KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Bootstrapping Algorithms for BGV and FV

Robin Geelen

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option ICT
Security and Networks

Thesis supervisor:
Prof. dr. ir. Frederik Vercauteren

Assessors:
Prof. dr. N. Smart
Dr. W. Castryck

Mentors:
Dr. ir. C. Bonte
Dr. I. Iliashenko

Academic year 2020 – 2021

Preface

This thesis has been an interesting research opportunity and a nice ending of my master's studies. I would therefore like to thank my promotor, Frederik Vercauteren, for offering the topic. I also appreciate his personal involvement and the many discussions we had. I am also grateful to my friends and family for their interest and continuous support. Finally, I would like to thank Charlotte, Ilia and everyone else at COSIC who took the time for an online conversation.

Robin Geelen

Contents

Abstract

Bootstrapping is the core operation that allows us to transform a somewhat homomorphic scheme into a fully homomorphic scheme. Although bootstrapping enables FHE, it also comes at a serious cost, both in time and memory. There is an implementation of bootstrapping for BGV in the `HElib` library, but the full algorithm was not yet worked out for the FV scheme.

In this thesis, we port `HElib`'s bootstrapping algorithm from BGV to FV. We support recryption of fully packed as well as sparsely packed ciphertexts. Our analysis treats bootstrapping for BGV and FV, which allows us to make the first fair comparison between those schemes. That is, we obtain exactly the same complexity (both time and depth) for the BGV and FV schemes, which we also verify experimentally with our Magma implementation. Finally, we compare the new algorithm for FV to an older bootstrapping approach and show that the new technique outperforms the old one by 6 orders of magnitude.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

R-LWE	Ring learning with errors
NTT	Number-theoretic transform
FFT	Fast Fourier transform
SHE	Somewhat homomorphic encryption
FHE	Fully homomorphic encryption
BGV	Brakerski-Gentry-Vaikuntanathan
FV	Fan-Vercauteren

Symbols

$\varphi(\cdot)$	Euler's totient function
$\mathcal{O}(\cdot)$	Big O notation
$\mathbb{E}[\,\cdot\,]$	Expected value
$\langle a_1, \ldots, a_n \rangle$	The subgroup generated by $a_1, \ldots, a_n$
$(a_1, \ldots, a_n)$	The ideal generated by $a_1, \ldots, a_n$
$\mathcal{K}_m$	The mth cyclotomic number field
$\mathcal{R}$	The ring of integers of $\mathcal{K}_m$
$\Phi_m(x)$	The mth cyclotomic polynomial
ω_m	A root of $\Phi_m(x)$ over $\mathbb{C}$
ζ_m	A formal root of $\Phi_m(x)$ over $\mathbb{Q}$
ζ	A formal root of $\Phi_m(x)$ over $\mathbb{Z}_{p^r}$

Chapter 1

Introduction

While classical cryptography can secure data at rest and in transit, it fails to protect computations on this data. Often, the motivation for collecting information is the desire to process it, so this opens the natural question whether we can encrypt data and at the same time allow computations on it. The last decades of research have answered this question in the affirmative. Many cryptographic algorithms were proposed for processing information in a secure way. In this thesis, we study the field of homomorphic encryption, which allows a third party to perform operations on encrypted data.

1.1 Background

The concept of homomorphic encryption was introduced in 1978 by Rivest, Adleman and Dertouzos [27]. In contrast to classical cryptography, it allows a third party to perform meaningful operations on encrypted data. In other words, we trust the third party to do computations on the data, while at the same time hiding its content. An example scenario is a person or company who stores his encrypted database in a cloud server. Although the data itself is private, it would be advantageous if the server is able to perform operations, like a query lookup. This is indeed possible if a homomorphic encryption scheme is used. In practice, the cloud server does this by performing some algebraic manipulations on the ciphertexts, which have a corresponding result on the plaintexts. The result is thereafter sent back to the owner, who will decrypt it and see that it has been manipulated in the indented way. Thus, the added value of homomorphic encryption is that one can store data securely and still let the cloud server perform operations on it.

Homomorphic encryption schemes can be thought of as evaluating circuits on the plaintexts that they encrypt. The schemes are often categorized based on which circuits they can evaluate. We make a distinction between three types. The first type is the *somewhat homomorphic* encryption scheme (SHE). It allows to evaluate an arithmetic or Boolean circuit of fixed complexity. The complexity of many schemes is mainly determined by the number of multiplications; additions are often neglected. The second type is the *leveled homomorphic* encryption scheme. It allows to evaluate a

circuit of unlimited complexity, but known beforehand. That is, when the complexity of a circuit is known, one can always choose the parameters of the scheme such that it is able to evaluate the circuit. Remark that any leveled homomorphic scheme is by definition also somewhat homomorphic. The third type is the *fully homomorphic* encryption scheme (FHE). It allows to evaluate a circuit of unlimited complexity, even when unknown beforehand. Every fully homomorphic encryption scheme is by definition also leveled and somewhat homomorphic.

In this thesis, we study two related homomorphic encryption schemes, namely BGV and FV. Both schemes can evaluate arithmetic circuits, i.e., consisting of additions and multiplications. The security of BGV and FV is based on the ring learning with errors problem (R-LWE) [20]. A common issue with R-LWE-based encryption schemes (and by extension all current homomorphic schemes) is that they add noise during encryption for security reasons. This noise gets larger when performing operations, and when it becomes too big, a decryption failure can happen. This is what prevents the schemes from being fully homomorphic.

The existence of fully homomorphic encryption was an open problem for many years, until Gentry [11] solved it in 2009. He proved that FHE is theoretically possible under some reasonable assumptions. The idea is to start from a somewhat homomorphic scheme (e.g., BGV or FV), and then add one extra algorithm called bootstrapping or recryption. The goal of bootstrapping is to reduce the noise of a ciphertext. If we can reduce it far enough, then there is additional capacity for evaluating extra operations, thereby making the scheme fully homomorphic. Today, this is still the only known way to create an FHE scheme.

1.2 Thesis goal and outline

The goal of this thesis is to study, implement and compare bootstrapping algorithms for BGV and FV. In particular, we analyze two entirely different techniques for recryption. The first one is a rather early approach for the FV scheme proposed by Fan and Vercauteren [10]. The second one is a more recent algorithm designed by Halevi and Shoup [16, 19] for `HElib`[1]. It is currently the state-of-the-art bootstrapping algorithm for BGV, but the full approach and most advanced techniques have not yet been used for FV. We therefore port the bootstrapping algorithm from BGV to FV, which allows us to make the first fair comparison between those schemes.

The implementation of BGV and FV, together with their recryption procedures, is done in the Magma Computational Algebra System. Magma is designed for algebra and number theory, among others [22]. It is a mathematical environment for working with algebra, but it is not designed for extremely efficient programming. Our goal is therefore not to provide a very fast implementation, but to develop all functionality for analyzing bootstrapping algorithms. The implementation can also be used to test various optimizations and to compare the differences between BGV and FV recryption.

[1] See https://github.com/homenc/HElib

This thesis consists of an introduction, five intermediate chapters and a conclusion. The content is divided over the intermediate chapters as follows:

- **Chapter 2 – Preliminaries:** this chapter explains the background on top of which the other chapters are built. This includes notations, the ring learning with errors problem and the definition of the BGV and FV schemes.
- **Chapter 3 – Bootstrapping for Fully Homomorphic Encryption:** this chapter explains the concept of bootstrapping as originally introduced by Gentry. Moreover, the bootstrapping algorithm of Fan and Vercauteren is discussed as an example.
- **Chapter 4 – Homomorphic Linear Transformations:** this chapter explains how a plaintext can "pack" multiple elements from a small ring and how we can perform operations on those elements independently. We also introduce the concept of linear transformations on the plaintext space because it is an important part of the bootstrapping algorithm in the next chapter.
- **Chapter 5 – Bootstrapping based on Digit Extraction:** this chapter introduces a bootstrapping algorithm for BGV that is based on digit extraction. The actual goal of the chapter is to extend the same technique to the FV scheme.
- **Chapter 6 – Implementation and Results:** this chapter gives details about the implementation of bootstrapping for BGV and FV. We also give experimental results for the execution time, noise capacity and memory usage.

1.3 Contributions

This section gives a summary of our most important contributions:

- We implement BGV and FV, together with their recryption procedures, in the Magma Computational Algebra System.
- We make an estimate of the average noise growth for BGV and FV when applying homomorphic operations. This estimate is also included in the Magma implementation.
- We propose some improvements to the bootstrapping algorithm of Fan and Vercauteren. Our adapted algorithm makes it possible to recrypt ciphertexts when the secret key has ternary instead of binary coefficients.
- We port the techniques of Halevi and Shoup's bootstrapping algorithm from BGV to FV. We describe the procedures for general and thin bootstrapping for both schemes in one big analysis.
- Our analysis and code allow us to make the first fair comparison between BGV and FV bootstrapping. The difference is much smaller than described in the literature.

Chapter 2

Preliminaries

This chapter explains the background on top of which the other chapters are built. In the first section, we explain the mathematical background and notations of cyclotomic number fields and the ring learning with errors problem. In the second section, we present the BGV and FV somewhat homomorphic encryption schemes, which are both based on the ring learning with errors problem. We also give an analysis of the average noise growth of both schemes.

2.1 Ring learning with errors

This section explains the ring learning with errors problem, which can be defined for an arbitrary number field. In this thesis, however, we restrict ourselves to the case of cyclotomic numbers fields because they are the most useful for homomorphic encryption.

2.1.1 Cyclotomic number fields

Let $m \geqslant 1$ be an integer, and let $\omega_m \in \mathbb{C}$ be a primitive mth root of unity. That is, for $i \in \mathbb{Z}$, the equality $\omega_m^i = 1$ holds if and only if i is a multiple of m. As a consequence of this definition, all other primitive mth roots of unity are given by ω_m^j for $j \in \mathbb{Z}_m^*$. The next definition shows that the mth cyclotomic polynomial is exactly the lowest degree, monic and non-zero polynomial that evaluates to zero in all primitive mth roots of unity.

Definition 2.1 (Cyclotomic polynomial)**.** The polynomial

$$\Phi_m(x) \stackrel{\text{def}}{=} \prod_{j \in \mathbb{Z}_m^*} (x - \omega_m^j)$$

is called the mth *cyclotomic polynomial*. Its degree is equal to $n \stackrel{\text{def}}{=} \varphi(m)$, where $\varphi(\cdot)$ is Euler's totient function.

Lemma 2.1 ([30, Section 1.1.1])**.** *Each cyclotomic polynomial is monic, has integer coefficients and is irreducible over the field of rationals* $\mathbb{Q}$*.*

A number field is a finite degree field extension of the rationals. It can be obtained by adjoining a formal root of some irreducible polynomial to $\mathbb{Q}$. In particular, let ζ_m denote a formal root of $\Phi_m(x)$ over $\mathbb{Q}$, then the mth cyclotomic number field is obtained by adjoining this root to $\mathbb{Q}$. This can also be seen from the following definition.

Definition 2.2 (Cyclotomic number field)**.** The field $\mathcal{K}_m \stackrel{\text{def}}{=} \mathbb{Q}(\zeta_m)$ is called the mth *cyclotomic number field*.

Lemma 2.2. *There exists a natural field isomorphism $\mathcal{K}_m \to \mathbb{Q}[x]/(\Phi_m(x))$, which is given by $\zeta_m \mapsto x$.*

Observe that the cyclotomic polynomial $\Phi_m(x)$ is the minimal polynomial of any primitive mth root of unity over $\mathbb{Q}$. The mth cyclotomic number field is the smallest field extension of $\mathbb{Q}$ containing all mth roots of unity, namely the powers of ζ_m. The extension degree is therefore equal to the degree of $\Phi_m(x)$. Computations in a cyclotomic number field can be performed by considering its elements as polynomials modulo $\Phi_m(x)$.

The set of all automorphisms of $\mathcal{K}_m$ that fix the base field $\mathbb{Q}$ forms a group under the function composition operator. This group is called the Galois group of $\mathcal{K}_m/\mathbb{Q}$ and is denoted by $\mathrm{Gal}(\mathcal{K}_m/\mathbb{Q})$. The following lemma shows that there are exactly n automorphisms fixing $\mathbb{Q}$, which are given by mapping ζ_m to any primitive mth root of unity in $\mathcal{K}_m$.

Lemma 2.3 ([30, Theorem 1.1.20])**.** *The Galois group of a cyclotomic number field contains the automorphisms τ_j that are given by $\zeta_m \mapsto \zeta_m^j$ for $j \in \mathbb{Z}_m^*$. There exists a group isomorphism between $\mathbb{Z}_m^*$ and $\mathrm{Gal}(\mathcal{K}_m/\mathbb{Q})$ given by $j \mapsto (\zeta_m \mapsto \zeta_m^j)$.*

It is sometimes useful to consider an embedding of a cyclotomic number field into the real or complex numbers. Two examples are the coefficient and canonical embeddings that are introduced next.

Definition 2.3 (Coefficient embedding)**.** The *coefficient embedding* $\iota\colon \mathcal{K}_m \hookrightarrow \mathbb{R}^n$ is defined by the map

$$\boldsymbol{a} = \sum_{i=0}^{n-1} a_i \zeta_m^i \mapsto (a_0, \ldots, a_{n-1})\,.$$

Definition 2.4 (Canonical embedding)**.** For each $\boldsymbol{a} \in \mathcal{K}_m$, there exists a polynomial $a(x) \in \mathbb{Q}[x]$ such that $\boldsymbol{a} = a(\zeta_m)$. The *canonical embedding* $\tau\colon \mathcal{K}_m \hookrightarrow \mathbb{C}^n$ is defined by the map

$$\boldsymbol{a} = a(\zeta_m) \mapsto \{a(\omega_m^j)\}_{j \in \mathbb{Z}_m^*}\,.$$

Note that this map is well-defined because ω_m^j is a root of $\Phi_m(x)$ for each $j \in \mathbb{Z}_m^*$.

Both the coefficient and canonical embedding are additive group embeddings because they preserve the addition. The canonical embedding is also a ring embedding because it preserves the multiplication. Both the addition and multiplication can therefore be performed component-wise in the embedding space $\mathbb{C}^n$.

Remark. The coefficient embedding interprets an element $\boldsymbol{a} \in \mathcal{K}_m$ as a polynomial of degree less than n. We set $\boldsymbol{a} = a(\zeta_m)$ for some $a(x) \in \mathbb{Q}[x]$, and then we map $\boldsymbol{a}$ to the vector of coefficients of $a(x)$.

Remark. There is a similarity between the canonical embedding and the Galois group of Lemma 2.3. In fact, the canonical embedding considers an element of $\mathcal{K}_m$ under the image of all automorphisms and then replaces the formal root of unity ζ_m by the complex root of unity ω_m.

Both embeddings will be particularly useful for analyzing the noise present in ciphertexts. This will be done by studying the norm of an element through its coefficient or canonical embedding. The notations

$$||\boldsymbol{a}||_p \overset{\text{def}}{=} ||\iota(\boldsymbol{a})||_p \qquad \text{and} \qquad ||\boldsymbol{a}||_p^{\text{can}} \overset{\text{def}}{=} ||\tau(\boldsymbol{a})||_p$$

denote the ℓ_p-norm on the coefficient and canonical embedding respectively.

2.1.2 Notations

All arithmetic in this thesis takes place in $\mathcal{K}_m$, namely the cyclotomic number field that was introduced in the previous section. However, the R-LWE problem is defined over the subring $\mathcal{R} = \mathbb{Z}[\zeta_m] \cong \mathbb{Z}[x]/(\Phi_m(x))$, which coincides with the ring of integers of $\mathcal{K}_m$. Quotient rings of the form $\mathcal{R}_N = \mathcal{R}/N\mathcal{R}$ for an integer $N \geqslant 2$ will also be used. All elements of $\mathcal{R}$, $\mathcal{R}_N$ and $\mathcal{K}_m$ are shown in bold lower case letters, e.g., $\boldsymbol{a} \in \mathcal{R}$. Row vectors are written as $\mathsf{v} \in \mathcal{R}^{1\times\ell}$ and column vectors as $\overrightarrow{\boldsymbol{v}} \in \mathcal{R}^{\ell}$. The inner product between vectors is indicated with angular brackets, e.g., $\langle \mathsf{v}_1, \mathsf{v}_2 \rangle \in \mathcal{R}$. For matrices, we use a hybrid notation, namely $\overrightarrow{\mathsf{v}} \in \mathcal{R}^{\ell_1\times\ell_2}$.

Using Lemma 2.2, we will often view elements of $\mathcal{K}_m$ as polynomials modulo $\Phi_m(x)$. Concretely, for each $\boldsymbol{a} \in \mathcal{K}_m$, there is a unique $a(x) \in \mathbb{Q}[x]$ of degree less than n such that $\boldsymbol{a} = a(\zeta_m)$. By identifying each element with such a unique polynomial representation, we can talk about the coefficients of $\boldsymbol{a}$, which in fact correspond to the coefficients of $a(x)$. This reasoning carries over to elements of $\mathcal{R}$ and $\mathcal{R}_N$ by restricting their coefficients to $\mathbb{Z}$ and $[-N/2, N/2) \cap \mathbb{Z}$ respectively. Note that we can now consider $\mathcal{R}$ and $\mathcal{R}_N$ as subsets of $\mathcal{K}_m$ and as such, elements of the three sets can be combined in one formula. Later in this thesis, there is no distinction anymore between ring elements and polynomials: we will write $\boldsymbol{a} = a(x)$, so reduction modulo $\Phi_m(x)$ is implicit.

For $\boldsymbol{a} \in \mathcal{K}_m$ and an integer $N \geqslant 2$, we denote the centered reduction modulo N by $[\boldsymbol{a}]_N$. More precisely, it is the unique element in $N\mathcal{R} + \boldsymbol{a}$ that has all coefficients in the set $[-N/2, N/2)$. Similarly, we denote flooring, ceiling and rounding to the nearest integer by $\lfloor \boldsymbol{a} \rfloor$, $\lceil \boldsymbol{a} \rceil$ and $\lfloor \boldsymbol{a} \rceil$ respectively. Coefficients are rounded upwards when they are in $\mathbb{Z} + 1/2$.

Probability distributions are the final ingredient for the R-LWE problem. The distribution $\mathcal{U}_N$ denotes the uniform distribution on $\mathcal{R}_N$. Moreover, two distributions on $\mathcal{R}$ are considered, namely χ_{key} and χ_{err}.

2.1.3 The ring learning with errors problem

The ring learning with errors problem is a mathematical problem proposed by Lyubashevsky et al. [20] It is an algebraic variant of the learning with errors problem [26]. The ring variant is more time- and memory-efficient to compute, so building cryptographic primitives on top of it can lead to more efficient protocols.

There are two versions of the R-LWE problem, namely the decisional and the search version. Both are based on the R-LWE distribution for an integer $q \geqslant 2$ and a secret $\boldsymbol{s}$ sampled from χ_{key}. The authors of the problem give a quantum reduction from the approximate shortest vector problem on ideal lattices to both versions of the R-LWE problem [20]. Informally, this means that both the decisional and search version are hard, assuming that worst-case problems on ideal lattices are hard for quantum computers. This hardness result makes the R-LWE problem attractive in the field of cryptography.

Definition 2.5 (R-LWE distribution)**.** Fix a secret $\boldsymbol{s} \in \mathcal{R}_q$. The *R-LWE distribution* $A^q_{\boldsymbol{s},\chi_{\text{err}}}$ is defined by first sampling $\boldsymbol{a} \xleftarrow{\$} \mathcal{U}_q$, $\boldsymbol{e} \xleftarrow{\$} \chi_{\text{err}}$ and then returning $(\boldsymbol{a}, [\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}]_q)$.

Definition 2.6 (Decision R-LWE)**.** Given access to polynomially many samples from $\mathcal{R}_q^2$, the *decision R-LWE problem* is to distinguish between the distributions $A^q_{\boldsymbol{s},\chi_{\text{err}}}$ and $\mathcal{U}_q^2$.

Definition 2.7 (Search R-LWE)**.** Given access to polynomially many samples from $A^q_{\boldsymbol{s},\chi_{\text{err}}}$, the *search R-LWE problem* is to find the underlying $\boldsymbol{s}$.

Remark. The original definition of the R-LWE distribution involved the dual fractional ideal $\mathcal{R}^\vee$. For example, the secret was considered to be an element of $\mathcal{R}_q^\vee$. However, it has been shown that one can also work in the ring $\mathcal{R}_q$ directly without making the problem any easier [9]. Also BGV and FV avoid the use of $\mathcal{R}^\vee$, so we adopt Definition 2.5.

The error distribution χ_{err} should be statistically indistinguishable from a discrete Gaussian distribution in the embedding space [20]. Choosing it badly can lead to R-LWE instantiations that are vulnerable to attacks [25, 5]. The key distribution χ_{key} can be chosen uniformly random in $\mathcal{R}_q$. However, it has been shown that taking the key distribution equal to χ_{err} does not yield security implications [20]. Sometimes, the key is even chosen very sparse because it has a beneficial effect on the noise growth of BGV and FV [19, 10].

2.1.4 Efficient arithmetic in cyclotomic rings

In order to use the ring learning with errors problem and related cryptographic protocols, it is necessary to implement all ring operations efficiently. This section explains how multiplications and automorphisms can be computed using a better approach than classical polynomial arithmetic. In particular, we discuss the *Double-CRT* representation that allows to implement very efficient operations in $\mathcal{R}_q$ [30, 18].

Throughout this section, it is assumed that the R-LWE modulus is given by $q = p_1 \cdot \ldots \cdot p_k$, where the factors p_i are pairwise coprime. This allows us to apply the Chinese remainder theorem to the ring of coefficients of $\mathcal{R}_q$, namely

$$\begin{aligned} \mathbb{Z}_q \to & \quad \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_k} \\ a \mapsto & \quad (a \bmod p_1, \ldots, a \bmod p_k). \end{aligned}$$

Since in practice the time complexity of modular integer multiplication in $\mathbb{Z}_q$ is $\mathcal{O}((\log_2 q)^2)$, we can roughly reduce its execution time with a factor of k assuming that the p_i's are of comparable size. Similarly, the Chinese remainder theorem applies coefficient-wise to $\boldsymbol{a} \in \mathcal{R}_q$, yielding the isomorphism

$$\begin{aligned} \mathcal{R}_q \to & \quad \mathcal{R}_{p_1} \times \cdots \times \mathcal{R}_{p_k} \\ \boldsymbol{a} \mapsto & \quad (\boldsymbol{a} \bmod p_1, \ldots, \boldsymbol{a} \bmod p_k). \end{aligned}$$

Hence all operations can be done in k smaller quotient rings of the form $\mathcal{R}_{p_i}$ instead of the original $\mathcal{R}_q$.

From now on denote the factors of the modulus by p instead of p_i for simplicity. The following lemma determines whether the coefficient ring $\mathbb{Z}_p$ contains a primitive mth root of unity.

Lemma 2.4 ([30, Lemma 1.3.9])**.** *Let p be a prime number, then the finite field $\mathbb{F}_p$ has an element of order $m \geqslant 1$ if and only if $p = 1 \pmod m$.*

We impose some additional restrictions on the factors of q that incorporate the above lemma: each p is assumed to be prime, and $p - 1$ should be divisible by m. Let $\omega \in \mathbb{F}_p$ be a primitive mth root of unity, then obviously the cyclotomic polynomial factors over $\mathbb{F}_p$ as

$$\Phi_m(x) = \prod_{j \in \mathbb{Z}_m^*} (x - \omega^j).$$

Applying the Chinese remainder theorem a second time now leads to the aforementioned Double-CRT representation. Recall that $\mathcal{R}_p$ can be represented as $\mathbb{Z}_p[x]/(\Phi_m(x))$, so the isomorphism

$$\begin{aligned} \mathbb{Z}_p[x]/(\Phi_m(x)) &\to \mathbb{F}_p^n \\ \alpha(x) &\mapsto \{\alpha(\omega^j)\}_{j \in \mathbb{Z}_m^*} \end{aligned} \tag{2.1}$$

gives rise to a component-wise addition and multiplication algorithm. Also automorphisms can be computed easily on the right-hand side by permuting some entries.

The evaluation in the powers of ω from Equation 2.1 can be done with the number-theoretic transform. The input of the transform is a vector with entries x_i that are equal to the coefficients of $\alpha(x)$. The output is a vector with entries X_j that correspond to the evaluations of $\alpha(x)$ in ω^j. The indices i and j both range over $\{0, \ldots, m-1\}$. Since we only care about the primitive roots of unity, all entries of X_j with $\gcd(m, j) \neq 1$ can be discarded. More details are given in Chapter 6.

2.2 Somewhat homomorphic encryption

This section describes the Brakerski-Gentry-Vaikuntanathan (BGV) [4] and Fan-Vercauteren (FV) [10] homomorphic encryption schemes. Both are based on the R-LWE problem, and therefore they add noise to a ciphertext during encryption. Because the noise grows during homomorphic operations, a decryption error can occur when it becomes too big. Both schemes are therefore leveled somewhat homomorphic: they support only a limited number of additions and multiplications, but the parameters can be chosen such that this number becomes arbitrarily high.

Two additional functions are needed in the definition of the BGV and FV schemes. The function $\mathcal{D}_{\omega,q}$ decomposes a ring element into a radix $\omega \geq 2$. The function $\mathcal{P}_{\omega,q}$ does the opposite by including additional powers of ω into the result. These functions are defined with respect to the parameters ω, q and $\ell_{\omega,q} = \lceil \log_\omega(q) \rceil$. Both $\mathcal{D}_{\omega,q}$ and $\mathcal{P}_{\omega,q}$ are necessary to control the noise growth during relinearization [4, 30]. Let $\boldsymbol{a}' = [\boldsymbol{a}]_q$ for $\boldsymbol{a} \in \mathcal{R}$, then we define

$$\mathcal{D}_{\omega,q}(\boldsymbol{a}) \stackrel{\text{def}}{=} \left([\boldsymbol{a}']_\omega, \left\lfloor \left[\frac{\boldsymbol{a}'}{\omega} \right]_\omega \right\rceil, \ldots, \left\lfloor \left[\frac{\boldsymbol{a}'}{\omega^{\ell_{\omega,q}-1}} \right]_\omega \right\rceil \right)^\top$$

and

$$\mathcal{P}_{\omega,q}(\boldsymbol{a}) \stackrel{\text{def}}{=} \left([\boldsymbol{a}']_q, [\boldsymbol{a}' \cdot \omega]_q, \ldots, \left[\boldsymbol{a}' \cdot \omega^{\ell_{\omega,q}-1} \right]_q \right)^\top.$$

Lemma 2.5 ([4, Lemma 2])**.** *For all $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{R}$, it holds that*

$$\langle \mathcal{D}_{\omega,q}(\boldsymbol{a}), \mathcal{P}_{\omega,q}(\boldsymbol{b}) \rangle = \boldsymbol{a} \cdot \boldsymbol{b} \pmod{q}.$$

2.2.1 Brakerski-Gentry-Vaikuntanathan

This section specifies the Brakerski-Gentry-Vaikuntanathan scheme, which is a leveled homomorphic encryption scheme [4, 30]. It relies on a crucial *modulus switching* procedure for managing the noise. This technique allows us to convert a ciphertext ct encrypted under ciphertext modulus q into a ciphertext ct' encrypted under a smaller ciphertext modulus q'. Doing so results in a constant noise level, which avoids an exponential noise blow-up in circuits with large multiplicative depth.

The first thing to do is selecting a set of publicly known parameters: these parameters are the cyclotomic index m, the plaintext modulus $t \geqslant 2$, the ciphertext *modulus chain* $\{q_0, \ldots, q_L = q\}$, the radix $\omega \geqslant 2$ and the probability distributions χ_{key} and χ_{err}. We assume that the moduli in the chain are divisible, that they are all equal modulo t and that they are coprime to t. More formally, we want

$$q_i \mid q_{i+1}, \qquad q_j = q_k \pmod{t}, \qquad \gcd(q, t) = 1$$

for $i = 0, \ldots, L-1$ and $j, k = 0, \ldots, L$. Additionally, the distributions χ_{key} and χ_{err} are assumed to output elements of small norm with respect to q.

Key generation. The key generation procedure is specified in Algorithm 1. Except for the public parameters, there are no inputs to the key generation. The algorithm returns three things: the public key for encryption, the secret key for decryption and the relinearization key. This last key is necessary for reducing the size of a ciphertext after multiplication, which will be explained later. Remark that the public key is a slightly adapted R-LWE sample, so it is hard to infer the secret key from it.

Algorithm 1 Key generation BGV

procedure KEYGEN(χ_{key}, χ_{err}, t, q, ω)
- $\boldsymbol{s} \xleftarrow{\$} \chi_{\text{key}}$
- $\mathsf{sk} \leftarrow (1, \boldsymbol{s})$
- $(\boldsymbol{a}, \boldsymbol{e}) \xleftarrow{\$} \mathcal{U}_q \times \chi_{\text{err}}$
- $\mathsf{pk} \leftarrow \left([\boldsymbol{a} \cdot \boldsymbol{s} + t\boldsymbol{e}]_q, -\boldsymbol{a}\right)$
- $(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{e}}) \xleftarrow{\$} \mathcal{U}_q^{\ell_{\omega,q}} \times \chi_{\text{err}}^{\ell_{\omega,q}}$
- $\overrightarrow{\mathsf{rlk}} \leftarrow \left([\mathcal{P}_{\omega,q}(\boldsymbol{s}^2) + \overrightarrow{\boldsymbol{a}} \cdot \boldsymbol{s} + t\overrightarrow{\boldsymbol{e}}]_q, -\overrightarrow{\boldsymbol{a}}\right)$
- **return** $(\mathsf{sk}, \mathsf{pk}, \overrightarrow{\mathsf{rlk}})$

Encryption. The encryption procedure is specified in Algorithm 2. The inputs are the message and the public key. The result is a ciphertext that encrypts the message under the public key. Note that the BGV scheme encrypts messages from $\mathcal{R}_t$, so homomorphic addition and multiplication also happen in that ring. A ciphertext is a row vector of elements in $\mathcal{R}_{q_i}$, which for the moment contains only two parts. Both ciphertext parts are pseudorandom R-LWE samples of which the first one is used as an encryption mask.

Remark that both the message and the noise are added to the ciphertext, but the noise is first multiplied with a factor of t. The message is therefore located in the "lower bits" and the noise can grow in the "upper bits" of the ciphertext.

Algorithm 2 Encryption BGV

procedure ENC($\boldsymbol{m}$, pk, χ_{key}, χ_{err}, t, q_i)
- $\boldsymbol{u} \xleftarrow{\$} \chi_{\texttt{key}}$
- $(\boldsymbol{e}_0, \boldsymbol{e}_1) \xleftarrow{\$} \chi_{\texttt{err}}^2$
- $\mathsf{ct} \leftarrow \left([\boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + t\boldsymbol{e}_0]_{q_i}, [\boldsymbol{u} \cdot \boldsymbol{p}_1 + t\boldsymbol{e}_1]_{q_i}\right)$ ▷ With $\mathsf{pk} = (\boldsymbol{p}_0, \boldsymbol{p}_1)$
- **return** ct

Decryption. The decryption procedure is specified in Algorithm 3. The inputs are the ciphertext and the secret key. The result is the message encrypted by the ciphertext under the public key. The decryption function should return the correct message, even though the ciphertext contains noise.

Assume that decryption is done immediately after encryption. Recall that the encryption algorithm computes

$$\mathsf{ct} = ([\boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + t\boldsymbol{e}_0]_{q_i}, [\boldsymbol{u} \cdot \boldsymbol{p}_1 + t\boldsymbol{e}_1]_{q_i}).$$

The decryption algorithm evaluates

$$\begin{aligned} \boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} &= \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + t\boldsymbol{e}_0 + (\boldsymbol{u} \cdot \boldsymbol{p}_1 + t\boldsymbol{e}_1) \cdot \boldsymbol{s} \\ &= \boldsymbol{m} + t\boldsymbol{v} \pmod{q_i} \end{aligned}$$

with $\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{e} + \boldsymbol{e}_1 \cdot \boldsymbol{s} + \boldsymbol{e}_0$. The ring element $\boldsymbol{v}$ is called the noise of the ciphertext [30]. Decryption yields the correct result as long as $||\boldsymbol{v}||_\infty < \lfloor q_i/(2t) \rfloor$.

Remark. A BGV ciphertext is a row vector of elements in $\mathcal{R}_{q_i}$. Alternatively, it can be interpreted as a polynomial that is evaluated in the secret key during decryption. A ciphertext of length $k+1$ is therefore also called a degree-k ciphertext. This polynomial interpretation is also important for homomorphic addition and multiplication.

Algorithm 3 Decryption BGV

procedure DEC(ct, sk, t, q_i)
  $\boldsymbol{m}' \leftarrow [\langle \mathsf{sk}, \mathsf{ct} \rangle]_{q_i}$ ▷ $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \dots)$
  return $[\boldsymbol{m}']_t$

Addition. Homomorphic ciphertext addition is specified in Algorithm 4. The inputs are two different ciphertexts, and the output is a ciphertext that encrypts the sum of the underlying plaintexts. The plaintext sum is computed in the quotient ring $\mathcal{R}_t$.

Remark that homomorphic addition can be seen as the polynomial addition of two ciphertexts. Their evaluations in the secret key will therefore be added as well. Indeed, assume that the two input ciphertexts encrypt messages $\boldsymbol{m}$ and $\boldsymbol{m}'$ respectively, then they are evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + t\boldsymbol{v} \pmod{q_i} \quad \text{and} \quad \boldsymbol{w}' = \boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} = \boldsymbol{m}' + t\boldsymbol{v}' \pmod{q_i}.$$

The resulting ciphertext evaluates as

$$\boldsymbol{w}_{\text{add}} = \boldsymbol{w} + \boldsymbol{w}' = (\boldsymbol{m} + \boldsymbol{m}') + t(\boldsymbol{v} + \boldsymbol{v}') \pmod{q_i}.$$

The new noise is $\boldsymbol{v}_{\text{add}} \approx \boldsymbol{v} + \boldsymbol{v}'$. This formula is an approximation because possible overflows modulo t in $\boldsymbol{m} + \boldsymbol{m}'$ are ignored.

Algorithm 4 Addition BGV

procedure ADD(ct, ct', q_i)
  $\mathsf{ct}_{\text{add}} \leftarrow ([\boldsymbol{c}_0 + \boldsymbol{c}'_0]_{q_i}, [\boldsymbol{c}_1 + \boldsymbol{c}'_1]_{q_i})$
  return ct_{add}

Multiplication. Algorithm 5 specifies homomorphic ciphertext multiplication. The inputs are two different ciphertexts, and the output is a ciphertext that encrypts the product of the underlying plaintexts. The plaintext product is computed in the quotient ring $\mathcal{R}_t$. A side effect of multiplication is that the ciphertext degree increases to 2.

Remark that homomorphic multiplication can be seen as the polynomial multiplication of two ciphertexts. Their evaluations in the secret key will therefore be multiplied as well. Indeed, assume again that the input ciphertexts encrypt messages $\boldsymbol{m}$ and $\boldsymbol{m}'$ respectively, then they are evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + t\boldsymbol{v} \pmod{q_i} \quad \text{and} \quad \boldsymbol{w}' = \boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} = \boldsymbol{m}' + t\boldsymbol{v}' \pmod{q_i}.$$

The resulting ciphertext evaluates as

$$\boldsymbol{w}_{\text{mult}} = \boldsymbol{w} \cdot \boldsymbol{w}' = (\boldsymbol{m} \cdot \boldsymbol{m}') + t(\boldsymbol{v} \cdot \boldsymbol{m}' + \boldsymbol{v}' \cdot \boldsymbol{m} + t\boldsymbol{v} \cdot \boldsymbol{v}') \pmod{q_i}.$$

The new noise is

$$\boldsymbol{v}_{\text{mult}} \approx \boldsymbol{v} \cdot \boldsymbol{m}' + \boldsymbol{v}' \cdot \boldsymbol{m} + t\boldsymbol{v} \cdot \boldsymbol{v}', \tag{2.2}$$

which is again an approximation because possible overflows are ignored.

Algorithm 5 Multiplication BGV

procedure MULT(ct, ct', q_i)
  $\mathsf{ct}_{\text{mult}} \leftarrow \left([\boldsymbol{c}_0 \cdot \boldsymbol{c}'_0]_{q_i}, [\boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0]_{q_i}, [\boldsymbol{c}_1 \cdot \boldsymbol{c}'_1]_{q_i}\right)$
  return $\mathsf{ct}_{\text{mult}}$

Relinearization. One problem of homomorphic multiplication is that the resulting ciphertext is quadratic. This significantly increases the time and memory requirements and causes a faster noise growth when going to ciphertexts of even higher degree. Fortunately, it is possible to convert a quadratic ciphertext back into a linear one, using the relinearization procedure from Algorithm 6. The idea is to provide a masked version of $\boldsymbol{s}^2$ in the so-called relinearization key. The third ciphertext part is then broken into digits, which are multiplied by both parts of the relinearization key. In this way, the quadratic part is divided over the constant and linear parts. The result is a ciphertext encrypting the same plaintext as the input, but of lower degree.

Assume that the input ciphertext encrypts the message $\boldsymbol{m}$, then it is evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} + \boldsymbol{c}_2 \cdot \boldsymbol{s}^2 = \boldsymbol{m} + t\boldsymbol{v} \pmod{q_i}.$$

Using Lemma 2.5, the resulting ciphertext evaluates as

$$\begin{aligned}\boldsymbol{w}_{\text{relin}} &= (\boldsymbol{c}_0 + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_0}\rangle) + (\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_1}\rangle) \cdot \boldsymbol{s} \\ &= \left(\boldsymbol{c}_0 + \left\langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \mathcal{P}_{\omega,q_i}(\boldsymbol{s}^2) + \overrightarrow{\boldsymbol{a}} \cdot \boldsymbol{s} + t\overrightarrow{\boldsymbol{e}} \right\rangle\right) + (\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), -\overrightarrow{\boldsymbol{a}}\rangle) \cdot \boldsymbol{s} \\ &= \boldsymbol{m} + t\left(\boldsymbol{v} + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{e}}\rangle\right) \pmod{q_i}.\end{aligned}$$

Hence the new noise is equal to $\boldsymbol{v}_{\text{relin}} = \boldsymbol{v} + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{e}}\rangle$.

Algorithm 6 Relinearization BGV

procedure RELIN(ct, $\overrightarrow{\mathsf{rlk}}$, q_i, ω)
  $\mathsf{ct}' \leftarrow \left(\left[\boldsymbol{c}_0 + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_0}\rangle\right]_{q_i}, \left[\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_1}\rangle\right]_{q_i}\right)$ ▷ With $\overrightarrow{\mathsf{rlk}} = (\overrightarrow{\boldsymbol{r}_0}, \overrightarrow{\boldsymbol{r}_1})$
  return ct'

Modulus switching. A second problem of homomorphic multiplication is that the size of the noise grows exponentially as a function of the multiplicative depth. Indeed, the last term of Equation 2.2 shows that the new noise is mainly determined by the product of the original noise terms. When operating on ciphertexts of similar noise level, this leads to an effective doubling of the number of bits in the noise (as seen through the norm induced by the coefficient or canonical embedding). Ideally, the number of bits in the modulus-to-noise ratio should only decrease with a constant amount at each multiplicative level. This can be achieved with the modulus switching procedure from Algorithm 7: instead of having a variable noise, the noise level is kept constant, and the ciphertext modulus decreases one or several levels before performing a multiplication.

Assume that the input ciphertext encrypts the message $\boldsymbol{m}$, then it is evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \boldsymbol{m} + t\boldsymbol{v} \pmod{q_i}.$$

The updated ciphertext parts are equal to

$$\boldsymbol{c}_0' = \frac{q_j}{q_i}(\boldsymbol{c}_0 + t\boldsymbol{\delta}_0) \qquad \text{and} \qquad \boldsymbol{c}_1' = \frac{q_j}{q_i}(\boldsymbol{c}_1 + t\boldsymbol{\delta}_1),$$

where the division is exact due to the special shape of the δ-term. The resulting ciphertext evaluates as

$$\begin{aligned}\boldsymbol{w}_{\text{switch}} = \boldsymbol{c}_0' + \boldsymbol{c}_1' \cdot \boldsymbol{s} &= \frac{q_j}{q_i}(\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} + t(\boldsymbol{\delta}_0 + \boldsymbol{\delta}_1 \cdot \boldsymbol{s})) \\ &= \frac{q_j}{q_i}(\boldsymbol{m} + t(\boldsymbol{v} + \boldsymbol{\delta}_0 + \boldsymbol{\delta}_1 \cdot \boldsymbol{s})) \pmod{q_j}. \qquad (2.3)\end{aligned}$$

Remember that all moduli are equal modulo t, so we can retrieve the message as

$$[\boldsymbol{w}_{\text{switch}}]_{q_j} = \boldsymbol{m} \pmod{t}.$$

The new noise is $\boldsymbol{v}_{\text{switch}} \approx (q_j/q_i)(\boldsymbol{v} + \boldsymbol{\delta}_0 + \boldsymbol{\delta}_1 \cdot \boldsymbol{s})$. Note that this formula is an approximation because it ignores the fact that Equation 2.3 also scales the message by q_j/q_i.

Algorithm 7 Modulus switching BGV

procedure ModSwitch(ct, t, q_i, q_j)
$u \leftarrow t^{-1} \pmod{q_i/q_j}$
$\delta \leftarrow \left([-u\boldsymbol{c}_0]_{q_i/q_j}, [-u\boldsymbol{c}_1]_{q_i/q_j}\right)$
$\mathsf{ct}' \leftarrow \left(\left[\frac{q_j}{q_i}(\boldsymbol{c}_0 + t\boldsymbol{\delta}_0)\right]_{q_j}, \left[\frac{q_j}{q_i}(\boldsymbol{c}_1 + t\boldsymbol{\delta}_1)\right]_{q_j}\right)$
return ct'

2.2.2 Fan-Vercauteren

Brakerski [3] proposed an alternative way of controlling the noise in a ciphertext, which is different than modulus switching. The idea is to switch the locations of the message and the noise in a ciphertext: hide the message in the "upper bits" and the noise in the "lower bits". The result is a new category of so-called *scale-invariant* schemes, in which the homomorphic properties only depend on the ratio between the ciphertext modulus and the current noise. In particular, a homomorphic multiplication does not square the size of the noise, but multiplies it by a constant factor that depends on the scheme's parameters.

Fan and Vercauteren [10, 30] have designed a ring variant of Brakerski's scale-invariant scheme. This section specifies their scheme, including a modulus switching procedure, which was not present in the original definition. Although modulus switching is not necessary in scale-invariant schemes, it can still be applied and is useful for two other reasons. Firstly, a smaller modulus significantly reduces the execution time of operations in $\mathcal{R}_q$. Secondly, a smaller modulus also reduces the number of operations in the decryption circuit. Modulus switching is therefore an important aspect of bootstrapping.

The first thing to do is selecting a set of publicly known parameters: these parameters are the cyclotomic index m, the plaintext modulus $t \geqslant 2$, the ciphertext modulus q, the radix $\omega \geqslant 2$ and the probability distributions χ_{key} and χ_{err}. To simplify notation, the parameter $\Delta = \lfloor q/t \rfloor$ is introduced. We assume again that the distributions χ_{key} and χ_{err} output elements of small norm.

Key generation. The key generation procedure is specified in Algorithm 8. Except for the public parameters, there are no inputs to the key generation. The algorithm returns three things: the public key for encryption, the secret key for decryption and the relinearization key.

Algorithm 8 Key generation FV

procedure KEYGEN(χ_{key}, χ_{err}, q, ω)

$\boldsymbol{s} \xleftarrow{\$} \chi_{\text{key}}$

$\mathsf{sk} \leftarrow (1, \boldsymbol{s})$

$(\boldsymbol{a}, \boldsymbol{e}) \xleftarrow{\$} \mathcal{U}_q \times \chi_{\text{err}}$

$\mathsf{pk} \leftarrow \left([\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}]_q, -\boldsymbol{a}\right)$

$(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{e}}) \xleftarrow{\$} \mathcal{U}_q^{\ell_{\omega,q}} \times \chi_{\text{err}}^{\ell_{\omega,q}}$

$\overrightarrow{\mathsf{rlk}} \leftarrow \left([\mathcal{P}_{\omega,q}(\boldsymbol{s}^2) + \overrightarrow{\boldsymbol{a}} \cdot \boldsymbol{s} + \overrightarrow{\boldsymbol{e}}]_q, -\overrightarrow{\boldsymbol{a}}\right)$

return $(\mathsf{sk}, \mathsf{pk}, \overrightarrow{\mathsf{rlk}})$

Encryption. The encryption procedure is specified in Algorithm 9. The inputs are the message and the public key. The result is a ciphertext that encrypts the message under the public key. Note that the FV scheme encrypts messages from $\mathcal{R}_t$, so homomorphic addition and multiplication also happen in that ring. A ciphertext is a row vector of elements in $\mathcal{R}_q$, which for the moment contains only two parts.

Remark that the message is first scaled up with a factor of Δ before being added to the first part of the ciphertext. Indeed, it was mentioned earlier that the FV scheme hides the message in the upper bits of the ciphertext. The scaling factor gives the desired result, and the noise can now grow in the $\log_2(\Delta)$ lower bits of the ciphertext.

Algorithm 9 Encryption FV

procedure ENC($\boldsymbol{m}$, pk, χ_{key}, χ_{err}, t, q)

$\boldsymbol{u} \xleftarrow{\$} \chi_{\mathtt{key}}$

$(\boldsymbol{e}_0, \boldsymbol{e}_1) \xleftarrow{\$} \chi_{\mathtt{err}}^2$

$\mathsf{ct} \leftarrow \left([\Delta \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + \boldsymbol{e}_0]_q, [\boldsymbol{u} \cdot \boldsymbol{p}_1 + \boldsymbol{e}_1]_q\right)$ ▷ With $\mathsf{pk} = (\boldsymbol{p}_0, \boldsymbol{p}_1)$

return ct

Decryption. The decryption procedure is specified in Algorithm 10. The inputs are the ciphertext and the secret key. The result is the message encrypted by the ciphertext under the public key. The decryption function should return the correct message, even though the ciphertext contains noise.

Assume that decryption is done immediately after encryption. Recall that the encryption algorithm computes

$$\mathsf{ct} = ([\Delta \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + \boldsymbol{e}_0]_q, [\boldsymbol{u} \cdot \boldsymbol{p}_1 + \boldsymbol{e}_1]_q).$$

The decryption algorithm evaluates

$$\begin{aligned} \boldsymbol{w} &= \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{p}_0 + \boldsymbol{e}_0 + (\boldsymbol{u} \cdot \boldsymbol{p}_1 + \boldsymbol{e}_1) \cdot \boldsymbol{s} \\ &= \Delta \boldsymbol{m} + \boldsymbol{v} \pmod q \end{aligned}$$

with $\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{e} + \boldsymbol{e}_1 \cdot \boldsymbol{s} + \boldsymbol{e}_0$. The ring element $\boldsymbol{v}$ is called the noise of the ciphertext [30]. Decryption yields the correct result as long as $||\boldsymbol{v}||_\infty < \Delta/2$.

Algorithm 10 Decryption FV

procedure DEC(ct, sk, t, q)
  $\boldsymbol{m}' \leftarrow \left\lfloor \frac{t}{q} [\langle \mathsf{sk}, \mathsf{ct} \rangle]_q \right\rceil$ ▷ $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \dots)$
  return $[\boldsymbol{m}']_t$

Addition. Homomorphic ciphertext addition is specified in Algorithm 11. The inputs are two different ciphertexts, and the output is a ciphertext that encrypts the sum of the underlying plaintexts. The plaintext sum is computed in the quotient ring $\mathcal{R}_t$.

Remark that homomorphic addition can be seen as the polynomial addition of two ciphertexts. Their evaluations in the secret key will therefore be added as well. Indeed, assume that the two input ciphertexts encrypt messages $\boldsymbol{m}$ and $\boldsymbol{m}'$ respectively, then they are evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta \boldsymbol{m} + \boldsymbol{v} \pmod q \quad \text{and} \quad \boldsymbol{w}' = \boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} = \Delta \boldsymbol{m}' + \boldsymbol{v}' \pmod q.$$

The resulting ciphertext evaluates as

$$\boldsymbol{w}_{\text{add}} = \boldsymbol{w} + \boldsymbol{w}' = \Delta(\boldsymbol{m} + \boldsymbol{m}') + (\boldsymbol{v} + \boldsymbol{v}') \pmod q.$$

The new noise is $\boldsymbol{v}_{\text{add}} \approx \boldsymbol{v} + \boldsymbol{v}'$. This formula is an approximation because possible overflows modulo t in $\boldsymbol{m} + \boldsymbol{m}'$ are ignored.

Algorithm 11 Addition FV

procedure ADD(ct, ct', q)
  $\mathsf{ct}_{\text{add}} \leftarrow ([\boldsymbol{c}_0 + \boldsymbol{c}'_0]_q, [\boldsymbol{c}_1 + \boldsymbol{c}'_1]_q)$
  return ct_{add}

Multiplication. Algorithm 12 specifies homomorphic ciphertext multiplication. The inputs are two different ciphertexts, and the output is a ciphertext that encrypts the product of the underlying plaintexts. The plaintext product is computed in the quotient ring $\mathcal{R}_t$. A side effect of multiplication is that the ciphertext degree increases to 2.

Remark that homomorphic multiplication can be seen as the polynomial multiplication of two ciphertexts, followed by a scaling over t/q. Their evaluations in the secret key will therefore be multiplied and scaled as well. Indeed, assume again that the input ciphertexts encrypt messages $\boldsymbol{m}$ and $\boldsymbol{m}'$ respectively, then they are evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta\boldsymbol{m} + \boldsymbol{v} + q\boldsymbol{r} \quad \text{and} \quad \boldsymbol{w}' = \boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} = \Delta\boldsymbol{m}' + \boldsymbol{v}' + q\boldsymbol{r}'.$$

The resulting ciphertext evaluates as

$$\boldsymbol{w}_{\text{mult}} \approx \frac{t}{q}(\boldsymbol{w} \cdot \boldsymbol{w}') + q\boldsymbol{r}'' \approx \Delta(\boldsymbol{m} \cdot \boldsymbol{m}') + \frac{t}{q}(\boldsymbol{v} \cdot \boldsymbol{w}' + \boldsymbol{v}' \cdot \boldsymbol{w} - \boldsymbol{v} \cdot \boldsymbol{v}') + q\boldsymbol{r}'''.$$

The term $q\boldsymbol{r}'''$ cancels exactly modulo q, so the new noise is

$$\boldsymbol{v}_{\text{mult}} \approx (t/q)(\boldsymbol{v} \cdot \boldsymbol{w}' + \boldsymbol{v}' \cdot \boldsymbol{w} - \boldsymbol{v} \cdot \boldsymbol{v}'),$$

which is again an approximation because possible overflows and some other terms are ignored.

Algorithm 12 Multiplication FV

procedure MULT(ct, ct', t, q)
  $\mathsf{ct}_{\text{mult}} \leftarrow \left(\left[\left\lfloor \frac{t}{q}(\boldsymbol{c}_0 \cdot \boldsymbol{c}'_0) \right\rceil \right]_q, \left[\left\lfloor \frac{t}{q}(\boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0) \right\rceil \right]_q, \left[\left\lfloor \frac{t}{q}(\boldsymbol{c}_1 \cdot \boldsymbol{c}'_1) \right\rceil \right]_q \right)$
  return $\mathsf{ct}_{\text{mult}}$

Relinearization. The relinearization procedure is specified in Algorithm 13. The result is a ciphertext encrypting the same plaintext as the input, but of lower degree. Note that the algorithm is identical to the one from BGV.

Assume that the input ciphertext encrypts the message $\boldsymbol{m}$, then it is evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} + \boldsymbol{c}_2 \cdot \boldsymbol{s}^2 = \Delta\boldsymbol{m} + \boldsymbol{v} \pmod q.$$

Using Lemma 2.5, the resulting ciphertext evaluates as

$$\begin{aligned}
\boldsymbol{w}_{\text{relin}} &= (\boldsymbol{c}_0 + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \vec{\boldsymbol{r}}_0 \rangle) + (\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \vec{\boldsymbol{r}}_1 \rangle) \cdot \boldsymbol{s} \\
&= \left(\boldsymbol{c}_0 + \left\langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \mathcal{P}_{\omega,q}(\boldsymbol{s}^2) + \vec{\boldsymbol{a}} \cdot \boldsymbol{s} + \vec{\boldsymbol{e}} \right\rangle \right) + (\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), -\vec{\boldsymbol{a}} \rangle) \cdot \boldsymbol{s} \\
&= \Delta\boldsymbol{m} + \boldsymbol{v} + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \vec{\boldsymbol{e}} \rangle \pmod q.
\end{aligned}$$

Hence the new noise is equal to $\boldsymbol{v}_{\text{relin}} = \boldsymbol{v} + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \vec{\boldsymbol{e}} \rangle$.

Algorithm 13 Relinearization FV

procedure RELIN(ct, $\overrightarrow{\mathsf{rlk}}$, q, ω)
  $\mathsf{ct}' \leftarrow \left(\left[\boldsymbol{c}_0 + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_0} \rangle \right]_q, \left[\boldsymbol{c}_1 + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{r}_1} \rangle \right]_q \right)$ ▷ With $\overrightarrow{\mathsf{rlk}} = (\overrightarrow{\boldsymbol{r}_0}, \overrightarrow{\boldsymbol{r}_1})$
  return ct'

Modulus switching. Algorithm 14 specifies the modulus switching procedure. The result is a ciphertext encrypting the same plaintext as the input, but with respect to a smaller ciphertext modulus.

Assume that the input ciphertext encrypts the message $\boldsymbol{m}$, then it is evaluated in the secret key as

$$\boldsymbol{w} = \boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta \boldsymbol{m} + \boldsymbol{v} \pmod{q}.$$

The updated ciphertext parts are equal to

$$\boldsymbol{c}_0' = \frac{q'}{q}\boldsymbol{c}_0 + \boldsymbol{n}_0 \qquad \text{and} \qquad \boldsymbol{c}_1' = \frac{q'}{q}\boldsymbol{c}_1 + \boldsymbol{n}_1,$$

where the ring elements $\boldsymbol{n}_0$ and $\boldsymbol{n}_1$ are the rounding errors. The resulting ciphertext evaluates as

$$\begin{aligned} \boldsymbol{w}_{\text{switch}} = \boldsymbol{c}_0' + \boldsymbol{c}_1' \cdot \boldsymbol{s} &= \frac{q'}{q}(\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}) + (\boldsymbol{n}_0 + \boldsymbol{n}_1 \cdot \boldsymbol{s}) \\ &= \frac{\Delta q'}{q}\boldsymbol{m} + \frac{q'}{q}\boldsymbol{v} + \boldsymbol{n}_0 + \boldsymbol{n}_1 \cdot \boldsymbol{s} \pmod{q'}. \end{aligned}$$

The new noise is $\boldsymbol{v}_{\text{switch}} \approx (q'/q)\boldsymbol{v} + \boldsymbol{n}_0 + \boldsymbol{n}_1 \cdot \boldsymbol{s}$. Note that this formula is an approximation because Δ' might not exactly be equal to $\Delta q'/q$.

Remark. A relinearization key generated for ciphertext modulus q is also valid for any divisor of q. The relinearization key can therefore be reused when switching to a modulus that is a divisor of the original one.

Algorithm 14 Modulus switching FV

procedure MODSWITCH(ct, q, q')
  $\mathsf{ct}' \leftarrow \left(\left[\left\lfloor \frac{q'}{q}\boldsymbol{c}_0 \right\rceil \right]_{q'}, \left[\left\lfloor \frac{q'}{q}\boldsymbol{c}_1 \right\rceil \right]_{q'} \right)$
  return ct'

2.3 Noise analysis of BGV and FV

One important aspect of BGV and FV is the noise of a ciphertext. When it becomes too big, decryption returns an erroneous result, which is obviously not desired. It is therefore important to have a good estimate of the noise magnitude. For that reason, this section presents an analysis of the average noise growth, assuming that all ciphertexts have been encrypted independently.

Recall that we view $\boldsymbol{v} \in \mathcal{K}_m$ as a polynomial, i.e., $\boldsymbol{v} = v(x) \in \mathbb{Q}[x]$. In this section, elements of a cyclotomic number field are even considered as complex numbers, that is, $\boldsymbol{v} = v(\omega_m)$. In particular, we denote the modulus of a complex number by $|\boldsymbol{v}| = |v(\omega_m)|$. The noise analysis then computes the expected squared value $\mathbb{E}[|\boldsymbol{v}|^2]$. This is done for encryption, addition, multiplication, relinearization and modulus switching. It is always assumed that the inputs are independent and freshly encrypted ciphertexts. Note that the analysis actually corresponds to a noise estimate on the first entry of the canonical embedding. The noise estimate in `HElib` employs a similar strategy [14].

The distribution χ_{err} is assumed to have zero mean and variance equal to σ^2. It is also assumed to be symmetric, such that the probability of sampling an element $\boldsymbol{e}$ is equal to the probability of sampling $-\boldsymbol{e}$. The distribution χ_{key} is defined with respect to the Hamming weight h, and it samples uniformly from the set

$$\{\, \boldsymbol{a} \in \mathcal{R} \;\mid\; ||\boldsymbol{a}||_\infty \leqslant 1 \text{ and } ||\boldsymbol{a}||_1 = h \,\}.$$

In other words, all coefficients are sampled from $\{0, \pm 1\}$, and there are exactly h non-zero coefficients.

The expected value $\mathbb{E}[|\boldsymbol{v}|^2]$ is computed with respect to the sampling of all random variables, except for the secret key. The latter is assumed to be constant because it is the same in each operation and can therefore not be studied as an independent random variable. The analysis is not always exact, but all approximations are motivated. For example, it is assumed that the two ciphertext parts have zero mean and are uncorrelated. This approximation is reasonable given that R-LWE samples behave pseudorandom.

The following lemmas are required.

Lemma 2.6 ([14, Section 3.1.4])**.** *Sample* $\boldsymbol{u} \xleftarrow{\$} \chi_{key}$*, then* $\mathbb{E}[|\boldsymbol{u}|^2] = h$*.*

Proof. Consider

$$\boldsymbol{u} = \sum_{i=0}^{n-1} u_i x^i,$$

then the expected value of its squared modulus is equal to

$$\mathbb{E}[|\boldsymbol{u}|^2] = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \mathbb{E}[u_i u_j] \omega_m^{i-j}. \tag{2.4}$$

The correlation is given by

$$\mathbb{E}[u_i u_j] = \begin{cases} \frac{h}{n} & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

All cross-terms in Equation 2.4 cancel, which immediately proves the lemma. □

Lemma 2.7. *Sample* $\boldsymbol{a} \xleftarrow{\$} \mathcal{U}_N$ *for large* N*, then* $\mathbb{E}[|\boldsymbol{a}|^2] \approx n \cdot N^2/12$.

Proof. All coefficients of $\boldsymbol{a}$ are sampled independently and uniformly from $S = [-N/2, N/2) \cap \mathbb{Z}$. Let a_0 be such a coefficient, then we know that

$$\mathbb{E}[a_0^2] = \frac{1}{N} \sum_{i \in S} i^2 \qquad \text{or} \qquad \frac{\mathbb{E}[a_0^2]}{N^2} = \frac{1}{N} \sum_{i \in S} (\frac{i}{N})^2.$$

If N goes to infinity, then this is a Riemann sum of the function $f(y) = y^2$ over the interval $\left[-\frac{1}{2}, \frac{1}{2}\right]$. It converges to

$$\lim_{N \to \infty} \frac{\mathbb{E}[a_0^2]}{N^2} = \int_{-1/2}^{1/2} y^2 \, dy = \frac{1}{12}.$$

The expected squared value is therefore $\mathbb{E}[a_0^2] \approx N^2/12$. We get the desired result using a similar reasoning as in the previous proof. □

Lemma 2.8. *Consider the independent random variables* X_1 *and* X_2*. If at least one of them has zero mean, then*

$$\mathbb{E}[X_1 X_2] = 0.$$

Proof. Because the variables are independent, the expected value of their product is equal to the product of their expected values. Moreover, at least one of them has zero mean, which implies that

$$\mathbb{E}[X_1 X_2] = \mathbb{E}[X_1] \cdot \mathbb{E}[X_2] = 0.$$

□

Lemma 2.9. *If two ciphertexts* ct *and* ct' *are encrypted independently, then their noise terms are uncorrelated.*

Proof. Recall from the previous section that the noise after encryption is given by

$$\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{e} + \boldsymbol{e}_1 \cdot \boldsymbol{s} + \boldsymbol{e}_0. \tag{2.5}$$

Let $\boldsymbol{v}$ and $\boldsymbol{v}'$ be the noise in ct and ct' respectively. We must show that they are uncorrelated, i.e., $\mathbb{E}[\boldsymbol{v}^*\boldsymbol{v}'] = 0$, where the asterisk denotes the complex conjugate. Using Equation 2.5, the product $\boldsymbol{v}^*\boldsymbol{v}'$ can be written as a sum of nine cross-terms, each of which contains at least one factor that was sampled with zero mean and independent from the other factors. The proof is concluded by applying Lemma 2.8 to each of these cross-terms. □

Lemma 2.10. *Let $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ be a freshly encrypted FV ciphertext with respect to a sufficiently large ciphertext modulus. If $\boldsymbol{v}$ denotes the noise in ct, then $\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}$ and $\boldsymbol{v}$ are approximately uncorrelated.*

Proof. During encryption, the ciphertext is computed as

$$\mathsf{ct} = ([\Delta \boldsymbol{m} + \boldsymbol{u} \cdot (\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}) + \boldsymbol{e}_0]_q, [-\boldsymbol{u} \cdot \boldsymbol{a} + \boldsymbol{e}_1]_q).$$

Assuming that q is large, the terms $\boldsymbol{u} \cdot \boldsymbol{e} + \boldsymbol{e}_0$ and $\boldsymbol{e}_1$ are negligible compared to the other terms. The equation simplifies to

$$\mathsf{ct} \approx ([\Delta \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{a} \cdot \boldsymbol{s}]_q, [-\boldsymbol{u} \cdot \boldsymbol{a}]_q).$$

Now assume that $\boldsymbol{u}$ and $\boldsymbol{a}$ are fixed, and consider all triplets of the form $\mathfrak{e} = (\boldsymbol{e}_0, \boldsymbol{e}_1, \boldsymbol{e})$. Except for the all-zeros triplet, they can be partitioned into disjoint sets S_1 and S_2 of opposite values, i.e., $\forall \mathfrak{e} \in S_1 \colon -\mathfrak{e} \in S_2$. Define $S = S_1 \cup S_2$, and let $f(\mathfrak{e})$ be the probability mass of $\mathfrak{e}$. Recall that χ_{err} is assumed to be symmetric, which implies that f is an even function; that is, $f(-\mathfrak{e}) = f(\mathfrak{e})$. Define

$$\boldsymbol{r} = ([\Delta \boldsymbol{m} + \boldsymbol{u} \cdot \boldsymbol{a} \cdot \boldsymbol{s}]_q + [-\boldsymbol{u} \cdot \boldsymbol{a}]_q \cdot \boldsymbol{s})^*(\boldsymbol{u} \cdot \boldsymbol{e} + \boldsymbol{e}_1 \cdot \boldsymbol{s} + \boldsymbol{e}_0),$$

then the conditional covariance is equal to

$$\begin{aligned}\mathbb{E}[\boldsymbol{r} \mid \boldsymbol{u}, \boldsymbol{a}] &= \sum_{\mathfrak{e} \in S} \boldsymbol{r} f(\mathfrak{e}) = \sum_{\mathfrak{e} \in S_1} \boldsymbol{r} f(\mathfrak{e}) + \sum_{\mathfrak{e} \in S_2} \boldsymbol{r} f(\mathfrak{e}) = \sum_{\mathfrak{e} \in S_1} \boldsymbol{r} f(\mathfrak{e}) + \sum_{\mathfrak{e} \in -S_1} \boldsymbol{r} f(\mathfrak{e}) \\ &= \sum_{\mathfrak{e} \in S_1} \boldsymbol{r} f(\mathfrak{e}) + \sum_{\mathfrak{e} \in S_1} -\boldsymbol{r} f(-\mathfrak{e}) = 0.\end{aligned}$$

The covariance is equal to

$$\mathbb{E}[\boldsymbol{r}] = \mathbb{E}[\mathbb{E}[\boldsymbol{r} \mid \boldsymbol{u}, \boldsymbol{a}]] = \mathbb{E}[0] = 0,$$

which proves that they are uncorrelated. □

Encryption. The noise after encryption is given by Equation 2.5, both for BGV and FV ciphertexts. Writing out the product and using Lemma 2.8, the expected value of the noise is

$$\mathbb{E}[|\boldsymbol{v}|^2] = \sigma^2 \cdot (1 + h + |\boldsymbol{s}|^2).$$

Addition. Consider the ciphertexts ct and ct' with noise $\boldsymbol{v}$ and $\boldsymbol{v}'$ respectively. If the magnitude of the noise is much bigger than t, then the noise in the ciphertext sum is approximately equal to $\boldsymbol{v}_{\text{add}} \approx \boldsymbol{v} + \boldsymbol{v}'$. Using Lemma 2.9, the expected value of the noise is

$$\mathbb{E}[|\boldsymbol{v}_{\text{add}}|^2] \approx \mathbb{E}[|\boldsymbol{v}|^2] + \mathbb{E}[|\boldsymbol{v}'|^2].$$

Multiplication. Consider the ciphertexts ct and ct' with noise $\boldsymbol{v}$ and $\boldsymbol{v}'$ respectively. For BGV, the dominant noise term is

$$\boldsymbol{v}_{\text{mult}} \approx t\boldsymbol{v} \cdot \boldsymbol{v}'.$$

The expected value of the noise in the ciphertext product is therefore

$$\mathbb{E}[|\boldsymbol{v}_{\text{mult}}|^2] \approx t^2 \cdot \mathbb{E}[|\boldsymbol{v}|^2] \cdot \mathbb{E}[|\boldsymbol{v}'|^2].$$

For FV, the dominant noise term is

$$\boldsymbol{v}_{\text{mult}} \approx \frac{t}{q}[\boldsymbol{v} \cdot (\boldsymbol{c}_0' + \boldsymbol{c}_1' \cdot \boldsymbol{s}) + \boldsymbol{v}' \cdot (\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s})].$$

Define

$$\boldsymbol{w} = \boldsymbol{v} \cdot (\boldsymbol{c}_0' + \boldsymbol{c}_1' \cdot \boldsymbol{s}) \qquad \text{and} \qquad \boldsymbol{w}' = \boldsymbol{v}' \cdot (\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s}).$$

Using Lemmas 2.8 to 2.10 and the observation that R-LWE samples behave as uniformly random, the expected value of the noise is

$$\mathbb{E}[|\boldsymbol{v}_{\text{mult}}|^2] \approx \left(\frac{t}{q}\right)^2 \left(\mathbb{E}[|\boldsymbol{w}|^2] + \mathbb{E}[|\boldsymbol{w}'|^2]\right)$$

with

$$\mathbb{E}[|\boldsymbol{w}|^2] \approx \mathbb{E}[|\boldsymbol{v}|^2] \cdot (\mathbb{E}[|\boldsymbol{c}_0'|^2] + \mathbb{E}[|\boldsymbol{c}_1'|^2] \cdot |\boldsymbol{s}|^2)$$

and

$$\mathbb{E}[|\boldsymbol{w}'|^2] \approx \mathbb{E}[|\boldsymbol{v}'|^2] \cdot (\mathbb{E}[|\boldsymbol{c}_0|^2] + \mathbb{E}[|\boldsymbol{c}_1|^2] \cdot |\boldsymbol{s}|^2).$$

Using Lemma 2.7, we find that

$$\mathbb{E}[|\boldsymbol{c}_0|^2] = \mathbb{E}[|\boldsymbol{c}_0'|^2] = \mathbb{E}[|\boldsymbol{c}_1|^2] = \mathbb{E}[|\boldsymbol{c}_1'|^2] \approx n \cdot \frac{q^2}{12}.$$

Relinearization. The noise after relinearization is equal to

$$\boldsymbol{v}_{\text{relin}} = \boldsymbol{v} + \langle \mathcal{D}_{\omega,q}(\boldsymbol{c}_2), \overrightarrow{\boldsymbol{e}} \rangle.$$

Let

$$\mathcal{D}_{\omega,q}(\boldsymbol{c}_2) = \left(\boldsymbol{c}_{2,0}, \boldsymbol{c}_{2,1}, \ldots, \boldsymbol{c}_{2,\ell_{\omega,q}-1}\right)^\top \qquad \text{and} \qquad \overrightarrow{\boldsymbol{e}} = (\boldsymbol{e}_0, \ldots, \boldsymbol{e}_{\ell_{\omega,q}-1})^\top.$$

All entries of $\overrightarrow{\boldsymbol{e}}$ are uncorrelated, so the new noise is given by

$$\begin{aligned}\mathbb{E}[|\boldsymbol{v}_{\text{relin}}|^2] &= \mathbb{E}[|\boldsymbol{v}|^2] + \sigma^2 \cdot \sum_{i=0}^{\ell_{\omega,q}-1} \mathbb{E}[|\boldsymbol{c}_{2,i}|^2] \\ &\approx \mathbb{E}[|\boldsymbol{v}|^2] + \sigma^2 \cdot \ell_{\omega,q} \cdot n \cdot \frac{\omega^2}{12}.\end{aligned}$$

Modulus switching. Although the formulas for BGV and FV look quite different, their noise growth under modulus switching is the same. Assume that the modulus is switched to $q' \ll q$ and that $q' \mid q$. For both schemes, the new noise is equal to

$$\boldsymbol{v}_{\text{switch}} \approx \frac{q'}{q}\boldsymbol{v} + \boldsymbol{n}_0 + \boldsymbol{n}_1 \cdot \boldsymbol{s},$$

where $\boldsymbol{n}_0$ and $\boldsymbol{n}_1$ behave as random elements from $\mathcal{K}_m$ with coefficients sampled uniformly from the interval $\left[-\frac{1}{2}, \frac{1}{2}\right]$. Using a similar argument as in the proof of Lemma 2.10, the terms in the above sum are approximately uncorrelated. The expected value of the noise after modulus switching is then

$$\mathbb{E}[|\boldsymbol{v}_{\text{switch}}|^2] \approx \left(\frac{q'}{q}\right)^2 \cdot \mathbb{E}[|\boldsymbol{v}|^2] + \frac{n}{12} \cdot (1 + |\boldsymbol{s}|^2).$$

Remark. The noise analysis from this section can be extended to $\mathbb{E}[|\tau_j(\boldsymbol{v})|^2]$ with $\tau_j \in \text{Gal}(\mathcal{K}_m/\mathbb{Q})$. When all automorphisms are considered simultaneously, it corresponds to a noise analysis on the entries of the canonical embedding. The asymptotic noise growth is then determined by $\arg\max_{\tau_j} |\tau_j(\boldsymbol{s})|$.

Chapter 3

Bootstrapping for Fully Homomorphic Encryption

Gentry [11] was the first to propose a fully homomorphic encryption scheme. His solution starts from a somewhat homomorphic scheme and then relies on a technique called *bootstrapping* or *recryption* to make it fully homomorphic. This chapter describes Gentry's construction from a high level, focusing mostly on the bootstrapping step. We also give an example of a relatively simple bootstrapping algorithm for the FV scheme.

3.1 Constructing an FHE scheme

3.1.1 General construction

All current somewhat homomorphic encryption schemes are restricted by noise growth when performing operations. The noise imposes an upper bound on the circuit complexity that a scheme can evaluate, which prevents it from being fully homomorphic. This problem can be overcome by bootstrapping, which is a technique for reducing the noise and thereby "refreshing" the ciphertext. The important insight is that evaluating the decryption circuit on a ciphertext removes the noise completely. Of course, an untrusted party cannot be given the decryption key itself. It is possible, however, to give him an encryption of the decryption key such that he can evaluate the decryption circuit homomorphically. Gentry's original proposal also involved a method for simplifying the decryption function, called *squashing*. We mention it for completeness, although most newer schemes avoid it. In summary, the construction of a fully homomorphic encryption scheme is now three-fold:

- **Initial scheme.** Create a somewhat homomorphic encryption scheme. This can be, for example, the BGV or FV scheme from the previous chapter.
- **Squashing.** Simplify the decryption function as much as possible such that it can be evaluated with a circuit of lower complexity. The idea is to split the decryption algorithm in two phases of which the first one doesn't need the decryption key and can therefore be evaluated in the clear.

- **Bootstrapping.** Let the somewhat homomorphic encryption scheme evaluate its own (squashed) decryption circuit. This is only possible when we are in possession of an encryption of the decryption key, called the *bootstrapping key*. The outcome of bootstrapping is an encryption of the same message as the input, but with a noise level determined by the complexity of the decryption circuit.

From the above analysis, it follows immediately that a somewhat homomorphic scheme can be turned into a fully homomorphic one if it can evaluate its own decryption circuit plus at least one additional operation. The idea is to recrypt a ciphertext just before it breaks, i.e., before the noise gets too high to evaluate an addition or multiplication. If the complexity of the decryption circuit is low enough, then this recrypted ciphertext contains less noise then the input. This allows us to compute one additional operation, which is necessary to make progress through the circuit that we want to evaluate.

3.1.2 Recrypting a ciphertext

The most important step in Gentry's construction and also the main interest of this thesis is bootstrapping. We therefore give some additional elaboration on how it works. A schematic representation of bootstrapping is given in Figure 3.1. The top of the diagram shows that the decryption function recovers the message $\boldsymbol{m}$ from the ciphertext ct and the secret key sk. Recryption works by following the bottom part of the diagram, which is similar to the top part, but has to be evaluated homomorphically. First we add a second "layer" of encryption to the ciphertext that is almost noise-free. In theory, this can be done by running the encryption procedure, but in practice, one can often simplify this step. Then the somewhat homomorphic scheme evaluates its own decryption function to remove the noisy inner layer of encryption. This can be done with the bootstrapping key $\mathsf{Enc}(\mathsf{sk})$ without having to reveal the secret key itself. The result is a new ciphertext that encrypts the original message $\boldsymbol{m}$, but with a noise level determined by the complexity of the decryption circuit.

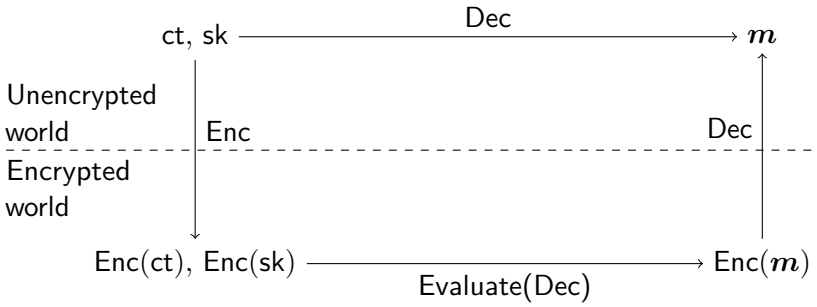

FIGURE 3.1: Bootstrapping diagram

3.2 Bootstrapping based on Boolean circuits

Next to the somewhat homomorphic FV scheme, Fan and Vercauteren [10] also proposed a bootstrapping algorithm to make it fully homomorphic. Their algorithm writes the decryption function as a binary circuit, which has to be implemented in the encrypted domain. This section describes their algorithm and some improvements. More details about the implementation are given in Chapter 6.

3.2.1 Overview of the algorithm

The bootstrapping algorithm of Fan and Vercauteren recrypts each plaintext coefficient separately. This is achieved by transforming the decryption function into a Boolean circuit. They also assume that the secret key is sampled with binary coefficients, i.e., they can only take values from $\{0, 1\}$. For the rest of this chapter, let the input ciphertext of the bootstrapping algorithm be denoted by $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$ and the underlying plaintext by

$$\boldsymbol{m} = \sum_{j=0}^{n-1} m_j x^j.$$

Bootstrapping a coefficient m_j of the plaintext is done as follows:

- **Switch to power-of-two modulus.** The decryption function is simplified with a scaling trick: the modulus is switched from q to 2^k with $k = \lfloor \log_2 q \rfloor$. Because the new modulus is a power of 2, we can write the decryption function as a binary circuit.
- **Ignore the bottom bits.** The decryption function is further simplified by ignoring the bottom bits of $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$. If not too many bits are ignored, decryption will still yield the correct result.
- **Binary addition.** In this step, the decryption function is rewritten as a binary addition circuit. A "matrix" of $n + 1$ binary numbers is assembled, and they are added using carry-save and ripple-carry adders.
- **Centered reduction.** Decryption requires to perform centered reduction of the result. Negative numbers are represented with a sign bit of '1' and positive number with a sign bit of '0'. This step is only necessary when the plaintext modulus is not a power of 2.
- **Second binary addition.** The next step is multiplying the result from centered reduction by t. This is done with a second addition circuit, based on the binary expansion of t.
- **Add rounding bit.** Finally, the rounding bit is defined as the first bit after the binary point. This bit is added to the rest of the result.

The above procedure is implemented by an untrusted party in the encrypted domain. This is only possible when that party is in possession of the bootstrapping key. Since the algorithm is based on a binary implementation of the decryption function, the bootstrapping key contains an encryption of the *bits* of the secret

key. If the secret key is equal to

$$\boldsymbol{s} = \sum_{i=0}^{n-1} s_i x^i,$$

then the untrusted party is given an encryption of the coefficients s_i.

3.2.2 Modifications and improvements

In this section, several modifications and improvements are applied to the algorithm of Fan and Vercauteren. A first improvement is obtained by combining step one and step two of the algorithm: the first step switches the modulus to a power of 2; the second step ignores the bottom bits of both ciphertext parts. These two operations are very similar and can therefore be replaced with only one step of modulus switching. Secondly, the improved algorithm also supports secret keys that are sampled from a ternary distribution. That is, the coefficients s_i can take values from $\{0, \pm 1\}$. The procedure for generating the bootstrapping key has to be modified accordingly. Thirdly, the algorithm is simplified by replacing rounding with flooring, i.e., using the equation $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$ for $x \in \mathbb{R}$. This formula is substituted into the decryption function. As a result, the addition of the rounding bit in the last step can be omitted. Finally, the adapted algorithm doesn't perform centered reduction. Instead, reduction is done into the set $\{0, \ldots, t-1\}$. This representation doesn't need a sign bit, which lowers the depth of bootstrapping. The steps of the modified algorithm are as follows:

Switch to power-of-two modulus. Switch to the smallest modulus $q' = 2^e$ such that the ciphertext still decrypts correctly. The new ciphertext is denoted by $\mathsf{ct}' = (\boldsymbol{c}'_0, \boldsymbol{c}'_1) = ([\lfloor q' \cdot \boldsymbol{c}_0 / q \rceil]_{q'}, [\lfloor q' \cdot \boldsymbol{c}_1 / q \rceil]_{q'})$.

Rewrite the decryption function. The decryption function is rewritten using flooring. This results in the formula

$$\boldsymbol{m} = \left[\left\lfloor \frac{t \cdot [\boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s}]_{q'}}{q'} \right\rceil \right]_t = \left[\left\lfloor \frac{t}{q'} \cdot \left[\boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} + \Gamma \left(\sum_{j=0}^{n-1} x^j \right) \right]_{q'} \right\rceil \right]_t$$

with $\Gamma = \lfloor q'/(2t) \rfloor$. The first step is to compute the sum modulo q':

$$\boldsymbol{w} = \left[\boldsymbol{c}'_0 + \boldsymbol{c}'_1 \cdot \boldsymbol{s} + \Gamma \left(\sum_{j=0}^{n-1} x^j \right) \right]_{q'} = \left[\boldsymbol{c}'_0 + \sum_{i=0}^{n-1} (s_i \cdot \boldsymbol{d}_i) + \Gamma \left(\sum_{j=0}^{n-1} x^j \right) \right]_{q'},$$

where $\boldsymbol{d}_i = \boldsymbol{c}'_1 \cdot x^i$. The algorithm evaluates this formula for each coefficient separately. Let w_j, $c'_{0,j}$ and $d_{i,j}$ be the coefficients of $\boldsymbol{w}$, $\boldsymbol{c}'_0$ and $\boldsymbol{d}_i$ respectively. Bootstrapping proceeds by implementing the sum

$$w_j = \left[c'_{0,j} + \sum_{i=0}^{n-1} (s_i \cdot d_{i,j}) + \Gamma \right]_{q'} \tag{3.1}$$

in the encrypted domain.

Binary addition. The simplified formula of Equation 3.1 is evaluated homomorphically using a binary addition circuit. A two-dimensional array of $n + 2$ rows containing the terms of the equation is constructed. The e columns of this array correspond to the bits of each term in its binary expansion. The numbers are added using carry-save adders [10]. Adding three bits x, y and z results in a sum

$$x \oplus y \oplus z$$

and a carry

$$\overline{\overline{x}\ \overline{y} \oplus \overline{y}\ \overline{z} \oplus \overline{x}\ \overline{z}}.$$

The NOT and XOR gates are implemented with a homomorphic addition; the AND gates use a homomorphic multiplication. It might happen that only two bits are left, and then the equations simplify to a ripple-carry adder. Note that carries beyond the size of the array can be ignored since the addition is performed modulo $q' = 2^e$.

The resulting noise heavily depends on the order in which the bits are added. A noise resistant algorithm performs the additions in an order such that each column always contains bits of similar noise magnitude. The implementation uses this insight as a heuristic: in each step, we combine the three (or two) bits of the column with lowest noise. This procedure is repeated until only one bit is left.

We now explain how the array of bits is constructed. Putting the terms $c'_{0,j}$ and Γ into the array is trivial. It is less clear how the terms $s_i \cdot d_{i,j}$ for $0 \leqslant i < n$ are taken care of, so we summarize it in Table 3.1. The table shows the bit inserted into the array based on $d_{i,j}$ and the corresponding s_i. When $s_i = 0$, a row of '0' bits must be inserted in the array. When $s_i = 1$, the binary expansion of $d_{i,j}$ must be inserted in the array. When $s_i = -1$, the two-complement of $d_{i,j}$ must be inserted in the array. The two-complement is computed by negating all bits and putting an additional '1' bit in the column of the least significant bits. Remark that this additional '1' bit turns out to be equal to the first row of the table. Exploiting this insight lowers the memory requirements for the bootstrapping key.

		s_i		
		-1	0	1
$d_{i,j}$	0	1	0	0
	1	0	0	1

TABLE 3.1: Array entries as a function of s_i and $d_{i,j}$

Second binary addition. The result from the previous step is multiplied by t. Let the digits of t be given by t_i for $i = 0, \ldots, \ell - 1$, then we compute

$$t \cdot w_j = \sum_{i=0}^{\ell-1} t_i \cdot (2^i w_j).$$

Multiplying by a power of 2 can be done with a shift, so the entire step is implemented by adding w_j to a shifted version of itself several times. The number of terms is equal to the Hamming weight of t's binary expansion. The output of bootstrapping is now located in the most significant bits of the sum.

The bootstrapping key must be adapted to support ternary sampled keys. In the adapted algorithm, it contains two entries per coefficient instead of only one: for each s_i, the bootstrapping key is augmented with an encryption of the bits a_i and b_i defined as

$$a_i = \begin{cases} 0 & \text{if } s_i = 0 \text{ or } s_i = 1 \\ 1 & \text{if } s_i = -1 \end{cases} \qquad b_i = \begin{cases} 0 & \text{if } s_i = -1 \text{ or } s_i = 0 \\ 1 & \text{if } s_i = 1. \end{cases}$$

Note that a_i corresponds to the first row and b_i to the second row of Table 3.1.

3.2.3 Multiplicative depth and time complexity

The noise after bootstrapping depends on the complexity of the decryption circuit. This is mainly determined by the two binary additions because they require homomorphic multiplications. The first addition operates on e-bit numbers, which leads to a multiplicative depth of $e - 1$. The depth of the second addition depends on the binary expansion of t: let $t = 2^x \cdot y$ with x and y non-negative integers and y odd, then it requires a multiplicative depth of $\lfloor \log_2 y \rfloor$. The total depth is therefore equal to $e + \lfloor \log_2 y \rfloor - 1$. Note that the noise growth resulting from homomorphic additions is neglected here.

The time complexity of bootstrapping is estimated based on the number of multiplications since they are much more expensive than additions. Remember that the array contains $n + 2$ numbers, each one consisting of e bits. Our approximate analysis is simplified by assuming n numbers instead of $n+2$. Additionally, the array also has n 1-bit numbers for computing the two-complement. Let the columns of the array be index by $i = 0, \ldots, e - 1$, where $i = 0$ corresponds to the least significant column. Define x_i as the number of bits that needs to be added in column i, then clearly we have that

$$x_{i+1} = \frac{x_i}{2} + n, \qquad x_0 = 2n.$$

The first term in the recurrence relation shows that there are $x_i/2$ bits originating from the carry-save adders in the previous column (the presence of possible ripple-carry adders is ignored). The second term comes from the n numbers that were originally present. The starting value $x_0 = 2n$ is due to the e-bit numbers as well as the 1-bit numbers in the least significant column. Solving the recurrence relation yields $x_i = 2n$ for all i. Since a carry-save adder requires three AND gates, the total number of homomorphic multiplications for recrypting one coefficient of a plaintext is equal to

$$\frac{3}{2}(x_0 + \ldots + x_{e-2}) = 3n(e - 1).$$

Recrypting all coefficients requires $3n^2(e - 1)$ multiplications.

3.2.4 Recombination of bits

A disadvantage of the algorithm from the previous section is that bootstrapping results in an encryption of bits. These bits are encrypted under a binary plaintext modulus, which is not necessarily the same as the original modulus t. This might not be desired for practical applications, and instead recryption should happen with respect to the original plaintext modulus. Fortunately, the problem can be fixed when t is a power of 2 using a technique similar to bit extraction [12].

Lemma 3.1. *Let* $f(y) \in \mathbb{Z}[y]$ *and* $d \in \mathbb{N}$*, then* $(2 \cdot f(y) + z)^{2^d} = z \pmod{2^{d+1}}$ *for each* $z \in \{0, 1\}$*.*

Proof. We prove the lemma by induction on d. The base case $d = 0$ is trivial, and we now assume that the lemma already holds for d. We get

$$\begin{aligned}(2 \cdot f(y) + z)^{2^{d+1}} &= ((2 \cdot f(y) + z)^{2^d})^2 \\ &= (2^{d+1} \cdot g(y) + z)^2 = z \pmod{2^{d+2}},\end{aligned}$$

where the existence of $g(y) \in \mathbb{Z}[y]$ is guaranteed by the induction hypothesis. The last equality follows by the binomial expansion and the fact that $z^2 = z$. □

Algorithm 15 specifies the recombination of ℓ bits into a single ciphertext. The trick is to repeatedly square the bit encryptions, which removes the "garbage polynomial" $f(y)$ from the upper bits. The input of the algorithm is a list of bit encryptions ordered from most to least significant bit. The output is an encryption of the message that the input bits represent, but with respect to the original plaintext modulus. Before recombination, the ciphertext ct_i has multiplicative depth $e - i - 1$. Recombination adds an additional depth of i multiplications, so the total depth of bootstrapping is equal to $e - 1$. Note that recombining the bits did not increase the multiplicative depth.

Algorithm 15 Recombination of bits

procedure RECOMB(ct_i, $\overrightarrow{\mathsf{rlk}}$, q, ω, ℓ)
  $\mathsf{ct}' \leftarrow (0, 0)$
  for $i \leftarrow 0$ to $\ell - 1$ **do**
    $\mathsf{ct}_i \leftarrow$ MODSWITCH(ct_i, q, $\lfloor q/2^i \rceil$) ▷ Move garbage into the upper bits
    for $j \leftarrow 1$ to i **do** ▷ Repeated squaring
      $\mathsf{ct}_i \leftarrow$ MULT(ct_i, ct_i, 2^{i+1}, q)
      $\mathsf{ct}_i \leftarrow$ RELIN(ct_i, $\overrightarrow{\mathsf{rlk}}$, q, ω)
    $\mathsf{ct}' \leftarrow$ ADD(ct', ct_i, q) ▷ Add all results together
  return ct'

Chapter 4

Homomorphic Linear Transformations

This chapter explains how the Chinese remainder theorem can be applied to encode multiple independent elements into one BGV or FV plaintext. This technique is called batching and makes it possible to perform homomorphic operations on elements of a small ring independently and in parallel. After introducing the concept of batching, we explain how the encoded elements can be permuted and how this gives rise to the concept of linear transformations. We also explain how these transformations can be computed efficiently in the homomorphic domain. Finally, we give an example of a linear transformation, the evaluation map, which is particularly useful for bootstrapping.

4.1 Plaintext slots and hypercube structure

4.1.1 Encoding elements in plaintext slots

Recall from Chapter 2 that BGV and FV operate in the plaintext ring $\mathcal{R}_t$. However, practical applications might not desire the algebraic structure of this ring, and we therefore need a technique to encode elements of a *different* ring into $\mathcal{R}_t$. One way to do it is by considering the subring $\mathbb{Z}_t \subseteq \mathcal{R}_t$ and thereby utilizing only the constant term of the plaintext. Although it is possible, this is quite wasteful because the higher degree coefficients are not employed in a useful manner. It would therefore be better to have a technique that encodes multiple elements of a small ring into one plaintext.

Smart and Vercauteren [29] have shown that it is indeed possible to encode multiple elements of a small ring into one plaintext and to perform operations component-wise. The plaintext can then be seen as encoding a "vector" of elements and the same homomorphic operation can be executed on the vector elements in parallel. Their method is called *batching*, and in fact, it corresponds to a single instruction, multiple data technique. It works for plaintext moduli t that are prime and coprime to the cyclotomic index m. Using the theory of p-adic fields, it has been

shown that the same result holds for any prime power as well [12]. We will therefore assume that the plaintext modulus is given by $t = p^r$, where p is a prime number that doesn't divide m. The following lemma is required.

Lemma 4.1 ([12, Section 4.2])**.** *Consider the positive integers p, r and $m > 1$, where p is a prime number that doesn't divide m. The mth cyclotomic polynomial over $\mathbb{Z}_{p^r}$ splits into ℓ distinct irreducible factors of degree d as*

$$\Phi_m(x) = F_1(x) \cdot \ldots \cdot F_\ell(x) \pmod{p^r}.$$

The parameter d is the multiplicative order of p modulo m, and the number of factors is $\ell = n/d$, where n is the degree of $\Phi_m(x)$.

Remember that the plaintext ring is $\mathcal{R}_{p^r} \cong \mathbb{Z}_{p^r}[x]/(\Phi_m(x))$. The Chinese remainder theorem allows us to write this ring as a direct sum of $\mathbb{Z}_{p^r}$-algebras through the isomorphism

$$\begin{aligned} \mathbb{Z}_{p^r}[x]/(\Phi_m(x)) &\to \bigoplus_{i=1}^{\ell} (\mathbb{Z}_{p^r}[x]/(F_i(x))) \\ \alpha(x) &\mapsto (\alpha(x) \bmod F_1(x), \ldots, \alpha(x) \bmod F_\ell(x)) \end{aligned} \tag{4.1}$$

that sends an element $\alpha(x)$ to its residue classes modulo the factors of $\Phi_m(x)$. The right-hand side of the isomorphism corresponds to a component-wise application of addition and multiplication, and the entries are called the *plaintext slots*. It can be proven that the rings $\mathbb{Z}_{p^r}[x]/(F_i(x))$ are all isomorphic [12], and we can therefore talk about a unique *slot algebra*. In order to see this, consider the Galois group of $\mathcal{K}_m/\mathbb{Q}$, which is given by the automorphisms

$$\begin{aligned} \tau_j \colon \mathbb{Q}[x]/(\Phi_m(x)) &\to \mathbb{Q}[x]/(\Phi_m(x)) \\ \alpha(x) &\mapsto \alpha(x^j). \end{aligned} \qquad (j \in \mathbb{Z}_m^*)$$

We need one special automorphism that we call the Frobenius map:

Definition 4.1. For a prime number p that doesn't divide m, we define the *Frobenius map* σ to be the automorphism given by

$$\alpha(x) \mapsto \alpha(x^p).$$

Remark. The Frobenius map is not exactly the same the Frobenius endomorphism that maps any element to its pth power. However, later we will show for $r = 1$ that the Frobenius map acts on each plaintext slot individually as the Frobenius endomorphism.

For the rest of this thesis, let ζ denote the residue class of x in the ring $\mathbb{Z}_{p^r}[x]/(F_1(x))$. Clearly ζ is a primitive mth root of unity, and we define the slot algebra [18, 17] as

$$E \stackrel{\text{def}}{=} \mathbb{Z}_{p^r}[\zeta].$$

The isomorphism $\mathbb{Z}_{p^r}[x]/(F_i(x)) \to E$ can be computed using the following lemma.

Lemma 4.2. *Consider the integers p, r and m as before, and let $S \subseteq \mathbb{Z}$ be a complete system of representatives for $\mathbb{Z}_m^*/\langle p \rangle$. Then for each $i \in \{1, \ldots, \ell\}$, there is a unique $h \in S$ such that the roots of $F_i(x)$ are given by the Frobenius powers of ζ^h, i.e.,*

$$F_i\left(\zeta^{h \cdot p^j}\right) = 0 \qquad \text{for } j = 0, \ldots, d-1.$$

In particular, the roots of $F_1(x)$ are given by ζ^{p^j}.

The above lemma can be proven using Galois field theory for the prime case and using the field of p-adic numbers for the prime-power case. An important insight is that the group $G = \mathrm{Gal}(\mathcal{K}_m/\mathbb{Q})$ acts transitively on the roots of $\Phi_m(x)$, i.e., on the set $X = \{\zeta^j\}_{j \in \mathbb{Z}_m^*}$. The subgroup $H = \langle \sigma \rangle \subseteq G$, which is generated by the Frobenius map, acts on the roots of each $F_i(x)$ individually. In fact, the lemma says that each $F_i(x)$ vanishes on exactly d primitive roots of unity that are related by taking Frobenius powers. Let $X_1, \ldots, X_\ell$ be the cosets of X containing the roots of $F_1(x), \ldots, F_\ell(x)$, then the quotient group $G/H \cong \mathbb{Z}_m^*/\langle p \rangle$ acts transitively on these cosets. The representative set S therefore links the roots of one factor to the roots of another one.

Using Lemma 4.2, we see that the isomorphism $\mathbb{Z}_{p^r}[x]/(F_i(x)) \to E$ is given by $\alpha(x) \mapsto \alpha(\zeta^h)$. This allows us to simplify Equation 4.1 by writing its domain as an ℓ-fold direct sum of the slot algebra:

$$\begin{aligned} \mathbb{Z}_{p^r}[x]/(\Phi_m(x)) &\to \bigoplus_{i=1}^{\ell} E \\ \alpha(x) &\mapsto \{\alpha(\zeta^h)\}_{h \in S}. \end{aligned} \tag{4.2}$$

The plaintext slots therefore contain the values $\alpha(\zeta^h)$, where h ranges over S.

4.1.2 Hypercube structure and permutations

For many algorithms including bootstrapping, it is necessary to permute the slots of a plaintext. Gentry et al. [13] have shown that implementing arbitrary permutations is possible and that they can be processed homomorphically. In this section, we give an overview of their method adopting the same point of view as `HElib` [18, 17].

We first explain that the plaintext slots are arranged in a *hypercube* structure. A visualization of a three-dimensional hypercube of size $3 \times 4 \times 3$ is given in Figure 4.1. Also one *hypercolumn* in the vertical dimension is highlighted. Each entry in the hypercube can be seen as storing one of the plaintext slots, and those are indexed by a multidimensional (hyper)index. A basic operation is permuting the plaintext slots circularly along one of the dimensions of the hypercube. These special permutations are called one-dimensional "rotations", and they are much more efficient to compute than arbitrary permutations. Moreover, it suffices for bootstrapping to support only these one-dimensional rotations, so we restrict our exposition to those as well.

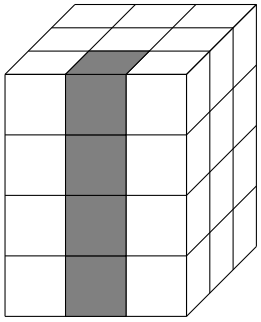

FIGURE 4.1: A hypercube and a hypercolumn, adapted from [18]

The hypercube relies on the set S from the previous section. Instead of choosing arbitrary representatives, we choose S in a different way, which enables us to impose a hypercube structure. Let

$$S = \{g_1^{e_1} \cdot \ldots \cdot g_t^{e_t} : 0 \leqslant e_i < \ell_i\}, \tag{4.3}$$

where t is the number of dimensions and ℓ_i is the size of dimension $i = 1, \ldots, t$. The number of plaintext slots is equal to $\ell = \ell_1 \cdot \ldots \cdot \ell_t$.

A simple way of computing the hypercube basis $\{g_1, \ldots, g_t\}$ is by iteration: first take $g_1 \in \mathbb{Z}$ and let ℓ_1 be its order in $\mathbb{Z}_m^*/\langle p \rangle$. The element should be taken different from the identity, i.e., $\ell_1 > 1$. Then take $g_2 \in \mathbb{Z}$ and let ℓ_2 be its order in $\mathbb{Z}_m^*/\langle p, g_1 \rangle$. Again, the element should be taken different from the identity, i.e., $\ell_2 > 1$. This procedure is repeated until $\mathbb{Z}_m^*/\langle p, g_1, \ldots, g_t \rangle$ is the trivial group containing only one element. It leads to a representative set S that is not very structured algebraically. Bootstrapping, however, needs a basis that depends on the factorization of the cyclotomic index m. Later in this chapter, we explain how it can be built differently.

Based on Equations 4.2 and 4.3, we can index the hypercube structure using tuples of the form $(e_1, \ldots, e_t)$ with $0 \leqslant e_i < \ell_i$. Let $h = g_1^{e_1} \cdot \ldots \cdot g_t^{e_t} \in S$, then we see that the hypercube contains the element $\alpha(\zeta^h)$ at the given index. Additionally, we can define one-dimensional *rotations* on the plaintext slots: a rotation with $0 \leqslant v < \ell_i$ positions in dimension i is indicated by ρ_i^v. It maps the plaintext slot at index $(e_1, \ldots, e_i, \ldots, e_t)$ to the plaintext slot at index $(e_1, \ldots, e_i', \ldots, e_t)$ with $e_i' = e_i + v \pmod{\ell_i}$. The rotations can be implemented using the following lemma.

Lemma 4.3. *Let $\boldsymbol{\mu}$ be the "mask" obtained by embedding the constant '0' in the plaintext slots with indices $(e_1, \ldots, e_i, \ldots, e_t)$, where $e_i < v$, and embedding the constant '1' in the other slots. Then for a plaintext $\boldsymbol{\alpha} \in \mathcal{R}_{p^r}$, the rotation ρ_i^v can be computed as*

$$\rho_i^v(\boldsymbol{\alpha}) = \boldsymbol{\mu} \cdot \tau_j(\boldsymbol{\alpha}) + (1 - \boldsymbol{\mu}) \cdot \tau_k(\boldsymbol{\alpha}),$$

where $j = g_i^{-v} \pmod{m}$ and $k = g_i^{\ell_i - v} \pmod{m}$.

Proof. We consider a ring element via its polynomial representation, i.e., $\boldsymbol{\alpha} = \alpha(x)$. Furthermore, let

$$\boldsymbol{\alpha}' = \boldsymbol{\mu} \cdot \tau_j(\boldsymbol{\alpha}) + (1 - \boldsymbol{\mu}) \cdot \tau_k(\boldsymbol{\alpha}).$$

Assume that $e'_i < v$, and consider $h = g_1^{e_1} \cdot \ldots \cdot g_i^{e'_i} \cdot \ldots \cdot g_t^{e_t}$. We know that the equality $\alpha'(\zeta^h) = \alpha(\zeta^{h \cdot k})$ holds, where

$$h \cdot k = g_1^{e_1} \cdot \ldots \cdot g_i^{e'_i + \ell_i - v} \cdot \ldots \cdot g_t^{e_t} \pmod{m}.$$

Define $e_i = e'_i + \ell_i - v$, then it follows that $0 \leqslant e_i < \ell_i$ and $e'_i = e_i + v \pmod{\ell_i}$. We conclude that the plaintext $\boldsymbol{\alpha}$ at index $(e_1, \ldots, e_i, \ldots e_t)$ contains the same value as $\boldsymbol{\alpha}'$ at index $(e_1, \ldots, e'_i, \ldots e_t)$. A similar reasoning can be given when $e'_i \geqslant v$. □

Lemma 4.3 allows us to implement a rotation using two constant-ciphertext multiplications and two automorphisms. Sometimes, it is even possible to perform a rotation using one automorphism only. We define a "good" hypercube dimension as one for which the order of g_i in $\mathbb{Z}_m^*$ is equal to ℓ_i; else the hypercube dimension is called "bad". In a good dimension, we have that $j = k \pmod{m}$, and it follows that $\tau_j = \tau_k$. The formula for a rotation in a good dimension can therefore be simplified to $\rho_i^v(\boldsymbol{\alpha}) = \tau_j(\boldsymbol{\alpha})$, which obviously needs only one automorphism instead of two.

In order to perform the rotations, we need to compute the automorphisms homomorphically. Consider the FV ciphertext $\mathsf{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$, which is evaluated in the secret key during decryption:

$$\boldsymbol{c}_0 + \boldsymbol{c}_1 \cdot \boldsymbol{s} = \Delta \boldsymbol{m} + \boldsymbol{v} \pmod{q}.$$

Applying an automorphism to both sides of the equation yields

$$\tau_j(\boldsymbol{c}_0) + \tau_j(\boldsymbol{c}_1) \cdot \tau_j(\boldsymbol{s}) = \Delta \tau_j(\boldsymbol{m}) + \tau_j(\boldsymbol{v}) \pmod{q}.$$

The ciphertext $\mathsf{ct}' = (\tau_j(\boldsymbol{c}_0), \tau_j(\boldsymbol{c}_1))$ encrypts $\tau_j(\boldsymbol{m})$, but under a different secret key $\tau_j(\boldsymbol{s})$. We can turn ct' into a ciphertext encrypting $\tau_j(\boldsymbol{m})$ under the original key with a *key switching* procedure that is very similar to relinearization. First we need to possess an *evaluation key*, which is the counterpart of the relinearization key. Then the second ciphertext part is broken into digits, which are multiplied by both parts of the evaluation key. Algorithms 16, 17 and 18 show the procedures for generating such an evaluation key and for computing an automorphism homomorphically. The last algorithm is identical for BGV and FV.

Algorithm 16 Evaluation key generation BGV

procedure EvkGen($\boldsymbol{s}$, τ_j, χ_{err}, t, q, ω)
  $(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{e}}) \xleftarrow{\$} \mathcal{U}_q^{\ell_{\omega,q}} \times \chi_{\text{err}}^{\ell_{\omega,q}}$
  $\overrightarrow{\mathsf{evk}} \leftarrow \left(\left[\mathcal{P}_{\omega,q}(\tau_j(\boldsymbol{s})) + \overrightarrow{\boldsymbol{a}} \cdot \boldsymbol{s} + t\overrightarrow{\boldsymbol{e}} \right]_q, -\overrightarrow{\boldsymbol{a}} \right)$
  return $\overrightarrow{\mathsf{evk}}$

Algorithm 17 Evaluation key generation FV

procedure EvkGen($\boldsymbol{s}$, τ_j, χ_{err}, q, ω)
  $(\overrightarrow{\boldsymbol{a}}, \overrightarrow{\boldsymbol{e}}) \xleftarrow{\$} \mathcal{U}_q^{\ell_{\omega,q}} \times \chi_{\text{err}}^{\ell_{\omega,q}}$
  $\overrightarrow{\mathsf{evk}} \leftarrow \left(\left[\mathcal{P}_{\omega,q}(\tau_j(\boldsymbol{s})) + \overrightarrow{\boldsymbol{a}} \cdot \boldsymbol{s} + \overrightarrow{\boldsymbol{e}} \right]_q, -\overrightarrow{\boldsymbol{a}} \right)$
  return $\overrightarrow{\mathsf{evk}}$

Algorithm 18 Automorphism and key switching BGV/FV

procedure Auto(ct, τ_j, $\overrightarrow{\mathsf{evk}}$, q_i, ω) ▷ Set $q_i \leftarrow q$ for FV
  $\mathsf{ct}' \leftarrow (\tau_j(\boldsymbol{c}_0), \tau_j(\boldsymbol{c}_1))$
  $\mathsf{ct}'' \leftarrow \left(\left[\boldsymbol{c}'_0 + \left\langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}'_1), \overrightarrow{\boldsymbol{k}_0} \right\rangle \right]_{q_i}, \left[\left\langle \mathcal{D}_{\omega,q_i}(\boldsymbol{c}'_1), \overrightarrow{\boldsymbol{k}_1} \right\rangle \right]_{q_i} \right)$ ▷ With $\overrightarrow{\mathsf{evk}} = (\overrightarrow{\boldsymbol{k}_0}, \overrightarrow{\boldsymbol{k}_1})$
  return ct''

4.2 One-dimensional linear transformations

Linear transformations are an important aspect of the bootstrapping algorithms for BGV and FV [19, 6]. This section gives an overview of the linear transformations that are useful for bootstrapping, as also implemented by `HElib` [17].

We first need to define what a linear transformation is exactly. Using the isomorphism of Equation 4.2, we see that the ring of plaintext elements is actually a module over the slot algebra E. The slot algebra itself is a module over $\mathbb{Z}_{p^r}$. The concept of linear transformations is then defined in the usual way:

Definition 4.2. A function $L\colon \mathcal{R}_{p^r} \to \mathcal{R}_{p^r}$ is *E-linear* if $L(\boldsymbol{\alpha} + \boldsymbol{\alpha}') = L(\boldsymbol{\alpha}) + L(\boldsymbol{\alpha}')$ and $L(c \cdot \boldsymbol{\alpha}) = c \cdot L(\boldsymbol{\alpha})$ for all $\boldsymbol{\alpha}, \boldsymbol{\alpha}' \in \mathcal{R}_{p^r}$ and $c \in E$.

Definition 4.3. A function $L\colon \mathcal{R}_{p^r} \to \mathcal{R}_{p^r}$ is *$\mathbb{Z}_{p^r}$-linear* if $L(\boldsymbol{\alpha} + \boldsymbol{\alpha}') = L(\boldsymbol{\alpha}) + L(\boldsymbol{\alpha}')$ and $L(c \cdot \boldsymbol{\alpha}) = c \cdot L(\boldsymbol{\alpha})$ for all $\boldsymbol{\alpha}, \boldsymbol{\alpha}' \in \mathcal{R}_{p^r}$ and $c \in \mathbb{Z}_{p^r}$.

We restrict ourselves to the case of one-dimensional linear transformations. A linear transformation is called one-dimensional if it can be decomposed as a series of linear transformations that act on the hypercolumns in a certain dimension separately. Figure 4.2 shows a schematic representation of a one-dimensional linear transformation. The $2 \times 4 \times 2$ hypercube is reinterpreted as a series of hypercolumns in the vertical dimension. Each hypercolumn is then interpreted as a vector (either an E-vector or a $\mathbb{Z}_{p^r}$-vector), and a matrix multiplication is done for each of the vectors separately. For example, the input-output relation for an E-linear transformation of the front right hypercolumn is given by the matrix-vector product

$$\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix} = M \begin{bmatrix} \eta_1 \\ \eta_2 \\ \eta_3 \\ \eta_4 \end{bmatrix}$$

with $M \in E^{4\times 4}$. The same equation holds for all other hypercolumns in the vertical dimension, where the matrix M can either be the same of different.

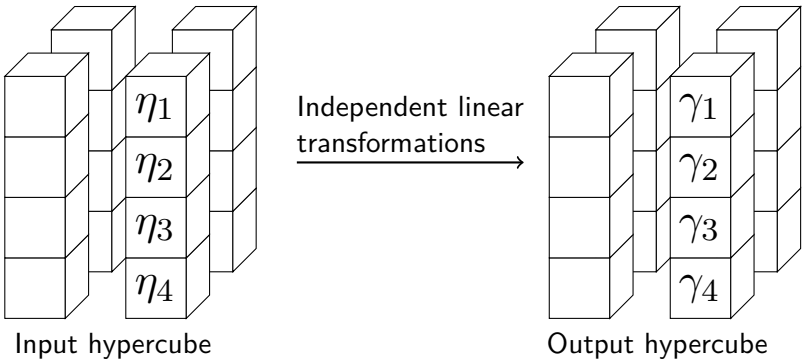

FIGURE 4.2: A one-dimensional linear transformation

4.2.1 E-linear transformations

As mentioned in the previous section, a one-dimensional E-linear transformation can be computed by multiplying each hypercolumn with a matrix $M \in E^{\ell_i \times \ell_i}$. Consider

$$M = \begin{bmatrix} m_{1,1} & \dots & m_{1,\ell_i} \\ \vdots & \ddots & \vdots \\ m_{\ell_i,1} & \dots & m_{\ell_i,\ell_i} \end{bmatrix}, \tag{4.4}$$

and for a vector $\alpha \in E^{\ell_i}$, let α_v be the vector obtained by rotating the entries of α by v positions. Using some algebraic manipulations, it was established by Halevi and Shoup [15] that we can rewrite the matrix-vector product as

$$M\alpha = \sum_{v=0}^{\ell_i - 1} M_v \alpha_v \tag{4.5}$$

with

$$M_v = \begin{bmatrix} m_{1,1-v} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & m_{\ell_i,\ell_i-v} \end{bmatrix}.$$

The second index is implicitly reduced modulo ℓ_i, so $m_{j,k} = m_{j,k+\ell_i}$ by definition.

Now instead of a vector, consider an entire plaintext $\boldsymbol{\alpha} \in \mathcal{R}_{p^r}$. Using the above theory, any one-dimensional E-linear transformation can be written as

$$L(\boldsymbol{\alpha}) = \sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}_v \cdot \rho_i^v(\boldsymbol{\alpha}), \tag{4.6}$$

where the constants $\boldsymbol{\kappa}_v$ are obtained by embedding the diagonal entries of M_v in the slots. A naive approach can implement Equation 4.6 with ℓ_i constant-ciphertext multiplications and $\ell_i - 1$ rotations. However, Halevi and Shoup [17] have proposed an algorithm that is cheaper. They rewrite the transformation as

$$L(\boldsymbol{\alpha}) = \sum_{k=0}^{h-1}\sum_{j=0}^{g-1} \boldsymbol{\kappa}_{j+gk} \cdot \Big(\boldsymbol{\mu}_{j+gk} \cdot \tau^{j+gk}(\boldsymbol{\alpha}) + (1-\boldsymbol{\mu}_{j+gk}) \cdot \tau^{j+gk-\ell_i}(\boldsymbol{\alpha})\Big)$$
$$= \sum_{k=0}^{h-1} \tau^{gk}\left[\sum_{j=0}^{g-1}\Big(\boldsymbol{\kappa}'_{j+gk} \cdot \tau^{j}(\boldsymbol{\alpha}) + \boldsymbol{\kappa}''_{j+gk} \cdot \tau^{j-\ell_i}(\boldsymbol{\alpha})\Big)\right],$$

where the masks are as in Lemma 4.3. We use $\tau = \tau_{h_i}$ for $h_i = g_i^{-1} \pmod m$, and the parameters are chosen as $g = \lceil\sqrt{\ell_i}\rceil$ and $h = \lceil \ell_i/g \rceil$. The adapted constants are

$$\boldsymbol{\kappa}'_{j+gk} = \tau^{-gk}(\boldsymbol{\mu}_{j+gk} \cdot \boldsymbol{\kappa}_{j+gk}) \qquad \text{and} \qquad \boldsymbol{\kappa}''_{j+gk} = \tau^{-gk}((1-\boldsymbol{\mu}_{j+gk}) \cdot \boldsymbol{\kappa}_{j+gk}).$$

In case of a good hypercube dimension, the above formula reduces to

$$L(\boldsymbol{\alpha}) = \sum_{k=0}^{h-1} \tau^{gk}\left[\sum_{j=0}^{g-1} \boldsymbol{\kappa}'''_{j+gk} \cdot \tau^{j}(\boldsymbol{\alpha})\right] \qquad \text{with} \qquad \boldsymbol{\kappa}'''_{j+gk} = \tau^{-gk}(\boldsymbol{\kappa}_{j+gk}).$$

Algorithm 19 shows the procedures for good and bad hypercube dimensions. The constants $\boldsymbol{\kappa}'_v$, $\boldsymbol{\kappa}''_v$ and $\boldsymbol{\kappa}'''_v$ are only non-zero in the range $0 \leqslant v < \ell_i$, and all counters are implicitly initialized to 0. The algorithm requires ℓ_i constant-ciphertext multiplications and $2\sqrt{\ell_i} + \mathcal{O}(1)$ automorphisms in a good dimension; in a bad dimension, it requires $2\ell_i$ constant-ciphertext multiplications and $3\sqrt{\ell_i} + \mathcal{O}(1)$ automorphisms.

Algorithm 19 E-linear transformations

procedure MatMulGood($\boldsymbol{\alpha}$, $\boldsymbol{\kappa}'''_v$, ℓ_i, g_i)
  for $j \leftarrow 0$ to $g-1$ **do**
    $\boldsymbol{\alpha}_j \leftarrow \tau^j(\boldsymbol{\alpha})$
  for $k \leftarrow 0$ to $h-1$ **do**
    for $j \leftarrow 0$ to $g-1$ **do**
      $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_k + \boldsymbol{\kappa}'''_{j+gk} \cdot \boldsymbol{\alpha}_j$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \tau^{gk}(\boldsymbol{w}_k)$
  return $\boldsymbol{w}$

procedure MatMulBad($\boldsymbol{\alpha}$, $\boldsymbol{\kappa}'_v$, $\boldsymbol{\kappa}''_v$, ℓ_i, g_i)
  $\boldsymbol{\alpha}' \leftarrow \tau^{-\ell_i}(\boldsymbol{\alpha})$
  for $j \leftarrow 0$ to $g-1$ **do**
    $\boldsymbol{\alpha}_j \leftarrow \tau^j(\boldsymbol{\alpha})$
    $\boldsymbol{\alpha}'_j \leftarrow \tau^j(\boldsymbol{\alpha}')$
  for $k \leftarrow 0$ to $h-1$ **do**
    for $j \leftarrow 0$ to $g-1$ **do**
      $\boldsymbol{w}_k \leftarrow \boldsymbol{w}_k + \boldsymbol{\kappa}'_{j+gk} \cdot \boldsymbol{\alpha}_j + \boldsymbol{\kappa}''_{j+gk} \cdot \boldsymbol{\alpha}'_j$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \tau^{gk}(\boldsymbol{w}_k)$
  return $\boldsymbol{w}$

4.2.2 Z-linear transformations

A one-dimensional $\mathbb{Z}_{p^r}$-linear transformation can be computed using roughly the same strategy as for an E-linear one. The rest of this section explains how it can be done based on the following lemmas.

Lemma 4.4. *The Frobenius map $\sigma = \tau_p$ acts on each plaintext slot individually as the slot-wise Frobenius map $\sigma_E\colon \beta(\zeta) \mapsto \beta(\zeta^p)$ for any $\beta(x) \in \mathbb{Z}_{p^r}[x]$. In the special case when $r = 1$, the slot-wise Frobenius map corresponds to the Frobenius endomorphism.*

Proof. Consider a plaintext $\boldsymbol{\alpha} = \alpha(x)$ and its image under the Frobenius map $\boldsymbol{\alpha}' = \sigma(\boldsymbol{\alpha})$. For $h \in S$, the plaintext slot of $\boldsymbol{\alpha}'$ at index h is given by $\alpha'(\zeta^h) = \alpha(\zeta^{h \cdot p}) = \sigma_E(\alpha(\zeta^h))$. When $r = 1$, the ring $\mathcal{R}_p$ has prime characteristic p, so the Frobenius map corresponds to the pth power map. □

Lemma 4.5. *The Frobenius map is a $\mathbb{Z}_{p^r}$-linear function from E to E. Conversely, any $\mathbb{Z}_{p^r}$-linear function $L\colon E \to E$ can be written as a linear combination of Frobenius powers, i.e.,*

$$L(\eta) = \sum_{f=0}^{d-1} \theta_f \cdot \sigma_E^f(\eta)$$

for some constants θ_f. When the image of L lies in $\mathbb{Z}_{p^r}$, the constants are related as $\theta_f = \sigma_E^f(\theta_0)$.

The above constants θ_f can be obtained with field theory for the prime case [28] and can then be Hensel lifted to the prime-power case [19].

Equations 4.4 and 4.5 remain valid for $\mathbb{Z}_{p^r}$-linear transformations, but now the matrix entries $m_{j,k}$ are themselves $\mathbb{Z}_{p^r}$-linear maps from E to E. Based on this insight and the previous two lemmas, any one-dimensional $\mathbb{Z}_{p^r}$-linear transformation can be written as

$$L(\boldsymbol{\alpha}) = \sum_{v=0}^{\ell_i - 1} \sum_{f=0}^{d-1} \boldsymbol{\kappa}_{v,f} \cdot \sigma^f(\rho_i^v(\boldsymbol{\alpha})), \tag{4.7}$$

where the constants $\boldsymbol{\kappa}_{v,f}$ are obtained by embedding appropriate E-values in the slots. Those values can be found by considering the diagonal entries of M_v and solving a system of linear equations to determine the θ_f's from Lemma 4.5. A naive approach can implement Equation 4.7 with $d \cdot \ell_i$ constant-ciphertext multiplications, $\ell_i - 1$ rotations and $(d-1) \cdot \ell_i$ Frobenius powers. However, Halevi and Shoup [17] have proposed an algorithm that is cheaper. They rewrite the transformation as

$$\begin{aligned} L(\boldsymbol{\alpha}) &= \sum_{f=0}^{d-1} \sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}_{v,f} \cdot \sigma^f\left(\boldsymbol{\mu}_v \cdot \tau^v(\boldsymbol{\alpha}) + (1 - \boldsymbol{\mu}_v) \cdot \tau^{v-\ell_i}(\boldsymbol{\alpha})\right) \\ &= \sum_{f=0}^{d-1} \sigma^f\left[\sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}'_{v,f} \cdot \tau^v(\boldsymbol{\alpha})\right] + \tau^{-\ell_i}\left(\sum_{f=0}^{d-1} \sigma^f\left[\sum_{v=0}^{\ell_i - 1} \boldsymbol{\kappa}''_{v,f} \cdot \tau^v(\boldsymbol{\alpha})\right]\right), \end{aligned}$$

where the masks are as in Lemma 4.3. Again, we use $\tau = \tau_{h_i}$ for $h_i = g_i^{-1} \pmod m$, and the adapted constants are

$$\boldsymbol{\kappa}'_{v,f} = \boldsymbol{\mu}_v \cdot \sigma^{-f}(\boldsymbol{\kappa}_{v,f}) \qquad \text{and} \qquad \boldsymbol{\kappa}''_{v,f} = \tau^{\ell_i}\left((1-\boldsymbol{\mu}_v)\cdot\sigma^{-f}(\boldsymbol{\kappa}_{v,f})\right).$$

In case of a good hypercube dimension, the above formula reduces to

$$L(\boldsymbol{\alpha}) = \sum_{f=0}^{d-1} \sigma^f \left[\sum_{v=0}^{\ell_i-1} \boldsymbol{\kappa}'''_{v,f} \cdot \tau^v(\boldsymbol{\alpha})\right] \qquad \text{with} \qquad \boldsymbol{\kappa}'''_{v,f} = \sigma^{-f}(\boldsymbol{\kappa}_{v,f}).$$

Algorithm 20 shows the procedures for good and bad hypercube dimensions, where again all counters are implicitly initialized to 0. The algorithm is very similar to the one for E-linear transformations. Again, the sum is split in an inner and outer loop with the purpose of minimizing the number of automorphism operations. The algorithm requires $d \cdot \ell_i$ constant-ciphertext multiplications and $d + \ell_i - 2$ automorphisms in a good dimension; in a bad dimension, it requires $2d \cdot \ell_i$ constant-ciphertext multiplications and $2d + \ell_i - 3$ automorphisms.

Algorithm 20 $\mathbb{Z}_{p^r}$-linear transformations

procedure BlockMatMulGood($\boldsymbol{\alpha}$, $\boldsymbol{\kappa}'''_{v,f}$, d, ℓ_i, g_i)
  for $v \leftarrow 0$ to $\ell_i - 1$ **do**
    $\boldsymbol{\alpha}_v \leftarrow \tau^v(\boldsymbol{\alpha})$
  for $f \leftarrow 0$ to $d-1$ **do**
    for $v \leftarrow 0$ to $\ell_i - 1$ **do**
      $\boldsymbol{w}_f \leftarrow \boldsymbol{w}_f + \boldsymbol{\kappa}'''_{v,f} \cdot \boldsymbol{\alpha}_v$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \sigma^f(\boldsymbol{w}_f)$
  return $\boldsymbol{w}$

procedure BlockMatMulBad($\boldsymbol{\alpha}$, $\boldsymbol{\kappa}'_{v,f}$, $\boldsymbol{\kappa}''_{v,f}$, d, ℓ_i, g_i)
  for $v \leftarrow 0$ to $\ell_i - 1$ **do**
    $\boldsymbol{\alpha}_v \leftarrow \tau^v(\boldsymbol{\alpha})$
    for $f \leftarrow 0$ to $d-1$ **do**
      $\boldsymbol{u}_f \leftarrow \boldsymbol{u}_f + \boldsymbol{\kappa}'_{v,f} \cdot \boldsymbol{\alpha}_v$
      $\boldsymbol{u}'_f \leftarrow \boldsymbol{u}'_f + \boldsymbol{\kappa}''_{v,f} \cdot \boldsymbol{\alpha}_v$
  for $f \leftarrow 0$ to $d-1$ **do**
    $\boldsymbol{u} \leftarrow \boldsymbol{u} + \sigma^f(\boldsymbol{u}_f)$
    $\boldsymbol{u}' \leftarrow \boldsymbol{u}' + \sigma^f(\boldsymbol{u}'_f)$
  $\boldsymbol{w} \leftarrow \boldsymbol{u} + \tau^{-\ell_i}(\boldsymbol{u}')$
  return $\boldsymbol{w}$

4.3 The evaluation map

This section discusses the *evaluation map*, which is a linear transformation designed by Halevi and Shoup [16, 19] for `HElib`. The evaluation map moves the plaintext slots of some element $\beta(x)$ into the coefficients of some element $\alpha(x)$. That is, if the slots of $\beta(x)$ encode $c_1, \ldots, c_n$, then the output of the evaluation map is the element $\alpha(x)$ of which the coefficients are precisely these numbers. The evaluation map crucially relies on the powerful basis representation of the ring $\mathcal{R}$, introduced by Lyubashevsky et al. [21] and worked out by Alperin-Sheriff and Peikert [1]. We will therefore first introduce the powerful basis before explaining the evaluation map itself.

4.3.1 The powerful basis

Throughout this section, consider a pairwise coprime factorization of the cyclotomic index $m = m_1 \cdot \ldots \cdot m_t$. An interesting result in algebraic number theory is that the mth cyclotomic number field is isomorphic to the tensor product of smaller cyclotomics, i.e.,

$$\mathcal{K}_m \cong \bigotimes_{i=1}^{t} \mathcal{K}_{m_i} = \mathbb{Q}(\zeta_{m_1}, \ldots, \zeta_{m_t}),$$

where the field isomorphism is given by $\zeta_{m_i} \mapsto \zeta_m^{m/m_i}$. We restrict our analysis to the ring of integers $\mathcal{R}$, and we identify its elements as polynomials in the ring $\mathbb{Z}[x]/(\Phi_m(x))$. The tensorial decomposition thereby reduces to

$$\begin{aligned} \mathbb{Z}[x_1, \ldots, x_t]/(\Phi_{m_1}(x_1), \ldots, \Phi_{m_t}(x_t)) &\to \mathbb{Z}[x]/(\Phi_m(x)) \\ f(x_1, \ldots, x_t) &\mapsto f(x^{m/m_1}, \ldots, x^{m/m_t}). \end{aligned} \tag{4.8}$$

When studying elements of $\mathcal{R}$ as univariate polynomials, it is common to represent them in the *power basis* consisting of $x^0, x^1, \ldots, x^{\varphi(m)-1}$. However, a more "natural" basis appears when studying this ring via the above tensor product: the *powerful basis* consists of $x_1^{j_1} \cdot \ldots \cdot x_t^{j_t}$ with $0 \leqslant j_i < \varphi(m_i)$ and $i = 1, \ldots, t$. Note that the power basis and the powerful basis do not coincide when considering the isomorphism in Equation 4.8.

4.3.2 Building the hypercube structure

Recall from Section 4.2 that the linear transformations rely on the representative set S for $\mathbb{Z}_m^*/\langle p \rangle$. Earlier in Section 4.1.2, a simple procedure was described for building such a set. However, in order to compute the evaluation map efficiently, it is necessary that S has a special structure corresponding to the factorization $m = m_1 \cdot \ldots \cdot m_t$. Consider the following lemma, where we define $\mathrm{CRT}(h_1, \ldots, h_t)$ to be the unique $h \in \{0, \ldots, m-1\}$ such that $h = h_i \pmod{m_i}$ for all $i \in \{1, \ldots, t\}$.

Lemma 4.6 ([19, Lemma 4.1])**.** *Consider the integers p and $m = m_1 \cdot \ldots \cdot m_t$ as before. Let d_i be the order of $p^{d_1 \cdot \ldots \cdot d_{i-1}}$ modulo m_i. Then the order of p modulo m is $d = d_1 \cdot \ldots \cdot d_t$, which is equal to the degree of the factors in Lemma 4.1.*

Moreover, suppose that $S_1, \ldots, S_t \subseteq \mathbb{Z}$ are such that each S_i forms a complete system of representatives for $\mathbb{Z}^*_{m_i}/\langle p^{d_1 \cdot \ldots \cdot d_{i-1}} \rangle$. Then the set $S = \mathrm{CRT}(S_1, \ldots, S_t)$ forms a complete system of representatives for $\mathbb{Z}^*_m/\langle p \rangle$.

In order to simplify the evaluation map, `HElib` places some additional constraints on the choice of the sets S_i and we incorporate them as well [19]. Firstly, we assume that each group $\mathbb{Z}^*_{m_i}/\langle p^{d_1 \cdot \ldots \cdot d_{i-1}} \rangle$ is cyclic of order $\ell_i = \varphi(m_i)/d_i$ and with generator $\tilde{g}_i$. The set S can then be built using Equation 4.3 by setting $g_i = \mathrm{CRT}(1, \ldots, 1, \tilde{g}_i, 1, \ldots, 1)$. This first assumption allows us to decompose the evaluation map as a series of one-dimensional linear transformations, namely one for each factor of m. Secondly, we assume that $d_1 = d$ and $d_i = 1$ for $i = 2, \ldots, t$. This second assumption will result in a $\mathbb{Z}_{p^r}$-linear transformation in the first dimension and an E-linear transformation in the other dimensions. With these assumptions, the first dimension of the hypercube can be good or bad, but the other dimensions are always good.

In the rest of this chapter, we denote the powerful basis representation of some element $\alpha(x)$ by $\alpha'(x_1, \ldots, x_t)$. For $i = 1, \ldots, t$, we define

$$\zeta_i \stackrel{\text{def}}{=} \zeta^{m/m_i} \qquad \text{and} \qquad \zeta_{i,e} \stackrel{\text{def}}{=} \zeta_i^{g_i^e}.$$

It follows from the definition of the powerful basis representation that for $h = \mathrm{CRT}(h_1, \ldots, h_t)$ with $h_i \in \mathbb{Z}$, we have that $\alpha(\zeta^h) = \alpha'(\zeta_1^{h_1}, \ldots, \zeta_t^{h_t})$. Recall that the plaintext slots of $\alpha(x)$ contain the values $\alpha(\zeta^h)$, where h ranges over S. Equivalently, using Lemma 4.6, the plaintext slots hold $\alpha'(\zeta_1^{h_1}, \ldots, \zeta_t^{h_t})$, where the h_i's range over S_i, or even $\alpha'(\zeta_{1,e_1}, \ldots, \zeta_{t,e_t})$, where the e_i's range over $\{0, \ldots, \ell_i - 1\}$.

4.3.3 Moving slots to coefficients

This section explains how the evaluation map can be decomposed as a series of one-dimensional linear transformations. Suppose that we start with the plaintext $\beta(x)$ and end with the plaintext $\alpha(x)$. Let the powerful basis coefficients of $\alpha(x)$ be given by $c_{j_1,\ldots,j_t}$, i.e.,

$$\alpha'(x_1, \ldots, x_t) = \sum_{j_1,\ldots,j_t} c_{j_1,\ldots,j_t} \, x_1^{j_1} \cdot \ldots \cdot x_t^{j_t}, \tag{4.9}$$

then the plaintext slots of $\beta(x)$ encode these coefficients in some way. Specifically, we assume that the coefficients are encoded using a normal element $\theta \in E$, such that the slot with index $(e_1, \ldots, e_t)$ contains

$$\beta'(\zeta_{1,e_1}, \ldots, \zeta_{t,e_t}) = \sum_{f=0}^{d-1} c_{e_1+\ell_1 \cdot f, e_2, \ldots, e_t} \, \sigma_E^f(\theta).$$

The evaluation map consists of t stages, and we denote the output of stage i by $\beta_i(x)$. The map is such that the slot of $\beta_i(x)$ with index $(e_1, \ldots, e_t)$ contains

$$\beta'_i(\zeta_{1,e_1}, \ldots, \zeta_{t,e_t}) = \sum_{j_1,\ldots,j_i} c_{j_1,\ldots,j_i,e_{i+1},\ldots,e_t} \, \zeta_{1,e_1}^{j_1} \cdot \ldots \cdot \zeta_{i,e_i}^{j_i}. \tag{4.10}$$

Stage 1. The first stage takes the plaintext $\beta(x)$ as input and transforms it into $\beta_1(x)$. Consider the slot of $\beta_1(x)$ at index $(e_1, \ldots, e_t)$, and observe by the above formulas that it is a function of the slots of $\beta(x)$ at indices $(e'_1, e_2, \ldots, e_t)$ with $0 \leqslant e'_1 < \ell_1$. In other words, the output slot only depends on the input slots of the corresponding hypercolumn in dimension 1, so the transformation is one-dimensional. It is easy to see that the transformation is also $\mathbb{Z}_{p^r}$-linear, and therefore it can be implemented using the theory of Section 4.2.2.

It was noticed by Halevi and Shoup that the above $\mathbb{Z}_{p^r}$-linear transformation can be interpreted as a series of multi-point evaluations: each hypercolumn in the first dimension is interpreted as a polynomial of degree $d \cdot \ell_1$ with coefficients in $\mathbb{Z}_{p^r}$ and then converted to an evaluation of that polynomial in ℓ_1 distinct points. It is given by the matrix M of Equation 4.4 with entries

$$m_{j+1,k+1} \colon \sum_{f=0}^{d-1} c_f \, \sigma_E^f(\theta) \mapsto \sum_{f=0}^{d-1} c_f \, \zeta_{1,j}^{k+\ell_1 \cdot f} \tag{4.11}$$

for $c_f \in \mathbb{Z}_{p^r}$. The matrix is identical for each hypercolumn.

Stages 2, …, *t*. The ith stage takes the plaintext $\beta_{i-1}(x)$, i.e., the output from the previous stage, and transforms it into $\beta_i(x)$. Consider the slot of $\beta_i(x)$ at index $(e_1, \ldots, e_t)$, and observe that it is a function of the slots of $\beta_{i-1}(x)$ at indices $(e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_t)$ with $0 \leqslant e'_i < \ell_i$. In other words, the output slot only depends on the input slots of the corresponding hypercolumn in dimension i, so the transformation is one-dimensional. Is is easy to see that the transformation is also E-linear, and therefore it can be implemented using the theory of Section 4.2.1.

It was noticed by Halevi and Shoup that the above E-linear transformation can be interpreted as a series of multi-point evaluations: each hypercolumn in the ith dimension is interpreted as a polynomial of degree ℓ_i with coefficients in E and then converted to an evaluation of that polynomial in ℓ_i distinct points. It is given by the matrix M of Equation 4.4 with entries $m_{j+1,k+1} = \zeta_{i,j}^k$. The matrix is identical for each hypercolumn.

The correctness of the above procedure can be shown by proving that $\beta_t(x)$ is equal to $\alpha(x)$. It is clear from Equation 4.10 that the former plaintext contains the value

$$\beta'_t(\zeta_{1,e_1}, \ldots, \zeta_{t,e_t}) = \sum_{j_1,\ldots,j_t} c_{j_1,\ldots,j_t} \, \zeta_{1,e_1}^{j_1} \cdot \ldots \cdot \zeta_{t,e_t}^{j_t}$$

in the slot with index $(e_1, \ldots, e_t)$. Evaluating Equation 4.9 in all ζ_{i,e_i} shows that the latter plaintext contains exactly the same value in each slot. The plaintexts are therefore equal to each other.

Chapter 5

Bootstrapping based on Digit Extraction

This chapter introduces a bootstrapping algorithm that is entirely different than the one from Chapter 3. Here we study a technique called *digit extraction*, which homomorphically removes the lower digits of a plaintext, or alternatively, extracts the upper digits. Bootstrapping based on digit extraction can be traced back to Gentry, Halevi and Smart [12] who originally proposed the concept for power-of-two moduli. Halevi and Shoup [19] have generalized their procedure to the prime-power case.

The first section describes what digit extraction means and how it can be implemented for the BGV and FV schemes. The second section explains bootstrapping for "fully packed" plaintext slots, and the third section explains bootstrapping for "sparsely packed" plaintext slots.

5.1 Digit extraction

Digit extraction is a procedure to remove (i.e., set to zero) the v least significant digits of a number $w \in \mathbb{Z}_{p^e}$ with $v < e$. In order to evaluate the procedure homomorphically, it must be written as a series of polynomial evaluations, additions, subtractions, multiplications and divisions. We explain two algorithms for digit extraction: the first one is based on the procedure of Halevi and Shoup [19], and the second one is developed by Chen and Han [6]. The Chen/Han procedure requires a lower multiplicative depth, especially when $e - v \gg 1$.

We always refer to the non-negative digits of a number $w \in \mathbb{Z}_{p^e}$. They are denoted by $w_i \in \{0, \ldots, p-1\}$, where we have that

$$w = \sum_{i=0}^{e-1} w_i p^i.$$

Moreover, we write $w_{i,j}$ for any integer of which the least significant digit is w_i, and the next j digits are all zeros. More formally, it holds that $w_{i,j} = w_i \pmod{p^{j+1}}$. In

addition, we sometimes consider multiple digits of a number at the same time. This is done using the notation

$$w\langle j, \ldots, k\rangle \stackrel{\text{def}}{=} \sum_{i=k}^{j} w_i p^{i-k},$$

which extends coefficient-wise to elements of $\mathcal{R}$. Note that we always refer to the powerful basis coefficients since they are the ones that we will apply digit extraction to. With this notation in mind, the output of digit extraction is $w\langle e-1, \ldots, v\rangle$ when given $w \in \mathbb{Z}_{p^e}$.

5.1.1 Halevi/Shoup digit extraction

The digit extraction procedure of Halevi and Shoup relies on the following lemma.

Lemma 5.1 ([19, Corollary 5.5])**.** *For every prime p and exponent $e \geqslant 1$, there exists a polynomial $F_e(y) \in \mathbb{Z}[y]$ of degree p such that for all integers $1 \leqslant e' \leqslant e$, $0 \leqslant w_0 < p$ and w_1, it holds that $F_e(w_0 + p^{e'} w_1) = w_0 \pmod{p^{e'+1}}$.*

The polynomial $F_e(y)$ is called the *lifting polynomial*, and the idea is to apply it repeatedly in order to extract the least significant digit. Then we subtract the obtained digit from the input, and these steps are repeated until enough digits are removed. Algorithm 21 gives the Halevi/Shoup-based digit extraction procedure. Its cost is dominated by $ev - v(v+1)/2$ evaluations of the lifting polynomial. Note that the value of $w_{i,0}$ at the end of the algorithm is actually not consistent with its definition when $i > v$.

Algorithm 21 Halevi/Shoup-based digit extraction

```
procedure DigitExtract(w, p, e, v)
    for i ← 0 to e − 1 do
        w_{i,0} ← w
    for i ← 0 to v − 1 do
        for j ← 0 to e − i − 2 do
            w_{i,j+1} ← F_{j+1}(w_{i,j})
            w_{i+j+1,0} ← (w_{i+j+1,0} − w_{i,j+1})/p
    return w_{e−1,0}
```

5.1.2 Chen/Han digit extraction

The digit extraction procedure of Chen and Han needs one more polynomial next to the lifting polynomial. Consider the following lemma.

Lemma 5.2 ([6, Section 3.2])**.** *For every prime p and exponent $e \geqslant 1$, there exists a polynomial $G_e(y) \in \mathbb{Z}[y]$ of degree at most $(e-1)(p-1)+1$ such that for all integers $0 \leqslant w_0 < p$ and w_1, it holds that $G_e(w_0 + p w_1) = w_0 \pmod{p^e}$.*

The polynomial $G_e(y)$ is called the *lowest digit retain polynomial*. It can be applied directly to the input instead of repeatedly applying the lifting polynomial. As such, we only need one polynomial evaluation to extract the least significant digit of a number. Algorithm 22 gives the Chen/Han digit extraction procedure. Its cost is dominated by v evaluations of the lowest digit retain polynomial and $v(v-1)/2$ evaluations of the lifting polynomial. Note that the lifting polynomial is still used, but it is only necessary to keep the multiplicative depth as low as possible.

Algorithm 22 Chen/Han digit extraction

procedure DIGITEXTRACT(w, p, e, v)
  $z \leftarrow w$
  for $i \leftarrow 0$ to $v-1$ **do**
    $w_{i,0} \leftarrow w$
  for $i \leftarrow 0$ to $v-1$ **do**
    $z \leftarrow (z - G_{e-i}(w_{i,0}))/p$
    for $j \leftarrow 0$ to $v-i-2$ **do**
      $w_{i,j+1} \leftarrow F_{j+1}(w_{i,j})$
      $w_{i+j+1,0} \leftarrow (w_{i+j+1,0} - w_{i,j+1})/p$
  return z

5.1.3 Homomorphic digit extraction

Evaluating the digit extraction procedure for BGV and FV requires to implement all steps homomorphically. The only operations in both algorithms are subtraction, polynomial evaluation and division-by-p. Subtraction can be done trivially by negating all ciphertext parts of the second ciphertext. For polynomial evaluation, we can use for example the Paterson-Stockmeyer algorithm because it requires less non-constant multiplications than a naive implementation [24, 6]. The division-by-p operation is less trivial, and it is implemented differently for BGV and FV. For BGV with plaintext modulus p^r and ciphertext modulus q, we multiply all ciphertext parts by $p^{-1} \pmod q$. It can easily be checked that this has the desired result of dividing the plaintext by p [19]. A side effect is that also the plaintext modulus gets divided by p, i.e., it decreases to p^{r-1}. This does not cause any problems for the bootstrapping algorithm in this chapter; as explained later, it is even desired. For the FV scheme, the ciphertext can just be reinterpreted as having a plaintext modulus of p^{r-1}, and thereby the division-by-p operation comes for free. Again, it can be checked that this has the desired result of dividing the plaintext by p [6]. Note that in both cases only exact division by p it possible, so it only works if the plaintext is indeed divisible by p.

When the digit extraction procedure is evaluated homomorphically, an additional important aspect is the multiplicative depth. The depth of Halevi/Shoup digit extraction is $e-1$ evaluations of the lifting polynomial. For Chen/Han, the depth is $v-1$ evaluations of the lifting polynomial and one evaluation of the lowest digit retain polynomial. The degree of this lowest digit retain polynomial is at

most $(e-v)(p-1)+1$. The depth of polynomial evaluation depends on the algorithm, but is usually around $\lceil \log_2 n \rceil$ multiplications for a degree-n polynomial. Assuming that these bounds hold exactly, the multiplicative depth is $(e-1)\lceil \log_2 p \rceil$ for Halevi/Shoup versus $(v-1)\lceil \log_2 p \rceil + \lceil \log_2((e-v)(p-1)+1) \rceil$ for Chen/Han. The second algorithm is therefore better when $e - v \gg 1$.

5.2 General bootstrapping

The goal of this section is to describe Halevi and Shoup's [16, 19] bootstrapping algorithm for BGV and to transfer the same technique to the FV scheme as well. The algorithm starts from the observation that we can rewrite the decryption function in such a way that it corresponds to a coefficient-wise digit extraction. We first obtain an encryption of a plaintext that has the message in the upper bits and contains noise in the lower bits. We then get rid of the noise by removing the least significant bits of the plaintext. This cannot be done by applying digit extraction immediately since multiplication for BGV and FV are not coefficient-wise. We therefore need to rely on the techniques of the previous chapter: first move the coefficients to the slots, do the digit extraction there, and finally move the slots back to the coefficients.

5.2.1 Simplifying the decryption function

We start by rewriting the decryption function for plaintext space $\mathcal{R}_{p^r}$, with the purpose of evaluating it homomorphically using digit extraction. For the BGV scheme, this has been described by Gentry et al. [12] for the binary case and by Halevi and Shoup [19] for the general case. As observed by Chen and Han [6], it is not necessary to alter the decryption function for the FV scheme.

BGV decryption. The simplified decryption function is defined with respect to the parameters e' and $e'' \geqslant r$. We start by modulus switching the input ciphertext to $\tilde{q} = p^{e'} + 1$. Then we add multiples of $\tilde{q}$ to both ciphertext parts such that they become divisible by $p^{e''}$. We choose the multiples such that the resulting ciphertext, denoted by ct, has coefficients that are as small as possible. Remark that the addition of $\tilde{q}$-multiples was not done in the original decryption function. However, it was shown by Gentry et al. [12] that this trick can lower the multiplicative depth of the decryption circuit, which is obviously desired for bootstrapping. Decryption now proceeds in two steps. First compute the inner product, i.e., set

$$\boldsymbol{w} \leftarrow [\langle \mathsf{sk}, \mathsf{ct}/p^{e''} \rangle]_{p^{e'-e''+r}}. \tag{5.1}$$

Define $\Gamma = \lfloor p^{e'-e''}/2 \rfloor$, and consider the constant $\boldsymbol{\gamma} \in \mathcal{R}$ of which the powerful basis coefficients are all equal to Γ. Now compute

$$\boldsymbol{u} \leftarrow -\boldsymbol{w} + \boldsymbol{\gamma} \qquad \text{and} \qquad \boldsymbol{m} \leftarrow \boldsymbol{u}\langle e' - e'' + r - 1, \ldots, e' - e'' \rangle .$$

The addition of $\boldsymbol{\gamma}$ is necessary because we extract the non-negative digits instead of the balanced ones. It was proven that this procedure indeed recovers the original message if the parameters are well chosen [19].

FV decryption. The simplified decryption function is defined with respect to the parameter e. We first obtain the ciphertext ct by modulus switching the input ciphertext to $\tilde{q} = p^e$. Decryption now proceeds in two steps. First compute the inner product, i.e., set

$$\boldsymbol{w} \leftarrow [\langle \mathsf{sk}, \mathsf{ct} \rangle]_{p^e}. \tag{5.2}$$

Define $\Gamma = \lfloor p^{e-r}/2 \rfloor$, and consider the constant $\boldsymbol{\gamma} \in \mathcal{R}$ of which the powerful basis coefficients are all equal to Γ. Now compute

$$\boldsymbol{u} \leftarrow \boldsymbol{w} + \boldsymbol{\gamma} \qquad \text{and} \qquad \boldsymbol{m} \leftarrow \boldsymbol{u}\langle e-1, \ldots, e-r \rangle .$$

Correctness follows immediately because this procedure is roughly the same as the original decryption function.

5.2.2 Bootstrapping fully packed ciphertexts

Bootstrapping is done by implementing the simplified decryption function homomorphically. We describe the entire procedure below, and all steps are also shown schematically in Figure 5.1.

Modulus switching. Switch the input ciphertext to the special modulus $\tilde{q}$. In the case of BGV, also add multiples of $\tilde{q}$ to both ciphertext parts as described in the previous section. The resulting ciphertext is denoted by ct.

Homomorphic inner product. Compute homomorphically the inner product between the secret key $\mathsf{sk} = (1, \boldsymbol{s})$ and the ciphertext ct using Equation 5.1 or 5.2. The outcome of this step is an encryption of $\boldsymbol{w}$ with respect to plaintext modulus $p^{e'-e''+r}$ for BGV and p^e for FV. To evaluate the expression homomorphically, we must be in possession of a bootstrapping key. That is, we must have an encryption of $\boldsymbol{s}$ with respect to the same plaintext modulus as we desire for $\boldsymbol{w}$. Once we obtain an encryption of $\boldsymbol{w}$, it is converted to an encryption of $\boldsymbol{u}$. The cost and depth of the inner product are one constant-ciphertext multiplication.

$$\begin{array}{rl}
\mathsf{Enc}(\boldsymbol{m}) & \\
\quad \downarrow \text{Modulus switching and inner product} & \\
\mathsf{Enc}(p^v \cdot \boldsymbol{m} + \boldsymbol{v}) & \\
\quad \downarrow \text{Linear transformation and unpacking} & \\
\mathsf{Enc}(p^v \cdot m_{\ell i} + v_{\ell i}, \ldots, p^v \cdot m_{\ell(i+1)-1} + v_{\ell(i+1)-1}) & \text{for } i = 0, \ldots, d-1 \\
\quad \downarrow \text{Digit extractions} & \\
\mathsf{Enc}(m_{\ell i}, \ldots, m_{\ell(i+1)-1}) & \text{for } i = 0, \ldots, d-1 \\
\quad \downarrow \text{Repacking and inverse linear transformation} & \\
\mathsf{Enc}(\boldsymbol{m}) &
\end{array}$$

FIGURE 5.1: General bootstrapping procedure, adapted from [6]

Linear transformation. Move the powerful basis coefficients of $\boldsymbol{u}$ to the plaintext slots. The result of this step is an encryption of $\boldsymbol{\beta}$ of which the slots encode the coefficients of $\boldsymbol{u}$. The transformation is the inverse of the evaluation map from Chapter 4, where we run all stages in reverse order and with matrix M^{-1} instead of M. The cost is approximately $c \cdot d + \ell_1 + 2\sqrt{\ell_2} + \ldots + 2\sqrt{\ell_t}$ automorphisms and $c \cdot d \cdot \ell_1 + \ell_2 + \ldots + \ell_t$ constant-ciphertext multiplications, where $c = 1$ if all dimensions are good and $c = 2$ otherwise. The depth is t constant-ciphertext multiplications.

Unpacking the slots. The plaintext $\boldsymbol{\beta}$ contains d coefficients per slot that are encoded using a normal element θ. However, digit extraction needs the plaintext slots to be encoded "sparsely", i.e., each slot can contain only one coefficient. We must therefore convert $\boldsymbol{\beta}$ into d distinct plaintexts $\boldsymbol{\beta}_0, \ldots, \boldsymbol{\beta}_{d-1}$ that encode only one coefficient per slot in the constant term. This step is called unpacking and can be computed based on the Frobenius map. Consider the constants $\boldsymbol{\kappa}_f$ corresponding to the linear map that acts on each slot as

$$L_0 \colon \sum_{f=0}^{d-1} c_f \, \sigma_E^f(\theta) \mapsto c_0. \tag{5.3}$$

for $c_f \in \mathbb{Z}_{p^r}$. We first precompute $\sigma^f(\boldsymbol{\beta})$ for $f = 0, \ldots, d-1$. The result is then

$$\boldsymbol{\beta}_i = \sum_{f=0}^{d-1} \boldsymbol{\kappa}_{f+i} \cdot \sigma^f(\boldsymbol{\beta}).$$

The index is implicitly reduced modulo d, so $\boldsymbol{\kappa}_{f+d} = \boldsymbol{\kappa}_f$ by definition. The motivation for using normal element encoding is clear from the above analysis: it allows to reuse the same set of constants for each $\boldsymbol{\beta}_i$, so we only need to store d of them. The cost of this step is $d - 1$ automorphisms and d^2 constant-ciphertext multiplications. The depth is one constant-ciphertext multiplication.

Digit extraction. Apply the digit extraction procedure to each $\boldsymbol{\beta}_i$ from the previous step. This corresponds to a slot-wise digit extraction, and it therefore acts on the coefficients of $\boldsymbol{u}$ separately. We denote the resulting plaintexts by $\boldsymbol{\beta}'_i$. They are obtained by running DigitExtract with inputs $e \leftarrow e' - e'' + r$ and $v \leftarrow e' - e''$ for BGV. For the FV scheme, we set $e \leftarrow e$ and $v \leftarrow e - r$. A side effect of digit extraction is that the plaintext modulus decreases to p^r. The cost and depth of digit extraction were discussed earlier in this chapter.

Repacking the slots. Repack the sparsely packed plaintexts $\boldsymbol{\beta}'_i$ into a fully packed plaintext $\boldsymbol{\beta}'$ that encodes d coefficients per slot. This step is the inverse of unpacking, and it can be done by implementing the sum

$$\boldsymbol{\beta}' = \sum_{f=0}^{d-1} \boldsymbol{\kappa}'_f \cdot \boldsymbol{\beta}'_f,$$

where $\boldsymbol{\kappa}'_0$ contains θ in each slot and $\boldsymbol{\kappa}'_f = \sigma^f(\boldsymbol{\kappa}'_0)$. The cost of repacking is d constant-ciphertext multiplications, and the depth is one constant-ciphertext multiplication.

Inverse linear transformation. Move the plaintext slots of $\boldsymbol{\beta}'$ back to the powerful basis coefficients, using the evaluation map from Chapter 4. The result of this step is an encryption of the original message $\boldsymbol{m}$. The cost and depth are the same as for the first linear transformation.

5.2.3 Time complexity

This section gives a summary of the most expensive operations for the general recryption algorithm. We ignore the modulus switching and homomorphic inner product steps because they are very cheap. In summary, the following components determine the cost of bootstrapping:

- Both linear transformations need approximately $c \cdot d + \ell_1 + 2\sqrt{\ell_2} + \ldots + 2\sqrt{\ell_t}$ automorphisms, and slot unpacking needs $d - 1$ automorphisms.
- Both linear transformations need $c \cdot d \cdot \ell_1 + \ell_2 + \ldots + \ell_t$ constant-ciphertext multiplications, slot unpacking needs d^2 constant-ciphertext multiplications, and slot repacking needs d constant-ciphertext multiplications.
- There are d digit extraction operations, and each one needs $ev - v(v+1)/2$ evaluations of the lifting polynomial if Algorithm 21 is used; Algorithm 22 needs v evaluations of the lowest digit retain polynomial and $v(v-1)/2$ evaluations of the lifting polynomial.

5.3 Thin bootstrapping

Thin bootstrapping has been proposed as an alternative algorithm for when the plaintext slots are sparsely packed. It was first described by Chen and Han [6] and later worked out by Halevi and Shoup [17]. The algorithm is designed for ciphertexts of which the slots encode only one coefficient in the constant term. It uses similar building blocks as the general bootstrapping algorithm, but reverses the order of the linear transformations to avoid d repetitions of digit extraction.

$$
\begin{array}{c}
\mathsf{Enc}(m_0, \ldots, m_{\ell-1}) \\
\downarrow \text{Linear transformation} \\
\mathsf{Enc}(\boldsymbol{m}) \\
\downarrow \text{Modulus switching and inner product} \\
\mathsf{Enc}(p^v \cdot \boldsymbol{m} + \boldsymbol{v}) \\
\downarrow \text{Inverse linear transformation} \\
\mathsf{Enc}(p^v \cdot m_0 + v_0, \ldots, p^v \cdot m_{\ell-1} + v_{\ell-1}) \\
\downarrow \text{Digit extraction} \\
\mathsf{Enc}(m_0, \ldots, m_{\ell-1})
\end{array}
$$

FIGURE 5.2: Thin bootstrapping procedure, adapted from [6]

5.3.1 Bootstrapping sparsely packed ciphertexts

We describe the thin bootstrapping procedure below, and all steps are also shown schematically in Figure 5.2.

Linear transformation. Move the plaintext slots of $\boldsymbol{\beta}$ to the powerful basis coefficients. The result of this step is an encryption of $\boldsymbol{m}$ of which the coefficients are equal to the numbers that $\boldsymbol{\beta}$ has in its slots. Because the slots of $\boldsymbol{\beta}$ are encoded sparsely, also the powerful basis coefficients of $\boldsymbol{m}$ are sparse, i.e., most of them are zero. The transformation roughly corresponds to the evaluation map from Chapter 4 with an exception for stage 1. It turns out that, by restricting the plaintext slots to encode only a coefficient in the constant term, we can alleviate the implementation of the first stage. Indeed, the matrix entries can be simplified to

$$m_{j+1,k+1} \colon \eta \mapsto \zeta_{1,j}^{k} \cdot \eta,$$

which corresponds to an E-linear transformation that is identical to the one from stages $2, \ldots, t$. This makes the evaluation map much cheaper to implement: the cost is approximately $(1+c)\sqrt{\ell_1}+2\sqrt{\ell_2}+\ldots+2\sqrt{\ell_t}$ automorphisms and $c\cdot\ell_1+\ell_2+\ldots+\ell_t$ constant-ciphertext multiplications, where $c=1$ if all dimensions are good and $c=2$ otherwise. The depth remains t constant-ciphertext multiplications.

Modulus switching. This step is identical to the corresponding step in the general bootstrapping algorithm.

Homomorphic inner product. Also this step is identical to the corresponding step in the general bootstrapping algorithm. The outcome is an encryption of $\boldsymbol{u}$ from which we can recover $\boldsymbol{m}$ by applying a coefficient-wise digit extraction. Note that although the powerful basis coefficients of $\boldsymbol{m}$ are sparse, this does not hold for $\boldsymbol{u}$ because they contain noise.

Inverse linear transformation. Move the powerful basis coefficients of $\boldsymbol{u}$ to the plaintext slots. We are only interested in the subset of coefficients that correspond to the original message $\boldsymbol{\beta}$. The remaining coefficients should not end up in the slots in any way. The transformation is roughly the inverse of the evaluation map from Chapter 4, with an exception for stage 1 (which is now actually the last stage because we are considering the inverse map). Recall that the coefficients of $\boldsymbol{u}$ are not sparse, so stage 1 cannot be directly implemented as an E-linear transformation. Instead, consider the matrix M with entries given by Equation 4.11, and consider the linear map L_0 from Equation 5.3. We must first apply M^{-1} to each hypercolumn and then perform a slot-wise unpacking with L_0. This is equivalent to a multiplication with LM^{-1}, where L is the diagonal matrix containing L_0 everywhere. The matrix LM^{-1} contains linear maps to the base ring, namely $\mathbb{Z}_{p^{e'-e''+r}}$ for BGV and $\mathbb{Z}_{p^e}$ for FV. Hence it follows from Lemma 4.5 that each of its entries can be written as

$$\eta \mapsto \sum_{f=0}^{d-1} \sigma_E^f(\theta_0 \cdot \eta)$$

for some $\theta_0 \in E$. More specifically, denote by $\theta_{j,k}$ the constant corresponding to the jth row and the kth column of LM^{-1}. Moreover, let

$$T_E \colon \eta \mapsto \sum_{f=0}^{d-1} \sigma_E^f(\eta)$$

be the trace map on E. We can now rewrite the matrix as

$$LM^{-1} = \begin{bmatrix} T_E & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & T_E \end{bmatrix} \begin{bmatrix} \theta_{1,1} & \dots & \theta_{1,\ell_1} \\ \vdots & \ddots & \vdots \\ \theta_{\ell_1,1} & \dots & \theta_{\ell_1,\ell_1} \end{bmatrix},$$

so stage 1 can be decomposed as an E-linear transformation, followed by a slot-wise trace map. We can implement the slot-wise trace map at the cost of at most $2 \cdot \log_2 d$ automorphism operations [15]. That is, for a plaintext $\boldsymbol{\eta} \in \mathcal{R}_{p^e}$, we choose $d' = \lfloor d/2 \rfloor$ and compute

$$\boldsymbol{y} = \sum_{f=0}^{d'-1} \sigma^f(\boldsymbol{\eta})$$

recursively. The result is then $\boldsymbol{y} + \sigma^{d'}(\boldsymbol{y})$ for even d and $\sigma(\boldsymbol{y} + \sigma^{d'}(\boldsymbol{y})) + \boldsymbol{\eta}$ for odd d. The slot-wise trace map makes the cost of the inverse transformation slightly higher than the cost of the forward transformation.

Digit extraction. Apply the digit extraction procedure to the ciphertext from the previous step. This corresponds to a slot-wise digit extraction, and it therefore acts on the coefficients of $\boldsymbol{u}$ separately. The only difference with the general bootstrapping algorithm is that we need to run only one digit extraction instead of d. The result of this step is an encryption of the original message $\boldsymbol{\beta}$.

5.3.2 Time complexity

This section gives a summary of the most expensive operations for the thin recryption algorithm. We ignore the modulus switching and homomorphic inner product steps because they are very cheap. In summary, the following components determine the cost of bootstrapping:

- Both E-linear transformations need approximately $(1+c)\sqrt{\ell_1}+2\sqrt{\ell_2}+\ldots+2\sqrt{\ell_t}$ automorphisms, and the slot-wise trace map needs between $\log_2 d$ and $2 \cdot \log_2 d$ automorphisms.
- Both linear transformations need $c \cdot \ell_1 + \ell_2 + \ldots + \ell_t$ constant-ciphertext multiplications.
- The digit extraction operation needs $ev - v(v+1)/2$ evaluations of the lifting polynomial if Algorithm 21 is used; Algorithm 22 needs v evaluations of the lowest digit retain polynomial and $v(v-1)/2$ evaluations of the lifting polynomial.

Chapter 6

Implementation and Results

This chapter gives a thorough overview of the actual Magma implementation. The first section discusses some design decisions and implementation details, covering both the cryptographic schemes and bootstrapping. The second section gives experimental results for the two recryption algorithms, including execution time, noise capacity and memory usage.

6.1 Design decisions

6.1.1 Efficient arithmetic in cyclotomic rings

Recall from Chapter 2 that the Double-CRT representation leads to an efficient arithmetic in $\mathcal{R}_q$. It significantly improves the execution time for multiplications and automorphisms when compared to standard polynomial algorithms. In order to convert between the polynomial representation and Double-CRT, we must have access to an algorithm for computing the number-theoretic transform (NTT). A drawback of Magma is that such an algorithm is not provided to the public interface. Various solutions can be considered to overcome this problem. A first option is to discard the Double-CRT representation and work with polynomial arithmetic instead. A second option is to implement a customized FFT algorithm.

The first option was chosen for polynomial multiplication and modular reduction. The reason is that Magma provides fast algorithms for polynomial arithmetic. It uses a method based on Chinese remaindering for small coefficients, whereas for large coefficients, it uses the Schönhage-Strassen method [23]. It was determined experimentally that this built-in functionality outperforms the solution with FFT.

In contrast to multiplication and modular reduction, there is no fast algorithm to compute automorphisms in Magma. We therefore choose the second option, so we need to implement an FFT algorithm. The Cooley-Tukey decomposition cannot be used directly because it only supports power-of-two lengths [7]. Instead, we use Bluestein's FFT algorithm that can compute an NTT of any length m [2]. It requires a prime p (not the plaintext modulus) as in Section 2.1.4. The finite field $\mathbb{F}_p$ must have a primitive $2m$th root of unity W and a primitive mth root of unity $\omega = W^2$. The NTT is then reformulated as a convolution by writing

$$X_j = \sum_{i=0}^{m-1} x_i \cdot \omega^{ij} = \sum_{i=0}^{m-1} x_i \cdot W^{2ij} \cdot W^{(i^2+j^2)} \cdot W^{-(i^2+j^2)}$$
$$= W^{j^2} \sum_{i=0}^{m-1} \left[x_i \cdot W^{i^2}\right] W^{-(j-i)^2} = W^{j^2} \left[x' * w'\right]_j .$$

The sequences x' and w' are defined as

$$x'_i = x_i \cdot W^{i^2}, \quad i = 0, \ldots, m-1$$
$$w'_i = W^{-i^2}, \qquad i = -m+1, \ldots, m-1,$$

and both are zero outside the specified range [8]. The convolution can be computed by observing its similarity with polynomial multiplication. Consider the polynomials

$$f(y) = \sum_{i=0}^{m-1} x'_i y^i \qquad \text{and} \qquad g(y) = \sum_{i=0}^{2m-2} w'_{i-m+1} y^i,$$

and let their product be denoted by

$$f(y) \cdot g(y) = \sum_{i=0}^{3m-3} c_i y^i.$$

It is not so difficult to see that $[x' * w']_j = c_{j+m-1}$, so again it is possible to use the built-in polynomial multiplication algorithm. In a low-level programming language, one would do it differently and compute the convolution using an NTT with power-of-two length. We also tried this in our original implementation, where we used a Cooley-Tukey decomposition. However, this turned out to be very slow due to random memory accesses, so instead we compute the convolution as a polynomial product. Note that this special way of computing an NTT should never be used in practice, but for us it is the best option because we have no FFT algorithm available.

6.1.2 Linear transformations

The following paragraphs discuss the implementation of the linear transformations, including the hypercube structure and details about the evaluation keys and linear algebra.

Choosing the factorization of m. The performance of the evaluation map is to a large extent determined by the factorization of the cyclotomic index $m = m_1 \cdot \ldots \cdot m_t$. It was explained earlier that the size of the first dimension is $\ell_1 = \varphi(m_1)/d$, and for all other dimensions, the size is $\ell_i = \varphi(m_i)$. On the one hand, we should choose the factors small enough because this leads to an efficient evaluation map. This can be done, for example, by considering only prime-power factorizations. On the other hand, the factors should not be too small because the multiplicative depth is equal to the number of factors. This trade-off typically results in two or three factors of reasonable size.

Also the order of the factors makes a difference for the evaluation map, but this is harder to quantify because it depends on many parameters. Assume that we want to apply thin bootstrapping with as few automorphisms as possible. If all dimensions are good, then the number of automorphisms is $2\sqrt{\varphi(m_1)/d} + \mathcal{O}(\log_2 d)$ for the first stage and $2\sqrt{\varphi(m_i)} + \mathcal{O}(1)$ for all other stages. The "cost" of choosing a factor as the first one can therefore be characterized as $\sqrt{\varphi(m_i)/d} - \sqrt{\varphi(m_i)}$. This expression decreases as a function of $\varphi(m_i)$, so the first factor should have $\varphi(m_1)$ as large as possible. Note that this analysis is only valid under the above assumptions, and the optimal first factor can be different in other circumstances.

Key switching strategy. Key switching is done by generating one evaluation key per automorphism. `HElib` has some more advanced key switching strategies like baby-step/giant-step, minimal key switching and hoisting [17], but our implementation does not support them. The reason is that these algorithms do not improve bootstrapping in most cases. We explain each of these algorithms and why they were not implemented.

The baby-step/giant-step key switching strategy reduces the number of evaluation keys in a certain dimension to $2\sqrt{\ell_i} + \mathcal{O}(1)$. However, for one-dimensional E-linear transformations, we use Algorithm 19 that reduces the number of evaluation keys to the same amount. Baby-step/giant-step key switching is therefore not helpful in this case. Only for $\mathbb{Z}_{p^r}$-linear transformations, it can reduce the number of evaluation keys even further.

The minimal key switching strategy reduces the number of evaluation keys to only one per dimension. On the downside, it requires up to $\ell_i - 1$ key switching operations per rotation. Consequently, minimal key switching is only useful when memory and storage are very expensive.

The hoisting technique reduces the number of conversions between polynomial and Double-CRT representation. However, this is not relevant to us because we perform ring multiplication in polynomial form anyway. This means that we need to convert to polynomial representation for key switching, and hoisting cannot prevent this.

Linear algebra. Recall that the linear transformations require a set of constants $\boldsymbol{\kappa}_i$ to be precomputed. They are obtained by embedding elements of the slot algebra in the plaintext slots. Calculations in the slot algebra, for example a matrix inversion, require to factor the cyclotomic polynomial modulo p. Both matrix inversion and factorization are handled by first computing the solution over a finite field and then Hensel lifting the result.

At some point in the precomputation of the inverse evaluation map, we need to invert a matrix of which the entries are themselves $\mathbb{Z}_{p^e}$-linear maps on E. This is achieved by representing elements of the slot algebra in the basis $1, \zeta, \ldots, \zeta^{d-1}$. A linear map on E can then be written as a matrix over $\mathbb{Z}_{p^e}$, so the problem comes down to inverting a block matrix. This is possible with built-in functionality because we can just see the block matrix as having entries from $\mathbb{Z}_{p^e}$.

6.1.3 Digit extraction

The following paragraphs discuss the implementation of digit extraction. We explain the polynomial evaluation algorithm and a lower memory implementation of digit extraction.

Polynomial evaluation. Digit extraction needs an algorithm to evaluate the lifting polynomial and the lowest digit retain polynomial. This evaluation is done homomorphically, so the algorithm should perform well on ciphertexts. The cost is measured in ciphertext-ciphertext multiplications because they are much more expensive than constant-ciphertext multiplications [6]. The Paterson-Stockmeyer algorithm is precisely designed for this cost measure, requiring only $\mathcal{O}(\sqrt{n})$ non-constant multiplications for a degree-n polynomial [24]. We therefore choose a variant of it that also keeps the multiplicative depth low. Let the parameters k and m be such that $n \leqslant km$, then we write

$$f(y) = \sum_{i=0}^{n} a_i y^i = a_0 + \sum_{j=0}^{m-1} \left[\sum_{i=1}^{k} a_{i+kj} y^i \right] y^{kj}.$$

The right-hand side of this equation can be evaluated efficiently by precomputing y^i for $i = 2, \ldots, k$ and $i = 2k, \ldots, (m-1)k$. Computing the above sum then results in $2m + k - 4$ non-constant multiplications when $m > 1$. The parameters k and m are chosen to minimize this number.

The implementation also contains a different version of the Paterson-Stockmeyer algorithm that recursively breaks down the polynomial in multiple polynomials of lower degree [24]. Here we chooses the parameters k and $p = 2^m$, and we write a polynomial $f(y)$ of degree $k(2p-1)$ as

$$f(y) = (y^{kp} + c(y)) \cdot q(y) + (y^{k(p-1)} + s(y)).$$

The degree of $c(y)$ is at most $k-1$, and the degrees of $q(y)$ and $y^{k(p-1)} + s(y)$ are exactly $k(p-1)$. In order to evaluate this expression, we precompute y^i for $i = 2, \ldots, k$ and $i = 2k, 4k, \ldots, 2^m k$. The evaluation of $c(y)$ is done with these precomputed values only, but for $q(y)$ and $y^{k(p-1)} + s(y)$, we need one recursive call. Although this version of the Paterson-Stockmeyer algorithm requires even less non-constant multiplications asymptotically, it performs worse when the degree is not so high. The implementation was therefore discarded, and the previously explained algorithm was used.

Memory requirements. Both the Halevi/Shoup and Chen/Han digit extraction procedures from Section 5.1 consist of two nested loops. Each iteration of the inner loop computes one new value, so a direct implementation requires at least $\mathcal{O}(v^2)$ memory. However, we can reduce this by noticing that for each i, we only need to store the value of $w_{i,j}$ that is computed most recently; all other values are never used anymore. It is therefore possible to reduce the memory to $\mathcal{O}(v)$, and this was incorporated in the implementation.

6.2 Experimental results

The following two sections give experimental results about the implementation. All experiments are run on an Intel® Xeon® E5-2630 v2 CPU with 128 GB memory, Ubuntu 16.04.7 LTS and Magma V2.26-1.

6.2.1 Bootstrapping based on Boolean circuits

We verify the complexity estimate of the bootstrapping algorithm from Chapter 3. Recall from Section 3.2.3 that we need approximately $3n(e-1)$ multiplications for recrypting a single plaintext coefficient. The parameter n is the degree of $\Phi_m(x)$, and $q' = 2^e$ is the ciphertext modulus used for bootstrapping. Although multiplications are the most expensive operations, there is of course some overhead in practice. Let t_{rec} be the time for recrypting one coefficient, and let t_{mul} be the time for one multiplication plus relinearization. We model the overhead as a constant factor, i.e.,

$$t_{\text{rec}} = c \cdot 3n(e-1) \cdot t_{\text{mul}} \tag{6.1}$$

with c close to 1. This simple model was checked for $m = 1024$, $q = 2^{300}$ and $q' = 2^9$, where the factor was approximately $c = 1.15$, i.e., an overhead of 15%.

The result from the previous paragraph is extrapolated to larger cyclotomic indices in Figure 6.1. This is done by measuring the time for one multiplication, and then applying Equation 6.1 under the assumption that the overhead factor $c = 1.15$ remains valid. Note that the plot shows the bootstrapping time for only one coefficient and that the security level of the first four data points is unrealistically low. Also the estimated execution time is very large, varying between 4.51e2 and 8.18e5 seconds. This algorithm is therefore not sufficient for practical applications.

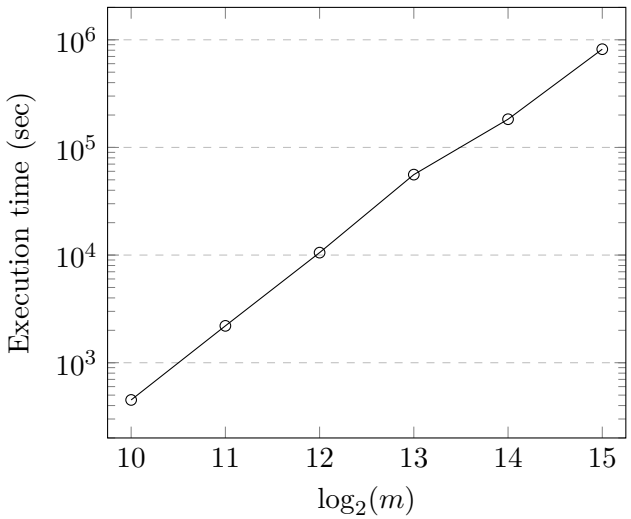

FIGURE 6.1: Estimated bootstrapping time

6.2.2 Bootstrapping based on digit extraction

This section gives experimental results for the recryption algorithms of Chapter 5. To ensure a fair comparison, we always take equal parameters for the BGV and FV schemes. Contrary to what is claimed by Chen and Han [6], this is always possible using the improved analysis of Halevi and Shoup [19]. In particular, this can be done for BGV by choosing the parameter e'' sufficiently large. The parameter v can then be chosen independent of r.

Tables 6.1 and 6.2 give experimental results for the general and thin bootstrapping algorithms respectively. The left values are for BGV and the right values for FV. Both tables have a plaintext modulus of 2 and a security level of approximately 80 bits. Digit extraction is always done with the Chen/Han procedure. The displayed capacity is the number of bits in the noise that each step uses. It is determined experimentally, so in practice, the remaining capacity is less if we want to have some safety margin. We also give the timing for each step and the total memory usage. Note that the forward and inverse linear maps include unpacking and repacking of the plaintext slots respectively.

Remember that the complexity of BGV and FV bootstrapping, expressed in number of homomorphic operations, is exactly the same for equal parameters. This can be verified experimentally with the result tables: the timing difference between BGV and FV is always less than 10%. The slightly better results for BGV can be explained due to a faster modulus switching strategy. Note that if Double-CRT were used in the implementation, then BGV would perform even better compared to FV. Indeed, the modulus of FV has to be increased temporarily during multiplication, which is not necessary for BGV. The small timing difference in our case is thus a consequence of not using Double-CRT for ciphertext multiplication.

Cyclotomic index m		$425 \cdot 41$	$257 \cdot 5 \cdot 17$	$241 \cdot 7 \cdot 17$
Lattice dimension n		12800	16384	23040
Bits in modulus $\log_2 q$		500	630	890
Number of slots		320	1024	960
Recrypt param v		7	8	8
Capacity (bits)	Inner product	24/24	25/25	26/26
	Linear map	83/83	120/120	154/153
	Digit extract	152/121	181/146	189/149
	Inverse map	29/31	39/41	44/45
	Remaining	210/239	263/296	475/515
Execution time (sec)	Linear map	4878/4789	2988/3164	8678/8688
	Digit extract	12520/11656	9366/10243	32238/32892
	Inverse map	1581/1549	1648/1815	4122/3982
	Total	18979/17994	14002/15222	45038/45562
Memory usage (GB)		3.4/3.4	4.1/4.1	7.7/7.7

TABLE 6.1: Results for fully packed plaintext slots, BGV (left) vs. FV (right)

A noteworthy fact is that the performances of the forward and inverse linear maps are completely different. In particular, the evaluation map adds less noise and is more efficient to compute than its inverse. There are four important reasons for this: firstly, the evaluation map is computed with respect to a smaller plaintext modulus. Consequently, the constant-ciphertext multiplications add less noise. Secondly, the inverse evaluation map is executed just after the homomorphic inner product. The noise is thus very low, so key switching adds a lot of noise relatively speaking. Thirdly, the evaluation map is computed with respect to a smaller ciphertext modulus, which makes the entire procedure more efficient to compute. Finally, the inverse evaluation map is followed either by a costly slot unpacking or a slot-wise trace map, which is not necessary in the forward direction.

Cyclotomic index m		$425 \cdot 41$	$257 \cdot 5 \cdot 17$	$241 \cdot 7 \cdot 17$
Lattice dimension n		12800	16384	23040
Bits in modulus $\log_2 q$		500	630	890
Number of slots		320	1024	960
Recrypt param v		7	8	8
Capacity (bits)	Linear map	17/16	29/30	33/31
	Inner product	24/25	25/25	26/27
	Inverse map	69/69	104/105	137/137
	Digit extract	151/120	181/145	189/150
	Remaining	237/268	289/323	503/543
Execution time (sec)	Linear map	89/84	133/145	223/237
	Inverse map	302/305	443/466	910/915
	Digit extract	272/295	555/610	1257/1326
	Total	663/684	1131/1221	2390/2478
Memory usage (GB)		1.7/1.7	3.1/3.1	5.4/5.4

TABLE 6.2: Results for sparsely packed plaintext slots, BGV (left) vs. FV (right)

Cyclotomic index m		$425 \cdot 41$	$257 \cdot 5 \cdot 17$	$241 \cdot 7 \cdot 17$
Lattice dimension n		12800	16384	23040
Bits in modulus $\log_2 q$		500	630	890
Number of slots		320	1024	960
Recrypt param v		7	8	8
Capacity (bits)	$r = 1$	126/120	154/145	157/150
	$r = 4$	212/173	247/201	252/204
	$r = 8$	355/229	397/260	403/264
Execution time (sec)	$r = 1$	202/295	418/610	910/1326
	$r = 4$	324/359	652/732	1432/1604
	$r = 8$	421/429	844/862	1985/1935

TABLE 6.3: Comparison of Halevi/Shoup (left) and Chen/Han (right) digit extraction

The bottleneck of the fully packed bootstrapping algorithm is clearly the digit extraction step. For thin bootstrapping, however, the execution times of the linear maps and digit extraction are similar. Remark that the time percentage of the linear maps in our implementation is somewhat higher than for state-of-the-art bootstrapping [19]. This can be explained by the differences regarding Double-CRT representation and not using hoisting for the linear transformations. An important aspect here is our customized FFT algorithm, which is still slower than a low-level implementation.

Digit extraction is the deepest part of bootstrapping in terms of multiplicative depth. Moreover, its complexity depends immensely on the Hensel lifting exponent of the plaintext space. Table 6.3 therefore shows a comparison of Halevi/Shoup digit extraction (left) and Chen/Han digit extraction (right) for the FV scheme using different values of r. As expected, Chen/Han performs much better in terms of capacity when r is large. On the other hand, the execution time is often slightly higher because we need to evaluate the lowest digit retain polynomial.

Chapter 7

Conclusion

Gentry's bootstrapping technique remains a costly operation in the construction of fully homomorphic encryption schemes. In this thesis, we studied two different bootstrapping algorithms for the FV scheme: one based on Boolean circuits and the other one based on digit extraction. We made a theoretical and experimental analysis of these two algorithms, and the second one was also compared to its BGV counterpart. The implementation of both schemes and their bootstrapping algorithms was done in the Magma Computational Algebra System.

We applied a few improvements to the Boolean recryption procedure of Fan and Vercauteren. Our adapted algorithm makes it possible to recrypt ciphertexts when the secret key has ternary coefficients. However, bootstrapping is still very slow in terms of time complexity and absolute timing, so this algorithm should not be used.

We ported the bootstrapping algorithm of `HElib` from BGV to FV, and we support recryption of fully packed as well as sparsely packed ciphertexts. The most expensive operations are the linear transformations and digit extraction. For the linear transformations, we use Halevi and Shoup's algorithm, which saves some automorphisms compared to a naive implementation. For digit extraction, we use the Halevi/Shoup or Chen/Han procedure, together with the Paterson-Stockmeyer algorithm for polynomial evaluation. Our theoretical analysis shows that BGV and FV bootstrapping can be done with exactly the same number of operations. This was verified experimentally, where we observe only a very small timing difference between BGV and FV.

In order to obtain a more efficient implementation of bootstrapping, it would be advantageous to have a native FFT algorithm in Magma. Currently, there is not even an FFT for power-of-two lengths, and a customized implementation leads to very slow memory accesses. A faster FFT would allow us to work with the Double-CRT representation, and consequently, both the linear maps and digit extraction would benefit from this.

For future work, we propose to concentrate on the digit extraction step. In practice, this is still the bottleneck when considering either execution time or multiplicative depth. A possible approach is to look for new techniques that evaluate digit extraction using less ciphertext multiplications.

Bibliography

[1] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In *Annual Cryptology Conference*, pages 1–20. Springer, 2013.

[2] L. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE transactions on audio and electroacoustics*, 18(4):451–455, 1970.

[3] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325. ACM, 2012.

[5] W. Castryck, I. Iliashenko, and F. Vercauteren. On error distributions in ring-based lwe. *LMS Journal of Computation and Mathematics*, 19(A):130–145, 2016.

[6] H. Chen and K. Han. Homomorphic lower digits removal and improved fhe bootstrapping. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 315–337. Springer, 2018.

[7] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):249–259, 1965.

[8] Bluestein's fft algorithm. https://www.dsprelated.com/freebooks/mdft/Bluestein_s_FFT_Algorithm.html. Accessed: 2021-05-13.

[9] L. Ducas and A. Durmus. Ring-lwe in polynomial rings. In *International Workshop on Public Key Cryptography*, pages 34–51. Springer, 2012.

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

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[12] C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

[13] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[14] S. Halevi and V. Shoup. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 6:12–15, 2013.

[15] S. Halevi and V. Shoup. Algorithms in helib. Cryptology ePrint Archive, Report 2014/106, 2014. https://eprint.iacr.org/2014/106.

[16] S. Halevi and V. Shoup. Bootstrapping for helib. In *Annual International conference on the theory and applications of cryptographic techniques*, pages 641–670. Springer, 2015.

[17] S. Halevi and V. Shoup. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*, pages 93–120. Springer, 2018.

[18] S. Halevi and V. Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. https://eprint.iacr.org/2020/1481.

[19] S. Halevi and V. Shoup. Bootstrapping for helib. *Journal of Cryptology*, 34(1):1–44, 2021.

[20] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[21] V. Lyubashevsky, C. Peikert, and O. Regev. A toolkit for ring-lwe cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 35–54. Springer, 2013.

[22] Magma computational algebra system. http://magma.maths.usyd.edu.au/magma/. Accessed: 2021-05-13.

[23] Overview of magma v2.13: Rings and their fields. http://magma.maths.usyd.edu.au/magma/overview/2/13/5/. Accessed: 2021-05-13.

[24] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.

[25] C. Peikert. How (not) to instantiate ring-lwe. In *International Conference on Security and Cryptography for Networks*, pages 411–430. Springer, 2016.

[26] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

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

[28] S. Roman. *Field theory*, volume 158. Springer Science & Business Media, 2005.

[29] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[30] V. Zucca. Towards efficient arithmetic for ring-lwe based homomorphic encryption, 2018.