithm: $\mathbf{EvalTr}_{N/n}\left(\mathbf{ct} \in R_q^2\right)$ The algorithm $\mathrm{EvalTr}_{N/n}\left(\mathrm{ct} \in R_q^2\right)$, i.e. Algorithm 4.5, homomorphically evaluates the trace function Tr_{K_N/K_n} for a (non valid) RLWE ciphertext and an integer $n \leq N$ which is a power-of-two, i.e. $n = 2^\ell$ with $\ell \in \mathbb{N}$.

Algorithm 4.5: $\mathrm{EvalTr}_{N/n}\left(\mathrm{ct} \in R_q^2\right)$: Homomorphic Evaluation of the Trace Function Tr_{K_N/K_n}. [Che+20, Algorithm 1]

```
   Input  : RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$,
            A power-of-two integer $n \leq N$
   Output : RLWE ciphertext $\mathrm{ct}' \in R_q^2$
1  $\mathrm{ct}' \leftarrow \mathrm{ct} \in R_q^2$
2  for $k = 1$ to $\log_2(N/n)$ do
3      $\mathrm{ct}' \leftarrow \mathrm{ct}' + \mathrm{EvalAuto}(\mathrm{ct}'; 2^{\log_2(N)-k+1} + 1) \in R_q^2$
4  return $\mathrm{ct}' \in R_q^2$
```

The algorithm for homomorphically evaluating the trace function is used for LWE-to-RLWE(), i.e. Algorithm 4.7, for $n = 1$ and LWEs-to-RLWE(), i.e. Algorithm 4.9.

4.3. Conversion Algorithms

In this section, we will use the mathematical building blocks from Section 4.1, and the algorithms of Section 4.2, to build and discuss the newly presented algorithms from [Che+20]. We can combine these new algorithms in three big blocks, namely

1. improving KS between LWE ciphertexts, see Section 4.3.1,
2. conversion of an LWE ciphertext into an RLWE ciphertext, see Section 4.3.2, and
3. packing of multiple LWE ciphertexts into a single RLWE ciphertext, see Section 4.3.3.

4.3.1. Improved Key Switching (KS) Between Learning With Errors (LWE) Ciphertexts

The first goal is to embed integers of $\mathbb{Z}_q$, or a vector of integers, i.e. elements of $\mathbb{Z}_q^N$, into the ring R_q, [Che+20, Sec. 1]:

Assume we have an LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$, with its phase $\mu_0 = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$, under the LWE secret key $\mathbf{s} \in \mathbb{Z}^N$. With using Equation (4.1), we identify the vector $\mathbf{a}$ as polynomial a, i.e. $a = \sum_{i\in[N]} \mathbf{a}[i] \cdot X^i$. With this polynomial a and the scalar b of the LWE ciphertext $(b, \mathbf{a})$, we create a (not completely valid) new RLWE ciphertext $\mathrm{ct} = (b, a) \in R_q^2$ with phase $\mu = b + a \cdot s \mod q \in R_q$. The corresponding RLWE secret key for this new RLWE ciphertext ct is calculated from the LWE secret key $\mathbf{s}$ by using Equation (4.7), i.e. we have the RLWE secret key $s = \sum_{i\in[N]} \mathbf{s}[i] \cdot X^{-i}$. Although the new RLWE ciphertext ct is not completely valid, its phase μ, which is a polynomial, contains the original LWE phase μ_0 in its constant term, i.e. $\mu[0] = \mu_0$.

In [Che+20] this idea is used to improve the performance of the KS from one LWE ciphertext to another LWE ciphertext: Let's assume, for an LWE ciphertext we want to switch from an old secret key $\mathbf{s}$ to a new secret key $\mathbf{s}'$. We first use Equation (4.7) to convert the two LWE secret keys $\mathbf{s}$ and $\mathbf{s}'$ into RLWE secret keys, i.e. $s = \sum_{i\in[N]} \mathbf{s}[i] \cdot X^{-i}$, resp. $s' = \sum_{i\in[N]} \mathbf{s}'[i] \cdot X^{-i}$. By using the idea from above, i.e. Equation (4.1), we then extract an RLWE ciphertext ct out of the LWE ciphertext. Now, this RLWE ciphertext ct corresponds to the RLWE secret key s. But after performing a (regular) RLWE KS procedure, i.e. Algorithm 4.2, on this RLWE ciphertext ct, i.e. from the old RLWE secret key s to the new RLWE secret key s', we get a new RLWE ciphertext ct', now under the new RLWE secret key s'. Because the phase of this new RLWE ciphertext ct' is approximately equal to the phase μ of the old RLWE ciphertext ct, we can reconstruct an LWE ciphertext out of the RLWE ciphertext ct', by using Equation (4.2).

Learning With Errors (LWE) to Learning With Errors (LWE) For an LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$ under the secret LWE key $\mathbf{s} \in \mathbb{Z}^N$ and its phase $\mu_0 = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$, we will present a conversion algorithm, which changes the underlying secret key $\mathbf{s} \in \mathbb{Z}^N$ to a new one $\mathbf{s}' \in \mathbb{Z}^N$ while almost preserving the phase μ_0 of the ciphertext, [Che+20, Sec. 3.2].

We first use Equation (4.1) and Equation (4.7) to convert the two vectors $\mathbf{a} \in \mathbb{Z}_q^N$ and $\mathbf{s} \in \mathbb{Z}^N$ into polynomials $a \in R_q$ and $s \in R$, i.e.

$$a := \iota(\mathbf{a}) \overset{(4.1)}{=} \sum_{i\in[N]} \mathbf{a}[i] \cdot X^i$$

and

$$s := \tau_{-1} \circ \iota(\mathbf{s}) \overset{(4.7)}{=} \sum_{i\in[N]} \mathbf{s}[i] \cdot X^{-i}.$$

With the scalar $b \in \mathbb{Z}_q$ interpreted as the constant polynomial $b \in R_q$, we then define the polynomial tuple $\mathrm{ct} = (b, a) \in R_q^2$. This tuple ct can be seen as an RLWE ciphertext under the secret s, which satisfies

$$\begin{aligned}\langle \mathrm{ct}, (1,s)\rangle [0] &= \langle (b,a), (1,s)\rangle [0] = \\ &= (1\cdot b + a\cdot s)[0] = \\ &\overset{(4.1),(4.7)}{=} \left(b + \left(\sum_{i\in[N]} \mathbf{a}[i]\cdot X^{i}\right)\cdot\left(\sum_{i\in[N]} \mathbf{s}[i]\cdot X^{-i}\right)\right)[0] = \\ &= b + \underbrace{\left(\left(\sum_{i\in[N]} \mathbf{a}[i]\cdot X^{i}\right)\cdot\left(\sum_{i\in[N]} \mathbf{s}[i]\cdot X^{-i}\right)\right)[0] =}_{=\langle \mathbf{a},\mathbf{s}\rangle} \\ &= b + \langle \mathbf{a}, \mathbf{s}\rangle = \mu_0.\end{aligned}$$

This means the phase $\mu = \langle \mathrm{ct}, (1,s)\rangle = b + a\cdot s \mod q \in R_q$ of the RLWE ciphertext ct contains the phase μ_0 of the LWE ciphertext in its constant term $\mu[0]$, i.e. $[0] = \mu_0$. However, the values of all other coefficients, $\mu[i]$, $0 \neq i \in [N]$, are not valid, meaning these coefficients contain random garbage.

The RLWE ciphertext ct is not valid, but we still can perform a KS which preserves the phase, i.e. Algorithm 4.2, from s to s', where s' again can be obtained by using Equation (4.7), i.e.

$$s' := \tau_{-1}\circ\iota(\mathbf{s}') \overset{(4.7)}{=} \sum_{i\in[N]} \mathbf{s}'[i]\cdot X^{-i}.$$

After the KS, the constant term $b'[0]$ of the resulting RLWE ciphertext $\mathrm{ct}' = (b', a')$ is again a valid value. We then only use Equation (4.2) to convert the polynomial a' into the vector $\mathbf{a}'$ and together with $b'[0]$ we get the LWE ciphertext $(b'[0], \mathbf{a}')$ under the secret LWE key $\mathbf{s}'$.

Conversion Algorithm: LWE-to-LWE $\left((b,\mathbf{a})\in\mathbb{Z}_q\times\mathbb{Z}_q^N, \mathbf{K}\in R_q^{L\times 2}\right)$ In order to convert an LWE ciphertext $(b,\mathbf{a})\in\mathbb{Z}_q\times\mathbb{Z}_q^N$ under the secret LWE key $\mathbf{s}\in\mathbb{Z}^N$ into an LWE ciphertext under a different secret LWE key $\mathbf{s}'\in\mathbb{Z}^N$, we have to generate a KS key using Algorithm 4.1. Because the two inputs of Algorithm 4.1 are RLWE secret keys, i.e. polynomials, we therefore use Equation (4.7) to transform the vectors $\mathbf{s}$ and $\mathbf{s}'$ into polynomials s and s'.

The actual algorithm LWE-to-LWE $\left((b,\mathbf{a})\in\mathbb{Z}_q\times\mathbb{Z}_q^N, \mathbf{K}\in R_q^{L\times 2}\right)$, i.e. Algorithm 4.6, now takes the LWE ciphertext $(b,\mathbf{a})$ and the just generated secret key $\mathbf{K}$ as input and returns the wanted LWE ciphertext, which is encrypted under the new secret LWE key $\mathbf{s}'$.

Although Algorithm 4.6 won't be part of our actual use case, see Section 6.1, it is a nice to have byproduct of the other algorithms. Especially, it might be applicable in Fast Fully Homomorphic Encryption Over the Torus (TFHE), another LWE based HE scheme, [Chi+20].

Algorithm 4.6: LWE-to-LWE $\left((b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N, \mathbf{K} \in R_q^{L\times 2}\right)$: Conversion of a LWE ciphertext under a secret key $\mathbf{s}$ into a LWE ciphertext under a new secret key $\mathbf{s}'$ [Che+20, Sec. 3.2]

```
Input  : LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$ under secret key $\mathbf{s} \in \mathbb{Z}^N$,
         KS key $\mathbf{K} \in R_q^{L\times 2}$ (from RLWE secret key $s$ to RLWE secret key $s'$,
         which are calculated from the LWE secret keys $\mathbf{s}$ and $\mathbf{s}'$ by (4.7))
Output : LWE ciphertext $\in \mathbb{Z}_q \times \mathbb{Z}_q^N$ under a new secret key $\mathbf{s}' \in \mathbb{Z}^N$
1 $a = \iota(\mathbf{a}) \in R_q$                                   // For $\iota(\mathbf{a})$ see (4.1)
2 RLWE ciphertext ct $\leftarrow (b, a) \in R_q^2$     // ct can be seen as an RLWE
    ciphertext under the RLWE secret key $s$ calculated by (4.7), but
    it's not a valid RLWE ciphertext, [Che+20, Sec. 3.2]
3 $\mathrm{ct}' = (b', a') \leftarrow \mathrm{KeySwitch}(\mathrm{ct}, \mathbf{K}) \in R_q^2$     // $\mathrm{ct}'$ can be seen as an RLWE
    ciphertext under the RLWE secret key $s'$ calculated by (4.7), but
    it's not a valid RLWE ciphertext, [Che+20, Sec. 3.2]
4 $\mathbf{a}' = \iota^{-1}(a') \in \mathbb{Z}_q^N$                        // For $\iota^{-1}(a')$ see (4.2)
5 return $(b'[0], \mathbf{a}') \in \mathbb{Z}_q \times \mathbb{Z}_q^N$
```

4.3.2. Conversion Learning With Errors (LWE) Ciphertext to Ring Learning With Errors (RLWE) Ciphertext

Our next main goal is to efficiently convert an LWE ciphertext into an RLWE ciphertext, [Che+20, Sec. 1]. We have seen during the LWE to LWE conversion, i.e. in Section 4.3.1, that the obtained RLWE ciphertext is not a valid ciphertext yet, when trying to convert it from an LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$ with its phase $\mu_0 = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$. Only the constant term of its phase is valid, all other coefficients are not. We will use the field trace $\mathrm{Tr}_{K/\mathbb{Q}}$, i.e. Equation (4.8), to solve this problem, because this field trace has the property to set all monomials to zero, except the constant term, which gets multiplied by N, see Equation (4.9). So, by homomorphically evaluating the trace function, we get an RLWE ciphertext which phase is approximately equal to $N \cdot \mu_0$.

In Section 4.5 we will show how to remove the factor N again.

Learning With Errors (LWE) to Ring Learning With Errors (RLWE) We will now efficiently convert an LWE ciphertext into an RLWE ciphertext, [Che+20, Sec. 3.3].

Assume an LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$ under the secret LWE key $\mathbf{s} \in \mathbb{Z}^N$ with its phase $\mu_0 = b + \langle \mathbf{a}, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$. Like we have seen before, a simple rewriting of the LWE ciphertext into an RLWE ciphertext $(b, a = \iota(\mathbf{a})) \in R_q^2$ with phase μ, using Equation (4.1), doesn't give us a valid RLWE ciphertext, because only the constant term of its phase is valid. So, in order to make the obtained RLWE ciphertext valid, we want to set all coefficients of the polynomial μ to zero, except the constant one, i.e $\mu[0]$. Luckily, the field trace function, i.e. Equation (4.8), has exactly the wanted property, see

Equation (4.9). This means, a homomorphic evaluation of this field trace function, i.e. applying Algorithm 4.5, would give us the wanted, valid, RLWE ciphertext.

Conversion Algorithm: LWE-to-RLWE $\left((b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N\right)$ After presenting an efficient algorithm to homomorphically evaluate the trace function, i.e. Algorithm 4.5, the final algorithm to convert an LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$ into a valid RLWE ciphertext is a simple application of Algorithm 4.5 for $n = 1$. So, the algorithm LWE-to-RLWE $\left((b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N\right)$, i.e. Algorithm 4.7, first converts the input LWE ciphertext $(b, \mathbf{a})$ into a non valid RLWE ciphertext $\mathrm{ct} = (b, a)$ by using Equation (4.1) and interpreting the scalar $b \in \mathbb{Z}_q$ as the constant polynomial $b \in R_q$. Next, we call Algorithm 4.5 for $n = 1$, and get the wanted, valid RLWE ciphertext.

Algorithm 4.7: LWE-to-RLWE $\left((b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N\right)$: Conversion of a single LWE ciphertext into a RLWE ciphertext. [Che+20, Sec. 3.3]

```
  Input  : LWE ciphertext $(b, \mathbf{a}) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$
  Output : RLWE ciphertext $\mathrm{ct}' \in R_q^2$
1 $a = \iota(\mathbf{a}) \in R_q$                         // For $\iota(\mathbf{a})$ see (4.1)
2 $\mathrm{ct} \leftarrow (b, a) \in R_q^2$    // ct can be seen as an RLWE ciphertext, but it's
    not a valid RLWE ciphertext yet, [Che+20, Sec. 3.3]
3 $\mathrm{ct}' \leftarrow \mathrm{EvalTr}_{N/1}(\mathrm{ct}) \in R_q^2$
4 return $\mathrm{ct}' \in R_q^2$
```

Note: The resulting RLWE ciphertext is not completely correct yet. Because, during the evaluation of the trace function the phase of the input LWE ciphertext gets multiplied by the scalar N. We will fix that in Section 4.5.

The resulting RLWE ciphertext ct' can be decrypted under the RLWE secret key s calculated from the LWE secret key $\mathbf{s}$, using Equation (4.7).

The conversion of an LWE into an RLWE ciphertext, i.e. Algorithm 4.7, is part of our use case, see Section 6.1. Therefore, we will implement it to perform some benchmarks and do a comparison, see Section 7.

4.3.3. Pack Multiple Ring Learning With Errors (RLWE) Ciphertexts Into a Single Learning With Errors (LWE) Ciphertext

The third, and last, efficient homomorphic conversion algorithm, is the merging of n LWE ciphertexts into a single RLWE ciphertext, [Che+20, Sec. 1]. The number of LWE ciphertexts in this case, is again an integer $n \leq N$ which is a power-of-two, i.e. $n = 2^\ell$ with $\ell \in \mathbb{N}$.

So let's assume, we have n LWE ciphertexts under the same secret key $\mathbf{s} \in \mathbb{Z}^N$ and their corresponding phases $\mu_j \in \mathbb{Z}_q$, $j \in [n]$. A simple, and already available, solution would be to use Algorithm 4.7 to convert every LWE ciphertext into an RLWE ciphertext and then sum up all the resulting RLWE ciphertexts. This would result in a single RLWE ciphertext, but would require n times the evaluation of Algorithm 4.5 (for $n = 1$), where each of them would perform $\log_2 (N/1)$ homomorphic evaluations of an automorphism, i.e. Algorithm 4.4. So, in total we would require an overall number of $n \cdot \log_2 (N)$ homomorphic automorphisms, including some automorphism to rotate the results before adding them up, producing a coefficient packing.

In [Che+20], the authors present a more efficient solution. They introduce a FFT-style ciphertext packing algorithm, and are able to reduce the number of needed homomorphic automorphisms to $(n-1) + \log_2 \left(\frac{N}{n}\right)$. Their solution is split into two parts, namely packing and evaluating the trace function.

Learning With Errors (LWE)s to Ring Learning With Errors (RLWE) Let's start by recapping the last problem in a more accurate way, like in [Che+20, Sec. 3.4]: In general a phase of an LWE ciphertext is a scalar, i.e it is an element in $\mathbb{Z}_q$. In contrast, a phase of an RLWE ciphertext is a polynomial of degree N, i.e. it is an element of the ring R_q. That means, a phase of an RLWE ciphertext can store up to N coefficients. Using Algorithm 4.7, i.e. convert an LWE ciphertext into an RLWE ciphertext, gives us an RLWE phase, which approximately has the LWE phase in its constant term, compare to Section 4.3.2. This means, only one of the N coefficients of the RLWE phase is used. So the question arises, how we can efficiently use the remaining coefficients, to merge up to N LWE ciphertexts into one RLWE ciphertext.

Assume, for an integer $n \leq N$ which is a power-of-two, i.e. $n = 2^\ell$ with $\ell \in \mathbb{N}$, we have n LWE ciphertexts $\{(b_j, \mathbf{a}_j)\}_{j\in[n]} \in \mathbb{Z}_q \times \mathbb{Z}_q^N$, all under the same secret LWE key $\mathbf{s} \in \mathbb{Z}^N$ and with their corresponding phase $\mu_j = b_j + \langle \mathbf{a}_j, \mathbf{s} \rangle \mod q \in \mathbb{Z}_q$, $j \in [n]$. By applying the simple solution from above, we would call Algorithm 4.7 on each LWE ciphertext $(b_j, \mathbf{a}_j)$ separately for all $j \in [n]$ and would therefore get n resulting RLWE ciphertexts $\{\mathrm{ct}'_j\}_{j\in[n]}$. Taking the linear combination of these resulting RLWE ciphertexts $\{\mathrm{ct}'_j\}_{j\in[n]}$ gives us a single RLWE ciphertext ct', i.e. $\mathrm{ct}' = \sum_{j\in[n]} \mathrm{ct}'_j \cdot Y^j$ for $Y^j = X^{\frac{N}{n}}$, and the phase of the RLWE ciphertext ct' would be approximately equal to $N \cdot \sum_{j\in[n]} \mu_j \cdot Y^j$. But, as we have seen before, there exists a better solution.

The algorithm to efficiently merge multiple LWE ciphertexts into a single RLWE ciphertext consists of two parts, [Che+20, Sec. 1]: Firstly, it uses a tree-based algorithm to generate an RLWE ciphertext with phase $\mu \in R_q$ and the property $\mu \left[\frac{N}{n} \cdot j\right] \approx n \cdot \mu_j$, $j \in [n]$, compare to [Che+20, Equation 4]. This means, it collects all the LWE phases $\mu_j \in \mathbb{Z}_q$, $j \in [n]$, in a polynomial $\sum_{j\in[n]} \mu_j \cdot Y^j \in K_n = \mathbb{Z}[Y] / (Y^n + 1)$. Secondly, it zeroises all useless coefficients $\mu[i]$ for $\frac{N}{n} \nmid i$, by using the field trace Tr_{K/K_n}, and in the end get an RLWE ciphertext with phase $\approx N \cdot \sum_{j\in[n]} \mu_j \cdot Y^j$.

Algorithm: PackLWEs $\left(\{\mathbf{ct}_j\}_{j\in[n]}\right)$ The recursive algorithm PackLWEs $\left(\{\mathrm{ct}_j\}_{j\in[n]}\right)$, i.e. Algorithm 4.8, merges $n = 2^\ell \leq N$ RLWE ciphertexts into a single one, [Che+20, Sec. 3.4]. It is an FFT-style algorithm. After Algorithm 4.8, the phase μ of the resulting RLWE ciphertext has the following two properties: Firstly, the constant term of the phases of the n input RLWE ciphertexts are stored in its coefficients $\mu[i]$ for $\frac{N}{n} \mid i$.

Secondly, all the other coefficients $\mu[i]$ for $\frac{N}{n} \nmid i$ are useless (and therefore will be removed in the next step).

Algorithm 4.8: PackLWEs $\left(\{\mathrm{ct}_j\}_{j\in[n]}\right)$: Homomorphic Packing of LWE Ciphertexts: merge multiple RLWE ciphertexts into one [Che+20, Algorithm 2]

```
Input  : $n = 2^\ell$ RLWE ciphertexts $\mathrm{ct}_j = (b_j, a_j) \in R_q^2$, for $j \in [n]$, encrypted
         under the same secret key $s \in R$
Output : RLWE ciphertext $\mathrm{ct} \in R_q^2$
1 if $\ell = 0$ then
2     return $\mathrm{ct} \leftarrow \mathrm{ct}_0 \in R_q^2$
3 else
4     $\mathrm{ct}_{even} \leftarrow \mathrm{PackLWEs}\left(\{\mathrm{ct}_{2j}\}_{j\in[2^{\ell-1}]}\right) \in R_q^2$          // recursive
5     $\mathrm{ct}_{odd} \leftarrow \mathrm{PackLWEs}\left(\{\mathrm{ct}_{2j+1}\}_{j\in[2^{\ell-1}]}\right) \in R_q^2$          // recursive
6     $\mathrm{ct} \leftarrow \left(\mathrm{ct}_{even} + X^{N/n} \cdot \mathrm{ct}_{odd}\right) + \mathrm{EvalAuto}\left(\mathrm{ct}_{even} - X^{N/n} \cdot \mathrm{ct}_{odd}, n+1\right) \in R_q^2$
7     return $\mathrm{ct} \in R_q^2$
```

We will use this packing algorithm for LWEs-to-RLWE(), i.e. Algorithm 4.9, and recursively in PackLWEs, i.e. Algorithm 4.8.

Conversion Algorithm: LWEs-to-RLWE $\left(\{(b_j, \mathbf{a}_j)\}_{j\in[n]}\right)$ The previous algorithm, i.e. Algorithm 4.8, packed all valid values into a single element in R_n, but it still has some useless coefficients in it.

The final conversion algorithm LWEs-to-RLWE $\left(\{(b_j, \mathbf{a}_j)\}_{j\in[n]}\right)$, i.e. Algorithm 4.9, therefore firstly utilises Algorithm 4.8 and secondly performs a evaluation of the field trace Tr_{K_N/K_n} afterwards, in order to annihilate these useless coefficients, [Che+20, Sec. 3.4]. This means, we are now finally able to merge $n = 2^\ell$ LWE ciphertexts $(b_j, \mathbf{a}_j) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$, for $j \in [n]$, which are all encrypted under the same secret key $\mathbf{s} \in \mathbb{Z}^N$ into a single RLWE ciphertext $\in R_q^2$.

Note: Like in Algorithm 4.7, the phases of the n input LWE ciphertexts get multiplied by the scalar N, and therefore the resulting RLWE ciphertext is not completely valid yet. We will fix that in Section 4.5.

The resulting RLWE ciphertext ct' can be decrypted under the RLWE secret key s calculated from the LWE secret key $\mathbf{s}$, using Equation (4.7).

The conversion of multiple LWE ciphertexs into a single RLWE ciphertext, i.e. Algorithm 4.9, is part of our use case, see Section 6.1. Therefore, we will implement it to perform some benchmarks and do a comparison, see Section 7.

Note, for $n = 1$, i.e. $\ell = 0$, Algorithm 4.9 equals Algorithm 4.7.

An example of Algorithm 4.9 is presented in Section 4.4.

Algorithm 4.9: LWEs-to-RLWE $\left(\{(b_j, \mathbf{a}_j)\}_{j\in[n]}\right)$: Conversion of multiple LWE ciphertexts into one RLWE ciphertext. [Che+20, Sec. 3.4]

```
    Input  : $n = 2^\ell$ LWE ciphertexts $(b_j, \mathbf{a}_j) \in \mathbb{Z}_q \times \mathbb{Z}_q^N$, for $j \in [n]$, encrypted
             under the same secret key $\mathbf{s} \in \mathbb{Z}^N$
    Output : RLWE ciphertext $\mathrm{ct}' \in R_q^2$
1   for $j \in [n]$ do
2       $a_j = \iota(\mathbf{a}_j) \in R_q$                         // For $\iota(\mathbf{a})$ see (4.1)
3       $\mathrm{ct}_j \leftarrow (b_j, a_j) \in R_q^2$
4   $\mathrm{ct} \leftarrow \mathrm{PackLWEs}\left(\{\mathrm{ct}_j\}_{j\in[n]}\right) \in R_q^2$
5   $\mathrm{ct}' \leftarrow \mathrm{EvalTr}_{N/n}(\mathrm{ct}) \in R_q^2$
6   return $\mathrm{ct}' \in R_q^2$
```

4.4. Example: Ciphertext Conversion

As discussed above, these algorithms enable us to merge multiple LWE ciphertexts into a single RLWE ciphertext. Let's illustrate these steps with an example.

Assume we have a fixed dimension N, for example $N = 4096$, a ciphertext modulus q, some plaintext modulus t and $n = 4$ LWE plaintexts m_0, m_1, m_2 and m_3 which are elements in $\mathbb{Z}_t$.

Using the LWE encryption to encrypt every integer separately, we get four LWE ciphertexts $\mathrm{LWE}(m_0)$, $\mathrm{LWE}(m_1)$, $\mathrm{LWE}(m_2)$ and $\mathrm{LWE}(m_3)$, which all are elements in $\mathbb{Z}_q^{N+1}$.

Next, we use Algorithm 4.9 to merge multiple LWE ciphertexts into a single RLWE ciphertext. I.e. we get one RLWE ciphertext $(b, a) \in R_q^2$.

After decrypting this RLWE ciphertext (b, a) we get a single RLWE plaintext, i.e. the polynomial

$$N \cdot m_3 \cdot x^{3072} + N \cdot m_2 \cdot x^{2028} + N \cdot m_1 \cdot x^{1024} + N \cdot m_0 \cdot x^0,$$

which is an element in R_t. This polynomial now has the original LWE plaintexts in its coefficients.

Note, the conversion is not quite finished yet. One will notice, that all coefficients are multiplied by the factor N. We will remove this factor in a preprocessing step, see Section 4.5. So in the end, the final conversion result doesn't include this factor anymore.

For further computation, for example for using parallel integer multiplication, the server can transform the RLWE ciphertext into a HE ciphertexts. Therefore, we have to transform the coefficients into slots by homomorphically evaluating the inverse of the NTT, which is discussed in Section 5.

4.5. Preprocessing

In [Che+20, Sec. 3.4 Removing the Leading Term] they show, that the conversion of a single or multiple LWE ciphertext(s) into a single RLWE ciphertext using LWE-to-RLWE (), i.e. Algorithm 4.7, or LWEs-to-RLWE (), i.e. Algorithm 4.9, introduces some additional term N into the phase of the resulting RLWE ciphertext. In order to get rid of that additional term N, one therefore has to preprocess the input LWE ciphertext(s). More precisely, one has to multiply every input LWE ciphertext with the constant

$$N^{-1} \mod q,$$

to get rid of the unwanted additional term N. In this case, the phase of the resulting RLWE ciphertext will then finally be approximately equal to

$$N \cdot \sum_{j \in [n]} \left(N^{-1} \cdot \mu_j\right) \cdot Y^j = \sum_{j \in [n]} \mu_j \cdot Y^j.$$

A requirement for this preprocessing is, that the ciphertext modulus q is coprime to the dimension N. In our case this assumption is always given, because for all the primes q_i it holds $q_i = 1 \mod 2N$ anyway, see Section 2.6.3.

As shown in [Che+20, Sec. 3.4], this preprocessing does not introduce additional noise.

Chapter 5.

Efficient Coefficients-to-Slots Conversion

After efficiently converting a single, or multiple LWE ciphertext(s) into an RLWE ciphertext, see Section 4, the resulting RLWE ciphertext does not have a multiplicative homomorphism. That is, because currently the original plaintext(s), i.e. some integers, are located in the coefficients of the ciphertexts. Though, in order to perform parallel multiplications on these encrypted original plaintexts, we want to pack these integers into the plaintext slots, instead of the current coefficients, [Che+20, Sec. 3.4, Further Computation on a Packed Ciphertext]. Because this conversion step depends on the underlying HE scheme, it is neither described in detail in [Che+20], nor benchmarks are performed there. I.e. they leave it to future work.

Therefore, in this section we present an efficient coefficients-to-slots conversion for the BFV scheme. In concrete terms, we transform an RLWE ciphertext into a HE ciphertext. I.e. we will transform an RLWE ciphertext from a coefficient representation into a slot, or CRT representation. This will result in a description of an RLWE-to-HE() algorithm. The basic idea is obtained from [CH18, Sec. A.2]. The conversion is also covered in [GHS12], [HS15], [Che+18c], and [Che+20, Sec. 3.4], but never explained in detail.

Note, in our case, see Section 6.1, the input RLWE ciphertext will be the result of the conversion of (multiple) LWE ciphertext(s) into a single RLWE ciphertext, using Algorithm 4.7 or Algorithm 4.9. This means, the RLWE ciphertext contains the original LWE ciphertexts in its coefficients. After converting an RLWE ciphertext into a HE ciphertext, i.e. performing the discussed coefficients-to-slots conversion, the actual data structure remains the same. Thus, the result is still a polynomial. The difference is, that the original plaintext(s) are now stored in its slots instead of its coefficients.

Further note, it does not make any difference if we say, that the RLWE ciphertext contains the LWE ciphertexts in its coefficients, or that we say, that decryption of the RLWE ciphertext contains the LWE plaintext in its coefficients. That is, because we are talking homomorphically.

5.1. Background

Before we get into the actual conversion, we start with some important building blocks.

Roots of Unity Like we have seen in Section 2.2.2 and Section 2.8, our polynomials are elements of $\mathbb{Z}[X]/\Phi_M(X)$, for the M-th cyclotomic polynomial $\Phi_M(X)$, modulo some modulus. For efficiency reasons, SEAL chooses the negacyclic polynomial $\Phi_{2N}(X) = X^N + 1$, for a power-of-two integer N, i.e. $N = 2^k$ with $k \in \mathbb{N}$, [Dob+21, Sec. 5.3.3]. In particular ciphertexts are elements in $\mathbb{Z}_q[X] / \left(X^N + 1\right)$, for the ciphertext modulus q and plaintexts are elements in $\mathbb{Z}_t[X] / \left(X^N + 1\right)$, for the ciphertext modulus t. We will focus on the latter.

For the coefficients-to-slot conversion, we have to recall the N primitive roots, or roots of unities, $\zeta_0, \zeta_1, \cdots, \zeta_{N-1}$: Like we have seen in Section 2.8, or in [Lai17, Sec. 5.6], the roots of unities ζ_ℓ, $\ell \in [N]$, are defined as follows: Given the primitive root ζ, we define the other primitive roots, or roots of unity, by

$$\zeta_\ell = \zeta^{2\ell+1} \mod t \tag{5.1}$$

for $\ell \in [N]$.

Note, to enable rotations, we need to encode the vector elements $\mathbf{a}[i]$ at ζ^{3^i} to enable rotations, see Equation (2.9), i.e. we need to permute the vector $\mathbf{a}$ before using the inverse of the NTT for encoding.

Utilising Number Theoretic Transform (NTT) The desired conversion from the coefficient representation into CRT representation is basically a homomorphic inverse NTT with permuted roots of unity, i.e. Equation (2.9).

The NTT, which in principle is a DFT, can be done as a matrix multiplication. We will investigate this matrix multiplication in more detail.

Using the N roots of unity ζ_ℓ, $\ell \in [N]$, from Equation (5.1), but in the same order as in Equation (2.9), we build up the $N \times N$ dimensional matrix V

$$\begin{aligned} V &= \left[\zeta_i^j \mod t\right]_{i\in[N], j\in[N]} = \\ &= \begin{bmatrix} 1 & \zeta_0 & \zeta_0^2 & \cdots & \zeta_0^{N-1} \\ 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{N-1} \\ 1 & \zeta_2 & \zeta_2^2 & \cdots & \zeta_2^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{N-1} & \zeta_{N-1}^2 & \cdots & \zeta_{N-1}^{N-1} \end{bmatrix} \in \mathbb{Z}_t^{N\times N}, \end{aligned} \tag{5.2}$$

which is a Vandermonde matrix, [Cor+09, Sec. D.2, Problem D-1]. Using the matrix V, i.e. Equation (5.2), would give us a negacyclic NTT. But we are interested in the inverse negacyclic NTT.

Utilising Inverse Number Theoretic Transform (NTT) The conversion of a ciphertext from coefficient representation into CRT representation is a negacyclic inverse NTT, wich can be described by a matrix vector multiplication of the matrix V^{-1} and the vector which results from decoding the input RLWE ciphertext using batching from Section 2.7. Hence, we have to find the inverse of the matrix V. Luckily, the inverse, i.e.

V^{-1}, has a relatively simple form, namely

$$V^{-1} = \left[\frac{\zeta_j^{-i}}{N} \mod t\right]_{i\in[N],j\in[N]} =$$
$$= \frac{1}{N}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \zeta_0^{-1} & \zeta_1^{-1} & \zeta_2^{-1} & \cdots & \zeta_{N-1}^{-1} \\ \zeta_0^{-2} & \zeta_1^{-2} & \zeta_2^{-2} & \cdots & \zeta_{N-1}^{-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \zeta_0^{-(N-1)} & \zeta_1^{-(N-1)} & \zeta_2^{-(N-1)} & \cdots & \zeta_{N-1}^{-(N-1)} \end{bmatrix} := V' \in \mathbb{Z}_t^{N\times N}. \tag{5.3}$$

We will prove that later on. The inverse will be called V' further on, in sake of simplicity. Because of the simplicity of the inverse matrix, we don't have to implement the computationally expensive general algorithm for the matrix inversion. Instead, we can build up the inverse for V, i.e. Equation (5.2), iteratively with simple loops using Equation (5.3). We only need the field inverses of the roots of unities, $\zeta_0, \zeta_1, \cdots, \zeta_{N-1}$, again in the same order as in Equation (2.9), in advance, i.e. we previously have to calculate $\zeta_0^{-1}, \zeta_1^{-1}, \cdots, \zeta_{N-1}^{-1}$. All powers can then be calculated iteratively out of them.

When using the matrix V', i.e. Equation (5.3), we are finally able to calculate the negacyclic inverse NTT. Performing matrix vector multiplication with the (full) matrix V', i.e. Equation (5.3), and the decoded input RLWE ciphertext using Batching from Section 2.7, would be a simple, and slow, solution for the coefficient to slots conversion, which is briefly mentioned in [Che+20, Sec. 3.4,]. But our goal is to efficiently perform the matrix vector multiplication in order to get a performant coefficients-to-slots conversion for RLWE ciphertexts. So we will do some improvements.

But before, we will prove that V^{-1} is actually the inverse of the matrix V.

Proof. We have to show, that multiplying the matrix V with its inverse V^{-1} gives us the $N \times N$ dimensional identity matrix I, i.e.

$$V \cdot V^{-1} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} =: I. \tag{5.4}$$

By simply reshaping the definitions of V and V^{-1}, i.e. Equation (5.2) and Equation (5.3), we get:

$$\begin{aligned}
V \cdot V^{-1} &= \left[\zeta_i^j \mod t\right]_{i\in[N],j\in[N]} \cdot \left[\frac{\zeta_j^{-i}}{N} \mod t\right]_{i\in[N],j\in[N]} = \\
&= \begin{bmatrix} 1 & \zeta_0 & \zeta_0^2 & \cdots & \zeta_0^{N-1} \\ 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{N-1} \\ 1 & \zeta_2 & \zeta_2^2 & \cdots & \zeta_2^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \zeta_{N-1} & \zeta_{N-1}^2 & \cdots & \zeta_{N-1}^{N-1} \end{bmatrix} \cdot \\
&\qquad \cdot \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ \zeta_0^{-1} & \zeta_1^{-1} & \zeta_2^{-1} & \cdots & \zeta_{N-1}^{-1} \\ \zeta_0^{-2} & \zeta_1^{-2} & \zeta_2^{-2} & \cdots & \zeta_{N-1}^{-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \zeta_0^{-(N-1)} & \zeta_1^{-(N-1)} & \zeta_2^{-(N-1)} & \cdots & \zeta_{N-1}^{-(N-1)} \end{bmatrix} = \\
&= \left[\sum_{k=0}^{N-1} \zeta_i^k \cdot \frac{1}{N} \cdot \zeta_j^{-k} \mod t\right]_{i\in[N],j\in[N]} = \\
&= \left[\frac{1}{N} \cdot \sum_{k=0}^{N-1} \zeta_i^k \cdot \zeta_j^{-k} \mod t\right]_{i\in[N],j\in[N]} .
\end{aligned}$$

In the first case, if $i = j$, i.e. considering the diagonal elements, we get:

$$\begin{aligned}
\frac{1}{N} \cdot \sum_{k=0}^{N-1} \zeta_i^k \cdot \zeta_i^{-k} \mod t &= \frac{1}{N} \cdot \sum_{k=0}^{N-1} \zeta_i^{k+(-k)} \mod t = \\
&= \frac{1}{N} \cdot \sum_{k=0}^{N-1} \zeta_i^0 \mod t = \\
&= \frac{1}{N} \cdot \sum_{k=0}^{N-1} 1 \mod t = \\
&= \frac{1}{N} \cdot N \mod t = \\
&= 1 \mod t = \\
&= 1.
\end{aligned}$$

In the second case, i.e. if $i \neq j$, by applying the definition of ζ_ℓ, i.e. Equation (5.1), we obtain:

$$\begin{aligned}\frac{1}{N} \cdot \sum_{k=0}^{N-1} \zeta_i^k \cdot \zeta_j^{-k} \mod t &\overset{(5.1)}{=} \frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\zeta^{2i+1}\right)^k \cdot \left(\zeta^{2j+1}\right)^{-k} \mod t = \\ &= \frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\zeta^{2(i-j)}\right)^k \mod t\end{aligned} \tag{5.5}$$

In general, the following property holds for $\alpha \neq 1$:

$$\sum_{k=0}^{N-1} \alpha^k = \frac{\alpha^N - 1}{\alpha - 1}, \tag{5.6}$$

see [Cor+09, Sec. A.1, Eq. (A.5)] for $n = N - 1$.

Additionally, for the primitive root ζ, it holds

$$\zeta^{2N} \mod t = 1, \tag{5.7}$$

because ζ is a primitive $2N$-th root of unity of the negacyclic polynomial $\Phi_{2N}(X) = X^N + 1$, see Section 2.7 and [Lai17, Sec. 7.4].

Using Equation (5.6) with $\alpha = \zeta^{2(i-j)}$ and the property for the primitive root, i.e. Equation (5.7), we get for Equation (5.5):

$$\begin{aligned}\frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\zeta^{2(i-j)}\right)^k \mod t &\overset{(5.6)}{=} \frac{1}{N} \cdot \frac{\left(\zeta^{2(i-j)}\right)^N - 1}{\zeta^{2(i-j)} - 1} \mod t = \\ &= \frac{1}{N} \cdot \frac{\left(\zeta^{2N}\right)^{i-j} - 1}{\zeta^{2(i-j)} - 1} \mod t = \\ &\overset{(5.7)}{=} \frac{1}{N} \cdot \frac{(1)^{i-j} - 1}{\zeta^{2(i-j)} - 1} \mod t = \\ &= \frac{1}{N} \cdot \frac{1 - 1}{\zeta^{2(i-j)} - 1} \mod t = \\ &= \frac{1}{N} \cdot \frac{0}{\zeta^{2(i-j)} - 1} \mod t = \\ &= 0.\end{aligned}$$

Note, $\alpha = \zeta^{2(i-j)}$ is always $\neq 1$, for $i \neq j$ and $\zeta \neq 1$. Thus, we are allowed to apply Equation (5.6).

To sum up, if $i = j$, then all coefficients of $V \cdot V^{-1}$ are 1. If $i \neq j$, the coefficients are 0. In total, we therefore showed $V \cdot V^{-1} = I$, i.e. Equation (5.4). So V' is actually the inverse of V.

$\square$

5.2. Squashing Matrix

Before we get to the actual matrix vector multiplication, respectively some optimisations of it, we make some changes to the matrix V', i.e. Equation (5.3). We say, we squash it. This squashing, does not change the actual resulting values in the output vector of the matrix vector multiplication, but it changes their order. In particular, squashing will implicitly move all non zero values of the output vector to the top.

After the conversion of (multiple) LWE ciphertext(s) into a single RLWE ciphertext, using Algorithm 4.7 or Algorithm 4.9, the i-th plaintext is not stored in the i-th coefficient, but in the $i \cdot \frac{N}{n}$-th one, which can be seen in the example in Section 4.4. With squashing the matrix V', we ensure that the i-th plaintext is in slot i and not $i \cdot \frac{N}{n}$.

Squashing the matrix V', i.e. Equation (5.3), will result in the matrix V'', i.e. Equation (5.8).

Investigating Number of Plaintexts We will start with an observation. The decryption of the input RLWE ciphertext, i.e. the result of the conversion of (multiple) LWE ciphertext(s) into a single RLWE ciphertext, using Algorithm 4.7 or Algorithm 4.9, contains the original LWE plaintexts in its coefficients. This is illustrated in the example in Section 4.4. But what if the number of LWE plaintexts, namely n, is much smaller than the number of available coefficients/slots N? This would mean, that most of the coefficients, i.e. $N - n$ coefficients, are zero.

And also only n values of the resulting output vector of the matrix vector multiplication are non zero. So we will focus on them.

Removing Matrix Rows Let's assume that the number of non-zero coefficients, i.e. the number of original LWE plaintext, n is also a power-of-two, i.e. $n = 2^\ell$. This assumption is always given, when looking at the input of Algorithm 4.7 and Algorithm 4.9.

The plan is, that we will change the matrix V', i.e. Equation (5.3), a little bit. I.e. we only use the matrix rows we really need. These are the rows, which lead to a non zero value, when performing the matrix vector multiplication. Therefore, we remove all rows of the matrix which correspond to the coefficients which are not used, i.e. are zero. We don't actually remove the rows, but we set them to zero and move them to the end of the matrix. I.e. we say, we squash the matrix. The resulting $N \times N$ matrix is then called $V'' \in \mathbb{Z}_t^{N \times N}$.

Let's describe the last step in detail. The n indices of the coefficients which are not zero are m_r with $r \in [n]$. We set all entries of the other rows, i.e. all rows with index not equal to m_r for $r \in [n]$, to zero and move them to the back of the matrix. The resulting matrix V'' then has the following form

$$V'' = \begin{bmatrix} V'_{m_0,0} & V'_{m_0,1} & \cdots & V'_{m_0,N-1} \\ V'_{m_1,0} & V'_{m_1,1} & \cdots & V'_{m_1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ V'_{m_{n-1},0} & V'_{m_{n-1},1} & \cdots & V'_{m_{n-1},N-1} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \in \mathbb{Z}_t^{N \times N}. \tag{5.8}$$

Note, these n non-zero coefficients are distributed equally in the polynomial. I.e. by setting

$$k = N/n, \tag{5.9}$$

we get, that each k-th coefficient is non-zero. One can see this behaviour in the example in Section 4.4 for $k = \frac{N}{n} = \frac{4096}{4} = 1024$. This means the new matrix V'', i.e. Equation (5.8), consists of the n rows which are all k-th rows of the matrix V', i.e. Equation (5.3), plus the remaining $N - n$ rows which are now set to zero. I.e. for the row index m_r it holds

$$m_r = r \cdot k,$$

for $r \in [n]$.

Note, squashing, i.e. changing matrix V', i.e. Equation (5.3), to matrix V'', i.e. Equation (5.8), does only change the order of the resulting output vector of the matrix vector multiplication. I.e. the matrix vector multiplication, which gives us the wanted conversion from coefficients to slots. In particular, it implicitly moves all values of the output vector, which correspond to non zero coefficients, to the top of the vector. These are exactly the rows, which result in a non zero value when performing the matrix vector multiplication. All others result in zeros anyway.

Looking at the example in Section 4.4, it moves the values of the output vector, which belong to the non zero coefficients, i.e. with indices $0, 1024, 2028$ and 3072, to the top of the output vector, i.e. indices $0, 1, 2$ and 3.

5.3. Diagonal Method

We aim to optimise the matrix vector multiplication in this section. The first optimisation is to use the Diagonal Method for the matrix vector multiplication, which will be described now.

Assume, we have a, general, $P \times P$ dimensional matrix

$$A = [A_{m,n}]_{m \in \{0,\ldots,P-1\}, n \in \{0,\ldots,P-1\}} \in \mathbb{Z}^{P \times P},$$

and some P dimensional vector

$$\mathbf{b} \in \mathbb{Z}^P.$$

The Diagonal Method, [Bam+20, Sec. 8.2 (3)], [HS18, Sec. 3 (4)], [HS15, Sec. 4.2 (3)], is illustrated for the case $P = 4$ in Equation (5.10).

$$\begin{aligned}
A \times \mathbf{b} &= \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \\
&= \begin{bmatrix} A_{0,0} & & & \\ & A_{1,1} & & \\ & & A_{2,2} & \\ & & & A_{3,3} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} + \\
&+ \begin{bmatrix} & A_{0,1} & & \\ & & A_{1,2} & \\ & & & A_{2,3} \\ A_{3,0} & & & \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_0 \end{bmatrix} + \\
&+ \begin{bmatrix} & & A_{0,2} & \\ & & & A_{1,3} \\ A_{2,0} & & & \\ & A_{3,1} & & \end{bmatrix} \cdot \begin{bmatrix} b_2 \\ b_3 \\ b_0 \\ b_1 \end{bmatrix} + \\
&+ \begin{bmatrix} & & & A_{0,3} \\ A_{1,0} & & & \\ & A_{2,1} & & \\ & & A_{3,2} & \end{bmatrix} \cdot \begin{bmatrix} b_3 \\ b_0 \\ b_1 \\ b_2 \end{bmatrix} = \\
&= \begin{bmatrix} A_{0,0} \\ A_{1,1} \\ A_{2,2} \\ A_{3,3} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} A_{0,1} \\ A_{1,2} \\ A_{2,3} \\ A_{3,0} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_0 \end{bmatrix} + \begin{bmatrix} A_{0,2} \\ A_{1,3} \\ A_{2,0} \\ A_{3,1} \end{bmatrix} \cdot \begin{bmatrix} b_2 \\ b_3 \\ b_0 \\ b_1 \end{bmatrix} + \begin{bmatrix} A_{0,3} \\ A_{1,0} \\ A_{2,1} \\ A_{3,2} \end{bmatrix} \cdot \begin{bmatrix} b_3 \\ b_0 \\ b_1 \\ b_2 \end{bmatrix} = \\
&= \sum_{i=0}^{P-1} \operatorname{diag}(A, i) \cdot \operatorname{rot}(\mathbf{b}, i)
\end{aligned} \tag{5.10}$$

In Equation (5.10), $\operatorname{diag}(A, i)$ is the i-th diagonal of the matrix A written into a vector of size P, and $\operatorname{rot}(\mathbf{b}, i)$ is a rotation of the vector $\mathbf{b}$ by the index i.

Thus, simplified, the general Diagonal Method has the form:

$$A \times \mathbf{b} \overset{(5.10)}{=} \sum_{i=0}^{P-1} \operatorname{diag}(A, i) \cdot \operatorname{rot}(\mathbf{b}, i) \tag{5.11}$$

Number of Operations One can easily see, that the Diagonal Method, i.e. Equation (5.11), requires P element-wise vector multiplications, $P - 1$ additions and $P - 1$ rotations, because we don't rotate for $i = 0$, since it gives us the identity.

Implementation Details In our case we use the Diagonal Method with $A = V''$, i.e. Equation (5.8), and the vector $\mathbf{b}$ is the decoding of the input RLWE ciphertext, using Batching from Section 2.7. This RLWE ciphertext is the result of Algorithm 4.7 or Algorithm 4.9.

During the implementation in SEAL, see Section 6, we don't actually perform these vector multiplications. Instead, we perform polynomial multiplications. To do this, we encode the diagonal vectors $\operatorname{diag}(A, i)$ into polynomials using Batching from Section 2.7. Then, the vector multiplications correspond to simple polynomial multiplications of an RLWE plaintext and an RLWE ciphertext. In particular, a polynomial multiplication of the encoded diagonal vector, which is an RLWE plaintext, and the corresponding homomorphic ciphertext rotation of the input RLWE ciphertext, which is again an RLWE ciphertext.

5.4. Baby-Step Giant-Step (BSGS) Algorithm

Ciphertext additions are not that expensive, in terms of computationally power. However, ciphertext multiplications are expensive, as well as homomorphic ciphertext rotations, which require a homomorphic evaluation of the Galois automorphism, [Dob+21, Sec. 5, Table 9]. In this section, in order to minimise the number of expensive operations, we optimise the matrix vector multiplication from Section 5.3, i.e. the Diagonal Method, even further, by introducing the Baby-Step Giant-Step (BSGS) Algorithm, see [Dob+21, Sec. 5.3.2 (4)], or [Bam+20, Sec. 8.2 (4)], which states [HS15] and [HS18, Sec. 6.3]:

$$\begin{aligned} A \times \mathbf{b} &\overset{(5.11)}{=} \sum_{i=0}^{P-1} \operatorname{diag}(A, i) \cdot \operatorname{rot}(\mathbf{b}, i) \\ &= \sum_{k=0}^{P_2-1} \sum_{j=0}^{P_1-1} \operatorname{diag}(A, kP_1 + j) \cdot \operatorname{rot}(\mathbf{b}, kP_1 + j) \\ &= \underbrace{\sum_{k=0}^{P_2-1} \operatorname{rot}\left(\underbrace{\sum_{j=0}^{P_1-1} \operatorname{diag}'(A, kP_1 + j) \cdot \operatorname{rot}(\mathbf{b}, j)}_{\text{Baby-Step}}, kP_1 \right)}_{\text{Giant-Step}} \end{aligned} \tag{5.12}$$

where P_1 and P_2 are some integers, such that

$$P = P_1 \cdot P_2$$

and

$$\operatorname{diag}'(A, i) = \operatorname{rot}\left(\operatorname{diag}(A, i), -\left\lfloor \frac{i}{P_1} \right\rfloor \cdot P_1 \right). \tag{5.13}$$

Note, that in Equation (5.12), i.e. when inserting Equation (5.13) into Equation (5.12), it holds: $\left\lfloor \frac{i}{P_1} \right\rfloor = k$. That is, because for $i = kP_1 + j$ it holds:

$$\begin{aligned}\left\lfloor \frac{i}{P_1} \right\rfloor &= \left\lfloor \frac{kP_1 + j}{P_1} \right\rfloor = \\ &= \left\lfloor \frac{kP_1}{P_1} + \frac{j}{P_1} \right\rfloor = \\ &= \left\lfloor k + \frac{j}{P_1} \right\rfloor = \\ &= k + \left\lfloor \frac{j}{P_1} \right\rfloor = k\end{aligned}$$

where $\left\lfloor \frac{j}{P_1} \right\rfloor = 0$, because $j < P_1$.

We also note, that the $\operatorname{rot}(\mathbf{b}, j)$ in the Baby-Step in Equation (5.12) depends on j only. This means, that they can be precomputed and stored for every $j \in [P_1]$ once and then reused in every Giant-Step. This brings us a performance boost.

Note, that BSGS is equivalent to Diagonal Method from Section 5.3. So it does not matter which method we use to calculate a matrix vector multiplication, the result is the same.

Number of Operations So, in total, we only have $(P_1 - 1) + (P_2 - 1) = P_1 + P_2 - 2$ homomorphic ciphertext rotations of the vector $\mathbf{b}$ when using this BSGS algorithm. Because, again, for $j = 0$ and $k = 0$ the rotation is the identity of the vector. Note, that the rotations in diag', i.e. Equation (5.13), can be done before the encoding using Batching from Section 2.7. That means these rotations happen in plain and therefore are much faster than homomorphic rotations. Therefore, the performance overhead is negligible.

In total, the BSGS algorithm, i.e. Algorithm (5.12), requires $P_1 \cdot P_2 = P$ element-wise vector multiplications, $(P_1 - 1) \cdot (P_2 - 1)$ additions and only $P_1 + P_2 - 2$ ciphertext rotations.

5.5. Simple Encrypted Arithmetic Library (SEAL)

Since the introduction of the Diagonal Method in Section 5.3, i.e. Equation (5.11), and therefore also in the BSGS algorithm in Section 5.4, i.e. Equation (5.12), we use rotations of vectors, respectively rotations of plaintexts/ciphertexts, when calculating a matrix vector multiplication.

Problems now arise, since the algorithms assume cyclic rotations of the whole vector, which is not possible in SEAL, as discussed in Section 2.8. As a recap, we can only rotate rows and columns of the inner $2 \times \frac{N}{2}$ matrix, like it is shown in Equation (2.10), or [Dob+21, Sec. 5.3.3 Inner Structure of HE ciphertexts].

Because of that, the Diagonal Method, respectively the BSGS algorithm, won't give us the correct result when we directly implement them in SEAL in Section 6. Therefore, we have to adapt our matrix vector multiplication for this case. Especially, we will use the homomorphic matrix multiplication similar to [Bam+20, Sec. 8.3].

Splitting the Matrix First, we split the matrix $V'' \in \mathbb{Z}_t^{N\times N}$, i.e. Equation (5.8), into four equally sized square sub-matrices:

$$\begin{aligned} V'' &= \begin{bmatrix} \begin{bmatrix} V''_{0,0} & \cdots & V''_{0,\frac{N}{2}-1} \\ \vdots & \ddots & \vdots \\ V''_{\frac{N}{2}-1,0} & \cdots & V''_{\frac{N}{2}-1,\frac{N}{2}-1} \end{bmatrix} & \begin{bmatrix} V''_{0,\frac{N}{2}} & \cdots & V''_{0,N-1} \\ \vdots & \ddots & \vdots \\ V''_{\frac{N}{2}-1,\frac{N}{2}} & \cdots & V''_{\frac{N}{2}-1,N-1} \end{bmatrix} \\ \begin{bmatrix} V''_{\frac{N}{2},0} & \cdots & V''_{\frac{N}{2},\frac{N}{2}-1} \\ \vdots & \ddots & \vdots \\ V''_{N-1,0} & \cdots & V''_{N-1,\frac{N}{2}-1} \end{bmatrix} & \begin{bmatrix} V''_{\frac{N}{2},\frac{N}{2}} & \cdots & V''_{\frac{N}{2},N-1} \\ \vdots & \ddots & \vdots \\ V''_{N-1,\frac{N}{2}} & \cdots & V''_{N-1,N-1} \end{bmatrix} \end{bmatrix} = \\ &=: \begin{bmatrix} V''_1 & V''_2 \\ V''_3 & V''_4 \end{bmatrix} \in \mathbb{Z}_t^{N\times N}, \end{aligned} \tag{5.14}$$

where

$$V''_i \in \mathbb{Z}_t^{\frac{N}{2}\times\frac{N}{2}}, \tag{5.15}$$

for $i \in \{1,2,3,4\}$.

Then we combine these matrices in a way, such that we get

$$V''_{1,4} := \begin{bmatrix} V''_1 \\ V''_4 \end{bmatrix} \in \mathbb{Z}_t^{N\times\frac{N}{2}},$$

and

$$V''_{3,2} := \begin{bmatrix} V''_3 \\ V''_2 \end{bmatrix} \in \mathbb{Z}_t^{N\times\frac{N}{2}}.$$

In contrast to Section 5.4, where we performed just one BSGS, we now perform two separate BSGS matrix vector multiplications. The first one with matrix $V''_{1,4}$ and the second one with matrix $V''_{3,2}$. In both cases, the upper limit of the initial sum in Equation (5.12) has to be $\frac{N}{2}$, instead of the originally used N. I.e. here the two upper limits of the two sums in BSGS P_1 and P_2 are some integers, such that

$$\frac{N}{2} = P_1 \cdot P_2. \tag{5.16}$$

We then have to rotate the columns of the result of the second BSGS. Finally, we take this rotated ciphertext and add it to the result of the first BSGS.

Matrix Split Explained We will discuss the last steps in a more detailed way. Lets assume, we want to multiply the matrix

$$M = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix} \in \mathbb{Z}^{N \times N},$$

with the vector

$$\mathbf{v} \overset{\text{SEAL}}{=} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \in \mathbb{Z}^N,$$

i.e. calculate $M \times \mathbf{v}$, where the matrix M can be equally split into 4 parts $M_1, M_2, M_3, M_4 \in \mathbb{Z}^{\frac{N}{2} \times \frac{N}{2}}$, and the vector v consists of two halfs $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{Z}^{\frac{N}{2}}$, like in SEAL.

By simply rewriting the multiplication, we get

$$\begin{aligned} M \times \mathbf{v} = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix} \times \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} &= \\ = \begin{bmatrix} M_1 \times \mathbf{v}_1 + M_2 \times \mathbf{v}_2 \\ M_3 \times \mathbf{v}_1 + M_4 \times \mathbf{v}_2 \end{bmatrix} &= \\ = \begin{bmatrix} M_1 \times \mathbf{v}_1 + M_2 \times \mathbf{v}_2 \\ M_4 \times \mathbf{v}_2 + M_3 \times \mathbf{v}_1 \end{bmatrix} &= \\ = \begin{bmatrix} M_1 \times \mathbf{v}_1 \\ M_4 \times \mathbf{v}_2 \end{bmatrix} + \begin{bmatrix} M_2 \times \mathbf{v}_2 \\ M_3 \times \mathbf{v}_1 \end{bmatrix} &= \\ = \begin{bmatrix} M_1 \times \mathbf{v}_1 \\ M_4 \times \mathbf{v}_2 \end{bmatrix} + \text{swap}\left(\begin{bmatrix} M_3 \times \mathbf{v}_1 \\ M_2 \times \mathbf{v}_2 \end{bmatrix} \right) &= \\ = \underbrace{\begin{bmatrix} M_1 \\ M_4 \end{bmatrix} \times \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}}_{\text{1st BSGS}} + \text{swap} \left(\underbrace{\begin{bmatrix} M_3 \\ M_2 \end{bmatrix} \times \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}}_{\text{2nd BSGS}} \right) &. \end{aligned}$$

Now, one can easily see, that instead of performing one matrix vector multiplication of size N using Diagonal Method or the equivalent BSGS algorithm, we can perform two separate multiplications of size $\frac{N}{2}$. The final result follows from swapping, i.e. rotating the columns, of the second intermediate result and summing the two intermediate results up.

Note, one may argue, that the dimensions of the two matrix vector multiplications do not fit. I.e. the matrices are $N \times \frac{N}{2}$, and the vectors are $N \times 1$ dimensional, and therefore a common matrix vector multiplication would not work. But by looking at the Diagonal Method, i.e. Equation (5.11), or also the equivalent BSGS algorithm, one can see, that it does not use the whole matrix. Instead, it extracts the diagonals of the matrix. Although the upper limit in Equation (5.11), or respectively the initial upper limit in Equation (5.12) is only $\frac{N}{2}$, these diagonals are always a vector of the length N, and therefore can be multiplied element wise to the corresponding rotations. Therefore, calculating

$$\begin{bmatrix} M_1 \\ M_4 \end{bmatrix} \times \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}$$

by using the Diagonal Method, or the BSGS algorithm, gives us the same result as directly calculating

$$\begin{bmatrix} M_1 \times \mathbf{v}_1 \\ M_4 \times \mathbf{v}_2 \end{bmatrix}.$$

This, of course, also holds for the second application of the Diagonal Method, or BSGS algorithm.

Note, it is important, that the two sub vectors of the vector $\mathbf{v}$, i.e. $\mathbf{v}_1$ and $\mathbf{v}_2$, don't get interchanged. This is, because during implementation we don't actually multiply vectors, but instead use the rotations of the original RLWE ciphertext, which is a polynomial. So we multiply the encoded diagonals, using Batching from Section 2.7, i.e. a plaintext polynomial, with the rotations of the RLWE ciphertext, i.e. a ciphertext polynomial.

Implementation Details Depending on the scalar k, i.e. Equation (5.9), the matrices V_i'', i.e. Equation (5.15), for $i \in \{1, 2, 3, 4\}$, are differently populated: Only for $k = 1$, i.e. $N = n$, all matrices are fully packed. In all other cases, i.e. $k \neq 1$, the two matrices V_3'' and V_4'' are zero. This is, because both N and n are power of two integers, i.e. $N = 2^{\ell_1}$ and $n = 2^{\ell_2}$.

Number of Operations The number of ciphertext rotations for a single BSGS for a general $N \times N$ dimensional matrix, is $P_1 + P_2 - 2$, for $N = P_1 \cdot P_2$, see Section 5.4.

But in this case, we have two multiplications using BSGS, but applied on smaller matrices. I.e. we here have $\frac{N}{2} = P_1 \cdot P_2$, like in Equation (5.16), and the total number of rotations is $2 \cdot (P_1 + P_2 - 2)$.

5.6. Minimise Ciphertext Rotations

The last goal is, to minimise the number of expensive operations even further. In particular, we want the number of ciphertext rotations to be dependent on the, probably much smaller, number of plaintexts n. So we assume, $n = 2^{\ell} < N$.

We start by observing the two Diagonal Methods in more detail. In particular, we recognize, that the intermediate results, i.e. the $\frac{N}{2}$ summands, of every single Diagonal Method, have a special structure. Namely, the first $\frac{n}{2}$ summands repeat themselves k times.

Therefore, it is sufficient to just sum up the first $\frac{n}{2}$ summands and multiply the result by k. In particular, we will lower the upper limit of the sums in the two Diagonal Methods, i.e. Equation (5.11), to $\frac{n}{2}$, and additionally multiply every entry of the matrix V'' by the scalar k.

Note, because the Diagonal Method from Section 5.3 is equivalent to the BSGS algorithm from Section 5.4, this behaviour also applies to the BSGS algorithm.

Lowering Sum Upper Limit So the first step is to lower the upper limit of the sums in the two Diagonal Methods, i.e. Equation (5.11), from the previously used $\frac{N}{2}$, i.e. Equation (5.16), to $\frac{n}{2}$.

Of course this also applies to the initial sum in the BSGS algorithm, i.e. Equation (5.12), because it is equivalent to the Diagonal Method. In this case, we also have to change P_1 and P_2 for the two BSGS algorithms, such that it holds

$$\frac{n}{2} = P_1 \cdot P_2, \tag{5.17}$$

instead of the previously used $\frac{N}{2} = P_1 \cdot P_2$ from Equation (5.16).

But, the results of the matrix vector multiplications are not correct yet So we have to compensate the lowering of the upper limit by another step.

Compensate Lower Limit We have to multiply the matrix V'', i.e. Equation (5.8), with the scalar k, i.e. Equation (5.9), in order to compensate the previous step of lowering the upper limit of the sum. This results in the $N \times N$ dimensional matrix V''', i.e.

$$V''' = k \cdot V'' \mod t \overset{(5.9)}{=} \frac{N}{n} \cdot V'' \in \mathbb{Z}_t^{N\times N}. \tag{5.18}$$

Final Procedure So we will alter the previous procedure from Section 5.5 in the following way: Like in Equation (5.14), we will extract four sub matrices

$$V_i''' \in \mathbb{Z}_t^{\frac{N}{2}\times\frac{N}{2}},$$

for $i \in \{1,2,3,4\}$, out of V''', i.e. Equation (5.18). Using these four sub matrices for our two Diagonal Methods, or BSGS algorithms, and additionally lowering the upper limits of the sums to $\frac{n}{2}$, will finally give us the same, correct, result but with less ciphertext rotations.

Note, when performing the calculations using two BSGS algorithms, it must hold $\frac{n}{2} = P_1 \cdot P_2$, like in Equation (5.17).

Number of Operations Note, that it always holds $n \leq N$. But here we additionally assume, that n is much smaller than N, i.e. $n = 2^\ell < N$.

Previously in Section 5.5, we had $2 \cdot (P_1 + P_2 - 2)$ ciphertext rotations, for $\frac{N}{2} = P_1 \cdot P_2$, like in Equation (5.16), when considering the inner structure of SEAL ciphertexts and therefore using two BSGS algorithms.

But we achieved to minimise the number of ciphertext rotations even more, by making it dependent on the number of original plaintexts n. In particular we get $2 \cdot (P_1 + P_2 - 2)$ ciphertext rotations, for $\frac{n}{2} = P_1 \cdot P_2$, like in Equation (5.17). Again by using two BSGS algorithms and considering the SEAL rotation behaviour, see Section 5.5.

5.7. Summary

To sum up, during implementation, see Section 6, we have to perform the following steps, in order to efficiently convert an RLWE ciphertext, which is the result of Algorithm 4.9, from the coefficient, into the CRT, i.e. slot, representation.

- First, we extract the N roots of unities, i.e. Equation (5.1), but in the order, which allows rotations of ciphertexts, i.e. Equation (2.9).
- Then we invert all roots of unity and use them to iteratively build up the matrix V', i.e. Equation (5.3).
- Next, we squash the matrix V', i.e. Equation (5.3), to reorder the slots of the resulting ciphertext from $i \cdot \frac{N}{n}$ to i. This results in the matrix V'', i.e. Equation (5.8).
- In order to lower the number of expensive ciphertext multiplications, which will be done in a later step, we have to multiply the matrix V'', i.e. Equation (5.8), with the scalar k, i.e. Equation (5.9). This results in the matrix V''', see Equation (5.18).
- Because of the uncommon behaviour of SEAL ciphertext rotations, we have to split up the matrix V''', i.e. Equation (5.18), into four sub matrices $V_i''' \in \mathbb{Z}_t^{\frac{N}{2} \times \frac{N}{2}}$, for $i \in \{1, 2, 3, 4\}$, like it is done in Equation (5.14), and perform two separate BSGS algorithms, see Section 5.5.
  Note, in Equation (5.14), the matrix V'', i.e. Equation (5.8), gets splitted and not matrix V''', i.e. Equation (5.18).
- Because we previously multiplied the matrix V'', i.e. Equation (5.8), with the scalar k, i.e. Equation (5.9), we are now allowed to lower the upper limits of the sums in the two BSGS algorithms. So, we first calculate
  $$\begin{bmatrix} V_1''' \\ V_4''' \end{bmatrix} \times \mathbf{b},$$
  and then
  $$\begin{bmatrix} V_3''' \\ V_2''' \end{bmatrix} \times \mathbf{b},$$
  both time by using the BSGS algorithm from Section 5.4, and $\frac{n}{2} = P_1 \cdot P_2$, i.e. Equation (5.17). The vector $\mathbf{b}$ is the decoding of the input RLWE ciphertext, using Batching from Section 2.7.
  Note, neither the Diagonal Method, nor the BSGS algorithm, internally perform vector multiplications. Instead, the diagonals get encoded using Batching from Section 2.7, and multiplied to the rotations of the input RLWE ciphertext. This is a plaintext ciphertext multiplication.
- We get the final result, i.e. the desired HE ciphertext, by rotating the columns of the result of the second BSGS and add it to the result of the first BSGS.

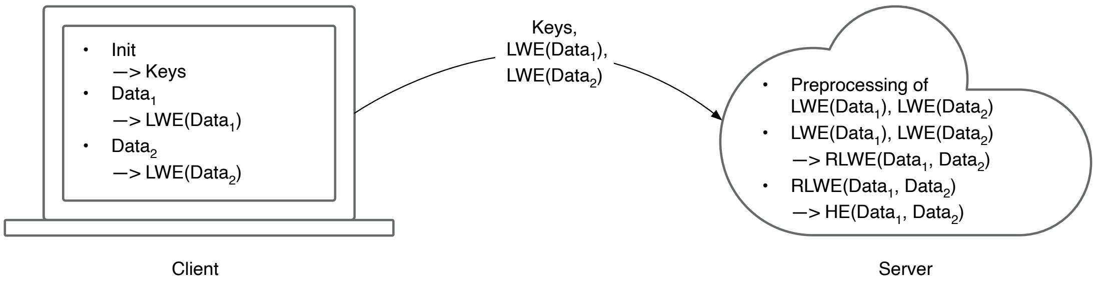
Chapter 6.

Use Case & Implementation

In this section we will describe the practical implementation of the thesis. In particular we describe our use case and the needed implementation steps in SEAL.

6.1. Use Case

A rough overview of our use case, i.e a solution which would mitigate the initial problem of sending too much data, is given in Figure 6.1 and each step gets briefly described below. We want to outsource the expensive homomorphic operations to the server, and,



• Init
—> Keys
• Data$_1$
—> LWE(Data$_1$)
• Data$_2$
—> LWE(Data$_2$)
Keys,
LWE(Data$_1$),
LWE(Data$_2$)
• Preprocessing of LWE(Data$_1$), LWE(Data$_2$)
• LWE(Data$_1$), LWE(Data$_2$)
—> RLWE(Data$_1$, Data$_2$)
• RLWE(Data$_1$, Data$_2$)
—> HE(Data$_1$, Data$_2$)
Client
Server

Figure 6.1.: Overview of the client-server setting. Illustrated for $n = 2$ plaintexts.

therefore, rely on a lightweight communication. The according client-server setting of our use case is described in Section 6.1.1 and Section 6.1.2 respectively below.

6.1.1. Client

Initialisation First, the client, on an edge device with few computational power, starts by performing an initialisation. It must prepare all necessary keys: Of course, the secret LWE key, but also some further keys which the server will need for future conversions, for example KS keys and Galois automorphism keys.

$\mathbf{Data}_i \to \mathbf{LWE}\,(\mathbf{Data}_i)$ For multiple plaintexts $\text{Data}_i, i = 1, \dots n$, $n = 2^\ell$, the client then calculates the symmetric lightweight LWE encryptions $\text{LWE}\,(\text{Data}_i)$, which are introduced in Section 3, all under the same secret key $\mathbf{s}$.

Data Transfer After the creation of the n lightweight ciphertexts, it sends them in addition to all required keys to the server. By using the symmetric lightweight LWE ciphertexts, the amount of data which has to be send is comparably small, see Section 3.

6.1.2. Server

LWE ciphertext conversion After receiving the n lightweight LWE ciphertexts, the server first converts them into regular LWE ciphertexts from Section 2.5.1, by extracting all random vectors $\mathbf{a}_i$ from the seed se.

Preprocessing of the LWE ciphertexts LWE (Data$_1$), ..., LWE (Data$_n$) After the conversion, we have got n regular LWE ciphertexts LWE (Data$_1$), ..., LWE (Data$_n$). The server then preprocesses these LWE ciphertexts, which is described in Section 4.5. I.e. it multiplies each of the input LWE ciphertexts with the constant $N^{-1} \mod q$, in order to get the correct result for algorithm LWE-to-RLWE(), see Algorithm 4.7, respectively algorithm LWEs-to-RLWE(), see Algorithm 4.9.

$\mathbf{LWE}\,(\mathbf{Data}_1)\,,\ldots,\mathbf{LWE}\,(\mathbf{Data}_n) \rightarrow \mathbf{RLWE}\,(\mathbf{Data}_1,\ldots,\mathbf{Data}_n)$ In the next step we will use the algorithms presented in Section 4. We want to create (transform or pack) a single RLWE ciphertext out of the received n LWE ciphertext(s): If we receive a single ciphertext, the server uses the algorithm LWE-to-RLWE(), see Algorithm 4.7. If we receive multiple ciphertexts, it uses the algorithm LWEs-to-RLWE(), see Algorithm 4.9.

$\mathbf{RLWE}\,(\mathbf{Data}_1,\ldots,\mathbf{Data}_n) \rightarrow \mathbf{HE}\,(\mathbf{Data}_1,\ldots,\mathbf{Data}_n)$ The last step is to perform the coefficient-to-slots conversion, introduced in Section 5. That means the RLWE ciphertext is transformed into a HE for further computation. I.e. the n plaintexts are finally encoded into slots of the encrypted polynomial.

6.1.3. Additional Steps

Two additional important steps for a practical application are function evaluation and decryption. However, they are not included in Figure 6.1, because we don't actually change any functionality there and therefore also don't include them in our benchmarks.

Function Evaluation After the server converted the RLWE ciphertext into a HE it is able to perform the actually intended operations on the homomorphically encrypted user data. After that, it sends back the result to the client.

Decryption After receiving the homomorphically encrypted result from the server, the client can finally decrypt it.

6.2. Implementation in Simple Encrypted Arithmetic Library (SEAL)

In the implementation part of this thesis, we extended SEAL, in Version 3.6.1, [20], with all the needed functionalities for our use case from Section 6.1. The goal was to achieve a lightweight communication between a client and a server for a secure homomorphic computation on the server-side, [Che+20, Sec. 4.2].

We first implemented the standard BFV HE LWE en-/decryption, see Section 2.5.1, because SEAL did only support the RLWE en-/decryption. For the actual lightweight communication for secure outsourced computation, there were three rather extensive implementation tasks:

- For achieving an efficient data transfer between the client and the server, described in Section 3, we first implemented the lightweight LWE ciphertexts.
- After that, all of the new algorithms for the efficient homomorphic conversion between LWE and RLWE ciphertexts, originally presented in [Che+20], which are covered in Section 4, were implemented in SEAL.
- Finally, the coefficient-to-slot transition from Section 5, was implemented.

Note, that the communication(-protocols) were not part of the implementation, because they don't depend on the used encryption scheme. We only focused on the individual functionalities on both the client and the server side.

We refer the reader to Section 8.2 for some further thoughts on the implementation.

Chapter 7.
Practical Evaluation

In this section, we first compare the results of our benchmarks to the results presented in [Che+20], see Section 7.3, and then also to HHE as presented in [Dob+21], see Section 7.4.

In our implementation we use a computational security level of 128 bit and multiple different combinations of dimension N, plaintext moduli t and ciphertext moduli q.

7.1. Benchmark Platform

We perform our benchmarks on a Server running Linux. It has 88 logical cores, 2 Intel Xeon CPU E5-2699 v4 with 2.2GHz (turboboost up to 3.6GHz) each and 512GB DDR4 RAM. Our use case and implementation does not have any multi threading or multi processing optimisation.

7.2. Preprocessing for RLWE-to-HE ()

The results of our benchmarks, i.e. Table 7.3, respectively Table 7.4, and Table 7.7, include multiple preprocessing steps.

The first one, i.e. Preprocessing of LWE(s)-to-RLWE (), is described in Section 4.5. It is the already known preprocessing step, i.e. multiplication of the input LWE ciphertexts with some constant value, for LWE-to-RLWE (), i.e. Algorithm 4.7, or LWEs-to-RLWE (), i.e. Algorithm 4.9.

In this section we will focus on the second one, i.e. preprocessing for RLWE-to-HE (), which belongs to the efficient coefficient-to-slots conversion of Section 5. It includes all preparations for the actual two matrix vector multiplications using the BSGS algorithm, which then gets timed separately. In particular it includes the creations of the needed matrices. This preparations are again divided into two separate preprocessing steps:

- Firstly, an independent one: This one does not depend on the number of input plaintexts n. For a specific dimension N, it always stays the same, and therefore can be reused for multiple conversions.
  This preparation step includes the extraction of the roots of unities, and the construction of the matrix V', i.e. Equation (5.3).

- Secondly, a dependent one: This one depends on the number of input plaintexts n. It therefore cannot be reused for other conversions, as long as n does not stay the same.
  In this step, the four sub matrices of the matrix V''', i.e. Equation (5.18), get calculated.

7.3. Comparison to Original Paper

The first comparison will be done against the results of the original paper, i.e [Che+20]. We will run the same benchmarks, i.e. time the three individual conversion algorithms presented in Section 4 individually, as in [Che+20, Sec. 4].

Reference Implementation In [Che+20], they use a 128 bit security, [Che+20, Sec. 4.1], and the following combinations of dimension N and ciphertext modulus q:

- $N = 2^{12} = 4096, \log_2 q = 72$,
- $N = 2^{13} = 8192, \log_2 q = 174$,
- $N = 2^{14} = 16384, \log_2 q = 389$.

For the LWEs-to-RLWE() algorithm, i.e. Algorithm 4.9, they use the following different numbers of input LWE ciphertexts: $n = 2$, $n = 8$ and $n = 32$. For this algorithm, they also state the amortised times, which correspond to the conversion time of a single LWE ciphertext into an RLWE ciphertext. I.e. it's the total time divided by the number of input ciphertexts n.

To the best of our knowledge, in [Che+20] they do not state the used plaintext modulus t. The reason for that is, that they do not want to limit themselves to one specific encoding μ of the message m, which depends on the used HE scheme, in our case BFV.

They use SEAL in version 3.5.1, and run their benchmarks single threaded at 3.50GHz, on a desktop with Intel core i7-4770K CPU, [Che+20, Sec. 4.1]. The final results of the reference implementation timings in [Che+20] are shown in Table 7.1. The total times, and also the amortised times are, given in milliseconds. The noise consumption is given in bits.

Our Implementation In our results, we extend the timings from [Che+20], shown in Table 7.1, with the times and noise consumption for LWEs-to-RLWE(), i.e. Algorithm 4.9, for $n = 64, 128, 256$.

We also include the final coefficients-to-slots conversion, i.e. RLWE-to-HE(), from Section 5, for the same parameters.

Additionally, we state the times for

- the creation of all lightweight LWE ciphertexts from Section 3, for the highest number of input ciphertexts, i.e. 256,

Table 7.1.: Performance of the Conversion Algorithms In [Che+20, Table 2].

$(N, \log_2 q)$	n	$(2^{12}, 72)$		$(2^{13}, 174)$		$(2^{14}, 389)$	
		Total Time (Amortised) ms	Noise bit	Total Time (Amortised)	Noise bit	Total Time (Amortised)	Noise bit
LWE -to- LWE	-	1.03	7	4.81	8	27.1	10
LWE -to- RLWE	-	11.2	18	57.7	21	361	23
LWEs -to- RLWE	2	11.4 (5.70)	18	58.7 (29.4)	21	364 (182)	23
	8	16.8 (2.10)	20	83.2 (10.4)	22	492 (61.5)	24
	32	45.0 (1.41)	20	209 (6.53)	22	1168 (36.5)	24

- the conversion of the 256 lightweight LWE ciphertexts into 256 regular LWE ciphertexts
- and the preprocessing of the 256 LWE ciphertexts for Algorithm 4.7 and Algorithm 4.9, from Section 4.5.

For the current comparison to [Che+20], we used some combinations of the plaintext moduli t listed in Table 7.2, and ciphertext moduli q which are a product of the primes q_i, i.e. $q = \prod_i q_i$. I.e.

- For $N = 2^{12} = 4096$ we used $t = 40961$ and q_i from Table A.1.
- For $N = 2^{13} = 8192$ we used $t = 1032193$ and q_i from Table A.2.
- For $N = 2^{14} = 16384$ we used $t = 786433$ and q_i from Table A.3.

Table 7.2.: Used Plaintext Moduli t for Comparison With [Che+20]

No.	t (dec)	t (hex)	$\log_2(t)$
1	40961	0xA001	16
2	1032193	0xFC001	20
3	786433	0xC0001	20

Our results are shown in Table 7.3 and continue in Table 7.4. The values are the means over 10 test runs and rounded to the second decimal place. The durations are given in milliseconds, and the noise consumption is given in bits.

Result Comparison We will compare the exact results of [Che+20], see Table 7.1, and our implementation, see Table 7.3, in more detail. Only a few things in Table 7.3 are directly comparable. Namely, the results of the LWE-to-LWE() algorithm, i.e. Algorithm 4.6, the LWE-to-RLWE() algorithm, i.e. Algorithm 4.7, and the LWEs-to-RLWE() algorithm, i.e. Algorithm 4.9. In case of LWEs-to-RLWE(), we can compare them for the following different numbers of input LWE ciphertexts: $n = 2$, $n = 8$ and $n = 32$.

One can easily see, that the differences in the results heavily depend on the chosen dimension N. Considering the average of the differences of the amortised timings per dimension N, we can see, that for $N = 2^{12}$, our implementation is around 11% slower. This is, because the LWE-to-LWE() algorithm, i.e. Algorithm 4.6, in this case is a little bit faster than in [Che+20]. But one have to mention, that both results are around one milliseconds, and therefore this result may be a bit inaccurate. When not considering the LWE-to-LWE() algorithm, i.e. Algorithm 4.6, our implementation is in average around 14.55% slower.

For $N = 2^{13}$, our implementation is around 32%, i.e. one third, slower. And for $N = 2^{14}$, it's around 49%, i.e. half the time, slower.

Nevertheless, in our implementation, the noise, which is consumed during the conversions, is in every case a bit lower. Namely, in average around 1.95 bits.

We do not know, whether or not the authors of [Che+20] included the times for the preprocessing step from Section 4.5 in their final results shown in Table 7.1. This preprocessing would be necessary in advance for every input LWE ciphertext of the LWE-to-RLWE() algorithm, i.e. Algorithm 4.7, and the LWEs-to-RLWE() algorithm, i.e. Algorithm 4.9 in order to get the correct result. Nonetheless, when looking at the amortised results of the preprocessing step in Table 7.3, one can see, that they do not play an important role in terms of time. I.e. it takes only 0.01 to 0.2 milliseconds, depending on the dimension N, to preprocess a single LWE ciphertext.

Table 7.3.: Performance of the Conversion Algorithms of Our Implementation For Comparison to [Che+20, Table 2]. Part 1/2, i.e. continues in Table 7.4.

$(N, \log_2 q)$	n	$(2^{12}, 72)$		$(2^{13}, 174)$		$(2^{14}, 389)$	
		Total Time (Amortised) ms	Noise bit	Total Time (Amortised) ms	Noise bit	Total Time (Amortised) ms	Noise bit
Lightw. LWE ct creation	256	90.30 (0.35)	-	318.90 (1.25)	-	1228.70 (4.80)	-
LWE ct conversion	256	76.90 (0.30)	-	300.20 (1.17)	-	1187.30 (4.64)	-
LWE -to- LWE	-	1.00	4.00	6.00	5.90	39.20	7.90
Prepr. LWE(s) -to- RLWE	256	3.00 (0.01)	-	13.00 (0.05)	-	52.00 (0.20)	-
LWE -to- RLWE	-	13.10	17.00	79.00	18.40	543.20	21.60
LWEs -to- RLWE	2	13.10 (6.55)	16.60	80.30 (40.15)	19.10	548.80 (274.40)	21.65
	8	19.20 (2.40)	17.00	110.80 (13.85)	20.10	732.40 (91.55)	22.23
	32	50.50 (1.58)	17.45	271.10 (8.47)	20.52	1731.70 (54.12)	22.36
	64	93.90 (1.47)	17.58	500.50 (7.82)	20.57	3143.00 (49.11)	22.63
	128	183.70 (1.44)	17.93	975.80 (7.62)	20.61	6035.50 (47.15)	22.73
	256	370.80 (1.45)	17.77	1945.50 (7.60)	20.70	11931.60 (46.61)	22.65
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 7.4.: Performance of the Conversion Algorithms of Our Implementation For Comparison to [Che+20, Table 2]. Part 2/2, i.e. continuation of Table 7.3.

⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮

$(N, \log_2 q)$	n	$(2^{12}, 72)$		$(2^{13}, 174)$		$(2^{14}, 389)$	
		Total Time (Amortised) ms	Noise bit	Total Time (Amortised) ms	Noise bit	Total Time (Amortised) ms	Noise bit
Prepr. RLWE -to- HE indep.	-	597.80	-	2215.60	-	9096.00	-
Prepr. RLWE -to- HE dep.	2	143.90	-	573.10	-	2244.10	-
	8	143.50	-	572.20	-	2241.80	-
	32	143.40	-	573.20	-	2244.20	-
	64	143.40	-	573.10	-	2241.80	-
	128	143.90	-	573.80	-	2242.50	-
	256	145.00	-	575.70	-	2247.10	-
RLWE -to- HE	2	3.00 (1.50)	16.90	13.00 (6.50)	21.60	67.00 (33.50)	20.70
	8	10.30 (1.29)	17.80	49.00 (6.12)	22.30	255.90 (31.99)	22.00
	32	36.10 (1.13)	19.20	157.50 (4.92)	24.00	779.50 (24.36)	23.80
	64	64.40 (1.01)	20.30	277.30 (4.33)	25.00	1322.30 (20.66)	24.60
	128	122.60 (0.96)	21.00	517.80 (4.05)	25.90	2410.00 (18.83)	25.60
	256	229.40 (0.90)	22.10	951.30 (3.72)	26.70	4285.20 (16.74)	26.80

7.4. Comparison to Hybrid Homomorphic Encryption (HHE)

In this section, we will compare our practical results to an alternative approach of reducing communication cost. This approach uses HHE with PASTA, a symmetric cipher optimised for homomorphic evaluation, [Dob+21], which is also described in Section 1.5.1.

Note: In [Dob+21, Sec. 3.3] the authors compare their solution, i.e. PASTA, with the solution in [Che+20], which is the basis for our implementation. But they don't state any detailed numbers. The benchmarks in [Che+20] are also done only for relatively small parameters and they don't perform the final coefficient-to-slots conversion. So one cannot compare these two approaches directly.

Reference Implementation Among other things, the authors of [Dob+21] perform multiple benchmarks for their own implementation PASTA, which is described in [Dob+21, Sec. 6], and a second symmetric cipher for HHE, called MASTA, which gets described in [Dob+21, Sec. 4.2], and was originally introduced in [Ha+20].

We will compare our results to the runtimes and noise budget results to different versions of PASTA, respectively MASTA, which are presented in [Dob+21, Sec. 8.2.2]. Especially, to PASTA-3 and PASTA-4, resp. MASTA-4 and MASTA-5, where PASTA/MASTA-m being m round instances of the ciphers, [Dob+21, Sec. 6.1].

In [Dob+21], they state the runtimes for two different encryptions, respectively decryptions: Firstly, the homomorphic encryption of the symmetric secret key. Secondly, the decompression of the HHE ciphertext. This decompression is the homomorphic decryption of the symmetrically encrypted ciphertexts, [Dob+21, Sec. 4.1.3]. For more specifics of these two steps, one can look up [Dob+21, Sec. 3.1, Definition 1], where they describe the public key HHE encryption scheme.

The ciphertext modulus q is chosen in a way, such that the HE scheme has a 128 bit security. The dimension N is the smallest possible integer, which allows a correct decryption. I.e. N is chosen in a way, such that the initial noise budget is large enough to correctly decrypt the ciphertext, [Dob+21, Sec. 4.1.3]. The used plaintext moduli t are listed in Table 7.5.

Table 7.5.: Used Plaintext Moduli t for Comparison With [Dob+21]

No.	t (dec)	t (hex)	$\log_2(t)$
1	65537	0x10001	17
2	8088322049	0x1E21A0001	33
3	1096486890805657601	0xF3781048B580001	60

They use SEAL in Version 3.6.2, [Dob+21, Sec. 2.2]. As their benchmark platform, [Dob+21, Sec. 4.1.2], they use the same platform we use in this thesis, which is described in Section 7.1.

The final results of the PASTA and MASTA implementation timings in [Dob+21] for homomorphically decrypting one block are shown in [Dob+21, Table 17 & Table 18 for 1 Block], and get explained in [Dob+21, Sec. 8.2.2]. For this thesis, we extracted both of them into Table 7.6. I.e. we combined the results of [Dob+21, Table 17] and [Dob+21, Table 18] for one block into Table 7.6. Here one block basically means, encrypting p field elements, where p is the state size of the underlying cipher. This state sizes p is comparable to n, i.e. the number of input plaintexts, and is extracted from [Dob+21, Table 8] and [Dob+21, Table 13] into Table 7.6 for every cipher. Runtimes are given in seconds, and the noise consumption is given in bits.

Table 7.6.: Performance of [Dob+21, Table 17 & Table 18 for 1 Block]

$\log t$ bit	Cipher	p	N	Time s		Noise bit
				Enc. Key	Decomp.	
17	PASTA-3	128	16384	0.016	9.22	269
	PASTA-4	32		0.016	4.19	340
	MASTA-4	128		0.016	11.6	332
	MASTA-5	64	32768	0.062	39.6	414
33	PASTA-3	128	32768	0.057	43.1	444
	PASTA-4	32		0.057	21.2	568
	MASTA-4	128		0.058	54.4	560
	MASTA-5	64		0.055	39.3	686
60	PASTA-3	128	32768	0.055	58.3	714
	PASTA-4	32	65536	0.220	119.2	931
	MASTA-4	128		0.220	284.3	921
	MASTA-5	64		0.219	213.3	1130

Note: In [Dob+21] they state the remaining noise budget, [Dob+21, Sec. 4.1.3], but in this thesis we state the consumed noise, i.e. noise consumption. Therefore we have to subtract the noise budget after decompressing the HHE ciphertext from the noise budget after encrypting the secret key, in order to compare our results.

Our Implementation For comparison with [Dob+21] we use multiple valid combinations of all the plaintext moduli t listed in Table 7.5, and ciphertext moduli q, which are a product of the primes q_i, i.e. $q = \prod_i q_i$:

- For $N = 2^{14} = 16384$, see the q_i in Table A.3.
- For $N = 2^{15} = 32768$, see the q_i in Table A.4.
- For $N = 2^{16} = 65536$, see the q_i in Table A.5.

Note: By default, [Lai17, Table 3], SEAL does not support the dimension $N = 65536$. Therefore we manually add it, and additionally the corresponding ciphertext moduli q_i from Table A.5, in order to perform our benchmarks.

For the number n of plaintexts, respectively input LWE ciphertexts, we use $n = 32$, 64, 128, and 256.

Our results are shown in Table 7.7. The results in Table 7.7 are the mean of 10 test runs. All runtimes are rounded to the third decimal place, and the noise consumption is rounded to the second decimal place. The runtimes are given in seconds, and the noise consumption is given in bits.

Observation: Looking at the, approximately equal, numbers in Table 7.7 one can come up with the following dependencies, i.e,

- the runtime depends on the dimension N,
- and the noise consumption depends on the plaintext modulus t.

Result Comparison The comparison to [Dob+21, Table 17 & Table 18 for 1 Block] is shown in Table 7.6 respectively in Table 7.7.

In [Dob+21, Sec. 3.3], the authors state, that the solution of [Che+20], which is used in this thesis, seems to be faster than their HHE approach. Nevertheless, the solution of [Che+20] does not have the good ciphertext expansion factor of 1, which is achieved by HHE. In contrast, the algorithms in [Che+20] have a ciphertext expansion factor of $\frac{\log q}{\log t} + \mathcal{O}(1)$, which could extremely blow up for big ciphertext moduli q.

One positive thing, that is easily visible, is that our approach needs much less noise than HHE. I.e. the noise consumption in Table 7.7 is, in every single experiment, much lower than in Table 7.6.

However, when comparing the final benchmarks in Table 7.6 and Table 7.7, it seems like the HHE approach is, in total, faster. Especially, the RLWE-to-HE() part including its independent and dependent preprocessing steps is slow. This is quite surprising.

But, one cannot directly compare the numbers in Table 7.6 and Table 7.7. So we will break it down in order to focus on the comparable parts:

PASTA, like many other symmetric ciphers, consists of the following: Firstly, a linear layer, i.e. essentially a BSGS matrix multiplication. Secondly, m rounds consisting of a non linear layer followed by a linear layer, i.e. again a BSGS matrix multiplication. In total we have $m + 1$ matrix multiplications for a m-round cipher.

Let's look, for example, at the numbers of PASTA-3 for $\log_2(t) = 17$ and $N = 16384$ in Table 7.6. There we have a combined runtime of $0.016 + 9.22 = 9.236$ seconds, where the former happens on the client side, and the latter on the server side. PASTA-3 has 3 rounds, that means it has a total number of 4 matrix multiplications.

Each of these 4 matrix multiplications should, at least in theory, be equally fast as the matrix multiplication in our RLWE-to-HE() without the preprocessing, i.e. just the two BSGS matrix multiplications, see Section 5. This means, this step should last around $\frac{9.236}{4} = 2.309$ seconds, because the runtime of the non linear layers is negligible.

In order to compare that to the corresponding runtimes of our implementation, we have to look at Table 7.7 for $\log_2(t) = 17$, $N = 16384$ and $n = 128$. We are interested in $n = 128$, because PASTA-3 has a state size of $p = 128$, and therefore the 1 block benchmark captures our case of $n = 128$.

There, we get $0.026 + 6.056 + 8.946 + 2.229 + 2.409 = 19.666$ seconds in total, and $0.026+6.056+2.409 = 8.491$ seconds without any RLWE-to-HE() preprocessing times. We see, that the RLWE-to-HE() step is quite time intensive, i.e. $8.946+2.229+2.409 = 13.584$ seconds. Especially its independent preprocessing step, which takes 8.946 seconds.

Firstly, like mentioned before, we will just directly compare the matrix multiplication, i.e. RLWE-to-HE() without the preprocessing step. We do this in order to verify, that the benchmarks match, i.e. to check whether the two implementations are comparable. This step takes 2.409 seconds. Compared to a single matrix multiplication in PASTA-3, which takes 2.309 seconds, thats approximately equally fast.

Secondly, we will perform the actual runtime comparison between HHE and our implementation: In total, all the steps in our implementation take $0.026 + 6.056 + 8.946 + 2.229 + 2.409 = 19.666$ seconds, see Table 7.7 for $\log_2(t) = 17$, $N = 16384$ and $n = 128$. In comparison, the decompression step for PASTA-3 for $\log_2(t) = 17$ and $N = 16384$ only takes 9.22 seconds, see Table 7.6. So one can see, that our implementation is much slower than the reference implementation of HHE in [Dob+21].

Additionally, one has to mention that the matrix V', i.e. Equation (5.3), which gets constructed in this independent preprocessing step, stays the same for a specific dimension N. So it only has to be calculated once, and can be reused.

Like mentioned before, a general downside of our approach is, that it does not have a ciphertext expansion factor of 1. In our approach, i.e. using lightweight LWE ciphertexts and efficient homomorphic conversion algorithms, we have to send $\log_2(q)$ bits, plus a seed, see Section 3. On the other side, when using the HHE approach, i.e. using PASTA, one has to send $\log_2(t)$ bits.

When using HHE, one additionally has to transmit the homomorphic encryption of the symmetric secret key. This is not necessary when using our solution, i.e. using efficient conversion algorithms. Both approaches, however, additionally have to send the needed Galois automorphism keys.

Table 7.7.: Performance of Thesis for Comparison to [Dob+21, Table 17 & Table 18]

$\log t$ bit	N	n	Time s					Noise bit	
			Prepr. LWEs -to- RLWE	LWEs -to- RLWE	Prepr. RLWE -to- HE indep.	Prepr. RLWE -to- HE dep.	RLWE -to- HE	LWEs -to- RLWE	RLWE -to- HE
17	16384	32	0.006	1.730	8.946	2.229	0.779	22.37	20.20
		64	0.013	3.140		2.229	1.322	22.74	20.80
		128	0.026	6.056		2.229	2.409	22.69	21.90
		256	0.051	11.920		2.232	4.282	22.57	22.90
	32768	32	0.025	11.013	36.352	9.046	4.071	23.61	20.20
		64	0.049	19.658		9.062	6.593	23.83	21.10
		128	0.098	37.319		9.109	11.675	23.84	21.90
		256	0.196	73.164		9.095	19.884	23.84	23.00
33	16384	32	0.007	1.735	9.099	2.252	0.779	22.66	37.20
		64	0.013	3.147		2.252	1.323	22.64	38.10
		128	0.026	6.045		2.251	2.411	22.79	39.00
		256	0.052	11.948		2.253	4.289	22.79	40.00
	32768	32	0.025	11.005	36.169	9.088	4.082	23.96	37.00
		64	0.050	19.637		9.069	6.613	23.79	38.00
		128	0.100	37.280		9.109	11.707	23.89	39.00
		256	0.199	73.068		9.106	19.884	23.96	39.90
	65536	32	0.095	73.448	146.718	36.408	23.433	26.20	37.40
		64	0.189	129.012		36.464	36.641	26.53	38.00
		128	0.379	242.288		36.413	63.207	26.68	38.90
		256	0.757	471.623		36.516	102.845	26.78	39.90
60	16384	32	0.007	1.727	8.956	2.253	0.908	22.59	64.10
		64	0.013	3.135		2.249	1.580	22.63	64.80
		128	0.026	6.022		2.252	2.927	22.85	65.90
		256	0.053	11.893		2.255	5.318	22.81	66.80
	32768	32	0.025	11.011	36.691	9.109	5.184	23.67	64.10
		64	0.050	19.645		9.100	8.806	23.85	64.90
		128	0.100	37.314		9.093	16.161	23.76	66.10
		256	0.200	73.097		9.112	28.802	23.97	66.90
	65536	32	0.094	73.518	149.018	36.387	23.419	26.61	64.50
		64	0.189	129.197		36.301	36.632	26.86	65.00
		128	0.378	242.624		36.407	63.233	26.94	66.10
		256	0.755	472.328		36.406	102.867	26.97	67.00

Chapter 8.

Conclusion and Future Work

Finally, we will sum up our thesis and briefly discuss some possible future work.

8.1. Conclusion

By introducing lightweight LWE ciphertexts, see Section 3, we managed to mitigate the problem of HE ciphertext expansion. Additionally we needed some efficient conversion algorithm, which were recently published in [Che+20], see Section 4.

The big contribution of the thesis is two-fold: First we have the first public implementation of the conversion algorithms from [Che+20]. Second, to the best of our knowledge we have the first detailed description of the coefficients-to-slots conversion and a corresponding public implementation in the SEAL library, [20].

We saw, that our implementation may be slower than the reference implementation in [Che+20]. However, we publicly showed the implementations of all these algorithms, and extensively performed benchmarks for numerous combinations of plaintext modulus t, dimension N, the corresponding ciphertext modulus q and the number of input LWE ciphertexts n, and compared them to HHE as discussed in [Dob+21].

8.2. Future Work

The C++ code, which we implemented in SEAL in Section 6, is not yet optimised, in terms of speed and memory. In order to get better results in our benchmarks in Section 7, one should consider optimising it in a more C++ way. Currently, it shall serve as a showcase model to publicly present the algorithms, because the authors of [Che+20] did not release the source code alongside their paper. Therefore it is written in a quite human readable, respectively easily understandable way, but not in a C++ efficient way. It is also written in a way, which correlates as much as possible with the descriptions in [Che+20] and this thesis.

This includes, that we tried to separate all our implementations from the original source code of SEAL. This means, that, as far as possible, our newly written code is located in separate files and is not simply added to the existing SEAL code. Only a few existing code lines had to be changed in order to guarantee a fully functional implementation. However, this approach may lead to a more complicated and also slower execution and can be changed if necessary.

One may also include some multi threading or multi processing support for certain tasks, such as the matrix creation in Section 5, which has a high potential of optimisation in terms of efficiency and speed.

Appendix A.

Primes Composing the Ciphertext Modulus

Table A.1.: Primes Composing the Ciphertext Modulus q for $N = 4096$

i	q_i (dec)	q_i (hex)	$\log_2(q_i)$
1	68719403009	0xFFFFEE001	36
2	68719230977	0xFFFFC4001	36
3	137438822401	0x1FFFFE0001	37

Table A.2.: Primes Composing the Ciphertext Modulus q for $N = 8192$

i	q_i (dec)	q_i (hex)	$\log_2(q_i)$
1	8796092858369	0x7FFFFFD8001	43
2	8796092792833	0x7FFFFFC8001	43
3	17592186028033	0xFFFFFFFC001	44
4	17592185438209	0xFFFFFF6C001	44
5	17592184717313	0xFFFFFEBC001	44

Table A.3.: Primes Composing the Ciphertext Modulus q for $N = 16384$

i	q_i (dec)	q_i (hex)	$\log_2(q_i)$
1	281474976546817	0xFFFFFFFD8001	48
2	281474976317441	0xFFFFFFFA0001	48
3	281474975662081	0xFFFFFFF00001	48
4	562949952798721	0x1FFFFFFF68001	49
5	562949952700417	0x1FFFFFFF50001	49
6	562949952274433	0x1FFFFFFEE8001	49
7	562949951979521	0x1FFFFFFEA0001	49
8	562949951881217	0x1FFFFFFE88001	49
9	562949951619073	0x1FFFFFFE48001	49

Table A.4.: Primes Composing the Ciphertext Modulus q for $N = 32768$

i	q_i (dec)	q_i (hex)	$\log_2(q_i)$
1	36028797017456641	0x7FFFFFFFE90001	55
2	36028797014704129	0x7FFFFFFFBF0001	55
3	36028797014573057	0x7FFFFFFFBD0001	55
4	36028797014376449	0x7FFFFFFFBA0001	55
5	36028797013327873	0x7FFFFFFFAA0001	55
6	36028797013000193	0x7FFFFFFFA50001	55
7	36028797012606977	0x7FFFFFFF9F0001	55
8	36028797010444289	0x7FFFFFFF7E0001	55
9	36028797009985537	0x7FFFFFFF770001	55
10	36028797005856769	0x7FFFFFFF380001	55
11	36028797005529089	0x7FFFFFFF330001	55
12	36028797005135873	0x7FFFFFFF2D0001	55
13	36028797003694081	0x7FFFFFFF170001	55
14	36028797003563009	0x7FFFFFFF150001	55
15	36028797001138177	0x7FFFFFFEF00001	55
16	72057594037338113	0xFFFFFFFFF70001	56

Table A.5.: Primes Composing the Ciphertext Modulus q for $N = 65536$

i	q_i (dec)	q_i (hex)	$\log_2(q_i)$
1	1152921504606584833	0xFFFFFFFFFFC0001	60
2	1152921504598720513	0xFFFFFFFFF840001	60
3	1152921504597016577	0xFFFFFFFFF6A0001	60
4	1152921504595968001	0xFFFFFFFFF5A0001	60
5	1152921504592822273	0xFFFFFFFFF2A0001	60
6	1152921504592429057	0xFFFFFFFFF240001	60
7	1152921504589938689	0xFFFFFFFFEFE0001	60
8	1152921504586530817	0xFFFFFFFFECA0001	60
9	1152921504583647233	0xFFFFFFFFE9E0001	60
10	1152921504581419009	0xFFFFFFFFE7C0001	60
11	1152921504580894721	0xFFFFFFFFE740001	60
12	1152921504578666497	0xFFFFFFFFE520001	60
13	1152921504578273281	0xFFFFFFFFE4C0001	60
14	1152921504577748993	0xFFFFFFFFE440001	60
15	1152921504577486849	0xFFFFFFFFE400001	60
16	1152921504570802177	0xFFFFFFFFDDA0001	60
17	1152921504570277889	0xFFFFFFFFDD20001	60
18	1152921504568836097	0xFFFFFFFFDBC0001	60
19	1152921504568442881	0xFFFFFFFFDB60001	60
20	1152921504565559297	0xFFFFFFFFD8A0001	60
21	1152921504565166081	0xFFFFFFFFD840001	60
22	1152921504563724289	0xFFFFFFFFD6E0001	60
23	1152921504563331073	0xFFFFFFFFD680001	60
24	1152921504559267841	0xFFFFFFFFD2A0001	60
25	1152921504556515329	0xFFFFFFFFD000001	60
26	1152921504555466753	0xFFFFFFFFCF00001	60
27	1152921504555073537	0xFFFFFFFFCEA0001	60
28	1152921504554156033	0xFFFFFFFFCDC0001	60
29	1152921504552583169	0xFFFFFFFFCC40001	60

Notation

$\approx$	approximate	46
$\mid$	divides	35, 36, 37, 46, 89
$\equiv$	equivalent	20
$\parallel$	fully divides (if $2^k \parallel i$, then k ist the maximal power of 2 which divides i. I.e. $2^k \mid i$ and $2^{k+1} \nmid i$)	35, 36, 89
$\cong$	isomorph	24, 25
$\gg$	much larger than (a much larger than b: $a \gg b$)	16, 18, 89
$\ll$	much smaller than (a much smaller than b: $a \ll b$)	17, 18, 19, 89
$\nmid$	not divides	35, 37, 46, 89

Acronyms

BFV	Brakerski/Fan-Vercauteren	v, vii, 3, 4, 9, 11, 15, 16, 18, 19, 24, 27, 39, 51, 69, 72
BSGS	Baby-Step Giant-Step	59, 60, 61, 62, 63, 64, 65, 71, 79
CPU	Central Processing Unit	23
CRT	Chinese Remainder Theorem	13, 14, 20, 21, 22, 24, 51, 52, 64
DFT	Discrete Fourier Transform	20, 21, 27, 52
EU	European Union	9
FFT	Fast Fourier transform	21, 46
FHE	Fully Homomorphic Encryption	15
GDPR	General Data Protection Regulation	9

HE	Homomorphic Encryption	v, vii, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 14, 15, 16, 17, 18, 23, 25, 27, 39, 43, 48, 51, 60, 65, 68, 69, 72, 77, 83
HHE	Hybrid Homomorphic Encryption	v, 8, 10, 71, 77, 78, 79, 80, 83
IAIK	Institute of Applied Information Processing and Communications	9
KS	Key Switching	33, 35, 37, 38, 39, 40, 41, 42, 43, 67

LWE	Learning With Errors	v, vii, 1, 2, 3, 5, 6, 7, 9, 11, 12, 15, 16, 17, 18, 29, 30, 31, 33, 36, 40, 41, 42, 43, 44, 45, 46, 47, 48, 51, 56, 67, 68, 69, 71, 72, 73, 74, 73, 78, 80, 83
MNO	mobile network operator	8
NTT	Number Theoretic Transform	4, 5, 9, 20, 21, 23, 25, 27, 48, 52, 53
PHE	Partially Homomorphic Encryption	14
PRNG	Pseudo-Random Number Generator	7, 29, 30

RLWE	Ring Learning With Errors	v, vii, 1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13, 15, 16, 18, 19, 27, 29, 30, 33, 34, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 51, 52, 53, 56, 58, 59, 63, 64, 65, 68, 69, 72
RNS	Residue Number System	9, 20
RSA	Rivest–Shamir–Adleman	14
SEAL	Simple Encrypted Arithmetic Library	v, vii, 7, 9, 11, 15, 18, 19, 20, 21, 22, 23, 26, 33, 38, 39, 40, 51, 59, 60, 61, 62, 64, 65, 67, 68, 69, 72, 77, 78, 83
SIMD	Single-Instruction-Multiple-Data	23
SWHE	Somewhat Homomorphic Encryption	15

TFHE	Fast Fully Homomorphic Encryption Over the Torus	43

Bibliography

[20] *Microsoft SEAL (release 3.6).* `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA. Nov. 2020.

[Bam+20] Alexandros Bampoulidis, Alessandro Bruni, Lukas Helminger, Daniel Kales, Christian Rechberger, and Roman Walch. "Privately Connecting Mobility to Infectious Diseases via Applied Cryptography". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 522.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. "Packed Ciphertexts in LWE-Based Homomorphic Encryption". In: *Public Key Cryptography.* Vol. 7778. Lecture Notes in Computer Science. Springer, 2013, pp. 1–13.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ITCS.* ACM, 2012, pp. 309–325.

[Bra12] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: *CRYPTO.* Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 868–886.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages". In: *CRYPTO.* Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 505–524.

[CH18] Hao Chen and Kyoohyung Han. "Homomorphic Lower Digits Removal and Improved FHE Bootstrapping". In: *EUROCRYPT (1).* Vol. 10820. Lecture Notes in Computer Science. Springer, 2018, pp. 315–337.

[Che+18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A Full RNS Variant of Approximate Homomorphic Encryption". In: *SAC.* Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 347–368.

[Che+18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for Approximate Homomorphic Encryption". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 153.

[Che+18c] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for Approximate Homomorphic Encryption". In: *EUROCRYPT (1).* Vol. 10820. Lecture Notes in Computer Science. Springer, 2018, pp. 360–384.

[Che+20] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. "Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts". In: *IACR Cryptol. ePrint Arch.* 2020 (Dec. 2020), p. 19.

[Chi+20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *J. Cryptol.* 33.1 (2020), pp. 34–91.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[Dob+21] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. "Pasta: A Case for Hybrid Homomorphic Encryption". In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 731.

[FV12] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption". In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[Gen09] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *STOC.* ACM, 2009, pp. 169–178.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Better Bootstrapping in Fully Homomorphic Encryption". In: *Public Key Cryptography.* Vol. 7293. Lecture Notes in Computer Science. Springer, 2012, pp. 1–16.

[Ha+20] Jincheol Ha, Seongkwang Kim, Wonseok Choi, Jooyoung Lee, Dukjae Moon, Hyojin Yoon, and Jihoon Cho. "Masta: An HE-Friendly Cipher Using Modular Arithmetic". In: *IEEE Access* 8 (2020), pp. 194741–194751.

[Har14] David Harvey. "Faster arithmetic for number-theoretic transforms". In: *J. Symb. Comput.* 60 (2014), pp. 113–119.

[HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. "An Improved RNS Variant of the BFV Homomorphic Encryption Scheme". In: *CT-RSA*. Vol. 11405. Lecture Notes in Computer Science. Springer, 2019, pp. 83–105.

[HS15] Shai Halevi and Victor Shoup. "Bootstrapping for HElib". In: *EUROCRYPT (1).* Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 641–670.

[HS18] Shai Halevi and Victor Shoup. "Faster Homomorphic Linear Transformations in HElib". In: *CRYPTO (1).* Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 93–120.

[Lai17] Kim Laine. *Simple Encrypted Arithmetic Library 2.3.1.* `https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf`. 2017.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *EUROCRYPT.* Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 1–23.

[NLV11] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. “Can homomorphic encryption be practical?” In: *CCSW*. ACM, 2011, pp. 113–124.

[Pai99] Pascal Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *EUROCRYPT*. Vol. 1592. Lecture Notes in Computer Science. Springer, 1999, pp. 223–238.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. “On data banks and privacy homomorphisms”. In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[Reg05] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *STOC*. ACM, 2005, pp. 84–93.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[SV14] Nigel P. Smart and Frederik Vercauteren. “Fully homomorphic SIMD operations”. In: *Des. Codes Cryptogr.* 71.1 (2014), pp. 57–81.