Jacob Sjødin

Privacy Preserving Computation with Fully Homomorphic Encryption

Master's thesis in Mathematics
Supervisor: Kristian Gjøsteen
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Norwegian University of
Science and Technology

Jacob Sjødin

Privacy Preserving Computation with Fully Homomorphic Encryption

Master's thesis in Mathematics
Supervisor: Kristian Gjøsteen
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Privacy Preserving Computation with Fully Homomorphic Encryption

Jacob Sjødin

May 31, 2021

Abstract. The goal of this thesis is to show how privacy preserving computation can be achieved through homomorphic encryption. The thesis explores the BGV cryptosystem and how it can be used to compute arbitrary polynomials on data. We establish different ways of encoding real world data into the plaintext space and how functions in the data space can be emulated in the plaintext space. We explore the SIMD structure of the plaintext space and show how we can implement different linear algebra algorithms. A rigorous definition of privacy preserving is given and we prove that our privacy preserving model satisfies IND-CPA security. Lastly we show the potential of homomorphic encryption by looking at some statistical and machine learning techniques and discuss how we can implement them homomorphically. We give an overview of the limits of the current secure implementations of the BGV cryptosystem.

Sammendrag. Målet med denne oppgaven er å vise hvordan personvernbevarende beregninger kan oppnås gjennom homomorf kryptering. Oppgaven utforsker BGV-kryptosystemet og hvordan det kan brukes til å beregne vilkårlige polynomer på data. Vi etablerer forskjellige måter å representere data i klartekstrommet på og hvordan funksjoner i datarommet kan emuleres i klartekstrommet. Vi utforsker SIMD-strukturen til klartekstrommet og viser hvordan vi kan implementere forskjellige lineær algebra-teknikker. En rigorøs definisjon av personvernbevarende blir gitt, og vi beviser at vår personvernbevarende modell tilfredsstiller IND-CPA sikkerhet. Til slutt viser vi potensialet med homomorf kryptering ved å se på noen statistiske beregningsmetoder og maskinlæringsmetoder, og diskuterer hvordan vi kan implementere dem homomorft. Vi gir en oversikt over hvor grensene går for de eksisterene sikre implementasjonene av BGV-kryptosystemet.

Contents

Abstract **i**

1 Introduction **1**

2 Fully homomorphic encryption **5**
2.1 Notation and algebraic background 5
2.1.1 Size of polynomials 7
2.1.2 SIMD structure 9
2.1.3 Galois automorphisms 12
2.2 Constructing the cryptosystem 13
2.2.1 Homomorphic operations 16
2.2.2 Key switching 19
2.2.3 Modulus switching 22
2.2.4 The homomorphic cryptosystem 25
2.3 Noise management 28
2.3.1 The probability distribution 28
2.3.2 Noise bounds 29
2.4 Bootstrapping 32

3 Implementation **35**
3.1 Generic properties 36
3.1.1 Encodings 36

3.1.2 Arithmetic . . . 42
3.1.3 Comparisons and conditionals . . . 47
3.2 Specific properties . . . 49
3.2.1 Moving between the slots . . . 49
3.2.2 Other properties . . . 54
3.3 Overview of useful techniques . . . 56
3.3.1 Linear Algebra . . . 59

4 Security model **65**
4.1 Security proof . . . 66

5 Privacy Preserving Computation **71**
5.1 Computing complicated functions . . . 71
5.1.1 Polynomial approximation . . . 72
5.1.2 Table lookup . . . 72
5.2 Computation techniques . . . 74
5.2.1 Principal component analysis . . . 75
5.2.2 Linear regression . . . 77
5.2.3 Logistic regression . . . 80
5.2.4 Neural networks . . . 81

6 Concluding remarks **89**

References **91**

Chapter 1

Introduction

In recent years, outsourcing of computation has become increasingly in demand with the rise of cloud platforms. This outsourcing requires sharing and storage of vast amounts of data, which raises some privacy concerns. The data owners have no control over the data they send to the cloud platforms, and the service providers can use the data for their own benefit or sell them to third parties. These concerns, along with other factors, have made the need for privacy preserving computation pressing. There are many potential applications for privacy preserving computation. Due to reasons such as privacy or copyright issues, there are data we do not want to share even if useful information can be gained from sharing them with the cloud. We can outsource the computation on the data to someone with better models with privacy preserving computation. One method of privacy preserving computation from the field of cryptography is to do the computation after the data is encrypted. This is known as *homomorphic encryption*.

The goal of this thesis is to show how we can compute on encrypted data. The tool we will use is called homomorphic encryption, more specifically *fully homomorphic encryption*. This method allows for addition and multiplication on the encrypted data in a very specific

polynomial ring. In essence we can compute any polynomial function on the data. This is in contrast to partial homomorphic encryption which can only do either addition or multiplication on the encrypted data, and which has been possible for a long time. Many famous cryptosystems such as RSA are partially homomorphic. Fully homomorphic encryption was first proven to be possible in a seminal paper in 2009 by Gentry [8]. There has been rapid development since then, but there is still room for improvements. The current FHE schemes are unfortunately still considered impractical and too inefficient for real world applications.

There are many different aspects to consider in order to do homomorphic computation. The first step is to choose a particular homomorphic cryptosystem to work with, which is presented in Chapter 2. To explain how the cryptosystem works, we include some algebraic background in the chapter. After this we establish a basic encryption scheme which we use as the blueprint for the homomorphic version. We show how we can do the basic homomorphic operations, namely addition and multiplication. To hide the messages we add some noise to make them seem random. We will see how this noise grows with homomorphic operations, and show some techniques we can use to mitigate the noise growth.

The structure of the cryptosystem requires the data to be in a very specific format. This is often not the case for real world data, and we therefore require some sort of encoding of the data into this particular format. In Chapter 3 we explore some options available for encoding of data and discuss their benefits and drawbacks. In addition we consider how we can implement basic techniques such as arithmetic and comparisons. We show the algebraic structure of the cryptosystem in more detail, and how it lets us encode multiple messages in a hypercube structure. We also show how the algebraic structure of the cryptosytem lets us implement linear algebra techniques in different ways.

One question we need to answer before we can do privacy preserving computation is what it means for our computation to be privacy preserving. In Chapter 4 we take a cryptographic point of view. If we want to compute on encrypted data, we need to establish a security model to ensure that the computation is actually the right kind of privacy preserving. We include the possibility for some purposeful information leak, in order to for example decrease computation time. We show security in the semi-honest model, where the adversary follows the protocol assigned to it, but want to violate privacy when possible. We give a security proof of our proposed model in this chapter.

Lastly, in Chapter 5 we examine how we can use the tools we have created, and how we can use them to do privacy preserving computation. We consider some specific computation models and illustrate what is feasible to implement homomorphically and what is not. In addition we look at different approaches to the same computation. It is often straightforward to convert computation to the homomorphic setting once we have established the security model, chosen parameters and chosen how we represent data. We still have to choose exactly what information leak the algorithm will allow.

The main focus of this thesis is to establish the foundations for homomorphic encryption and how to use it in practice, which we do in Chapter 2 and 3. After this, the hardest part is done. The security model we establish in Chapter 4 and the concrete examples of computation methods in Chapter 5 serves as illustrating examples of the framework and purpose of privacy preserving computation. What we show throughout this thesis is that efficient implementation is highly application dependent. It is therefore impossible to give a general method for implementation. As we have not done an explicit implementation of a computation technique here, we refrain from giving a method for specific implementation.

Chapter 2

Fully homomorphic encryption

Fully homomorphic encryption is the ability to do additions and multiplication on encrypted data. In order to do this we need cryptosystems which are specifically designed for this purpose. The cryptosystem we will use is based on a system by Brakerski et al. [2], which is a system that uses a polynomial ring $R_q = \mathbb{Z}_q/\langle f(x)\rangle$ for some special polynomial f and an integer q. Its hardness to crack is based on the *learning with errors over rings (RLWE)* problem, which is described in Section 2.2. We first need some algebraic background before we can construct the cryptosystem.

2.1 Notation and algebraic background

We start by establishing some notation. We write $R = \mathbb{Z}[X]/\langle f(X)\rangle$, where the polynomial $f(X)$ will be defined later. Similarly we define $R_q = \mathbb{Z}_q[X]/\langle f(X)\rangle$ for an integer q. For $a \in R$ we write $[a]_q$ to symbolise that all the coefficients of a are reduced mod q into

$(-q/2, q/2]$. When q is odd this is the same as being reduced into the range $(-q/2, q/2)$. We write $k \in [n]$ to denote $k \in \{0, 1, \ldots, n-1\}$. In general we write vectors in bold and denote the ith element of $\mathbf{v}$ by $\mathbf{v}[i]$, and sometimes as v_i when it is obvious from context. We denote the inner product of two vectors $\mathbf{u}, \mathbf{v}$ as $\langle \mathbf{u}, \mathbf{v} \rangle = \sum_i \mathbf{u}[i] \cdot \mathbf{v}[i]$. For a field $\mathbb{F}$ we define an *mth root of unity* to be an element ω that satisfies $\omega^m = 1$. We call ω *primitive* if $\omega^k \neq 1$ for all k less than m. We denote $\mathbb{Z}_m^*$ the group of units in $\mathbb{Z}_m$. The number of elements in $\mathbb{Z}_m^*$ is $\phi(m)$, where $\phi(\cdot)$ is Euler's totient function.

Three important properties of the mth roots of unity are

- If $k \equiv l \mod m$ then $\omega^k = \omega^l$.
- If ω is primitive then every mth root of unity can be written as ω^j for some $j \in [m]$.
- If ω is primitive then every primitve mth root of unity can be written as ω^j for some $j \in [m]$ where $\gcd(j, m) = 1$.

We use the first property to simplify notation a bit. We write X^k with $k \in \mathbb{Z}_m$ to denote X to the power of the representative of $k + m\mathbb{Z}$ which lies in $[m]$. Similarly we write X^k with $k \in \mathbb{Z}_m^*$ as the same as above when $\gcd(k, m) = 1$. We now define $f(X)$ in the quotient of R as the *mth cyclotomic polynomial* $\Phi_m(X)$.

Definition 2.1. Let $\omega = e^{2i\pi/m} \in \mathbb{C}$ The mth cyclotomic polynomial $\Phi_m(x)$ is the product of the primitive mth roots of unity. Symbolically we write

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

The mth roots of unity are roots of the polynomial $X^m - 1 \in \mathbb{Z}[X]$. The mth cyclotomic polynomial then divides $X^m - 1$. In fact we have

that

$$\prod_{d|m} \Phi_d(X) = X^m - 1$$

by the fact that the primitive dth roots of unity are only represented in the dth cyclotomic polynomial.

Two well-known properties of the cyclotomic polynomials are that $\Phi_m(X) \in \mathbb{Z}[X]$ and that $\Phi_m(X)$ is irreducible over $\mathbb{Q}$. The degree of $\Phi_m(X)$ is $\phi(m)$.

2.1.1 Size of polynomials

In this section we will develop a notion for the *size* of polynomials. This is a crucial property in the cryptosystem we will define in Section 2.2. How we measure a polynomials size depends on the kind of *embedding* we use. One naive way is to look at the vector formed by the coefficients of the polynomial. This is known as the *coefficient embedding*. One notion for size of the polynomial is then to take a norm of that vector. Another important embedding is the *canonical embedding*, which evaluates the polynomial at the primitive roots of unity. If we assume we work in the mth cyclotomic quotient ring, then all polynomials are represented by a polynomial of maximum degree $\phi(m) - 1$. For a given polynomial $r(X) = \sum r_i x^i$ we denote the embeddings symbolically as

$$\begin{aligned} \text{coeff} : r(X) &\mapsto c(r) = (r_0, \ldots, r_{\phi(m)-1}) \\ \text{canon} : r(X) &\mapsto \sigma(r) = (r(\omega^t))_{t \in \mathbb{Z}_m^*} \end{aligned}$$

where ω is a primitive complex root of unity. The canonical embedding has the added benefit that multiplication works component-wise. This turns out to be very important when we look at size. We define two

different notions of the size of a polynomial by

$$\|r\| := \|\sigma(r)\|_\infty = \max_{t \in \mathbb{Z}_m^*} |r(\omega^t)|$$
$$\|r\|^c := \|c(r)\|_\infty = \max |r_i|.$$

The size of the canonical embedding has the property of being *sub-multiplicative*, namely

$$\|ab\| \leq \|a\| \cdot \|b\|$$

which is a convenient property. It makes the proofs involving the norms of polynomials much cleaner. With the coefficient embedding we only have the sub-multiplicative with a scaling factor. For example in the ring $R_q/(X^d + 1)$ we get that the norm of a product is bounded by the product of the norms scaled by a factor of $\sqrt{d}$. This is weaker than being sub-multiplicative.

The coefficient norm is useful, as it encapsulates when ciphertexts decrypt correctly. If a ciphertext $\mathbf{c}$ encrypted under the key $\mathbf{s}$ has the property $\|\langle \mathbf{c}, \mathbf{s} \rangle\|^c < q/2$, then it decrypts correctly. We explain this in detail in Section 2.2.

We can bound the canonical norm by the coefficient norm for any given parameters (q, m). Thus even though the coefficient norm is what must be kept low to avoid decryption errors, we can operate with the canonical norm, which has better properties such as the sub-multiplicative property. We get the bound from the following theorem.

Theorem 2.1. *For all $r \in R$, we have*

$$\|r\|^c \leq E_m \cdot \|r\|$$

where E_m is the infinity norm of the inverse of the Vandermonde matrix of the mth primitive roots of unity. In other words

$$E_m = \|V_m^{-1}\|_\infty \qquad V_m = (\omega^{ij})_{i \in \mathbb{Z}_m^*, j \in [\phi(m)]}$$

where for a matrix $A = (a_{ij})$ we define $\|A\|_\infty = \max \sum_j |a_{ij}|$.

Proof. We notice first that $V_m \cdot \text{coeff}(r) = \sigma(r)$. From this we get that

$$\|c(r)\|_\infty = \|V_m^{-1} \cdot \sigma(r)\|_\infty \leq \|V_m^{-1}\|_\infty \|r\| = E_m \|r\|$$

□

Similarly, we can get the bound $\|r\| \leq \|V_m\| \|r\|^c$. Since the absolute value of all entries in the Vandermonde matrix is 1, we get the bound $\|r\| \leq \phi(m) \|r\|^c$.

The bound E_m is not very tight, but somewhat useful. It depends only on the prime factors of m. If $m = p_1^{e_1} \dots p_k^{e_k}$ and $r = p_1 \dots p_k$ then $E_m = E_r$. A proof of this and computed values for E_m was given by Damgård et al. in [6].

2.1.2 SIMD structure

The plaintext space we will be working in is $R_p = \mathbb{Z}_p[X]/\langle \Phi_m(X) \rangle$ for some prime number p. It turns out that with the right parameters this ring splits into a product of finite fields. We can utilise this to put a message inside each of the finite fields and encrypting a vector instead of a single message. We show how R_p splits in the following theorem:

Theorem 2.2. *Let $R_p = \mathbb{Z}_p[X]/(\Phi_m(X))$ where p is prime and Φ_m is a cyclotomic polynomial, where $p \nmid m$. Then $R_p \simeq \mathbb{F}_{p^d} \times \dots \times \mathbb{F}_{p^d}$ where d is the smallest integer such that $p^d \equiv 1 \mod m$.*

Proof. The polynomial $\Phi_m(X) \in \mathbb{Z}_p[X]$ divides $X^m - 1 \in \mathbb{Z}_p[X]$, which is a separable polynomial, i.e. the roots in the algebraic closure are distinct. Thus $\Phi_m(X)$ has distinct roots in the algebraic closure. Additionally, the roots of $\Phi_m(X)$ are primitve mth roots of unity, and thus has multiplicative order m.

Let $\Phi_m = F_1 \dots F_l$ where the F_i's are monic, irreducible and distinct from each other. Then each $\mathbb{Z}_p[x]/(F_i(X))$ is a field, each of

which is generated by a root of the polynomial $F_i(X)$, which has multiplicative order m.

We now show that if $\mathbb{E}$ is a field such that $[\mathbb{E} : \mathbb{Z}_p] = d$ (in other words $\mathbb{E} \simeq \mathbb{F}_{p^d}$) and $\mathbb{E}$ is generated by an element ω of multiplicative order m, then d is the multiplicative order of p modulo m.

We know that $\omega^{|\mathbb{E}^*|} = \omega^{p^d-1} = 1$, so we have that m divides $p^d - 1$. Let u be the multiplicative order of p modulo m. Since $p^d \equiv 1 \mod m$, we know that $u|d$. Let $\mathbb{E}' = \{z \in \mathbb{E} | z^{p^u} = z\}$, which contains ω and $\mathbb{Z}_p$. This is a subfield, and since it contains the generating element ω, it equals $\mathbb{E}$. Since $\mathbb{E}'$ is the roots of the polynomial $X^{p^u} - X$, we have that $|\mathbb{E}| = |\mathbb{E}'| \leq p^u$. Thus $d \leq u$, and because $u|d$, we have $d = u$.

Since the assumption that $\mathbb{Z}_p[x]/(F_i(X))$ is generated by an element of order m holds for all i, we can conclude that they are all isomorphic to $\mathbb{F}_{p^d}$. Thus $R_p \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$ as required. □

Since $R_p \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$ we can, given the right parameters, use the algebraic structure to give us some useful properties. Rather than encrypting a single message into R_p we can put a message into each copy of $\mathbb{F}_{p^d}$. We call each copy of $\mathbb{F}_{p^d}$ a plaintext *slot*.

Example 2.3. Let $m = 2^{15} - 1$ and $p = 2$. Then $\phi(m) = 27000$, and we know that $d = 15$ is the smallest integer such that $2^d \equiv 1 \mod m$. Thus we would get $\phi(m)/d = 27000/15 = 1800$ slots, where each slot is $\mathbb{F}_{2^{15}}$.

We should note that although we use a prime plaintext modulus throughout this thesis, it is possible to generalise to high prime powers $P = p^r$ for $r > 1$. The plaintext space R_P is then isomorphic to a product of $\mathbb{Z}_P$-algebras, instead of a product finite fields of order p^d. Thus we get larger plaintext slot spaces, which can give more options for implementation. The isomorphism from R_P to the product of $\mathbb{Z}_P$-algebras can be found through a process known as Hensel lifting [13]. We will largely ignore this generalisation for simplicity of explanation.

Addition or multiplication in R_p corresponds to an addition or multiplication in each slot. This structure with multiple slots where the same operation is applied to all the slots is often referred to as *Single Instruction, Multiple Data* (or *SIMD* for short). This is a useful property for computation, as it allows us to do a bunch of computations in parallel. The downside of this parallelisation is that we have to do the same operation on all the slots. Fortunately, the algebraic structure gives us a way of moving between slots directly. We first have to describe the structure a bit more in detail before we look at how we move between slots.

From Theorem 2.2 we get the isomorphism

$$\begin{aligned} R_p &\to \mathbb{Z}_p[X]/(F_1(X)) \times \cdots \times \mathbb{Z}_p[X]/(F_l(X)) \\ f(x) &\mapsto ([f(X) \mod F_1(X)], \ldots, [f(X) \mod F_l(X)]) \end{aligned}$$

From this we have that R_p can be split into slots, each of which is a finite field $\mathbb{F}_{p^d}$. The polynomial $\Phi_m(X)$ has $\phi(m)$ roots and each of the polynomials F_i has d roots mod p. We focus on a single slot. We arbitrarily choose the first slot and let $E = \mathbb{Z}_p[X]/(F_1(X)) \simeq \mathbb{F}_{p_d}$. We can view E differently by noticing that $E \simeq \mathbb{Z}_p[\omega]$ for a root ω of $F_1(X)$. Now $\Phi_m(X)$ has $\phi(m)$ roots in E, the primitive roots of unity ω^j for $j \in \mathbb{Z}_m^*$. Each irreducible $F_i(X)$ then has d roots in E. We can use these roots for the movement between the slots. First we look at how they are distributed.

We look at the subgroup $H = \langle p \rangle \leq \mathbb{Z}_m^*$. Notice that $|\mathbb{Z}_m^*/H| = l$ since $\phi(m) = dl$ and $|H| = d$. We choose representatives $k_1, \ldots, k_l \in \mathbb{Z}_m^*$ such that $k_i H$ are different cosets of H for all i. These are representatives for elements in $\mathbb{Z}_m^*/H$, each of which represents a slot in the plaintext. We can choose the k_i such that $F_i(X)$ has d roots in E of the form ω^k, where $k \in k_i H$. Thus we get an isomorphism.

$$\begin{aligned} \mathbb{Z}_p[X]/(F_i(X)) &\to E \\ [f(X) \mod F_i(X)] &\mapsto f(\omega^{k_i}) \end{aligned}$$

Combining this with the previous isomorphism we get

$$\begin{aligned} R_p &\to E^l \\ f(X) &\mapsto (f(\omega^{k_1}), \ldots, f(\omega^{k_l})). \end{aligned}$$

Now that we have described how we can view the algebraic structure, we look at a transformation which we later will use to move between the slots.

2.1.3 Galois automorphisms

The Galois automorphisms are the tool we later will use to move between slots. They are ring automorphisms defined in the following way:

$$\begin{aligned} \theta_j : R &\longrightarrow R \\ f(X) &\longmapsto f(X^j) \end{aligned}$$

for $j \in \mathbb{Z}_m^*$. The reason we only look at j in $\mathbb{Z}_m^*$ and not in $\mathbb{Z}_m$ is that we need j to be invertible. We will see why in the following lemma.

Lemma 2.4. *The operation θ_j is a well defined ring automorphism.*

Proof. We first show it is well defined. We first note that $\Phi_m(X)$ divides $\Phi_m(X^j)$. To see why, notice that if ω is a primitive mth root of unity then ω^j is a root of $\Phi_m(X)$, and therefore ω is a root of $\Phi_m(X^j)$. Since $\Phi_m(X)$ is the minimal polynomial of ω, we have that $\Phi_m(X)$ divides $\Phi_m(X^j)$. Therefore we have that

$$\begin{aligned} \theta_j[f(X) + h(X)\Phi_m(X)] &= f(X^j) + h(X^j)\Phi_m(X^j) \\ &= f(X^j) + g(X)\Phi_m(X) \end{aligned}$$

where $g(X) = h(X^j)\Phi_m(X^j)/\Phi_m(X) \in R$. To see that it is bijective, note that

$$\theta_j \circ \theta_k = \theta_{jk} = \theta_k \circ \theta_j$$

In particular, if we let k be the inverse of j in $\mathbb{Z}_m^*$, then we have an inverse automorphism. □

Using these automorphisms we can move between slots by applying the correct automorphisms $\theta_j : f(X) \to f(X^j)$ for $j \in \mathbb{Z}_m^*/H$. By choosing the representatives k_i appropriately, we can do various movements between slots. More precisely, we will choose the representatives so that a Galois automorphism corresponds to a rotation of the slots. More details on this and how more general movement is implemented is described in Section 3.2.1.

2.2 Constructing the cryptosystem

Now that we have established the necessary algebraic background, we can move on to defining the cryptosystem.

Definition 2.2. A *discrete Gaussian distribution* with standard deviation r is the Gaussian distribution with standard distribution r where the elements drawn are rounded to their nearest integer.

The one we will describe is a variant of the BGV cryptosystem, so named after the discoverers Brakerski, Gentry and Vaikuntanathan [2]. The cryptosystem relies on the following security assumption.

Definition 2.3. The *PLWE problem* (polynomial-learning with errors) is to distinguish polynomially many samples from the distribution (a_i, b_i) and the same number of samples (a_i', b_i') where the a_i's, a_i''s, b_i''s and s are all drawn uniformly from R_q, e_i is drawn from a discrete Gaussian distribution χ and $b_i = a_i \cdot s + pe_i$. The assumption that this problem is hard is called the *PLWE assumption*.

This assumption holds even if s is drawn from discrete Gaussian distribution χ and not from the uniform distribution. We call this the PLWE problem because the problem is to distinguish polynomial many samples. The assumption can be reduced to the *shortest vector problem (SVP)* on ideal lattices over R. We refer to [19] for a proof.

We now construct our first (non-homomorphic) cryptosystem. We will later construct a homomorphic variant based on the one below:

Basic Encryption Scheme

- E.Setup($m, p, 1^\lambda$): Set $R = \mathbb{Z}[X]/(\Phi_m(X))$ and choose a discrete Gaussian distributions χ, χ'. The prime p will be the plaintext modulus. Choose the integer q (the ciphertext modulus) so that we get 2^λ security for known attacks. Set $params = (q, m, p, \chi, \chi')$.
- E.KeyGen($params$): Draw s from χ. Draw e from χ. Draw a uniformly from R_q. Set $b = a \cdot s + pe$. Output $sk = \mathbf{s} = (1, s)$ as the secret key and $pk = (a, b)$ as the public key.
- E.Enc($params, pk, m$): draw r from χ, f from χ and g from χ'. Let $\mu \in R_p$ be our message. Set the encryption to be $\mathbf{c} = (c_0, c_1)$ where $c_0 = b \cdot r + pg + \mu$ and $c_1 = -a \cdot r + pf$.
- E.Dec($params, sk, \mathbf{c}$): output $[[\langle \mathbf{c}, \mathbf{s} \rangle]_q]_p$.

The *noise* in this encryption scheme is the term $\langle \mathbf{c}, \mathbf{s} \rangle$. We often refer to the parameters e, f, g as noise as well, as these are small added noise terms that obfuscates the message. Keeping the size of the total noise $\langle \mathbf{c}, \mathbf{s} \rangle$ low lets us decrypt correctly. We show this in the following lemma.

Lemma 2.5. *The decryption is correct provided that* $\|p \cdot (r \cdot e + f \cdot s + g) + \mu\|^c < \frac{q}{2}$.

Proof.

$$\begin{aligned}
[[\langle \mathbf{c}, \mathbf{s}\rangle]_q]_p &= [[c_0 + c_1 \cdot s]_q]_p \\
&= [[(b \cdot r + pg + \mu) + (-a \cdot r + pf) \cdot s]_q]_p \\
&= [[p \cdot (r \cdot e + f \cdot s + g) + \mu]_q]_p \\
&= [p \cdot (r \cdot e + f \cdot s + g) + \mu]_p \\
&= \mu
\end{aligned}$$

□

We prove that the encryptions are indistinguishable from uniformly selected elements, or in other words that the cryptosystem is secure. We rely on the following fact from [3].

Lemma 2.6. *Let a, b, c, d be drawn from a discrete Gaussian distribution with standard deviation r, and let D be drawn from a discrete Gaussian distribution with standard deviation $2^{\omega(\log n)}r$. Then $ab + cd + D$ is statistically indistinguishable from D.*

We can now prove the security of our cryptosystem.

Theorem 2.7. *Let e, f, r, s be drawn from a Gaussian distribution χ, and g be drawn from a Gaussian distribution with larger standard deviation χ'. Let a be drawn uniformly and $b = a \cdot s + pe$. Let $x = a \cdot r + pf$ and $w = b \cdot r + pg$. Then under the PLWE assumption, it is hard to distinguish between (a, b, x, w) and (a', b', x', w') where a', b', x', w' are drawn uniformly from R_q.*

Proof. We can see directly from the PLWE assumption that it is hard to distinguish between (a, b) and (a', b'). Next we look at (x, w):

$$\begin{aligned}
w &= b \cdot r + pg \\
&= (a \cdot s + pe) \cdot r + pg \\
&= x \cdot s + p(e \cdot r - f \cdot s + g)
\end{aligned}$$

If the standard deviation of χ' is sufficiently large compared to χ, we can by Lemma 2.6 say that $e \cdot r - f \cdot s + g \approx g$. We then have that

$$(a, b, x, w) \approx (a, b, x, xs + pg).$$

However (a, x) is by the PLWE assumption indistinguishable from (a', x'). Combining this with the fact that $(x', x's + pg)$ is indistinguishable from (x', w') we get that (a, b, x, w) is indistinguishable from (a', b', x', w'). □

Now that we have a secure and functional cryptosystem, we can describe how to make it homomorphic.

2.2.1 Homomorphic operations

Addition of two ciphertexts encrypted by the same secret key is done via coordinate-wise addition:

$$\mathbf{c}_{\text{add}} = \mathbf{c} + \mathbf{c}' = (c_0 + c_0', c_1 + c_1').$$

This decrypts correctly by the bilinearity of the inner product. Multiplication of two ciphertexts encrypted under the same secret key is a little bit more complicated:

$$\begin{aligned} \mathbf{c}_{\text{mult}} = \mathbf{c} \cdot \mathbf{c}' &= (c_0c_0',\ c_0c_1' + c_0'c_1,\ c_1c_1') \\ &:= (c_{\text{mult},0}, c_{\text{mult},1}, c_{\text{mult},2}) \end{aligned}$$

The reason that we define it this way will become clear in a moment.

To better understand the multiplication of two ciphertexts, we give some facts about tensor products of two vectors. The tensor product of two 2-dimensional vectors $\mathbf{c} = [c_0\ c_1]^T, \mathbf{c}' = [c_0'\ c_1']^T$ is

$$\mathbf{c} \otimes \mathbf{c}' = \begin{bmatrix} c_0c_0' & c_0c_1' \\ c_1c_0' & c_1c_1' \end{bmatrix}.$$

Tensoring the secret key $\mathbf{s} = (1, s)$ with itself gives

$$\mathbf{s} \otimes \mathbf{s} = \begin{bmatrix} 1 & s \\ s & s^2 \end{bmatrix}$$

which we notice is a symmetric matrix. The inner product on the tensor product is defined by

$$\langle \mathbf{c} \otimes \mathbf{c}', \mathbf{s} \otimes \mathbf{s}' \rangle := \langle \mathbf{c}, \mathbf{s} \rangle \langle \mathbf{c}', \mathbf{s}' \rangle.$$

The reason we include a discussion about tensor products is we can view $\mathbf{c} \otimes \mathbf{c}'$ as a ciphertext encrypted under $\mathbf{s} \otimes \mathbf{s}$, and we can compute the decryption via the inner product in the usual way. Thus we have that

$$\begin{aligned} \langle \mathbf{c} \otimes \mathbf{c}', \mathbf{s} \otimes \mathbf{s} \rangle &= \langle \mathbf{c}, \mathbf{s} \rangle \langle \mathbf{c}', \mathbf{s} \rangle \\ &= (c_0 + c_1 s) \cdot (c_0' + c_1' s). \end{aligned}$$

We know that the multiplication of the decrypted texts can be written as

$$\begin{aligned} (c_0 + c_1 s) \cdot (c_0' + c_1' s) &= c_0 c_0' + (c_0 c_1' + c_0' c_1) s + c_1 c_1' s^2 \\ &= c_{\text{mult},0} + c_{\text{mult},1} s + c_{\text{mult},2} s^2 \\ &= \langle \mathbf{c}_{\text{mult}}, (1, s, s^2) \rangle. \end{aligned}$$

The key insight is that we can make the multiplication of two ciphertexts decryptable at the expense of adding a term to the ciphertext. We also need the powers of the secret key s. In this case, since we were evaluating a multivariate polynomial of degree 2, we needed 2 powers of s. In general, to evaluate a polynomial of degree D, we need to compute D powers of s to decrypt the polynomial.

We can generalise our tensor product approach. We call $\mathbf{c}$ a *fresh* ciphertext if it only has two terms. If we multiply r fresh ciphertexts

$\mathbf{c^1}, \ldots, \mathbf{c^r}$, we can use the tensor product representation to write

$$\langle \mathbf{c^1} \otimes \cdots \otimes \mathbf{c^r}, \mathbf{s} \otimes \cdots \otimes \mathbf{s} \rangle = \prod_{i=1}^{r} \langle \mathbf{c^i}, \mathbf{s} \rangle$$

If we write out this product, we see that the k'th term in $\mathbf{c}_{\text{mult}}$ is described by the following sum:

$$c_{\text{mult},k} = \sum_{j_1 + \cdots + j_r = k} c^1_{j_1} \ldots c^r_{j_r}$$

where $\mathbf{c^k}$ is the k'th ciphertext we are multiplying and $j_k \in \{0, 1\}$. The reason we use $\mathbf{c}_{\text{mult}}$ instead of $\mathbf{c} \otimes \mathbf{c}'$ as the product of the ciphertexts is that it is more compact. This effect is much more notable if we have ciphertexts encrypted under longer keys, as the tensor product of r fresh ciphertexts has 2^r entries while the ciphertext product has $r+1$ entries. The only reason we can use this compact version is that $\mathbf{c}$ and $\mathbf{c}'$ are encrypted under the same key $\mathbf{s}$.

Example 2.8. If we want to multiply three ciphertexts, we get

$$\begin{aligned}
\mathbf{c^1 c^2 c^3} &= \Big(\sum_{j_1+j_2+j_3=0} c^1_{j_1} c^2_{j_2} c^3_{j_3}, \ldots, \sum_{j_1+j_2+j_3=3} c^1_{j_1} c^2_{j_2} c^3_{j_3} \Big) \\
&= \big(c^1_0 c^2_0 c^3_0,\ c^1_1 c^2_0 c^3_0 + c^1_0 c^2_1 c^3_0 + c^1_0 c^2_0 c^3_1, \\
&\quad c^1_1 c^2_1 c^3_0 + c^1_0 c^2_1 c^3_1 + c^1_1 c^2_0 c^3_1,\ c^1_1 c^2_1 c^3_1 \big) \\
&= \big(c_{\text{mult},0}, c_{\text{mult},1}, c_{\text{mult},2}, c_{\text{mult},3} \big)
\end{aligned}$$

and we decrypt it by computing $c_{\text{mult},0} + c_{\text{mult},1} s + c_{\text{mult},2} s^2 + c_{\text{mult},3} s^3$.

Of course, the ciphertexts cannot just continue to grow, so we need a method for reducing the new ciphertexts. We do this with the *key switching* technique, which we introduce in the following section.

2.2.2 Key switching

Multiplication of two ciphertexts produce a ciphertext with more terms. This makes it unpractical to use, because we cannot expand the ciphertexts indefinitely. To prevent the ciphertexts from growing in length, we use a key switching method which reduces the ciphertext to the original size, with only 2 terms. For our method to work, we need more than one secret key to use. Let us illustrate with an example. When we do multiplication of two fresh ciphertexts we get a message encrypted under $(1, s, s^2)$. We would like to get the same message encrypted under some new key $(1, s')$ with only two terms.

The technique we show here is for general length keys, but the most common case is after we do a multiplication. We first construct a weak version, which will add too much noise. We assume the keys $\mathbf{s}_1, \mathbf{s}_2$ are of the form $\mathbf{s}_i = (1, \mathbf{s}_i')$.

KeySwitch($\mathbf{s}_1 \in R_q^{n_1}, \mathbf{s}_2 \in R_q^{n_2}$)

1. $A \leftarrow R_q^{n_1 \times (n_2 - 1)}$
2. $\mathbf{b} = A\mathbf{s}_2' + \mathbf{e}$. $B = (\mathbf{b} + \mathbf{s}_1, A)$
3. To get a new ciphertext $\mathbf{c}'$ of the same message encrypted under $\mathbf{s}_2$, set $\mathbf{c}' = \mathbf{c}^T B$

We can see that $\mathbf{c}'$ decrypts correctly:

$$\begin{aligned}\langle \mathbf{c}', \mathbf{s}_2 \rangle &= \langle \mathbf{c}^T B, \mathbf{s}_2 \rangle \\ &= \mathbf{c}^T B \mathbf{s}_2 = \mathbf{c}^T (\mathbf{s}_1 + \mathbf{e}) \\ &= \langle \mathbf{c}, \mathbf{s}_1 \rangle + \langle \mathbf{c}, \mathbf{e} \rangle.\end{aligned}$$

We see here that we have obtained some additional noise $\langle \mathbf{c}, \mathbf{e} \rangle$, which we would like to reduce. To do this, we refine our key switching technique with two important subroutines:

Definition 2.4. The *bit decomposition* BitDecomp$(\mathbf{x}, q)$ takes in a vector $\mathbf{x} \in R_q^n$ and the modulus q and outputs the bit representation of $\mathbf{x}$. In other words, if $\mathbf{x} = \sum_{j=0}^{\lfloor \log q \rfloor} 2^j \cdot \mathbf{u}_j$ where $\mathbf{u}_j \in R_2^n$ then BitDecomp$(\mathbf{x}, q)$ outputs $(\mathbf{u}_0, \ldots, \mathbf{u}_{\lfloor \log q \rfloor}) \in R^{n \cdot \lceil \log q \rceil}$. The function Powersof2$(\mathbf{x}, q)$ takes in $\mathbf{x} \in R_q^n, q$ and outputs $\mathbf{x}$ multiplied by the powers of 2: $(\mathbf{x}, 2 \cdot \mathbf{x}, \ldots, 2^{\lfloor \log q \rfloor} \cdot \mathbf{x}) \in R^{n \cdot \lceil \log q \rceil}$

These routines have the following useful property

Lemma 2.9. *For vectors* $\mathbf{c}, \mathbf{s}$ *of equal length, we have*

$$\langle \text{BitDecomp}(\mathbf{c}, q), \text{Powersof2}(\mathbf{s}, q) \rangle = \langle \mathbf{c}, \mathbf{s} \rangle \mod q$$

.

Proof.

$$\begin{aligned} \langle \text{BitDecomp}(\mathbf{c}, q), \text{Powersof2}(\mathbf{s}, q) \rangle &= \sum_{j=0}^{\lfloor \log q \rfloor} \langle \mathbf{u}_j, 2^j \cdot \mathbf{s} \rangle \\ &= \langle \sum_{j=0}^{\lfloor \log q \rfloor} 2^j \cdot \mathbf{u}_j, \mathbf{s} \rangle \\ &= \langle \mathbf{c}, \mathbf{s} \rangle \end{aligned}$$

□

We use these subroutines to hide the powers of 2 of the secret key in the key switching matrix, instead of the key itself. This is done in two steps: We construct a key switching matrix which is an encryption of the powers of 2 of the first key under the second key. This key switching matrix is public. After this we give a simple method for how to switch the keys. The keys $\mathbf{s}_1, \mathbf{s}_2$ are assumed to be on the form $\mathbf{s}_i = (1, \mathbf{s}_i')$.

KeySwitchGenerator$(\mathbf{s}_1 \in R_q^{n_1}, \mathbf{s}_2 \in R_q^{n_2})$:

1. Let $N = n_1 \cdot \lceil \log q \rceil$, let A be uniformly drawn from $R^{N\times(n_2-1)}$, let $\mathbf{e}$ be drawn from χ^N and set $b = A\mathbf{s}_2' + p\mathbf{e}$

2. Let $B = (b, A) + (\text{Powersof2}(\mathbf{s}_1), 0)$. Output $\tau_{\mathbf{s_1}\to\mathbf{s_2}} = B \in R^{N\times n_2}$.

Here $tau_{\mathbf{s_1}\to\mathbf{s_2}}$ is the information needed to change the keys. It does not reveal anything about the keys, by a similar argument to the security of the basic encryption scheme. What this key switching procedure essentially does is to encrypt all powers of 2 times $\mathbf{s}_2$ under the key $\mathbf{s}_2$. This information is then used as a key switching mechanisms.

To switch keys, compute $\mathbf{c_2} = \text{BitDecomp}(\mathbf{c_1})^T \cdot B$.

Example 2.10. Say we have the ciphertext $\mathbf{c}_{\text{mult}} = (c_{\text{mult},0}, c_{\text{mult},1}, c_{\text{mult},2})$ encrypted under $\mathbf{s} = (1, s, s^2)$ and we want to reduce it to a ciphertext with two terms. Then we compute the key switching matrix $\tau_{\mathbf{s}\to\mathbf{s}'}$ where $\mathbf{s}' = (1, s')$ is a new key. From there, we compute $\tilde{\mathbf{c}} = \text{BitDecomp}(\mathbf{c}_{\text{mult}})^T \cdot \tau_{\mathbf{s}\to\mathbf{s}'}$, which have two terms. This is the encryption of the same message under the key $(1, s')$.

The added noise of this new key switching procedure is as mentioned smaller than with the first procedure. We prove how small in the following lemma.

Lemma 2.11. *Let all the parameters be as above. Then the noise of the new ciphertext can be described by*

$$\langle \mathbf{c_2}, \mathbf{s_2} \rangle = p\langle \text{BitDecomp}(\mathbf{c_1}), \mathbf{e} \rangle + \langle \mathbf{c_1}, \mathbf{s_1} \rangle \mod q$$

Proof.

$$\begin{aligned}
\langle \mathbf{c_2}, \mathbf{s_2} \rangle &= \text{BitDecomp}(\mathbf{c_1}) \cdot B\mathbf{s_2} \\
&= \text{BitDecomp}(\mathbf{c_1}) \cdot (p\mathbf{e} + \text{Powersof2}(\mathbf{s_1})) \\
&= p\langle \text{BitDecomp}(\mathbf{c_1}), \mathbf{e} \rangle + \langle \text{BitDecomp}(\mathbf{c_2}), \text{Powersof2}(\mathbf{s_1}) \rangle \\
&= p\langle \text{BitDecomp}(\mathbf{c_1}), \mathbf{e} \rangle + \langle \mathbf{c_1}, \mathbf{s_1} \rangle
\end{aligned}$$

□

From this we see that we gain an error term $p\langle \text{BitDecomp}(\mathbf{c_1}), \mathbf{e}\rangle$, which is less than with our first key switching procedure. We name this error term the *key switch noise* and if we denote it $\|e\|$ we see that it can be bounded by $\|e\| \leq p\sum_i \|\mathbf{e}[i]\|$. This is small if we choose small noise polynomials. Say we bound the $\mathbf{e}[i]$ by a constant β. Then the noise is bounded by $\|e\| \leq pn_1\lceil \log q\rceil\beta$.

2.2.3 Modulus switching

The BGV cryptosystem is build on essentially hiding the plaintexts in some noise. When we multiply ciphertexts, this noise grows. If we have a bound on the noise B of the ciphertexts, then the bound on the noise of the multiplied ciphertexts is B^2. If this noise becomes too big, it will wrap around modulo q, and the decryption becomes incorrect.

In this section we look at a technique to limit noise growth called *modulus switching*. This technique lets us convert a ciphertext $\mathbf{c}$ encrypted under a modulus Q to a ciphertext $\mathbf{c}$ encrypted under a modulus q. If $\mathbf{c}'$ is the integer vector closest to $(q/Q)\mathbf{c}$ such that $\mathbf{c}' \equiv \mathbf{c} \mod p$, then we have $\langle \mathbf{c}, \mathbf{s}\rangle \mod Q = \langle \mathbf{c}', \mathbf{s}\rangle \mod q$, which we show in Theorem 2.13.

We illustrate with an example. Say we want to compute c^8 for some ciphertext c with noise bound B and that we have a modulus chain $q_0, \dots, q_3$ so that $q_i/q_{i+1} \approx B$. With regular multiplication and no modulus switching, the noise of c^8 would be B^8. But with modulus switching, we can scale down the noise each time we multiply, so that by reducing modulus three times, we get that the noise of c^8 is still B. Even though we have a smaller modulus now, we still have more room for noise growth after this operation. In practice, the modulus switching adds some small noise in addition to scaling down the noise. Thus the analysis above is only approximate. We start by defining a scaling operation.

Definition 2.5. Let $c \in R$ and let $Q > q > p$. We define $c' \leftarrow \text{Scale}(c, Q, q, p)$ to be the R-vector closest (with the coefficient norm) to $(q/Q) \cdot c$ that satisfies $c' \equiv c \mod p$.

Lemma 2.12. *The difference between the coefficients of* c' *and* $(q/Q)c$ *is at most* $(p/2)$.

Proof. Let $a = \lceil (q/Q)c \rfloor$. Then $c' = a + u$ for some $u \in R$. If u is outside $[-p/2, p/2]$, then we can add some multiple of p to c' to get something that is still congruent to $c \mod p$, but is closer. Therefore the coefficients of u lie in $[-p/2, p/2]$. Let $v = (q/Q)c - a$. The coefficients of v lie in $[-1/2, 1/2]$. Now if p is odd, then u is uniquely determined. If p is even, then let u have the same sign as v. We have that

$$\begin{aligned} c' - (q/Q)c &= c' - a - ((q/Q)c - a) \\ &= u - v. \end{aligned}$$

If p is odd, then the coefficients of u lie in $(-p/2, p/2)$ and so the coefficients of $c' - (q/Q)c$ lie in $[-p/2, p/2]$. If p is even, the sign of u and v is the same, and so the coefficients of $u - v$ lies in $[-p/2, p/2]$. Therefore we have that the difference between the coefficients of c' and $(q/Q)c$ lies in $[-p/2, p/2]$. □

Now that we have some results to lean on, we show the following:

Theorem 2.13. *Let* $Q > q > p$ *be integers and let* $\mathbf{c}$ *be a ciphertext and* $\mathbf{c}' \leftarrow \textit{Scale}(c, Q, q, p)$ *such that* $Q \equiv q \equiv 1 \mod p$. *Let* β_i *be a bound on the size of the secret key term* $\|\boldsymbol{s}[i]\|$. *Then* $\mathbf{c}'$ *decrypts to the same message as* $\mathbf{c}$ *provided that the noise* e_Q *of* $\mathbf{c}$ *satisfy*

$$\|e_Q\| < Q/(2E_m) - (p/2)\phi(m) \sum_i \beta_i.$$

Proof. Let e_Q be the noise of $\mathbf{c}$ and set

$$\begin{aligned} e_Q &= [\langle \mathbf{c}, \mathbf{s}\rangle]_q = \langle \mathbf{c}, \mathbf{s}\rangle - kQ \\ e_q &:= \langle \mathbf{c}', \mathbf{s}\rangle - kq. \end{aligned}$$

We will argue that e_q is so small that $e_q = [\langle \mathbf{c}', \mathbf{s}\rangle]_q$. We have

$$\begin{aligned} \|e_q\| &= \|\langle \mathbf{c}', \mathbf{s}\rangle - kq\| \\ &= \| - kq + \frac{q}{Q}\langle \mathbf{c}, \mathbf{s}\rangle + \langle \mathbf{c}' - \frac{q}{Q}\mathbf{c}, \mathbf{s}\rangle \\ &\leq \frac{q}{Q}\|[\langle \mathbf{c}, \mathbf{s}\rangle]_Q\| + \sum_i \|\mathbf{c}'[i] - \frac{q}{Q}\mathbf{c}[i]\| \cdot \|\mathbf{s}[i]\| \\ &\leq \frac{q}{Q}\|e_Q\| + (p/2)\phi(m)\sum_i \beta_i. \end{aligned}$$

We say the first term is the *mod scaled noise* and the second term the *mod added noise*. If we put in the bound on the noise e_Q we see that

$$\begin{aligned} \|e_q\| &\leq \frac{q}{Q}\|e_Q\| + (p/2)\sum_i \beta_i \\ &< \frac{q}{Q}(Q/(2E_m) - (p/2)\phi(m)\sum_i \beta_i) + (p/2)\phi(m)\sum_i \beta_i \\ &= q/2E_m. \end{aligned}$$

Since $\|e_q\|^c \leq E_m\|e_q\| < q/2$, we have that the decryption is correct. We can see that $\mathbf{c}'$ decrypts to the same message by noticing that

$$\begin{aligned} [\langle \mathbf{c}', \mathbf{s}\rangle]_q &\equiv \langle \mathbf{c}', \mathbf{s}\rangle - kq \\ &\equiv \langle \mathbf{c}, \mathbf{s}\rangle - kQ \\ &\equiv [\langle \mathbf{c}, \mathbf{s}\rangle]_q \mod p \end{aligned}$$

□

We have now showed that we can scale down the noise of a ciphertext, at the cost of adding some noise. If we bound the secret key properly, this noise is small. We have then successfully reduced the noise of the ciphertext.

2.2.4 The homomorphic cryptosystem

We now sketch a setup of the homomorphic encryption system. This system uses the cryptosystem sketched in the beginning of Section 2.2 as subroutines. The homomorphic cryptosystem is *leveled*, meaning that we have to specify the number of levels L (typically the multiplication depth we want to achieve) when we set up the cryptosystem. A more flexible system can be constructed by implementing a technique called *bootstrapping*. We sketch the idea behind bootstrapping in Section 2.4. For now we construct our leveled homomorphic cryptosystem. The key generation involves constructing the key switch matrices as well as the secret keys and public keys. The homomorphic cryptosystem includes three extra algorithms. There is one for addition, one for multiplication and one for refreshing ciphertexts and correcting them to the same moduli or key. Without the refresh step we cannot compute addition and multiplication, as we always assume that two ciphertexts we want to add or multiply are encrypted under the same key and modulus.

- HE.Setup$(1^\lambda, 1^L)$ takes as input the security parameter and the number of levels L. For $j = L$ down to 0, run E.Setup$(m, p, 1^\lambda)$ to obtain a ladder of moduli from q_L down to q_0. We choose $m_j = m_L, \chi_j = \chi_L$ so that we have the same ring dimension and noise distribution for every level.

- HE.KeyGen$(\{q_j, m, \chi)\})$: For $j = L$ down to 0, do:

    1. Run $\mathbf{s}_j \leftarrow$ E.SecretKeyGen and $(a_j, b_j) \leftarrow$ E.PublicKeyGen

2. Set $\mathbf{s'}_j \leftarrow \mathbf{s}_j \otimes \mathbf{s}_j$. That is, $\mathbf{s'}_j$ is a tensoring of $\mathbf{s}_j$ with itself
3. Set $\mathbf{s''}_j \leftarrow \text{BitDecomp}(\mathbf{s'}_j, q_j)$
4. Run $\tau_{\mathbf{s''}_{j+1} \to \mathbf{s}_j} \leftarrow \text{KeySwitchGen}(\mathbf{s''}_{j+1}, \mathbf{s}_j)$

The secret key sk consists of the $\mathbf{s}_j$'s and the public key consists of the (a_j, b_j)'s and the $\tau_{\mathbf{s''}_{j+1} \to \mathbf{s}_j}$'s.

- HE.Enc($params, pk, \mu$): Take the message μ in R_p. Run the basic encryption with the top level keys E.Enc($(a_L, b_L), \mu$).

- HE.Dec($params, sk, \mathbf{c}$): Suppose the ciphertext is encrypted under $\mathbf{s}_j$. Run E.Dec($\mathbf{s}_j, \mathbf{c}$) (the index of the key can be known without compromising security).

- HE.Refresh($\mathbf{c}, \tau_{\mathbf{s''}_j \to \mathbf{s}_{j-1}}, q_j, q_{j-1}$) Takes a ciphertext encrypted under $\mathbf{s}'_j$, the auxillary information $\tau_{\mathbf{s''}_j \to \mathbf{s}_{j-1}}$ to do the key switching and the moduli q_j, q_{j-1} to switch between. Do:

  1. Expand: Set $\mathbf{c}_1 \leftarrow \text{Powersof2}(\mathbf{c}, q_j)$, so $\mathbf{c}_1$ is a ciphertext encrypted under $\mathbf{s''}_j$.
  2. Switch moduli: Set $\mathbf{c}_2 \leftarrow \text{Scale}(\mathbf{c}_1, q_j, q_{j-1}, p)$, which is a ciphertext under the key $\mathbf{s}''_j$ modulus q_{j-1}.
  3. Switch Keys: Output $\mathbf{c}_3 \leftarrow \text{KeySwitch}(\tau_{\mathbf{s''}_j \to \mathbf{s}_{j-1}}, \mathbf{c}_2, q_{j-1})$, which is a ciphertext under the key $\mathbf{s}_{j-1}$ modulus q_{j-1}.

- HE.Add($pk, \mathbf{c_1}, \mathbf{c_2}$): Takes two ciphertexts encrypted under the under the same $\mathbf{s}_j$. If they are not, use HE.Refresh to make them so. Set $\mathbf{c}_3 \leftarrow \mathbf{c}_1 + \mathbf{c}_2 \mod q_j$ and use HE.Refresh if necessary to reduce errors. Output $\mathbf{c_3}$.

- HE.Mult($pk, \mathbf{c_1}, \mathbf{c_2}$): Takes two ciphertexts encrypted under the same $\mathbf{s}_j$. If they are not, use HE.refresh to make them so. Set $\mathbf{c}_3 \leftarrow \mathbf{c}_1 \cdot \mathbf{c}_2 \mod q_j$ and use HE.Refresh to make $\mathbf{c_4}$. Output $\mathbf{c_4}$.

This scheme follows the blueprint of Brakerski et al. [2] closely. There have been many developments since their paper, and this is as such not the most effective known implementation of their cryptosystem. We could for example modify the routines so that we do not refresh after addition, or so that modulus switching and key switching is not part of the same routine. This version does provide the core concepts, and more efficient implementations (such as [13]) include implementation details that would get us too much of track.

We note that the key switching matrices are in general quite big, so it places a restriction on how big parameters we can set. One way we can get around this is by encryption the powers of two of the secret key under the secret key itself, instead of encrypting under a new key each time. This is illustrated in Figure 2.1. We could remove a lot of space requirement from the key switching matrices this way. The assumption that we can safely encrypt the secret key under itself is known as the *circular security assumption*. Brakerski and Vaikuntanathan described a variant of our scheme which do not rely on this assumption, but their circular secure scheme is dependent on essentially doing a nested encryption D times to use the same key D times [3]. This is not that different from using D keys in our system, so the system described does not solve the issue.

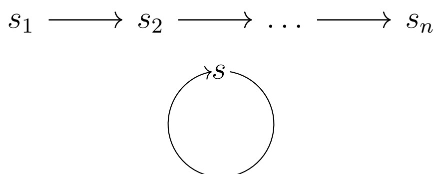


Figure 2.1: Assuming circular security we can work with one key instead of n keys.

2.3 Noise management

To show correctness of the scheme, we have to show how the noise behaves.

We here go a bit in detail in how the different Gaussian distribution we draw elements from are created and properties of them.

2.3.1 The probability distribution

We first consider the same polynomial ring, but with real coefficients $R_{\mathbb{R}} = \mathbb{R}[X]/\langle\Phi_m(X)\rangle$. Let us consider an element $a(X) \in R_{\mathbb{R}}$ where

$$a(X) = \sum_{i \in I} a_i X^i$$

for some index set I, where each a_i is a real-valued random variable with 0 mean and variance σ_i^2 and all the a_i are independent. Let ω be a primitive mth root of unity and consider the complex random variable $a(\omega)$. Denoting the variance σ^2 we get

$$\begin{aligned} E[a(\omega)] &= E[\sum_{i \in I} a_i \omega^i] = \sum_{i \in I} E[a_i]\omega^i = 0 \\ \sigma^2 &= E[a(\omega) \cdot \overline{a(\omega)}] = E[\sum \sum_{i,j} a_i a_j \omega^{i-j}] \\ &= \sum_i E[a_i^2] = \sum_i \sigma_i^2 \end{aligned}$$

Now let us set this index to be $I = [m]$. We can model $a(\omega)$ as being drawn from a complex normal distribution with variance σ^2, which is the same as a 2-D normal distribution with variance $\sigma^2/2$. We can do this because we assume that the complex roots ω^i are distributed evenly along the unit circle. This holds even if the index

set is a subset of I, provided the roots are still evenly distributed. This is expected to be the case if we randomly select indices with the same probabilities. The probability that any term exceeds B is then

$$\Pr[|a(\omega)| > B] = \exp(-B^2/\sigma^2)$$

If we set our bound as $B = \sigma\sqrt{\log(\phi(m))}$ we get that $\Pr[|a(\omega)| > B] = \frac{1}{\phi(m)}$. Since the coefficients of a are real, we know that $a(\overline{\omega}) = \overline{a(\omega)}$. Thus we get that $|a(\omega)| > B$ if and only if $|a(\bar{\omega})| > B$, which means we only need to account for half the probabilities. Thus if we account for all the probabilities and by applying the union bound we get that

$$\Pr[\|a\| > B] \leq \frac{1}{\phi(m)} \cdot \frac{\phi(m)}{2} = \frac{1}{2}$$

We can use this to bound various parts of our cryptosystem. The bound B changes depending on the variance for any given application. For our applications we often want to draw a polynomial where all the coefficients are drawn from a discrete Gaussian distribution. We can determines the standard deviation of this distribution by setting $\sigma = \sqrt{m}\hat{\sigma}$ for a tuneable parameter $\hat{\sigma}$. From this we get the bound $B = \hat{\sigma}\sqrt{m\log(\phi(m))}$, which we will call B_{gauss}.

2.3.2 Noise bounds

The noise associated with key generation

We choose the secret key according to the following method. Let $I \subseteq [m]$ be a random subset such that each index is chosen with probability

$$\alpha = \frac{\phi(m)}{2m}$$

and construct the secret by letting $a_i \in \{\pm 1\}$ and setting

$$s = \sum_{i \in I} a_i X^i$$

Then the variance is $\sigma^2 = \phi(m)$ and the bound on $\|s\|$ is $B = \sqrt{\phi(m)\log(\phi(m))/2}$. We now have a maximum bound on the the secret key B_{sk} such that $\Pr[\|s\| > B_{sk}] \leq 1/2$, which we want as close to $1/2$ as possible. Then we can guarantee that $\|s\| < B_{sk}$ if we sample keys until we get a key within the bound. This looses essentially one bit of security, but guarantees us that we have a key of small size instead of it being a high probability.

We can bound the public key in a similar way. The public key is essentially the same as an encryption of 0 (mod p) by the secret key $(1, s)$. We can bound the added error term e by the bound of the probability distribution it is taken from χ, which we call B_{gauss}. Here the variance can be chosen according to application.

The error is multiplied by the plaintext modulus p, and as such the bound on the public key noise is $B_{pk} = pB_{gauss}$. This is a bound on the error term $\|pe\|$.

The noise associated with encryption

For the encryption, we use the public key (a, b) and some parameters f, g from Gaussian distributions. The noise of an encryption is $\|p(re + fs + g) + m\|$. We can bound this noise by $B_{enc} = B_{pk}B_{small} + pB_{gauss}(B_{sk} + 1) + B_{ptxt}$. Here B_{small} is exactly the same as B_{sk}, as they are generated in the same way, we just denote them differently to emphasise the special role of the secret key.

The noise associated with key switching

The extra noise associated with key switching is bounded by p times the sum of the coefficient of the error term $\mathbf{e}$ which is "fresh" noise. If

each term is bounded by B_{gauss} then the key switch noise is bounded by $p\sum_i B_{gauss}$.

The noise associated with modulus switching

We get two new noise terms, the scaled noise $q/Q\|e\|$ of the original noise, and an added term which is bounded by $\|e'\|\beta$ if we set the bound on the secret key $\|s\|$ to be β. We also know $\|e'\|$ is not too big, since the coefficients of e' are in $[-p/2, p/2]$.

The noise associated with addition and multiplication

If we have a bound B of the noise on two ciphertexts, then the bound the noise of their sum is $2B$. If we have a bound B of the noise on two ciphertexts, then the bound the noise of their product is B^2.

The noise associated with Galois automorphisms

If we apply the Galois automorphism θ_j on the plaintext a where $j \in \mathbb{Z}_m^*$ we essentially permute the slots in the canonical embedding and so we get that $\|\theta_j(a)\| = \|a\|$. If the ciphertext $\mathbf{c}$ has the secret key $\mathbf{s}$ then the ciphertext $\theta_j(\mathbf{c})$ has secret key $\theta_j(\mathbf{s})$.

The noise associated with the HE.Refresh routine

If we have the ciphertext $\mathbf{c}$ with noise bound B we see that the expanding step does not affect the noise. The expanded ciphertext $\mathbf{c}_1$ then has the same noise bound as $\mathbf{c}$.

We look at how modulus switching is used in the HE.Refesh procedure. The noise of the new ciphertext $\mathbf{c}_2$ is at most $(q_{j-1}/q_j)B + 2p\phi(m)\lceil\log q_j\rceil B_{gauss}$ where the factor $4\lceil\log q_j\rceil$ is because this is the number of entries the bit decomposed secret key $\mathbf{s}_j''$.

After that we have the key switching step. If the ciphertext $\mathbf{c}_2$ is has noise B_1 then the new ciphertext has noise $B_1 + p \cdot \lceil \log q_j \rceil^2 \cdot B_{gauss}$, where the factor $\lceil \log q_j \rceil^2$ is because this is the number of entries in the bit decomposition matrix of $\mathbf{c}_2$.

Adding all this together we see that if the noise of the original ciphertext is B then the new noise is $(q_{j-1}/q_j)B + \epsilon_{mod} + \epsilon_{key}$ after applying the refresh routine, where $\epsilon_{mod} = 2p\phi(m)\lceil \log q_j \rceil B_{gauss}$ and $\epsilon_{key} = p \cdot \lceil \log q_j \rceil^2 \cdot B_{gauss}$.

Now with this information we can say something on how we build the modulus chain and by extension the bounds on the ciphertexts. We set up our modulus ladder and noise bound such that the following properties hold:

$$B \geq 2(\epsilon_{mod} + \epsilon_{key}) \tag{2.1}$$

$$(q_j/q_{j-1}) \geq 2B. \tag{2.2}$$

Let us say we apply the refresh routine after a multiplication. The bound on the noise after multiplication is B^2. After a refresh we have that $(q_{j-1}/q_j)B^2 + \epsilon_{mod} + \epsilon_{key} \leq 1/2 \cdot B + 1/2 \cdot B = B$. Thus with equation 2.1 satisfied we get a new ciphertext after the multiplication and the HE.Refresh routine with the same noise bound as the factor ciphertexts had.

2.4 Bootstrapping

Bootstrapping is an important technique for getting fully homomorphic encryption. We will give an overview of the idea behind bootstrapping. The bootstrapping procedure is essentially evaluating the decryption circuit homomorphically. In other words, computing the decryption without revealing any information to get a fresh ciphertext.

Say we have two key pairs (sk_1, pk_1) and (sk_2, pk_2) and a ciphertext c which is the encryption of m under pk_1. Let $\bar{sk_1}$ be the encryption

of the bits of sk_1 under the public key pk_2. Similarly, set $\bar{c}$ to be the encryption of the bits of c under pk_2. Then if we compute $c' = \mathrm{Eval}(pk_2, Dec, \bar{sk_1}, \bar{c})$ we get a new ciphertext c' which is an encryption of m under pk_2. Amazingly, the noise in the first ciphertext c disappears when we evaluate the decryption circuit. The evaluation of the decryption circuit introduces some noise, so the ciphertext c' has only the noise introduced from the evaluation. If this new noise is smaller than the first noise, we have made progress.

There is nothing inherent in using two different key pairs in this process. We could just as well use the same key pair twice, provided that the encryption of the secret key under itself is safe to publish, in other words that we have circular security. The circular security assumption and bootstrapping is used in many applications of homomorphic encryption [22, 5].

We have the following way of turning a somewhat homomorphic encryption scheme into a fully homomorphic encryption scheme: Let (G, E, D) be a CPA-secure somewhat homomorphic cryptosystem that has circular security. Suppose (G, E, D) can compute functions in a family of functions F homomorphically and that for every two ciphertexts c, c' the map $d \to D_d(c) \uparrow D_d(c')$ is in the family F, where $\uparrow$ is the logical NAND operation. Then we can turn (G, E, D) into a fully homomorphic encryption system. The reason the NAND gate is used is that it is *functionally complete*. This means that any boolean function $f : \{0, 1\}^n \to \{0, 1\}$ can be made using only NAND gates. Equivalently we could use other sets of boolean operators which are functionally complete, such as the set $\{NOT, AND\}$. These can be constructed through addition, subtraction and multiplication. We can thus use our leveled homomorphic encryption scheme given that there exists parameters that let us evaluate these operations in addition to compute the bootstrapping procedure.

If the crypto scheme we want to use has many levels and the size of the parameters grows too big, then bootstrapping can be used as

an optimisation. This means we have to do costly bootstrapping, but the parameters can be set smaller to compensate. Brakerski et al [2] did an analysis of when to use bootstrapping as an optimisation in the scheme.

The cryptosystem we have constructed allows for encryption of messages in the polynomial ring R_p. Data in the real world is seldom taken from this polynomial ring though. In the next chapter we will explore how we can convert data from other spaces into the polynomial ring and how this changes the computation we want to do.

Chapter 3

Implementation

When doing homomorphic computation, there are a lot of restrictions and choices to consider. All computation is more time consuming and the encrypted objects take more space than the unencrypted counterpart. We also have to limit multiplicative depth. Moreover, we have to consider how we represent numbers, how we do arithmetic and implementation of conditional statements. Furthermore, we have to choose parameters that give us the fastest implementation, and this is highly based on the computation we want to do. There is no general way of choosing optimal parameters. In short: implementation is complicated.

In this chapter we will discuss how we can represent numbers and add some details concerning the cryptosystem. This is important when doing computations in practice, as it decides how efficiently we can do any computation. This chapter is divided into two parts: generic properties and algebraic properties. The generic properties are properties which apply to most implementations, such as how we represent numbers and do arithmetic on them. The algebraic properties are specific ways of implementing our scheme, with focus on the plaintext structure and basic operation such as linear algebra operations.

3.1 Generic properties

The properties we will discuss in this section are more general than for our specific cryptosystem. Often the plaintext space is not the natural space to do specific computation in, because the numbers we want to compute on belong to another space. This is where encodings and the arithmetic on the encodings come in. Encodings are ways of representing numbers that are not actually in the plaintext space. The way we encode the numbers determine how we implement arithmetic on the numbers.

3.1.1 Encodings

Since we often want to compute on data that do not necessarily lie in our plaintext space, we have to find a way to represent the data in the plaintext space. We do this by an encoding. If we have a data space S (for example rational numbers, real numbers, etc.) and plaintext space $\mathcal{M}$ we can find an encoding $\pi : S \rightarrow \mathcal{M}^k$ that makes it possible to compute on our data . If we want to compute the function $f : S^n \rightarrow S$, then the emulated function $g : (\mathcal{M}^k)^n \rightarrow \mathcal{M}^k$ on the plaintext space is heavily dependent on the encoding π. Therefore the efficiency of the function g is also heavily dependent on π.

We stress that this encoding may not be perfect, for example if S is an infinite set and $\mathcal{M}$ is finite, then π cannot be injective and more than one element in S has to be encoded into the same element in $\mathcal{M}^k$ for any finite k. Since we are working in very particular plaintext spaces (often copies of finite fields), the evaluation of the emulated function g often becomes much less efficient than evaluation of the original function f. This holds even when working in the unencrypted plaintext domain. Jäschke and Armknecht showed that the evaluation of the perceptron is several orders of magnitude slower encoded in the plaintext space $\mathbb{F}_2$ than computing in the rationals numbers [15]. The

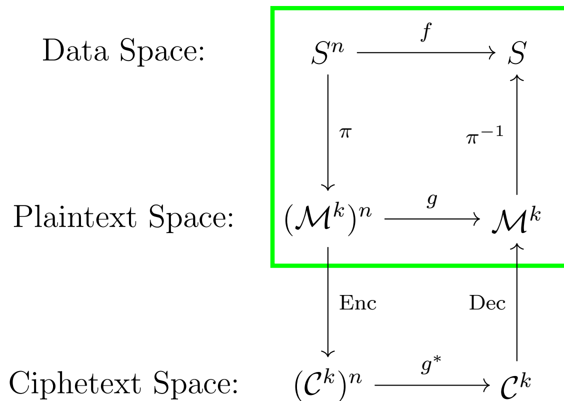


Figure 3.1: Illustration of how the emulated function g and encrypted variant g^* depends on the encoding π we use. In this chapter we will focus on the content of the green rectangle. The figure is based on one by Jäschke and Armknecht [15].

choice of encoding π is therefore crucial to efficiently implement more advanced computation. We now look at some different encodings and the benefits and drawbacks of each of them.

Word-wise encoding

One alternative is to encrypt *word-wise*. This is essentially encrypting an integer directly into the the plaintext space. Thus we limit ourselves to encryption of numbers less than the plaintext modulus p. This limits our computation in a big way, since the product of two integers (or more general polynomial functions of ciphertexts) cannot get bigger than the plaintext modulus without introducing errors. Thus we need a much bigger plaintext modulus than the integer we are working with in order for this approach to work. One way of mitigating this issue is to encode the integer as a polynomial instead of as an integer since the

plaintext space is a polynomial space. For a given integer value z we can first reduce it into the bit-representation $z = sign(z)(z_s \dots z_1 z_0)_2$ and then encode it as follows:

$$\begin{aligned} &\mathbb{Z} \to R_p, \\ &z \mapsto \mu_z = sign(z)(z_0 + z_1 x + \cdots + z_s x^s) \end{aligned}$$

To turn μ_z back to z we simply evaluate the polynomial at $x = 2$. This gives a lot more flexibility, because now each bit is essentially in $\mathbb{Z}_p$, and so each bit can grow to be as big as p. The good thing about word-wise encryption is that every arithmetic operation is baked into the operations of the cryptosystem.

A drawback of this encoding is that we still have the limit on arithmetic operations by the plaintext modulus, we cannot do a sum of p numbers for example. The plaintext modulus in the BGV-scheme influences so much of the other parameters, from ciphertext modulus to key switching noise, so having a high plaintext modulus reduces efficiency. Another drawback is that multiplication increases the degree of the encoded polynomials, and if the degree becomes higher than the maximum degree (which is $\phi(m)$), it will wrap around and introduce errors. Multiplying two polynomials of degree s gives polynomial of degree $2s$. Thus we quickly introduce errors with this encoding.

Digit-wise encoding

Digit-wise encoding of a number z consists of simply encoding each digit of the n-ary representation of z separately. The positive aspect of this is that we can encode as many digits as we want into different plaintexts without depending on the plaintext modulus or degree of the quotient polynomial. The downside is that this is much less space efficient, as encrypting many plaintexts uses much more space than

encrypting a single plaintext. Another downside is that we have to implement arithmetic for how the digits of the number interact when adding or multiplying, which is less time efficient.

We still have to choose how many plaintext spaces we will use before we encode the numbers. This is since the numbers are encrypted, we cannot know when the number becomes so large as to require more digits. Thus we must choose a way of cutting of digits when we have a too large number. The way we do this depends on the arithmetic we want to use.

Packed encoding

The *packed encoding* is a way of combining the other encodings which is somewhat specific to our cryptosytem. It utilises the structure of R_p, more precisely it uses the fact that R_p can be thought of as a product of finite fields: $R_p \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$. We can encode the digits of a number into the different slots. This combines the upside of the word-wise encoding's use of a single copy of the plaintext space R_p and the flexibility of digit-wise encoding. It also utilises the SIMD structure of R_p.

To use this encoding we need to implement arithmetical operations on the digits. This has an extra level of difficulty compared to the digit-wise encoding, as doing computation on the digits within a packed plaintext requires some extra effort. We describe how in detail in Section 3.2.1. The upside is that we can do l additions or multiplications at ones with a single addition or multiplication, and thus significantly reduce the number of operations.

Example 3.1. Let $m = 2^{15} - 1$ and $p = 2$ so that we have 1800 slots. Let us write $\mathbb{F}_{2^{15}} \simeq \mathbb{F}[\omega_k]$ for slot k. Then if we want to represent 15-bit integers we can put each bit j of integer k as $\omega_k^j \in \mathbb{F}[\omega_k]$ and thus represent 1800 integers using this packed encoding. An alternative is to encrypt a single bit into each slot, so to represent the same

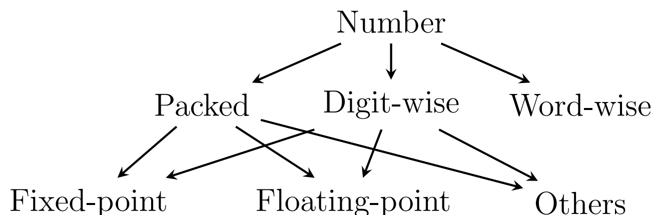


Figure 3.2: An overview of the decisions that go into an encoding. There are more nuances, such as how there are different ways to represent non-integers and negative numbers. Different encodings give different arithmetic.

represent 15-bit integers 15-bit integers we need 15 ciphertexts, where ciphertext j contains bit j of all the integers. The reason we might use more ciphertexts is that it can make the arithmetic faster, as we will see in Section 3.1.2.

Both the digit-wise and packed encodings encrypts the digits of the number we want to encrypt. Therefore we have to implement the arithmetic of the digits and show how they interact with each when we add and multiply numbers. There are different methods of representing the same number in terms of digits. We sketch some different *representation* for numbers. These representations are essentially the same as encodings, just that we look at how we encode integers into a finite field instead of into the plaintext space. The choices involved when choosing encodings is sketched in Figure 3.2.

Fixed point representation

The fixed point representation of an integer z is a way of representing it in digits. We can split z into digits z_i with a specific base r called the *radix* so that $z = \sum_{i=0}^{s} z_i r^i$. It is often practical to let the radix be the same as the plaintext modulus, i.e. $r = p$. The radix is an important

decision, where the main difference is between setting $r = 2$ and $r > 2$. In $\mathbb{F}_2$ the computation is much simpler than in other finite fields, so we usually use $r = p = 2$. We then call the digits for *bits*.

To represent negative numbers with bits we can use the *twos complement* encoding: $z = z_{s+1}(-2^{s+1})\sum_{i=0}^{s} z_i 2^i$. For general digit encodings we can use p's complement, where for $z_{s+1} \in \{0, 1\}$ and $z_i \in [p]$ for $i < s + 1$ we get $z = -z_{s+1} \cdot p^{s+1} + \sum_{i=0}^{s} z_i p^i$.

If we want to represent rationals, we can scale the rational by a scaling factor p^e to make it an integer, do arithmetic with integers and scale back after the computation. This causes a complication when we multiply two numbers. Let $a = a' \cdot p^e$ and $b = b' \cdot p^e$ where a', b' are rationals. Then $a \cdot b = (a' \cdot p^e) \cdot (b' \cdot p^e) = a'b' \cdot p^{2e}$ and so when we scale back with the factor p^e we get the wrong answer. What we can do instead is to cut the last e digits after every multiplication, to do the scaling in the ciphertext domain directly. Then we get a number on the correct form.

Floating point representation

The *floating point representation* is another way of representing a number using bits. In floating point representation the numbers are represented as some mantissa m times some signed exponent exp to get a number $\pm m \cdot 2 \pm exp$. The magnitude of the mantissa is usually normalised to the range $[0.5, 1]$ since we otherwise could shift the exponent to get values in this range. This is not as easy with homomorphic encryption, as we have no way of knowing what the magnitude of the mantissa is. The reason they are called floating point, is that the point "floats" depending on the exponent.

The floating point representation and floating point arithmetic cuts of the least significant bits (LSB) after arithmetic operations. There is no known way of representing this cutoff operation as a low degree polynomial, so floating point arithmetic is not that practical. There

are other cryptosystems where the floating point arithmetic is designed into the system. Chen et al. [4] designed a cryptosystem based on approximate arithmetic, where the floating point representation is much more natural than with the BGV cryptosystem.

There are many other representations one can use, but we limit the scope to describe these representations. We will focus on the arithmetic of the fixed point representation, and discuss how we can compute it the most efficiently using the packed encoding.

3.1.2 Arithmetic

Before we show how we can do arithmetic, we mention the important factors to keep in mind while looking at a computation. There are two main properties we need to keep in mind when evaluating a circuit: the time complexity of the evaluation and the depth of the circuits. The depth of multiplications is much more important than the depth of additions because it uses levels in our cryptosystem. Additions are much faster than multiplications, so multiplication dominates in run time as well. Thus our focus will in general be the number of and depth of multiplications.

We restrict to working with the base $p = 2$ in this section, as this significantly simplifies both the algorithms and their explanation. We call the digits in these computation for bits. Jäschke and Armknecht [16] proved that working in binary gives the most efficient arithmetic of any finite field $\mathbb{F}_q$. Their frame of reference was packing a single message into the plaintext space. Thus they did not account for the packed encoding, since it is somewhat (but not entirely) specific to the BGV scheme which we are working with. For certain parameters and certain computation it might in some cases therefore be more efficient to use a different base. We restrict to binary in our discussion both for simplicity, and because it is the most efficient method in general.

Addition

Arithmetic with the digit-wise and packed encodings is as mentioned more complicated than with the word-wise encoding. For example, if we encrypt two numbers bit wise, addition of each bit is dependent on the addition of the previous bit. We first illustrate how we can add with the naive *Ripple Carry Adder (RCA)*, which is similar to the addition technique taught to children in grade school. After this we show a more efficient implementation, which utilises the parallel structure of the packed encoding. The RCA outputs the sum $s = s_{n+1} \dots s_0$ of two numbers $a = a_n \dots a_0, b = b_n \dots b_0$ where the s_i gets computed in the following way:

$$s_i = \begin{cases} a_i + b_i & \text{if } i = 0 \\ a_i + b_i + c_{i-1} & \text{if } i \neq 0 \end{cases}$$

where

$$c_i = \begin{cases} a_i \cdot b_i & \text{if } i = 0 \\ (a_i \cdot b_i) + ((a_i + b_i) \cdot c_{i-1}) & \text{if } i \neq 0. \end{cases}$$

Since each c_i is dependent on the previous c_{i-1}, this computation is difficult to parallelise. It also has high multiplicative depth. Although one can rearrange the carry operation to get one multiplication for each bit by noticing that $(a_i \cdot b_i) + ((a_i + b_i) \cdot c_{i-1}) = (a_i + c_{i-1}) \cdot (b_i + c_{i-1}) + c_{i-1}$, this still leaves n multiplications and multiplicative depth of n. We can reduce this by implementing more efficient adders.

We can do better than this by using a dynamic programming approach. To do this we introduce generator carries g_i and propagator carries p_i. These are defined for bits as $g_i = a_i b_i$ and $p_i = a_i + b_i$, which we notice can be computed in parallel. We extend these carries to generator intervals $g_{[i,j]}$ and propagator intervals $p_{[i,j]}$ which we

define in the following way:

$$p_{[i,j]} = \prod_{k=i}^{j} p_k$$
$$g_{[i,j]} = g_i \prod_{k=i+1}^{j} p_k$$
$$c_j = \sum_{i=0}^{j} g_{[i,j]}.$$

The key decision in implementation is how we decide to implement the computation of the carry bits c_j. There are many different adder designs, but the one we use here is the *Kogge-Stone Adder (KSA)*. The KSA utilises a so called *carry operator* that produces propagator intervals and sums of generator intervals by computing

$$(G, P) = (G'' + G'P'', P'P'')$$

for inputs (G', P') and (G'', P''). We arrange these carry operators in a directed acyclic graph (DAG) so that we compute these operators in a very parallel manner. An illustration for 8-bit inputs can be seen in Figure 3.3.

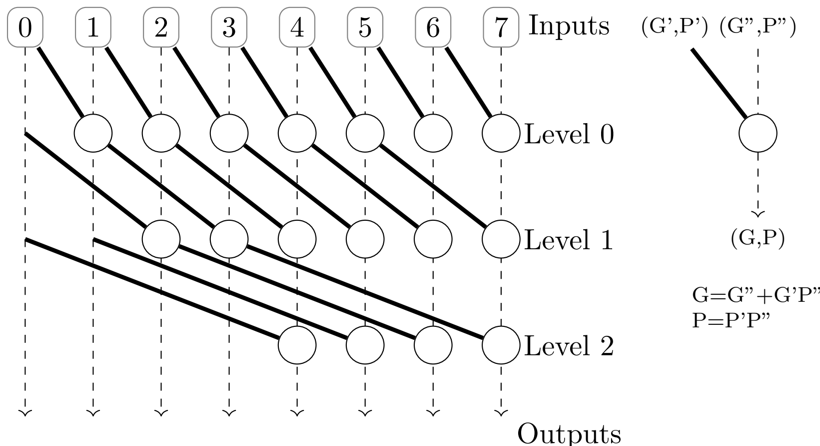


Figure 3.3: An illustration of the KSA for 8-bit inputs, which illustrates how we can compute the intervals. The illustration is based on one from Basilakis and Javadi [1].

This has much lower multiplicative depth than the RCA. The highest number of multiplications is in the $(n+1)$-product $g_{[0,n-1]} = a_0 b_0 \cdot \prod_{k=1}^{n-1}(a_k + b_k)$, which has depth $\lceil \log_2(n+2) \rceil$. This adder illustrates why the packed encoding is so helpful. In the bit-wise encoding the adder does roughly $2n \log_2(n)$ multiplications. In the packed version, the number of multiplications is two per level, so the total number of multiplications is $2\lceil \log_2 n \rceil$. Note that in order to be able to multiply and add together in the packed encoding, we have to shift around the bits. This is somewhat expensive in time complexity, but has the benefit of not increasing the multiplicative depth. We discuss how we can do these shifts in Section 3.2.1.

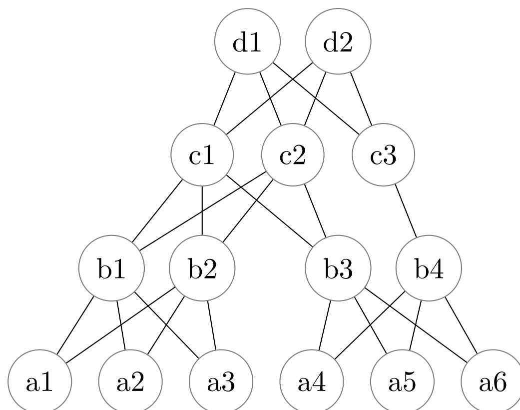


Figure 3.4: The three-for-two procedure used on six numbers $a_1, \ldots, a_6$ to reduce them into two numbers d_1, d_2.

Adding more than two integers

We introduce the three-for-two procedure to add three numbers and convert them to two numbers that sum to the same number. The concept is that we add all three numbers together at the same time. From this we get a number which represents the sum and one number which represent the carry. If we have three numbers u, v, w, then the we get two new numbers x, y where $x_i = u_i + v_i + w_i$ and $y_i = u_i v_i + v_i w_i + w_i u_i$. These two numbers satisfy $u + v + w = x + 2y$ over the integers. These new numbers can be computed using only two multiplications and multiplicative depth one, as $y_i = (u_i + w_i)v_i + w_i u_i$.

We can use this procedure to add k integers together by placing the numbers in a graph like illustrated in Figure 3.4. We place the k numbers at the lowest level and we use the three-for-two procedure to go higher towards the top. This gives multiplicative depth $d \approx \log_{3/2}(k)$ and $2d$ total multiplications.

Multiplication

Let a be an n'-bit number and b be an n-bit number and say $n \geq n'$. Then multiplication of a and b is done by multiplying each bit a with b and then adding the result. For example, if we set $n' = 3$ and we multiply a, b we get

$$\begin{array}{rcccccccc} a_0 b: & 0 & 0 & a_0 b_n & \dots & a_0 b_2 & a_0 b_1 & a_0 b_0 \\ 2a_1 b: & 0 & a_1 b_n & a_1 b_{n-1} & \dots & a_0 b_1 & a_1 b_0 & 0 \\ 2^2 a_2 b: & a_2 b_n & a_2 b_{n-1} & a_2 b_{n-2} & \dots & a_2 b_0 & 0 & 0 \end{array}.$$

We then add up the integers using integer addition. If we want to use negative integers, we have to compute what is known as a sign extension. Essentially, it increases a k-bit number to a l-bit number for $l > k$ by letting all the bits that have index greater than k be the same as the kth bit (the sign bit). To compute multiplication of two numbers a, b which are n, n' bits respectively in twos complement encoding, we do the following: First we do sign extension on both numbers to a $n + n'$ bit number. Then we multiply as normal, but only keeping the last $n + n'$ bits of each partial product. We then add the numbers as usual.

We note that there are other multiplication algorithms with lower multiplicative depth, such as the Karatsuba algorithm, which was implemented homomorphically by Fontaine et al. [20]. These algorithms are more complicated, and we limit our scope to the simpler implementation to not get too far off track.

3.1.3 Comparisons and conditionals

When comparing unencrypted bit-wise encoded data we can check the bits from most significant to least significant and halt when the bits do not coincide. Comparisons between encrypted entries is harder than between unencrypted entries since we cannot see the result of

the comparisons. Therefore we have to compare all the bits when we do the comparison, and not just the iterate down the most significant bits until we find a difference.

Comparison of two integers can be done in a similar way as the addition circuit. Given we have two integers $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$ we want to compute $x = \max(a, b)$ and $y = \min(a, b)$, as well as the indicator bits $\mu = (a > b)$ and $\nu = (a < b)$. We also want to compute the equality circuit, testing whether $(a = b)$. Similarly as in addition we compute the indicator bits

$$
\begin{aligned}
e_i &= a_i + b_i + 1 \\
g_i &= a_i + a_i b_i \\
e_i^* &= \prod_{j \geq i} e_j \\
g_i^* &= g_i \prod_{j > i} e_j \\
\tilde{g}_i &= \sum_{j \geq i} g_j^*
\end{aligned}
$$

The indicator e_i determines if $(a_i = b_i)$, while g_i determines if $a_i > b_i$. The indicator $\tilde{g}_i$ determines if $a_{n-1,\ldots,i} > b_{n-1,\ldots,i}$ for the truncated versions of a, b. Note that computing $\tilde{g}_i$ is somewhat similar to computing the carries in the addition. We can similarly optimise the computation of all the indicator bits to get an effective solution to our problems. The equality circuit is the easiest to compute, as it can be computed simply as $eq(a, b) = \prod_{i=0}^{n-1} e_i$. This can be computed with $\log(n)$ multiplications and depth $\log(n)$ by employing shifts in the packed encoding. The others require that we use nearly all the indicator bits. We compute $\mu = \tilde{g}_0$ and $\nu = 1 + \tilde{g}_0 + e_0^*$ and set $x_i = (a_i + b_i)\tilde{g}_i + b_i$ and $y_i = x_i + a_i + b_i$. Of course we can modify these operations when we do not need to compute both the maximum and the minimum to not

be expressed in terms of each other. Similarly to addition, computing the comparison bits requires $\log(n)$ multiplications and multiplicative depth, and we need an extra multiplication to get the maximum or minimum. The digit-wise and packed encodings are better suited for comparisons. Here we require multiplicative depth of $\log(n)$, while comparisons in the word wise encoding comparison requires multiplicative depth of $O(t)$ [21].

We now have a method for encoding data into our plaintext space and how we can do arithmetic. These are so called generic properties that we have to consider for any cryptosystem. Next we describe how we can use the specific properties of the BGV cryptosystem to make computation faster.

3.2 Specific properties

In this section we will explore some specific properties of using the BGV cryptosystem. In particular we highlight some specific properties from the implementation of HElib[11, 13]. This is a homomorphic encryption library for C++ which implements the BGV cryptosystem. Remember that our plaintext space $R_p \simeq \mathbb{F}_{p^d} \times \cdots \times \mathbb{F}_{p^d}$ so we can view our plaintext as having multiple slots. In this section we explore the structure in more detail. We also look at the structure of the ciphertext space and include a discussion on how to choose parameters.

3.2.1 Moving between the slots

The space R_p corresponds to l slots and that we can use Galois automorphisms $\theta_j : f(X) \mapsto f(X^j)$ to move between the slots for $j \in \mathbb{Z}_m^*/\langle p \rangle$. We now describe how these movements work in more detail.

One-dimensional rotations

Say that $g \in \mathbb{Z}_m^*$ and that $1, g, \ldots, g^{l-1}$ is a complete set of representatives of the cosets of H. We know that $g^n \in H$. If we are lucky, then $g^n = 1$. Then the automorphism θ_g acts as a rotation to the left of the plaintext slots:

$$\begin{aligned} f(x) &\longleftrightarrow (f(\eta^1), f(\eta^g), \ldots, f(\eta^{g^{l-1}})) \\ \theta_g(f(x)) &\longleftrightarrow (f(\eta^g), f(\eta^2), \ldots, f(\eta^{g^l})) \\ \theta_g(f(x)) &\longleftrightarrow (f(\eta^g), f(\eta^2), \ldots, f(\eta^1)). \end{aligned}$$

The last line follows from $g^l = 1$. We see that in this case the automorphism θ_g acts as a rotation of the slots one step to the left. To generalise this, we look at the automorphism θ_{g^j} which rotates the slots j places to the left and $\theta_{g^{-j}}$ which rotates the slots j places to the right.

If we are not so lucky, then $g^l \in H$, but is not equal to one. In this case $g^l = p^s \in \mathbb{Z}_m^*$ for some $s \in \{1, \ldots, d-1\}$. This is the same as applying the transformation $f(X) \mapsto f(X^p)$ a total of s times. This transformation $f(X) \mapsto f(X^p)$ is known as the *Frobenius automorphism* and we denote it σ. This is an important automorphism which fixes all elements of $\mathbb{Z}_p \subseteq \mathbb{F}_{p^d}$.

Applying θ_g rotates most of the slots, except for the last slot, which is "perturbed" by a Frobenius-power of degree s. In other words:

$$\theta_g(f(x)) \longleftrightarrow (f(\eta^g), f(\eta^2), \ldots, \sigma^s(f(\eta^1))).$$

We can still obtain a regular rotation of the slots from these transformation. We construct a *masking element* M_e which is one in the first e slots, and is zero on the last $l-e$ slots.

$$M_e \in R_p \longleftrightarrow (1, \ldots, 1, 0, \ldots, 0) \in E^n$$

Say a plaintext $a \in R_p$ corresponds to the slot plaintext $(\alpha_0, \dots, \alpha_{l-1})$. Then applying the automorphism θ_{g^e} and multiplying with M_e gives us

$$M_e \cdot \theta_{g^e}(a) \longleftrightarrow (\alpha_e, \dots, \alpha_{l-1}, 0 \dots, 0)$$

Furthermore, we have

$$(1 - M_e) \cdot \theta_{g^{e-l}}(a) \longleftrightarrow (0, \dots, 0, \alpha_0, \dots, \alpha_{e-1})$$

Combining these two gives us the desired rotation by adding them together. In other words, we can rotate e slots by applying the transformation

$$(M_e \cdot \theta_{g^e}(a)) + (1 - M_e) \cdot \theta_{g^{e-l}}(a)$$

to an element in R_p. We can achieve the same transformation by first applying the masking and then the automorphism

$$\theta_{g^e}((1 - M_{l-e}) \cdot a) + \theta_{g^{e-l}}(M_{l-e} \cdot a)$$

The general hypercube

Up until now we have viewed the slots structured as essentially a vector. In general, we can get a multidimensional array which we call a *hypercube* structure. The shape of the hypercube is determined by the structure of the quotient group $\mathbb{Z}_m^*/H$. The number of dimensions r of the hypercube is determined by the factorisation of $l = |\mathbb{Z}_m^*/H|$. The *length* of each dimension is determined by the relevant factor. The term hypercube is used, but the length of each dimension can in general be different (making it a hyperrectangle in a sense). The reason we want to utilise the hypercube structure is that it gives us more flexible movement between slots, and it lets us represent other data structures such as a matrix.

If $l = l_1 \dots l_r$, then a complete set of representatives of the cosets of H are of the form $g_1^{e_1} \dots g_r^{e_r}$ where $e_i \in [l_i]$. Each such representative

corresponds to a plaintext slot. If we keep all indices except e_i fixed, we are looking at a *hypercolumn*. If we keep only the index i fixed, we say we have a *slice* of the hypercube. An advantage gained from representing the plaintext in this hypercube structure is that we can maneuver between different slots in more ways than if we represent the plaintext as a vector. We should note that even though $\mathbb{Z}_m^*/H$ might give a high dimensional hypercube structure, there are ways to consider it as a linear array and implement the linear rotations from the hypercube rotations. We refer to [11] for details.

Example 3.2. Let $m = 2^{15} - 1$ and $p = 2$. The prime factorisation of m is $7 \cdot 31 \cdot 151$. Therefore the structure of $\mathbb{Z}_m^* \simeq \mathbb{Z}_7^* \times \mathbb{Z}_{31}^* \times \mathbb{Z}_{151}^* \simeq \mathbb{Z}_6 \times \mathbb{Z}_{30} \times \mathbb{Z}_{150}$. It is possible to show that $\mathbb{Z}_{2^{15}-1}^*/\langle 2 \rangle \simeq \mathbb{Z}_{30} \times \mathbb{Z}_6 \times \mathbb{Z}_{10}$. This does not have the same structure as $\mathbb{Z}_{1800}$, so we cannot give the slots the structure of a linear array as there are no single generators. Thus the slots have the hypercube structure of dimensions $30 \times 6 \times 10$.

Say we have $l = l_1 \cdot \dots \cdot l_r$ with generating set $g_1, \dots, g_r$. Then we move in dimension j by applying the automorphism associated to g_j, namely $\theta_{g_j^e}$ when we want to move e steps in the left direction. In the case where we get a regular rotation by applying θ_g, we say that it is a *good* dimension. If $g_j^{l_j} \neq 1$, but $g_j^{l_j} \in H$, then we say that we have a *bad* dimension. In this case, all the slots that are "wrapped around" are perturbed by a Frobenius power. If we do not even have this, we say we have a *very bad* dimension. In this case, the wrapped around slots are perturbed by a Frobenuis power, and then permuted in the slice corresponding to the dimension. We can choose appropriate parameters so that we can avoid very bad dimensions, and only use good and bad dimensions in the implementation. We can do

so with the following algorithm

Algorithm 1: Choosing generators for hypercubes in HElib

$H_0 \leftarrow H$
$i \leftarrow 1$
while $H_{i-1} \neq \mathbb{Z}_m^*$ **do**
  let l_i be the maximal order of any element in $\mathbb{Z}_m^*/H_{i-1}$
  choose $g_i \in \mathbb{Z}_m^*$ such that
    (a) the order of $g_i \mod H_{i-1}$ is l_i, and
    (b) $g_i^{l_i} \in H$ (and if possible, $g_i^{l_i} = 1$)
  let H_i be the subgroup generated by $H_{i-1} \cup \{g_i\}$
  $i \leftarrow i + 1$
return $\{g_i\}_{i \in [r]}$

Proposition 3.3. *By using algorithm 3.2.1 we can find a generating set $g_1, \dots, g_r$ such that all dimensions are either good or bad, but not very bad.*

Proof. Notice first that $g_1, \dots, g_r$ is indeed a generating set, regardless of condition b). Since l_i is the maximal order in $\mathbb{Z}_m^*/H_{i-1}$, it is also the exponent. From standard results about exponents and quotient groups, we get that $l_i | l_{i-1} | \dots | l_1$. Let i be the index of the generator g_i. If $i = 1$, there is nothing we can do. If $i \geq 2$, we can find a new generator g_i' such that i becomes at least a bad dimension. Choose g_i such that it has order l_i. We know that $g_i^{l_i} \in H_{i-1}$, which means that $g_i^{l_i} = g_{i-1}^s h$ for some $h \in H_{i-2}, s \in \mathbb{Z}$. We know that $g_i^{l_{i-1}} \in H_{i-2}$, so we get

$$H_{i-2} \ni g_i^{l_{i-1}} = g_i^{l_i \frac{l_{i-1}}{l_i}} = g_{i-1}^{s \frac{l_{i-1}}{l_i}} h'$$

for some $h' \in H_{i-2}$. This implies that $g_{i-1}^{s \frac{l_{i-1}}{l_i}} \in H_{i-2}$, which in turn implies that $l_i | s$ since l_{i-1} is the order of g_{i-1}. We now set $g_i' = g_i \cdot g_{i-1}^{-s/l_i}$ as the new generator. Since $g_{i-1}^{-s/l_i} \in H_{i-1}$, we see that g_i, g_i' are in

the same coset of H_{i-1}. They therefore have the same order, and in addition $(g_i')^{l_i} = g_i^{l_i} \cdot (g_{i-1}^{-s/l_i})^{l_i} = g_{i-1}^s h g_{i-1}^{-s} = h \in H_{i-2}$. We can do this process inductively until we get $(g_i^*)^{l_i} \in H_0 = H$. Thus we get a generator with at least not a very bad dimension. □

In the same way as with one-dimensional rotations, we can use multiplicative maskings to get rotations even in bad dimensions. Thus where the cost of a rotation in a good dimension is a single application of a Galois automorphism, we need two Galois automorphisms, two maskings and an addition in a bad dimension to make a rotation. This is more than twice as time consuming, and may require using another level in the modulus chain.

More complicated movement

Once we have the ability to rotate whole or parts of a plaintext we can use these rotations to implement arbitrary permutation. The way a given permutation should be constructed from rotations and shifts is not immediately clear however, as there are many possibilities of how to combine the rotations into a given permutation. The implementation of HElib [11] uses *shift networks* to construct any given given permutation from additions, rotations and multiplicative maskings.

3.2.2 Other properties

We list some other properties of the specific implementation we have here. We consider the structure of the ciphertexts and a method for choosing specific parameters.

Ciphertexts

The ciphertexts c are in general in the space R_q^n, and most often in R_q^2. We will now look a little more into the structure of R_q. One way of

representing an element r in R_q is through the coefficient embedding, where we look at r as a vector of the coefficients. This has the drawback that multiplication is not done component wise, and therefore multiplication of ciphertexts is inefficient. This is the representation of ciphertexts when they get encrypted, and the representation we need for decryption.

Another way of representing R_q involves the prime factorisation of q. Say that $q = \prod_{i=0}^{k} q_i$ where the q_i are relative prime. Then we need that the primes q_i are such that $\Phi_m(X)$ factors modulo q_i into linear terms of the form $(X - \omega_i^j)$ for $j \in \mathbb{Z}_m^*$ and primitive mth root ω_i. Then we can form a representation for the element $a \in R_q$ as

$$a(X) \leftrightarrow (a(\omega_i^j) \mod q_i)_{(i,j)\in[k]\times\mathbb{Z}_m^*}$$

This representation is called the *DoubleCRT representation*, as it uses the Chinese Remained Theorem both on the polynomial $\Phi_m(X)$ and the integer q. This representation has the benefit that both addition and multiplication is done component wise. The drawback is that encryption and decryption is done naturally through the coefficient embedding, so we have to convert in order to decrypt.

One can convert from the coefficient representation to the DoubleCRT representation by reducing the coefficients and evaluating the polynomial at the powers of the primitive roots. One can convert back by interpolating the polynomial based on the values $a(\omega_i^j) \mod q_i$. Both evaluating and interpolation can be done by applying the Fast Fourier Transform (FFT). This is not very efficient and should be avoided, even with FFTs [13].

Choosing parameters

Choosing secure parameters is a research field of its own. The way secure parameters are set is by looking at the best known attacks and

choosing parameters that are secure against them. Of course, this is not a guaranteed safe way of choosing parameters, as there could be unknown attacks that make the parameters less secure than assumed. Since the RLWE-assumption (and related LWE-assumption) has the potential of being a foundation for post-quantum cryptography, there is a lot of research into attacks. An analysis of how to choose safe parameters can be found in [9, Appendix C.3]. This gives us a way of choosing $\phi(m)$ in terms of the number of levels L and the security level k by

$$\phi(m) > \frac{(L(\log(\phi(m)) + 23) - 8.5)(k + 110)}{7.2}.$$

The scheme must be practical in addition to being secure. We would like to get a hypercube structure that has both suitable dimension sizes and good dimensions for a given implementation. There may not be that many parameters which are above the security threshold, gives the correct hypercube structure and results in relatively small parameters. Therefore we may have to choose whether to settle for parameters which give a slightly different hypercube structure, or to have large and sub-optimal parameters.

3.3 Overview of useful techniques

In this section we consider some computation techniques which are crucial when we want to compute with the packed encoding. We also consider different ways of doing linear algebra over our plaintexts. This is heavily dependent on the structure of the plaintext slots. In particular, there is an important distinction between considering the slots as a linear array or a more general hypercube.

Linear array

When considering computation on encrypted data, there is an extra property of the algorithms that becomes important. Normally, one analyses the algorithm with respect to time complexity (and sometimes space complexity), but with homomorphic encryption one also has to consider the depth of the calculation. If one multiplies the same ciphertext many times with other ciphertext, the layers of the modulus chain is used up quickly. We therefore have to keep track of the number of different computations, but also the depths the computation require. In this section n will denote the number of slots.

Masking is a multiplication by a vector where all the entries are 0 and 1. This is as costly as a multiplication by a constant, since this is what it essentially is. If we do a rotation followed by a masking so that all entries that are wrapped around are turned to zero, we have a *shift*. These are useful for a variety of purposes. Shifts are used in implementing rotations where the dimension is bad, as explained in Section 3.2.1.

TotalSum takes in a vector and produces a new vector where all the entries are the sum of the entries in the original vector. This operation uses at most $2\lfloor \log n \rfloor$ rotations and additions.

RunningSum takes in a vector u and produces a new vector v such that $v_k = \sum_{i=1}^{k} u_i$. This is implemented similarly to TotalSum, except that we replace the rotations by shifts. This gives $\lceil \log n \rceil$ additions, rotations and maskings.

We can make a *replication* of a single value from a vector v by $\mathrm{Replicate}(v, i)$. This gets us a vector u such that $u_j = v_i$ for all j. This is done by first turning all other entries to zero via masking, and then doing the TotalSum of the vector. It uses $2\lfloor \log n \rfloor$ rotations, additions and uses a single masking.

We can implement a *full replication* procedure, which produces n vectors $w_1, \ldots, w_n$ from a vector v such that $w_j = (v_j, \ldots, v_j)$. The

	Additions	Rotations	Maskings	Add. Depth	Rot. Depth	Mask. Depth
TotalSum	$2\lfloor\log n\rfloor$	$2\lfloor\log n\rfloor$	-	$2\lfloor\log n\rfloor$	$2\lfloor\log n\rfloor$	-
RunningSum	$\lceil\log n\rceil$	$\lceil\log n\rceil$	$\lceil\log n\rceil$	$\lceil\log n\rceil$	$\lceil\log n\rceil$	$\lceil\log n\rceil$
Replicate	$2\lfloor\log n\rfloor$	$2\lfloor\log n\rfloor$	1	$2\lfloor\log n\rfloor$	$2\lfloor\log n\rfloor$	1
Fullreplicate	$3n$	$2n$	n	$2\lceil\log n\rceil$	$\lceil\log n\rceil$	$\lceil\log n\rceil$

Table 3.1: The cost of some useful techniques. Fullreplicate uses the recursive method.

naive way is to use the replication method above n times, which uses $2n\lfloor\log n\rfloor$ rotations, additions and n maskings. We can do this more efficiently by a recursive method, to get $3n$ additions, $2n$ rotations and n maskings. This has the drawback of having larger masking depth than the naive method, with depth $\lceil\log n\rceil$. The algorithm is given in [11]. There is a way of combining these two methods which gives masking depth $O(\log\log n)$. Describing this combination in detail would get us too much off track, but see Halevi and Shoup [11] for an elaboration.

A table for cost of the different techniques can be found in Figure 3.1. The cost of multiplications is not considered, as none of the techniques uses multiplications.

Hypercube

The operation described above can be extended to the hypercube setup. TotalSum works in the exact same way as in the linear array case, except that it sums over the dimensions in the hypercube and rotates in the dimensions of the hypercube. We can either do a TotalSum in a single dimension, or over multiple or all dimensions.

RunningSum is more complicated to implement, since it involves transferring data which are not inside a hypercolumn. This involves the operation of doing a "cyclic rotation" where we pretend that the slots are in a linear array, and then perform a rotation. This is a

bit more complicated operation, where we essentially do an addition where the carries are computed on the different indices. Halevi and Shoup [11] gives a thorough explanation.

Replication of a single value can be done by a masking which gives zero in all other slots, and then TotalSum over the whole hypercube. We can also replicate slices in the hypercube by making all other entries than the slice zero and then doing TotalSum over a single dimension. Full replications can also be implemented similarly to on a linear array.

3.3.1 Linear Algebra

How we do linear algebra is heavily dependent on how we represent structures such as the matrix. In this section we highlight the differences between the different representations, and show how richer hypercube structure gives more options for faster algorithms. We show the methods for square matrices, but this can easily be extended by partitioning the matrices into square blocks and multiply the matrices block wise.

The linear algebra techniques we will describe requires the use of multiplications. Compared to the other basic operations (addition, rotations, maskings), multiplication dominates the computation time and is much more important to the number of levels. We therefore consider the number of multiplications and multiplication depth in exact terms, while we only consider big O notation for the other operations. This is to avoid tedium from computing the exact number of operations where it is not really that relevant, as they will be dominated by multiplications anyway.

Linear array

A natural way to represent a matrix is a collection of columns. Say we have the matrix $A = (A_1, \ldots, A_n)$ and that we want to multiply it by a vector v. Then we can compute $w = Av = \sum_{i=1}^{n} A_i \cdot \text{Replicate}(v, i)$. Since we want to replicate all entries of v, we can use full replicate to get the replicated values, reducing computation time. Thus we can compute matrix-vector multiplication with n multiplications and $O(n)$ maskings, additions and rotations. The depth is $O(n)$ additions, $O(\log n)$ maskings and rotations and a single multiplication. We can also represent a matrix as a collection of rows. Then the computation requires the same number of operations, but it requires some extra nuances. We refer to [11] for a thorough explanation.

We can do better than this if we have the matrix in *diagonal representation*. If we have the $n \times n$ matrix $A = (a_{i,j})$ then the diagonal representation places each diagonal $d_i = (a_{j,j+i})$ where the index arithmetic is modulo n. We can then compute $w = Av$ as $\sum_{i=1}^{n} d_i \cdot (v \lll i)$ where $(v \lll i)$ is the rotation of v by i places to the left. We can see that this is correct by seeing that

$$w[j] = \sum_{i=1}^{n} d_i[j] \cdot (v \lll i)[j] = \sum_{i=1}^{n} a_{j,j+i} v[i+j] = \sum_{i=1}^{n} a_{j,i} v[i]$$

as intended. This method uses $O(n)$ additions, rotations and n multiplications, and has depth one multiplication, one rotation and $O(n)$ additions. The drawback of this method is that matrices are more often represented in column or row representation, and it is expensive to move from one of these representations to the diagonal representation. If we can pre-process the matrix before encryption (negating the cost of changing representation), then this method is the best.

Matrix multiplication can be done in a similar way, depending on the representation. If we have the matrices X, Y in row order such

that $X = (X_1, \ldots, X_n)$, then we can compute XY by setting

$$(XY)_i = \sum \text{Replicate}(X_i, j) \cdot Y_i.$$

Again we use the full replication procedure on each row X_i to get the replications of X_{ij}. This method requires n^2 multiplications, additions and n full replications. This corresponds to n^2 multiplications, $O(n^2)$ additions, rotations and maskings. The depth of the method is $O(n)$ additions, $O(\log n)$ rotations and maskings and a single multiplication.

Hypercube

With more dimensions comes faster implementation. The drawback is that it is harder to find appropriate parameters to construct the hypercube.

With the number of dimensions being two, we can implement faster algorithms. Say we want to multiply a matrix A with a vector v. We let A be encrypted as a single ciphertext in this scenario, while we have two choices on how to encrypt v. We can either replicate it so we have a matrix where all the rows are the same vector (row order), or where all the columns are the same vector (column order). If we encode v in row order and call it V then we compute $A \odot V$, where we use the notation $\odot$ to emphasize that this is not matrix multiplication, but entry wise multiplication (also known as the Hadamard product). If we encode v in column order (the transpose of V, denoted V^T) then we compute $A^T \odot V^T$. We then take TotalSum over the other axis then the one we replicated over. If we multiply a matrix and a vector in row/column-order, we get a vector in the opposite order as the output. Symbolically we get that if $w = Av$ and $\text{TotalSum}(A, i)$ is the TotalSum over dimension i of A then the encrypted vector is

$$[w]_{i \in [n]} = \text{TotalSum}(A \odot V, 1)$$
$$[w^T]_{i \in [n]} = \text{TotalSum}(A^T \odot V^T, 2).$$

Note that we can convert from the vector in row order to column order and visa versa by multiplying by the identity matrix. We can illustrate the multiplication with the following example

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \odot \begin{bmatrix} a & b \\ a & b \end{bmatrix} = \begin{bmatrix} a & 2b \\ 3a & 4b \end{bmatrix} \to \begin{bmatrix} a+2b & a+2b \\ 3a+4b & 3a+4b \end{bmatrix}$$

The number of operations depends on if the vector is encrypted in matrix form or not. If it is not, then we only need a single replication. This is because replication in the hypercube replicates the slice, and the slice in a matrix is a vector. It it is encrypted in matrix form, we need one TotalSum and one multiplication. This translates to $O(\log n)$ additions and rotations, and one multiplication. The depth is $O(\log n)$ additions and rotations, and one multiplication. If we include the replication, we also get $O(1)$ maskings and masking depth $O(1)$. This is a clear improvement on the linear array arrangement, both in time and depth, see Tables 3.2 and 3.3 for comparisons.

Matrix multiplication of two matrices A, B also becomes faster. Instead of thinking of matrices in row- or column-order, we can think of them in diagonal order. We denote the ith diagonal $d_i(A) = (a_{0,i}, a_{1,i+1}, \ldots, a_{n-1,i-1}$ so that $d_i[j] = a_{i,i+j}$ where the indexes are modulo n. We can extract each diagonal by a single replication into a matrix D_i which is n replication of d_i. Now we can multiply each diagonal D_i by a rotation of the rows of B by i, denoted $(B \lll_1 i)$, and sum the results. This gives the correct answer because if we set $C = A \cdot B$, then

$$C[i,j] = \sum_k D_k[i,j] \cdot (B \lll_1 k)[i,j] = \sum_k a_{i,i+k} \cdot b_{i+k,j} = \sum_k a_{i,k} b_{k,j}$$

which is what we wanted. It should be noted that this diagonal represented matrix multiplication can not be applied consecutively without some processing, as the output of the algorithm is a matrix in column

Time	Additon	Rotation	Masking	Multiplication
1DMatVec	$O(n)$	$O(n)$	$O(n)$	n
1DMatVecDiag	$O(n)$	$O(n)$	-	n
1DMatMul	$O(n^2)$	$O(n^2)$	$O(n^2)$	n^2
2DMatVec	$O(\log n)$	$O(\log n)$	$O(1)$	1
2DMatMul	$O(n \log n)$	$O(n \log n)$	$O(n)$	n
3DMatMul	$O(\log n)$	$O(\log n)$	$O(1)$	1

Table 3.2: Table for time complexity of linear algebra techniques.

representation. This method uses $O(n \log n)$ additions and rotations, $O(n)$ maskings and n multiplications. It has depth of $O(\log n)$ additions, $O(\log n)$ rotations, and $O(1)$ maskings and a single multiplication.

With the number of dimensions being three, we can implement matrix multiplication in with a single multiplication. This is based on the DNS algorithm as in [7]. For two matrices A, B, it computes $C_{ij} = \sum_{j=1}^{n} A_{ik}B_{kj}$. It replicates the rows of A so that $A(i,j,k) = a_{ik}$ and the columns of B so that $B(i,j,k) = b_{kj}$. By a single multiplication and a Totalsum along the k-axis we then get $C[i,j,0] = \sum_k A[i,j,k] \cdot B[i,j,k] = \sum_k a_{ik}b_{kj}$. This then has $O(\log n)$ additions and rotations, and a single multiplication. The depth is $O(\log n)$ additions and rotations, and a single multiplication. If we count the replication to initialise the matrices, we get $O(1)$ maskings and masking depth additionally.

There are optimisations for publicly known linear transformations on ciphertexts. The optimisations used in HElib are described in detail by Halevi and Shoup [12].

Example 3.4. Let $m = 2^{15} - 1$ and $p = 2$ so that we have the hypercube structure isomorphic to $\mathbb{Z}_{30} \times \mathbb{Z}_6 \times \mathbb{Z}_{10}$. With these parameters we can do matrix multiplications in three dimensions if the dimensions of the

Depth	Additon	Rotation	Masking	Multiplication
1DMatVec	$O(n)$	$O(n)$	$O(\log n)$	1
1DMatVecDiag	$O(n)$	$O(1)$	-	1
1DMatMul	$O(n)$	$O(\log n)$	$O(\log n)$	1
2DMatVec	$O(\log n)$	$O(\log n)$	$O(1)$	1
2DMatMul	$O(\log n)$	$O(\log n)$	$O(1)$	1
3DMatMul	$O(\log n)$	$O(\log n)$	$O(1)$	1

Table 3.3: Table for depth complexity of linear algebra techniques.

matrices are less than 6×10, since these are the smallest dimensions of the hypercube. These are pretty small matrices compared to the degree of the polynomial $\phi(m) = 27000$.

We have now considered how we can encode data in our cryptosystem and constructed tools we need in order to compute on the encrypted data.

Chapter 4

Security model

In this section we will sketch a security model for computing on encrypted data. We choose to use a security model where only the data is secret, while the functions and model parameters are not necessarily kept secret. To get a model for secure computing, we first look at regular computing on unencrypted data. A regular machine learning model looks something like this:

We are interested in keeping privacy of data. One can also consider privacy of the algorithm, where the parameters in the machine learning algorithm are a corporate secret. However, we do not consider this case in this thesis. A public machine learning algorithm has the advantage of not complicating our security proofs.

When we are working with homomorphic encryption, the model is complicated by the encryption and decryption process. This looks like the following

In this model we include a bit of leakage, which can be information about convergence or some other piece of data that does not reveal too much information. Otherwise we would have to choose a fixed number of iterations, which can be inefficient in some cases.

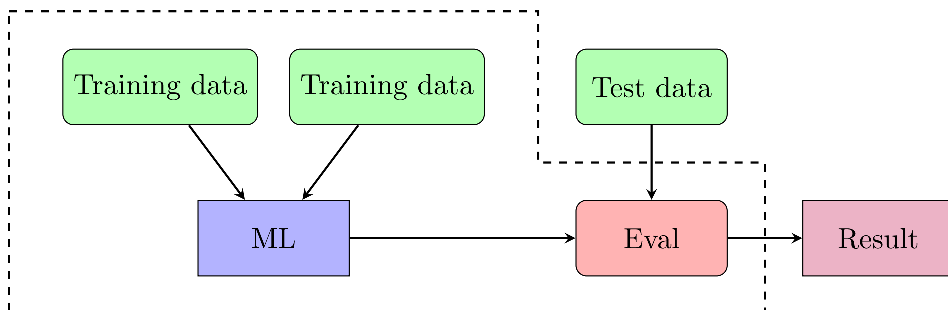


Figure 4.1: A model for machine learning. The training data is used by the machine learning algorithm to construct an evaluator which can take test data in and predict accurate results. We are interested in the steps inside the dashed box.

4.1 Security proof

We will now prove the security of this model. In the security proofs, we illustrate with a single data-holder, but in practice there can be many. Sending encrypted data to an honest server and then sending them to the computer is the same as sending the data directly in this security model. Thus the distinction between one and multiple data-holder is not important in the honest, but curious model.

We start by assuming that the machine learning provider represents an honest, but curious adversary. Such an adversary does not interfere with the communication or computation, as a malicious adversary would.

We assume that we have an IND-CPA secure cryptosystem. Homomorphic encryption can in general not be totally CCA2 secure, because if the adversary queries the decryption of a and $a + b$ then it is easy for the adversary to get the decryption of b. Therefore security must assume that the adversary cannot query the decryption oracle after the target message is chosen. It is possible to construct

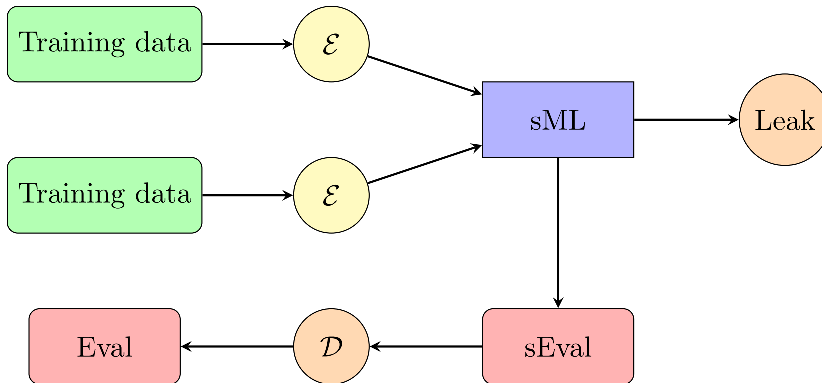


Figure 4.2: A homomorphic encryption-based machine learning model. The secure machine learning algortihm takes the encrypted training data and constructs a secure evaluator, which we decrypt. An alternative to this approach is to keep the evaluator encrypted when using the training data afterwards. We include a leak of information in this model, which can be used for example to accelerate computation.

CCA1-secure cryptosystem, as was done in [18]. We limit our analysis to CPA-security for simplicity.

Theorem 4.1. *Assume the homomorphic encryption protocol is IND-CPA secure and that the adversary is honest, but curious. Then the model is secure.*

Proof. We say Alice has the data, Bob does the calculations and Carol does the decryption. In game G_0, everything works as in the sketch provided in figure 4.3. The interesting case is when we identify Bob as the adversary (as Alice has the data and Carol has the decryption key). He has access to the ciphertexts, the evaluation of the ciphertexts and some leakage which is provided by Carol, which is used in computation.

The leakage in the computation can be included in the queries Bob asks of Carol, and as such is included in game G_0. We show that Bob

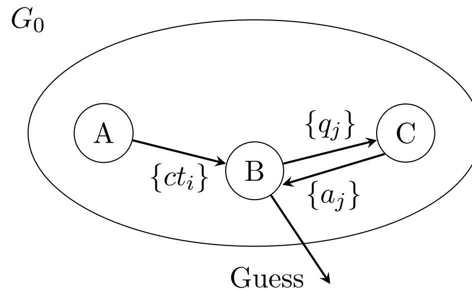


Figure 4.3: The game G_0 is the setup we have in the real world. Here ct_i denotes a ciphertext, q_j denotes a query, and a_j denotes the answer to the query.

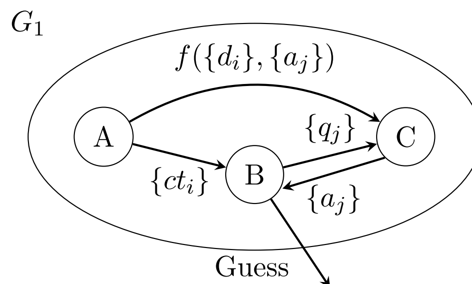


Figure 4.4: In game G_1, we send a computation from Alice to Carol, which encapsulates the result of the computation done by Bob and the leakage provided by Carol to Bob. This does not affect the security of the scheme. Here d_i denotes plaintext data.

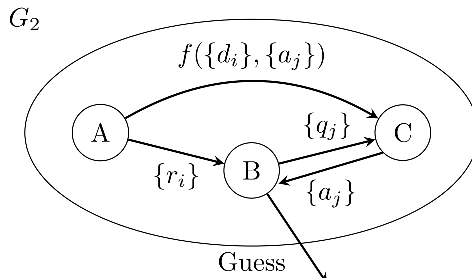


Figure 4.5: In the last game we send real-or-random ciphertexts to Bob, denoted r_i, giving no advantageous information to Bob.

does not gain any information by game hopping. In the game G_1, we send everything as before, but Alice also sends the computation to Carol, so she can decrypt without Bob's input. We sketch the process in Figure 4.4.

For the last game, we send a real-or-random encryption to Bob, so that he cannot distinguish between the encryptions of real data and random noise. This is because of the assumption that the cryptosystem is secure.

Since we send all the relevant computation of the data from Alice to Carol directly, Carol can supply the decryption of the data, without decrypting anything at all. This is the case even when Bob has only access to the random noise. Thus we have constructed a simulation which gets all the leakage. □

Security with a malicious adversary can be done, but requires a more complicated proof. This can be done by a zero-knowledge proof, but this is out of scope of this thesis.

The guess of the adversary can be thought of as a function from the data space X to a yes/no-space (essentially $\{0, 1\}$). We say the

adversary has no advantage if

$$\Pr[f(\text{data}) = 0 | \text{data} \xleftarrow{r} X] = \frac{1}{2}.$$

The fact that no information is gained by the leakage can be formulated as

$$\Pr[f(\text{data}) = 0 | \text{data} \xleftarrow{r} X \wedge g(\text{data}) = z] = \frac{1}{2}.$$

For this to hold, f cannot be correlated with g. This is highly dependent on the data space X. We therefore cannot do any general analysis on whether our specific leakage gives rise to new possible attacks without specifying the data space. This task is left to those who will implement specific algorithms and handle specific data sets.

Chapter 5

Privacy Preserving Computation

In this section we will finally use the foundation we have built in the previous sections. We will go through some different computation techniques to highlight how different the approach to each problem can be. We first look at another problem when computing with homomorphic encryption, namely how do we compute complicated functions?

5.1 Computing complicated functions

Homomorphic cryptosytems can compute polynomial functions. This is often not enough for practical usage, as we often need more complicated functions. When we refer to complicated functions, we actually mean non-polynomial functions, such as the exponential function, natural logarithm, etc. This also includes division of numbers. In this section we present two approaches to computing approximations of non-polynomial functions using only polynomials.

5.1.1 Polynomial approximation

Many techniques require polynomial approximations. We know that any continuous function on a closed interval can be approximated by a polynomial, and that the approximation can get arbitrarily close to the function. For a high precision this may in practice require a high degree polynomial, which is impractical when using homomorphic encryption. There are different ways of approximating using polynomials. One method is to use a truncated Taylor series. This is a local approximation, which works best when the the data is in some limited domain.

An example of a global approximation over an interval I is the least square error method, which chooses the polynomial with least mean square error (MSE). This chooses the polynomial g of degree d or less that minimises $\int_I (g(x) - f(x))^2 dx$. Ghasemi et al. [14] gives a detailed method for implementation of homomorphic polynomial approximation.

Another approach to non-polynomial function computation is to create lookup tables in advance. This means that we compute the values for a given range of values for the function before we do our encryption.

5.1.2 Table lookup

For a given function f we can construct a table $T_f[X]$ of outputs for a limited set of inputs, such that $T_f[X] = f(X)$ on the limited set of inputs. This method is faster and uses fewer levels than polynomial approximation, but has limited precision. We can pre-compute all the values in the table, which saves significant time. We have to do the table look-ups homomorphically, so it still has some cost associated with it, but not nearly as high as with a polynomial approximation. This method is suitable for SIMD operations, so it matches well with

the packed encoding.

We do this as follows: we have the parameters (p, s, ν), where p is the number of bits, s is the scale and ν is a boolean which determines whether the number is signed ($\nu = 1$) or unsigned ($\nu = 0$). The p-bit string $(x_{p-1} \dots x_1 x_0)$ is interpreted as the rational number

$$R_{p,s,\nu}(x_{p-1} \dots x_1 x_0) = 2^{-s} \cdot (\sum_{i=0}^{p-2} 2^i x_i + (-1)^\nu \cdot 2^{p-1} x_{p-1})$$

A given table has implicit parameters (p, s, ν) for the input of the function and parameters (p', s', ν') for the output of the function. For these parameters we get 2^p entries in the table, each of which has a p'-bit number. We construct the table as follows:

A given p-bit string x_i has an assigned index $i \in [2^p]$. The table entry $T_f[i]$ is given by the integer z_i with binary expansion y_i

$$R_{p',s',\nu'}(y_i) = \lceil f(R_{p,s,\nu}(x_i) \rfloor_{p',s'}$$

For values that are not representable in p bits, we give them the maximum or minimum possible values. If the size of the integer is bounded by the plaintext slot, we can store the value in each slot in a plaintext, to accommodate more SIMD operations.

Once we have the function table, we only need to extract the correct value homomorphically for a given bit string. We do this by computing subset products of the bits. Given encrypted bits $\sigma_{p-1}, \dots, \sigma_0$ we compute

$$\begin{array}{rccccccc}
\rho_0 = & (1-\sigma_0) & \cdot & \dots & \cdot & (1-\sigma_{p-2}) & \cdot & (1-\sigma_{p-1}) \\
\rho_1 = & (1-\sigma_0) & \cdot & \dots & \cdot & (1-\sigma_{p-2}) & \cdot & \sigma_{p-1} \\
\rho_2 = & (1-\sigma_0) & \cdot & \dots & \cdot & \sigma_{p-2} & \cdot & (1-\sigma_{p-1}) \\
 & \vdots & & & & \vdots & & \\
\rho_{2^p-1} = & \sigma_0 & \cdot & \dots & \cdot & \sigma_{p-2} & \cdot & \sigma_{p-1}
\end{array}$$

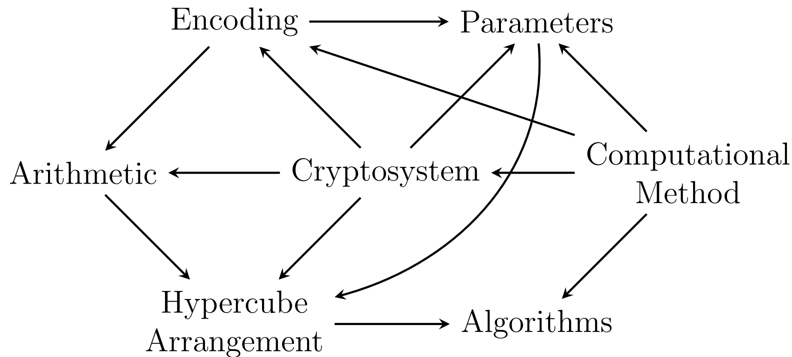


Figure 5.1: Overview of choices in how to do a specific computation homomorphically, and how the choices affects other choices. This is a sketch to illustrate the complexity of homomorphic encryption, and is not necessarily useful for specific implementation.

We can compute these subset products recursively, so we can get $\lceil \log p \rceil$ multiplication depth by using a tree like structure when we multiply. We do have to compute 2^p products, so the total number of multiplications is $O(2^p)$. With the function table and the subset products ρ_r computed, the table evaluation is easy to compute: $\sum_r T_f[r] \cdot \rho_r$ gives the desired evaluation of the function.

5.2 Computation techniques

In this section we will look at some different computation techniques, and look at how we can do them homomorphically. We will show which are feasible with which representation, and which are unfeasible all together. We start with some easy ones and move to more complicated ones.

We stress that for any given computation one want to do homo-

morphically, there are many choices to make in how one does it. This is illustrated in Figure 5.1. Therefore comparisons between different methods is difficult, and there is seldom a single "correct" way of doing a given computation.

5.2.1 Principal component analysis

Principal component analysis (PCA) is a dimension reduction technique, where we are given a set of possibly correlated variables and we want to reduce them to a set of independent variables. We do this by looking at the covariance matrix of the variables and extracting the dominant eigenvectors from it. We are given a data matrix X and want to compute the eigenvalues and eigenvectors of its covariance matrix

$$\Sigma = \frac{1}{N} X^T X - \boldsymbol{\mu}\boldsymbol{\mu}^T \qquad \boldsymbol{\mu}^T = \frac{1}{N}\sum_{i=0}^{N-1} \mathbf{x}_i^T.$$

We can extract the dominant eigenvalue and eigenvector by using the power method. This is a method that computes $\mathbf{v}^{(k)} = \Sigma^k \mathbf{v}^{(0)}$ for some random vector $\mathbf{v}^{(0)}$. The vector is stretched in the direction of the dominant eigenvector each time we multiply by the covariance matrix. We only compute the most dominant eigenvalue and eigenvector here. We show how in Algorithm 2.

The power method is suitable for homomorphic encryption, as it only uses additions and multiplications except at the very end. We could decrypt before doing the division, leaking a bit of information. Alternatively we could implement a look-up table for the functions $1/\|\mathbf{x}\|$ and compute $\mathbf{u} = \mathbf{v}^{(T)} \cdot 1/\|\mathbf{v}^{(T)}\|$. This would depending on the precision of the look-up table require a comparatively high amount of multiplications.

Algorithm 2: Power method for eigenvalues

Input:
- Σ Covariance matrix
- T Number of iterations

Output: Dominant eigenvalue λ and eigenvector $\mathbf{u}$

Choose random vector $\mathbf{v}$

for $i = 1$ *to* T **do**

  $\mathbf{v}^{(i)} = \Sigma\mathbf{v}^{(i-1)}$

return $\mathbf{u} = \mathbf{v}^{(T)}/\|\mathbf{v}^{(T)}\|$ and $\lambda = \|\mathbf{v}^{(T)}\|/\|\mathbf{v}^{(T-1)}\|$

A problem we face is with computing the covariance matrix Σ itself, as it requires division by N. We can solve this by multiplying with N^2 and working with the scaled covariance matrix. The algorithm for computing secure PCA is in Algorithm 3.

We get a normalised eigenvector by computing $\mathbf{v}^{(T)}/\|\mathbf{v}^{(T)}\|$ in the clear and we get the eigenvalue by $\|\mathbf{v}^{(T)}\|/\|\mathbf{v}^{(T-1)}\|$. The performance is highly dependent on how we represent matrices in the plaintext space. In total it uses T matrix vector multiplications, one outer product and $2N$ additions and the depth is the same. The matrix vector multiplications will dominate here. We therefore would probably like to use a two dimensional hypercube for this computation. Rathee et al. [23] showed that for some specific data sets the two dimensional hypercube gives significantly faster implementation than with the linear array arrangement. The difference in performance is more pronounced the more attributes the data has. This makes sense, as the one as the two dimensional matrix vector multiplication requires a single multiplication, while with the linear array the vector matrix multiplication requires n multiplications for n data attributes.

Algorithm 3: Secure PCA

Input:
- $\mathbf{x}_i^T$ the ith row of X
- $\mathbf{x}_i\mathbf{x}_i^T$ the outer product
- T number of iterations

Output: first principal component u_1 of X and its magnitude λ_1

$N\boldsymbol{\mu}^T = \sum \mathbf{x}_i^T$
$N^2\boldsymbol{\mu}\boldsymbol{\mu}^T = (N\boldsymbol{\mu}) \cdot N\boldsymbol{\mu}^T$
$N^2\Sigma = N\sum \mathbf{x}_i\mathbf{x}_i^T + N^2\boldsymbol{\mu}\boldsymbol{\mu}^T$
for $i = 1$ *to* T **do**
| $\mathbf{v}^{(i)} = N^2\Sigma\mathbf{v}^{(i-1)}$
return $\mathbf{v}^{(T)}, \mathbf{v}^{(T-1)}$

5.2.2 Linear regression

Linear regression is simple way of modeling the relationship between variables. This is typically done by methods which are difficult to implement homomorphically, such as singular value decomposition. We leverage the homomorphic implementation of PCA to compute linear regression in this section.

We get data on the form $(X, \mathbf{y}) = \{\mathbf{x}_i^T, y_i\}_{i=1}^N$. Our goal is to find weights $\mathbf{w}$ such that $\mathbf{y} \approx X\mathbf{w}$. We find this by

$$\mathbf{w} = \arg\min_{\mathbf{w}^*} \frac{1}{N}\sum_{i=1}^{N} \|y_i - \mathbf{x}_i^T\mathbf{w}^*\|^2.$$

There are many other techniques for solving linear regression, but we need a technique suitable for homomorphic computation. For small dimensions we can use the normal equation method, which computes $(X^TX)^{-1}X^T\mathbf{y}$. The problem here is that we have to compute the

inverse, where many methods are numerically unstable. Fortunately, there is a division free algorithm for finding a scaled matrix inverse, based on one by Guo and Higham [10]. This is a stable method if we have a dominant eigenvalue. We can therefore use the information gained from the secure PCA to use a division free matrix inversion algorithm. We show it in Algorithm 4.

Algorithm 4: Division free scaled matrix inverse

Input:
- M matrix
- λ dominant eigenvalue
- T number of iterations

Output: $\lambda^{2^T} M^{-1}$ the scaled inverse of M

Initialise $A^{(0)} = M, R^{(0)} = I, \alpha^{(0)} = \lambda$

for $i = 0$ *to* T **do**

  $B = 2\alpha^{(i-1)} I - A^{(i-1)}$

  $R^{(i)} = B \cdot R^{(i-1)}$

  $A^{(i)} = B \cdot A^{(i-1)}$

  $\alpha^{(i)} = \alpha^{(i-1)}\alpha^{(i-1)}$

return $R^{(T)}$

The output of the algorithm is the inverse of M scaled by the constant $\alpha^{(T)}$, so we get $M^{-1} = R^{(T)}/\lambda^{2^T}$. This is an algorithm we can evaluate homomorphically. We now present the algorithm for secure linear regression.

We see that λ^{2^T} can grow really fast, so we need either a large plaintext modulus or a high bit number of bits, namely $\log(\lambda) \cdot 2^T$ bits. The division free matrix inverses takes $2T$ matrix multiplications and additions, and the depth is T matrix multiplications. In addition we have to do N outer products and a sum of N variables. We have one final matrix vector multiplication in the end.

In both of these techniques, we multiply by a large constant (λ^{2^T}

Algorithm 5: Secure Linear Regression

Input:
- $y_i \mathbf{x}_i^T$
- $\mathbf{x}_i \mathbf{x}_i^T$ outer product of $\mathbf{x}_i$
- λ dominant eigenvalue
- T number of iterations

Output: $\lambda^{2^T} w$, λ^{2^T}

$X^T y = \sum y_i \mathbf{x}_i^T$
$X^T X = \sum \mathbf{x}_i \mathbf{x}_i^T$
Does the division free inverse algorithm on $X^T X, \lambda, T$
$\lambda^{2^T} \mathbf{w} = \lambda^{2^T} (X^T X)^{-1} \cdot X^T y$
return $\lambda^{2^T} \mathbf{w}$, λ^{2^T}

in secure linear regression and N^2 in secure PCA), which will be unpractical to represent in a bit wise encoding. If we want to represent λ^{2^T} we would need $\log(\lambda) \cdot 2^T$ bits, which becomes really impractical for larger T's. These computations are suitable for using word wise encoding, since we know how many levels we need and we do not need comparisons, which are less efficient in word wise encodings.

An alternative is to compute $1/\lambda^{2^T}$ homomorphically in order to avoid the leak we get from publishing both λ^{2^T} and $\lambda^{2^T}\mathbf{w}$. This requires the implementation of a look-up table for the function $1/x$ in the required precision neighbourhood.

In this method we have to leak information just before we are done, but the process requires a fixed number of iterations. In the next technique we can leak convergence rate instead of leaking something at the end.

5.2.3 Logistic regression

Logistic regression fits data to a logistic curve in order to predict probabilities based on this curve. We are given data of the form $(\mathbf{x}_i, y_i)$ where $\mathbf{x}_i \in \mathbb{R}^d, y_i \in \{\pm 1\}$ where $y_i \in \{\pm 1\}$ is a variable that depends on $\mathbf{x}_i$. Set $\mathbf{z}_i = y_i \cdot (1, \mathbf{x}_i)$.

The goal is to find the optimal $\boldsymbol{\beta} \in \mathbb{R}^{d+1}$ that minimises the loss function

$$J(\boldsymbol{\beta}) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-\mathbf{z}_i^T \boldsymbol{\beta})).$$

Finding the optimal value can be done by gradient descent, which moves toward a local extremum of the function by moving along the gradient. For our loss function, the gradient can be computed as

$$\nabla J(\boldsymbol{\beta}) = -\frac{1}{n} \sum_{i=1}^{n} \sigma(-\mathbf{z}_i^T \boldsymbol{\beta}) \cdot \mathbf{z}_i$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the logistic function, which acts component wise on vectors. We see here that we have to deal with complicated functions such as the logistic and the logarithmic functions. Let us say that we use look-up tables.

The gradient descent algorithm then updates $\boldsymbol{\beta}$ by setting

$$\boldsymbol{\beta}^{t+1} = \boldsymbol{\beta}^t + \frac{\alpha}{n} \sum_{i=1}^{n} \sigma(-\mathbf{z}_i^T \boldsymbol{\beta}^t) \cdot \mathbf{z}_i$$

for some learning rate α. Gradient descent is known to converge somewhat slowly. There are many ways of tweaking the gradient descent algorithm to make it converge faster, but we keep it simple and somewhat homomorphically implementable.

For each step we have one inner product, one logistic function evaluation and one multiplication. The value $1/n$ needs only to be

computed once. Inner products can be implemented efficiently in all hypercube arrangements, with a multiplication and a TotalSum over all the entries. The bottleneck in this computation will be computing the logistic function. It will in general require $O(2^p)$ multiplications for a precision p, which gives it a bit of trade-off. We can either choose to do a set number of iterations or leak the difference between two iterations of $\boldsymbol{\beta}$ and halt when the convergence is good enough.

Logistic regression has successfully been implemented homomorphically. Kim et al. [17] implemented logistic regression with the approximate arithmetic cryptosystem of Cheon et al. [4]. They used an accelerated gradient descent algorithm which converges faster, but which has additional hyperparameters. Crawford et al. [5] used a slightly more complicated method than the gradient descent method. They settled for an implementation using bootstrapping, which massively slows down performance, but which was possible to implement in reasonable time regardless. Their input data $\mathbf{x}_i$ had only binary valued attributes, and as such it was more suitable to this cryptosystem than more general valued data would be.

We have looked at some successful implementations of privacy preserving computing. Next we look at a harder technique to implement homomorphically, namely neural networks.

5.2.4 Neural networks

Neural networks is a popular machine learning method for classification of data. They comprise of connected nodes where each node computes a nonlinear function based on the outputs of its input nodes. The network is divided into a number of levels, and each level contains a number of nodes. In this section we only consider so called fully connected feed forward networks, for ease of explanation. This means that each node in a level depends on all the nodes from the previous level.

Combining neural nets with homomorphic encryption can be done in a number of ways. We can both train and classify using homomorphic encryption, or we can do only one of those things. Training requires three computations: feed forward, the cost function and backpropagation. If we only want to classify data, the feed forward step is sufficient.

We first consider the feed forward mechanisms which is used to classify the training data. In each node we compute a linear combination of the previous level and evaluate an activation function. The linear combination of nodes in level $l-1$ can be written as a weight matrix W^l. From one level to the next we write $\mathbf{a}^l = f(W^l \mathbf{a}^{l-1})$ where $\mathbf{a}^l$ is the vector of outputs in level l. The activation is a nonlinear function, typically the logistic function ($f(x) = \frac{1}{1+e^{-x}}$) or the ReLU ($f(x) = \max(x, 0)$). The logistic function is often used in the last level in classification problems, as its output is in $(0, 1)$ and is therefore suitable for describing a probability. We compute the derivative of the activation function in the backpropagation step, so it is useful that the derivative is easy to compute. For example, we have $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ for the logistic function.

Secondly, we compute the cost function which measures the correctness of the prediction from the feed forward step. The cost function varies depending on usage. A commonly used one is the quadratic cost function $C_k = \|\mathbf{a}_k^L - \mathbf{y}_k\|^2/2$, because the derivative of the cost function is easy to compute homomorphically, which is useful in backpropagation. This is a commonly used cost function because it intuitively captures the distance between the prediction and the result. Notice that we denote the cost function C_k, which emphasizes that it computes the cost related to the data and classification $(\mathbf{x}_k, \mathbf{y}_k)$.

Thirdly we use backpropagation, which uses the gradient of the cost function to tune the weights in the neural net. Backpropagation can be done using stochastic gradient descent. This can be implemented as a series of matrix multiplications. Gradient descent updates the

weight matrices by setting

$$W^l = W^l - \alpha \frac{\partial C_k}{\partial W^l}$$

for some learning rate α and for level $l \in \{1, \ldots, L\}$. Stochastic gradient descent computes the same, except that instead of computing the cost for each single data point, it computes the cost with respect to an average of a small sample B, called a mini-batch, to get more efficient convergence. In this case we denote the cost C_B instead of C_k to indicate that it is the cost of a mini-batch.

The name backpropagation comes from the fact that the change in the weights is propagated backwards through the levels. The derivatives $\frac{\partial C_B}{\partial W^l}$ are computed in a backward sequence. Set $\mathbf{z}^l = W^l \mathbf{a}^{l-1}$. We will compute $\frac{\partial C_B}{\partial W^l}$ in terms of the following:

$$\begin{aligned}
\nabla C_B &= (\mathbf{a}_B^L - \mathbf{y}_B) \\
(f^l)' &:= \frac{\partial f^l}{\partial \mathbf{z}^l} = \operatorname{diag}(\frac{\partial f^l}{\partial z_1^l}, \ldots, \frac{\partial f^l}{\partial z_n^l}) \\
\frac{\partial \mathbf{z}^l}{\partial \mathbf{a}^{l-1}} &= (W^l)^T
\end{aligned}$$

where $\operatorname{diag}(\mathbf{x})$ refers to the diagonal matrix with x_i on its diagonal. The term $(f^l)'$ is a diagonal matrix with the activation function on the diagonal since the activation function is applied entry-wise. We now define the *error* $\boldsymbol{\delta}^l$ at level l recursively as

$$\boldsymbol{\delta}^l = \begin{cases} (f^l)' \cdot (W^{l+1})^T \cdot \boldsymbol{\delta}^{l+1} & l < L \\ (f^l)' \cdot \nabla C_B & l = L. \end{cases}$$

We can then express the changes by the error as

$$\frac{\partial C_B}{\partial w^l_{jk}} = \boldsymbol{\delta}^l_j \cdot a^{l-1}_k$$
$$\frac{\partial C_B}{\partial W^l} = \boldsymbol{\delta}^l \cdot (\mathbf{a}^{l-1})^T.$$

A full round of feed forward, cost function and backpropagation for all the data sets is known as an *epoch*. To get desirable accuracy we can do a set number of epochs. This method gives us little flexibility. If say the accuracy after 30 epochs is 95,7% and after 50 epochs it is 96%, it may not be worth it to compute the last 20 epochs. This is where leakage in our security model from Chapter 4 comes in. While we can do a fixed amount of epochs, we can also stop the computation when it is at a fixed accuracy level, or the growth in accuracy is at a small enough level. The drawback of this is that we have to know the test accuracy or its growth, which is encrypted. What we can do is to decrypt the test accuracy or growth, and conclude from that whether to stop or not. In this method we would have to choose an accuracy test, which should depend on what kind of data space we are using for best effect.

Neural networks requires one activation function evaluation and one matrix vector multiplication in each level in the feed forward step. The cost function requires evaluation of the cost function. Backpropagation requires two matrix vector multiplications and an outer product per level. The matrix vector multiplication between $(f^l)'$ and $(W^{l+1})^T\boldsymbol{\delta}^{l+1}$ can be computed as an entry-wise multiplication of two vectors, as we can view $(f^l)'$ as a vector instead of the diagonal of a diagonal matrix. This is a cheaper operation, both in the one dimensional and two dimensional hypercube structures.

One downside with neural nets is that the number of levels in the in the leveled FHE scheme is roughly proportional to the number of

levels in the neural network, since the number of network levels dictates the number of multiplications in both the feed-forward and back-propagation step. Thus the neural nets are much more demanding to implement homomorphically then the earlier computation techniques. We now look at some different approaches to computing neural nets homomorphically.

Approaches to homomorphic neural networks

Ghasemi et al. [14] tried an approach to homomorphic neural nets. The problem with their approach is that they required regular communication with the client. After using up every modulus chain, they would send the result back to the client to refresh the modulus chain without bootstrapping. Since bootstrapping requires more computation time than the actual computation they wanted to do, this saved a lot of time. The regular communication with a client is not only a problem in of itself, but the decryption of the temporary computation leaks information we rather not leak. They used a polynomial approximation to the activation functions instead of a look-up table. The implementation was reasonably fast, but at the expense of high communication rate and perhaps reduced security.

Halevi et al. [22] used the logistic activation function, with a look-up table. The ReLU is preferred as activation function for general neural nets in the machine learning community. It also have the benefit of being suitable for a homomorphic implementation in two's complement representation. We can use the sign bit x_n to compute the max by multiplying the number x with $1 - x_n$. If the signed bit is 0 (i.e. a positive number or 0 itself) then we keep the number, or else it becomes zero. Thus the ReLU activation is obtained with the cost of one replication and one multiplication in this representation. The derivative of the ReLU is zero when x is less than zero, one when x is bigger than zero and undefined in $x = 0$. This can be expressed by

$(1 - x_n)$, defining the output at $x = 0$ to be one. Both the ReLU and its derivative needs much less computation time then the look-up table, and a lower multiplicative depth. We also do not have to sacrifice any precision.

Another thing about the implementation of Halevi et al. is that they considered the plaintext structure of a linear array. This means the time complexity of the matrix multiplications is higher than if they were to use a hypercube structure with higher dimension. In their implementation they used a toy parameter $m = 2^{10} - 1$ which gives $\phi(m)/10 = 60$ slots and a secure parameter $m = 2^{15} - 1$ with $\phi(m)/15 = 1800$ slots. With the toy parameters there is no room for advanced hypercube structure without making the dimensions unpractically small. This may be a reason they chose to use a linear array structure. With the secure parameters, one can arrange the data in 30×60 slots in matrix form with the secure parameters. This reduces the number of multiplications by a factor of 30.

Their best supported implementation processed a single neuron in 14 seconds, and it took 919 seconds for the secure parameters, which is roughly 60 times as much. The secure parameters has 30 times as many slots, so the amortised computation time of the secure parameters can be estimated to be roughly twice that of the toy parameters. This is just with the evaluation of a single neuron. A complete mini-batch computation with 60 training examples took roughly 40 minutes with the toy parameters. As the per-gate computation is quasi-linear in the degree of our cyclotomic polynomial [2], we can scale up the computation time of a complete mini-batch to secure parameters by multiplying by 60. Thus a single mini-batch with secure parameters would take roughly 40 hours. Even by accounting for the fact that we could pack more input samples with the secure parameters, this is still so slow as to be completely unpractical.

Back-propagation and computing the cost function can be expensive. Thus if we train the network first, then classify, we would get a

much more efficient solution. The application of this method is more sparse than the general solution. In this scenario, we would need that some data of the same type as we are encrypting can be read in plain-text. This is difficult in practice, as you don't want to do computation on medical data at the expense of revealing half the data.

We have established some implementation of computation methods one can achieve with homomorphic encryption. Some of these are possible to implement securely with the current best cryptosystems and specially designed algorithms. As we see in the literature, fully implemented and secure deep learning with homomorphic encryption is yet a practical reality. With more optimisation in the homomorphic system and new discoveries, this could be practical in the future.

Chapter 6

Concluding remarks

In this thesis we have explored how we can achieve privacy preserving computation and how many aspects we have to consider to make computation privacy preserving. We have constructed a cryptosystem, discussed how we can encode real-world data, proved a security model and investigated some examples of privacy preserving computation.

A fully fleshed out covering of the topic would have included an implementation, a deeper discussion about bootstrapping, an improved security model which allows malicious attackers and better multiplication algorithms. Some of these subjects are quite advanced and their explorations could have been master theses of their own.

The future of homomorphic encryption is bright, and we have exciting times ahead of us. The scheme of Cheon et al. [4] allows for floating point arithmetic in the plaintexts natively, and is a possible candidate for even better computation than with the BGV system, which is best for integers. The floating point arithmetic makes the scheme especially good for machine learning tasks over the reals. It has the drawback of using approximate arithmetic, in essence trading accuracy for efficiency.

Computing on encrypted data is more complicated than computing

on regular data. By encryption the data increases in size and the operations become slower. Noise growth mitigation takes time and the data might need to be represented in a specific way for encryption. Lastly, many operations which are trivial to compute unencrypted are complicated to compute homomorphically. The available operations and their efficiency are dependent on the particular structure of the cryptosystem. Therefore the current homomorphic encryption techniques are still several orders of magnitude slower than their unencrypted counterparts. Homomorphic encryption has vast potential, and that potential drives the development of this tool for privacy preserving computation further.

References

[1] J. Basilakis and B. Javadi. “Efficient Parallel Binary Operations on Homomorphic Encrypted Real Numbers”. In: *IEEE Trans. Emerg. Top. Comput.* 9.1 (2021), pp. 507–519.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Trans. Comput. Theory* 6.3 (2014), 13:1–13:36.

[3] Z. Brakerski and V. Vaikuntanathan. “Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages”. In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings.* Ed. by P. Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, 2011, pp. 505–524.

[4] J. H. Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I.* Ed. by T. Takagi and T. Peyrin. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 409–437.

[5] J. L. H. Crawford et al. “Doing Real Work with FHE: The Case of Logistic Regression”. In: *Proceedings of the 6th Workshop on*

Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018, Toronto, ON, Canada, October 19, 2018. Ed. by M. Brenner and K. Rohloff. ACM, 2018, pp. 1–12.

[6] I. Damgård et al. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings.* Ed. by R. Safavi-Naini and R. Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 643–662.

[7] E. Dekel, D. Nassimi, and S. Sahni. “Parallel Matrix and Graph Algorithms”. In: *SIAM J. Comput.* 10.4 (1981), pp. 657–675.

[8] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009.* Ed. by M. Mitzenmacher. ACM, 2009, pp. 169–178.

[9] C. Gentry, S. Halevi, and N. P. Smart. “Homomorphic Evaluation of the AES Circuit”. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings.* Ed. by R. Safavi-Naini and R. Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 850–867.

[10] C.-H. Guo and N. J. Higham. “A Schur-Newton Method for the Matrix \lowercase{\boldmathp}th Root and its Inverse”. In: *SIAM J. Matrix Anal. Appl.* 28.3 (2006), pp. 788–804.

[11] S. Halevi and V. Shoup. “Algorithms in HElib”. In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I.* Ed. by J. A. Garay and R. Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 554–571.

[12] S. Halevi and V. Shoup. “Faster Homomorphic Linear Transformations in HElib”. In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I.* Ed. by H. Shacham and A. Boldyreva. Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 93–120.

[13] S. Halevi and V. Shoup. “Design and implementation of HElib: a homomorphic encryption library”. In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1481.

[14] E. Hesamifard et al. “Privacy-preserving Machine Learning as a Service”. In: *Proc. Priv. Enhancing Technol.* 2018.3 (2018), pp. 123–142.

[15] A. Jäschke and F. Armknecht. “Accelerating Homomorphic Computations on Rational Numbers”. In: *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings.* Ed. by M. Manulis, A.-R. Sadeghi, and S. A. Schneider. Vol. 9696. Lecture Notes in Computer Science. Springer, 2016, pp. 405–423.

[16] A. Jäschke and F. Armknecht. “(Finite) Field Work: Choosing the Best Encoding of Numbers for FHE Computation”. In: *Cryptology and Network Security - 16th International Conference, CANS 2017, Hong Kong, China, November 30 - December 2, 2017, Revised Selected Papers.* Ed. by S. Capkun and S. S. M. Chow. Vol. 11261. Lecture Notes in Computer Science. Springer, 2017, pp. 482–492.

[17] A. Kim et al. “Logistic Regression Model Training based on the Approximate Homomorphic Encryption”. In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 254.

[18] J. Loftus et al. “On CCA-Secure Somewhat Homomorphic Encryption”. In: *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*. Ed. by A. Miri and S. Vaudenay. Vol. 7118. Lecture Notes in Computer Science. Springer, 2011, pp. 55–72.

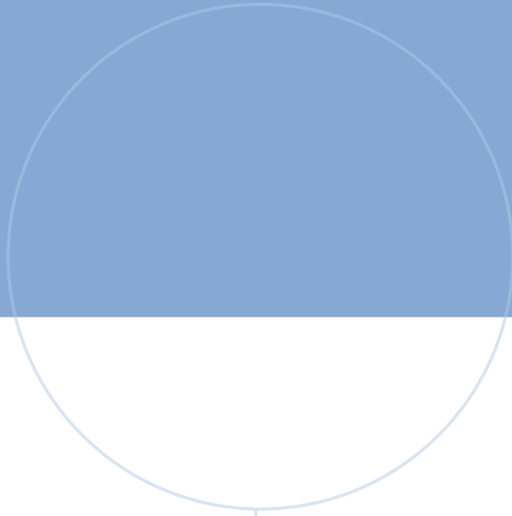
[19] V. Lyubashevsky, C. Peikert, and O. Regev. “On Ideal Lattices and Learning with Errors over Rings”. In: *J. ACM* 60.6 (2013), 43:1–43:35.

[20] V. Migliore et al. “A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm”. In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017), 138:1–138:17.

[21] M. Naehrig, K. E. Lauter, and V. Vaikuntanathan. “Can homomorphic encryption be practical?” In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*. Ed. by C. Cachin and T. Ristenpart. ACM, 2011, pp. 113–124.

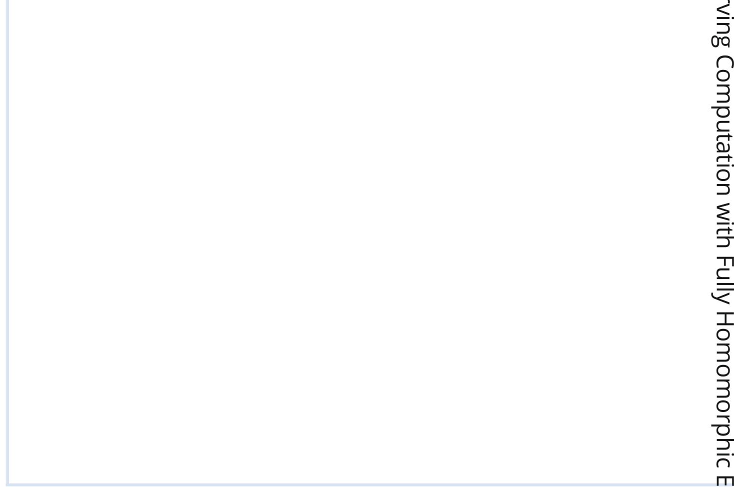
[22] K. Nandakumar et al. “Towards Deep Neural Network Training on Encrypted Data”. In: *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 40–48.

[23] D. Rathee, P. K. Mishra, and M. Yasuda. “Faster PCA and Linear Regression through Hypercubes in HElib”. In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by D. Lie, M. Mannan, and A. Johnson. ACM, 2018, pp. 42–53.



Jacob Sjødin

Privacy Preserving Computation with Fully Homomorphic Encryption



NTNU

Norwegian University of
Science and Technology