Fully Homomorphic Encryption Applications: The Strive Towards Practicality

by

Jack Lik Hon Crawford

A thesis submitted to the University of London for the degree of
Doctor of Philosophy

Department of Electronic Engineering and Computer Science
Queen Mary, University of London
United Kingdom

January 2019

I, Jack Lik Hon Crawford, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.

I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.

Signature: Jack Crawford

Date: 31/01/2019

Details of collaboration and publications:

1. **JLH. Crawford**, C. Gentry, S. Halevi, D, Platt and V. Shoup, "Doing Real Work with FHE: The Case of Logistic Regression," *WAHC '18: 6th Workshop on Encrypted Computing.* (Accepted)

2. **JLH. Crawford**, H. Hunt, "Fractions for Fully Homomorphic Computations" 2018 (Revising)

3. FHE patent with IBM (Filed)

I would like to dedicate this thesis to my family and friends who continually provide me support and inspiration.

Acknowledgements

I would like to thank my primary supervisor Dr. Søren Riis, my second supervisor Dr. Greta Yorsh and independent assessor Prof. Pasquale Malacaria for their input and feedback throughout the process of my doctoral programme.

I would like to thank my collaborators at IBM Research for their incredible insights and support. Most notably, Hamish Hunt, Shai Halevi and Victor Shoup whom I worked with during the course of my doctoral research.

Finally I would like to thank my fellow doctoral students for providing their advice, support and friendship.

Abstract

Fully Homomorphic Encryption (FHE) schemes are becoming evermore prevalent in the cryptography domain. They allow computation on encrypted data without the necessity of decryption, thus opening a plethora of new applications relating to cloud computing and cryptography.

FHE schemes have been viewed generally as being impractical in a real-world scenario, thus leading to a relatively slow uptake within industry despite the high level of interest in the topic. This has caused a lack of FHE applications and thus various practical questions have not been tackled due to such problems not arising or going unnoticed within research.

This thesis explores three contrasting FHE applications, each of which contain new ideas and overcome challenges within FHE. Namely, we analyse applications that require significant levels of bootstrapping, alternative data representations as well as the possibility of using FHE in the anonymity domain. Proofs of concept have been developed for each application to display the feasibility of each idea.

The aim of this research is to present the mathematics of FHE in a comprehensive manner to improve the accessibility of concepts within FHE. Furthermore we analyse the usability and versatility of FHE in various scenarios with the aim to demonstrate the practicality of using FHE in a real-world setting.

Table of Contents

Abstract **5**

Table of Contents **6**

List of Figures **10**

List of Tables **11**

1 Introduction **1**

1.1 Background of HE 1

1.1.1 BGV Scheme 2

1.2 The Current State-of-Art 3

1.3 Outline of the thesis 3

2 History of FHE **5**

2.1 Before FHE 5

2.1.1 RSA 5

2.1.2 El Gamal 6

2.1.3 Paillier 7

2.2 The Birth of FHE 8

2.2.1 Gentry's Breakthrough 9

2.2.2 The BGV Scheme 11

2.2.3 The FV Scheme 11

2.2.4 The GSW Scheme 12

2.2.5 Non-LWE-Based FHE Schemes 13

2.3 Summary 13

3 Mathematics of FHE 15

3.1 Introduction To Lattices and Rings 15

3.2 The Learning with Errors Problem 16

3.2.1 The Ring Learning with Errors Problem 19

3.2.2 The General Learning with Errors Problem 20

3.3 The BGV Scheme 21

3.3.1 Setup 22

3.3.2 Key generation 23

3.3.3 Encryption and Decryption 24

3.3.4 Arithmetic operations 25

3.3.5 Modulus Switching 28

3.3.6 Key-Switching 29

3.3.7 Batching 30

3.3.8 Cryptographic Usage 32

3.4 Summary 32

4 Anonymous Routing 34

4.1 Introduction 34

4.2 Existing Methods for Anonymity 36

4.3 Mathematics 38

4.4 The Algorithm 39

4.5 Implementation 42

4.5.1 Pre-Computation 42

4.5.2 Anonymous Routing 45

4.5.3 Retrieving The Message 49

4.6 Discussion 49

4.7 Conclusion 51

5 Implementing Fractional Arithmetic in FHE 52

5.1 Introduction 52

5.2 Implementation of Fractions 55

5.2.1 The Plaintext Space 55

5.2.2 The Quasi-Rationals 56

5.2.3 Simple Linear Regression 59

5.3 Implementation of Linear Regression 60

5.4 Evaluation of the use of QuasiQs 68

5.5 Comparison with Previous Methods 70

5.6 Conclusion 72

6 Performing Logistic Regression using FHE 73

6.1 Introduction 73

6.1.1 Somewhat vs. Fully Homomorphic Encryption 73

6.1.2 The iDASH Competition 74

6.1.3 Our Logistic-Regression Procedure 76

6.1.4 Homomorphic Computation of the Approximation Procedure . . . 77

6.1.5 The End Result 79

6.1.6 Related Work 80

6.1.7 Organization 81

6.2 Logistic Regression and Our Approximation 81

6.2.1 A Closed-Form Approximation Formula for Logistic Regression . . 82

6.2.2 Validity of the Approximation 84

6.3 Overview of Our Solution 85

6.3.1 The procedure that we implement 85

6.3.2 Homomorphic Evaluation 87

6.4 Using Table Lookup to Compute Arbitrary Functions 96

6.5 Binary Arithmetic and Comparisons 100

6.5.1 Adding Two Integers 101

6.5.2 Adding Many Integers 103

6.5.3 Integer Multiplication 104

6.5.4 Comparing Two Integers 105

6.5.5 Accumulating the bits in a ciphertext 106

6.6 Solving a Linear System 109

6.6.1 Randomized Encoding with Rational Reconstruction 109

6.6.2 Are We Still Leaking Too Much? 113

6.7 Implementation and Performance 113

6.7.1 Timing Results 114

6.7.2 Is this Procedure Accurate Enough? 115

6.7.3 Timing results for the Various Components 116

6.8 Conclusions and Discussion 117

7 Conclusions and future work 119

7.1 Summary 119

7.2 Conclusion 120

7.3 Future work 120

List of Figures

4.1 Plaintext slot representation where $a \in \mathbb{A}_p$ is a polynomial and each slot $R_i \equiv a \bmod F_i$ is a polynomial modulo a factor F_i of the cyclotomic polynomial $\Phi_m(x)$. 39

5.1 Random noise distribution from two Gaussian distributions. 61

5.2 Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0, 1)$. 63

5.3 Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0, 9)$. 65

5.4 Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0, 25)$. 65

5.5 Regression lines with gradients belonging to the same equivalence class. . 66

5.6 Regression lines with intercepts belonging to the same equivalence class. . 67

6.1 Randomized encoding for the rational linear-system solver, $f(A, \vec{b}) = A^{-1}\vec{b}$.112

List of Tables

6-A Timing results (minutes) of different phases of the logistic-regression program . . . 115

6-B Complexity measures and performance results. Time in seconds, RAM in MB. . . . 118

Chapter 1

Introduction

Throughout history, the security of the transmission and storage of data has always been an issue of high importance. Numerous cryptographic schemes have been developed to accommodate this need for information security. However due to the ever increasing prominence of cloud computing, merely the process of encrypting data is no longer sufficient. Encrypting data provides a means of securely storing and transmitting data to trusted parties. Yet what if the intended recipient of the data is not trusted by the sender? In the general case of communication this may not be the case but there are numerous scenarios where the user will not want to reveal their data to other parties but require them to have access. An example of a common use case is when the user wants to outsource computations on sensitive data.

1.1 Background of HE

Homomorphic Encryption (HE) schemes allow for computations to be performed directly on ciphertexts without the need for decryption. This allows users to outsource computations to the cloud without revealing their data. Achieving such a scheme that can evaluate useful circuit depths has been an open problem for numerous years [69]. Since then numerous different types of HE schemes have been proposed by the cryptographic community. Among them are the Partially Homomorphic Encryption (PHE) schemes that only allow limited types of homomorphic operations, namely addition but not multiplication or vice versa.

The community began to look at lattice based cryptography to perform HE and it was not until Gentry's breakthrough in 2009 [39] that Fully Homomorphic Encryption (FHE) schemes were shown to be possible. FHE schemes, unlike partially homomorphic schemes, allow for arbitrary computations on encrypted data. Gentry [39] presented a scheme that was Somewhat Homomorphic (SWHE) that through a novel technique called "bootstrapping" could evaluate its own decryption circuit in addition to a NAND gate to construct a Fully Homomorphic scheme.

Currently there are public-key encryption schemes that rely on the factorisation of prime numbers for their security, namely RSA [70]. Since the formulation of Shor's algorithm [71], which presents an algorithm that can factorise an integer N in polynomial time using a quantum computer. Prior to this discovery there has been increased interested in developing quantum secure cryptographic schemes. A previously well-studied set of problems that was explored was lattice problems, thus leading to lattice based cryptography.

1.1.1 BGV Scheme

The scheme proposed in 2012 by Brakerski, Gentry and Vaikuntanathan [13], known as the BGV scheme, was one of the more prominent FHE schemes. It was an extension upon Gentry's original scheme that mitigated the need for bootstrapping through the introduction of a concept called "levels".

Rather than building a scheme that is required to perform the computationally expensive bootstrapping technique, the BGV scheme produces a ciphertext that has an arbitrary number of pre-computed "computation levels". As operations are performed upon the ciphertext levels are consumed until the ciphertext reaches the base level and no further computations can be performed.

This scheme produced considerable gains in the efficiency of FHE thus improving the practicality of using such schemes. Theoretically, one can build a scheme with infinitely many levels although this is infeasible in practice thus technically making the BGV scheme a Somewhat Homomorphic Encryption (SWHE) scheme in practice.

In later implementations of the scheme [46] would utilise both the “leveled” scheme with the inclusion of Gentry’s bootstrapping technique. This allows a user to evaluate circuits using the more efficient leveled scheme until the ciphertext reaches the base level and then use the bootstrapping technique to “refresh” the ciphertext back to the top level before performing additional operations. The BGV scheme is described later in the thesis in Section 2.

1.2 The Current State-of-Art

Research and development on FHE schemes is ongoing in both academia and industry. Particular organisations of note are Microsoft Research and IBM Research who have developed the homomorphic encryption libraries S.E.A.L. [18] and HElib [46] respectively. In addition to this, other notable encryption libraries that implement FHE are PALISADE [66] and Fast Fully Homomorphic Encryption over the Torus (TFHE) [24].

Despite the various available FHE libraries, currently the libraries are not mature enough for practical use within the real world in a commercial environment. This further enforces the overall impracticality of FHE schemes however as improvements in hardware, algorithm efficiency and implementational techniques continue, practical uses of FHE have been shown and the feasibility of FHE schemes will continue to grow.

1.3 Outline of the thesis

Chapter 2 covers the history of FHE from the initial statement of the open problem, to Gentry’s breakthrough thesis through to the current stage of FHE research.

Chapter 3 provides an in-depth description of the mathematics behind FHE schemes, providing a detailed breakdown of how FHE schemes work mathematically.

Chapter 4 describes an anonymous routing algorithm implemented using an open-source homomorphic encryption library that protects against global adversaries.

Chapter 5 presents a method for representing and computing on rational numbers within FHE.

Chapter 6 proposes an FHE algorithm that performs logistic regression homomorphically.

Chapter 7 presents the conclusion and some thoughts for future work.

Chapter 2

History of FHE

This chapter provides an extensive literature review of the history of how homomorphic encryption schemes developed over time and aims to describe the differing types of FHE schemes that have emerged.

2.1 Before FHE

The ability to perform computations on encrypted data without the need to decrypt it has been an interesting open problem for a long time since its first mention by Rivest et al. [68]. Prior to the development of fully homomorphic encryption schemes, there existed encryption schemes that allowed the computation of only certain types of operations directly on the ciphertext itself. These schemes were known as partially homomorphic encryption schemes (PHE) which will be described in this section.

2.1.1 RSA

The Rivest-Shamir-Adelman (RSA) scheme [70] is one of the first public-key cryptosystems and is the earliest known encryption scheme capable of homomorphic computations. This scheme is multiplicatively homomorphic, although is not additively homomorphic. However, textbook RSA is deterministic and thus is not used in practice due to it being insecure. In other words, there is no element of randomness during the encryption of plaintexts under textbook RSA resulting in multiple encryptions of the same plaintext message yielding identical ciphertexts.

Below is a description of the multiplicatively homomorphic property of textbook RSA:

Definition 2.1.1 (Rivest-Shamir-Adelman)**.** Given two randomly generated primes r and s, generate a public key (q, e) where $q = rs$,$\gcd(e, \phi(q)) = 1$ and $\phi(q) = (r-1)(s-1)$. The secret key is $d = e^{-1}(\bmod\ \phi(q))$. Given a message m, encryption of the message produces ciphertext $c = m^e(\bmod\ q)$. To decrypt, the ciphertext is raised to the power of the secret key $c^d(\bmod\ q) = (m^e)^{e^{-1}} = m$. Multiplying two ciphertexts c_1 and c_2 encrypting m_1 and m_2 respectively, produces the encryption of $m_1 \cdot m_2$.

In order to improve the security of RSA, randomised padding is introduced prior to the encryption of the message. Therefore we are encrypting $p(m)$ for random padding function p instead of just encrypting m. Unfortunately, this removes the homomorphic properties of padded-RSA as we have $E(p(m_1) \cdot p(m_2)) \neq E(p(m_1)) \cdot E(p(m_2))$.

2.1.2 El Gamal

The El Gamal cryptosystem [37] is another multiplicatively homomorphic public-key cryptosystem which is based upon the Diffie-Hellman (DH) key exchange. The security of this scheme is reliant on the hardness of discrete logarithm problem, more specifically the Decisional Diffie-Hellman (DDH) assumption:

Definition 2.1.2 (Decisional Diffie-Hellman)**.** Given a group $\mathbb{G}$ with generator g, distinguishing between the distributions $\langle g^x, g^y, g^{xy} \rangle$ and $\langle g^x, g^y, g^z \rangle$ is known to be hard, where x, y and z are randomly chosen integers in $\mathbb{G}$.

Given this definition we can now define the El Gamal cryptosystem as follows:

Definition 2.1.3 (El Gamal)**.** Let g be a generator of the cyclic group $\mathbb{G}$ with order q. Choose a random element $s \in \mathbb{Z}_q$ as the secret key and generate the public key $h = g^s$.

To encrypt the message $m \in \mathbb{G}$, we define the ciphertext (c_1, c_2) where $c_1 = g^u$ and $c_2 = mh^u$, where $u \in \mathbb{Z}_q$ is chosen randomly. To decrypt the ciphertext, compute $m = c_2 \cdot c_1^{-s}$. Multiplying two ciphertexts $c_1 = (g^{u_1}, m_1 h^{u_1})$ and $c_2 = (g^{u_2}, m_2 h^{u_2})$ component-wise to produce $c_3 = (g^{u_1+u_2}, m_1 m_2 h^{u_1+u_2})$ is a valid encryption of the message $m_1 m_2$.

It is noted that although standard El Gamal only supports multiplicative homomorphisms it can be modified to become additively homomorphic, this variant sometimes being known as exponential El Gamal. This additively homomorphic variant has been used in electronic voting systems[30] and is preferred over the use of the Paillier cryptosystem when handling messages of "small" size.

The construction of the exponential El Gamal encryption scheme is based on altering the message space to $\mathbb{Z}_q$ with addition modulo the prime q as a group operation. In contrast to using the standard message space G_q with multiplication as its group operation as used in regular El Gamal. This is achieved by encrypting messages m as exponents, in other words, the messages that are to be encrypted are of the form g^m for group generator $g \in G_q$. This produces the necessary additive homomorphism for we have $g^{m_1} \cdot g^{m_2} = g^{m_1+m_2}$ as required. However, encrypting messages of this form increases the complexity of the decryption procedure as it requires the computation of the discrete log, which is notably a difficult problem. Therefore exponential El Gamal is only used for "small" messages where the discrete log can be computed efficiently using either a lookup table or algorithmically. Otherwise if one was looking to use an additively homomorphic encryption scheme that supported messages of arbitrary length efficiently, they would consider using the following encryption scheme.

2.1.3 Paillier

Paillier introduced an additively homomorphic encryption scheme [64] that is used for electronic voting systems for homomorphically counting votes. The security of this scheme is based upon the Decisional Composite Residuosity Assumption (DCRA):

Definition 2.1.4 (Decisional Composite Residuosity Assumption)**.** Given a composite integer q and $x \in \mathbb{Z}_{q^2}$ it is hard to find $y \in \mathbb{Z}_{q^2}$ such that $x \equiv y^q (\text{mod } q^2)$.

Subsequently we can define the Paillier cryptosystem as follows:

Definition 2.1.5 (Paillier)**.** Two prime numbers r, s are chosen at random to generate the public key $q = rs$. From this the secret key is defined as the tuple (λ, μ) where $\lambda = \text{LCM}(r-1, s-1)$ and $\mu = \lambda^{-1} (\text{mod } q)$. Encrypting a plaintext message $m \in \mathbb{Z}_q$ produces the ciphertext $c = (1+q)^m u^q (\text{mod } q^2)$ where $u \in \mathbb{Z}_q^*$.

The Paillier cryptosystem is only additively homomorphic as there is no known method for multiplying two given ciphertexts encrypted under the same Paillier encryption without knowing the secret key. However it is possible to multiply an encrypted plaintext $E(m_1)$ with an ordinary plaintext m_2 via exponentiation $E(m_1)^{m_2}$. We obtain $D(E(m_1)^{m_2} \bmod n^2) = m_1 \cdot m_2$.

2.2 The Birth of FHE

Having seen various partially homomorphic encryption schemes based on group homomorphisms, the subsequent natural step was to develop fully homomorphic encryption schemes. It must be noted that before the breakthrough in 2009 there were previous attempts at proposing such schemes.

The most notable of these is the work of Boneh et al. [6]. This homomorphic, public key encryption scheme is based upon finite groups which support a bilinear map. It is additively homomorphic via the use of a construction based upon the Paillier cryptosystem but allows a single multiplication operation via the bilinear map.

The main idea behind this scheme is we have two finite multiplicative cyclic groups $\mathbb{G}$ and $\mathbb{G}_1$ and a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_1$. Thus for all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}$ we have $e(u^a, v^b) = e(u, v)^{ab}$. Therefore given two ciphertexts $c_1 := g^{m_1} h^{r_1} \in \mathbb{G}$ and $c_2 := g^{m_2} h^{r_2} \in \mathbb{G}$ we set the ciphertext $c := e(c_1, c_2) h_1^r \in \mathbb{G}_1$. Resulting in the ciphertext $c = g_1^{m_1 m_2} h_1^{\tilde{r}} \in \mathbb{G}_1$. Since the groups $\mathbb{G}$ and $\mathbb{G}_1$ are additively homomorphic, an arbitrary number of additions can be performed followed by a single multiplication that can then be succeeded by arbitrarily more additions.

Prior to the following subsection, it is useful to note the difference between a somewhat homomorphic encryption (SHE) scheme and a fully homomorphic encryption scheme. An encryption scheme is known as fully homomorphic if it can evaluate a binary arithmetic circuit of unbounded size and depth. An encryption scheme is known as somewhat homomorphic if it can evaluate a non-empty set of binary arithmetic circuits of limited size and depth.

2.2.1 Gentry's Breakthrough

It was not until 2009 that a breakthrough into fully homomorphic encryption was made in Gentry's thesis [39]. The work proposed building a somewhat homomorphic encryption (SHE) scheme from ideal lattices.

Given a plaintext element which exists within an ideal, the encryption of this element alters the plaintext to another element within the ideal such that it is infeasible to compute the nearest ideal element to the ciphertext without having access the secret key to remove the alteration. To ensure that the scheme is circularly secure the encryption procedure must be non-deterministic so that multiple encryptions of the same plaintext produce different ciphertexts. This is achieved by adding a small "error" or "noise" element that is chosen at random.

This "noise" element is removed during the decryption procedure, however the noise is only removed correctly if it is "small". This is due the addition and multiplication operations not only affecting the original plaintext element but the noise element as well. Each operation has the affect of moving the ciphertext further away from the original ideal element as the noise grows in size. This continues to happen until the decryption algorithm incorrectly removes the noise as it erroneously correlates the ciphertext to another element in the ideal.

Subsequently the number of operations that can be performed on a single ciphertext is limited by the growth of the noise variable. The problem was trying to make this somewhat homomorphic encryption scheme into one that is fully homomorphic, in other words a scheme that allowed an unbounded number of operations to be performed without the requiring the secret key.

Gentry proposed a solution known as bootstrapping, which was a technique of connecting a succession of SHE schemes together an arbitrary number of times to form a fully homomorphic encryption scheme. Computations are specified as arithmetic circuits within FHE schemes and the decryption procedure can also be represented as such. The concept of bootstrapping is a FHE cryptosystem that can homomorphically evaluate its own decryption circuit. This insinuates that if one can create an SHE scheme that can evaluate its own decryption circuit followed by at least one NAND gate then they would obtain an FHE scheme.

The decryption circuit has two input gates, one that accepts a ciphertext and the other accepting a secret decryption key, which outputs a single plaintext message. The general idea to homomorphically evaluate this circuit involves providing an encryption of the secret key to the circuit along with the ciphertext that you want to refresh the noise level for. This produces another ciphertext encrypting the same plaintext message under the same public key, except the noise level is at the level of a freshly encrypted ciphertext. Thus further operations can be computed without removing the correctness of decryption. In theory this bootstrapping technique can be computed arbitrarily many times thus producing an FHE scheme. However this is a computationally expensive procedure and has led to cryptographers searching for an alternative method to bootstrapping for managing the noise levels. A variant of this bootstrapping method will be described in more detail in Chapter 3.

Gentry's scheme was one of the first of many schemes to begin using lattice based cryptography which relied on a known mathematical problem known as the learning with errors (LWE) problem [67] which will be introduced formally later in this thesis in section 3.2. This lattice problem also forms the basis of the security for the next three schemes described in this chapter.

Ultimately Gentry's breakthrough led to considerable developments on lattice based fully homomorphic encryption schemes as described in the following subsections.

2.2.2 The BGV Scheme

Brakerski, Gentry and Vaikuntanathan [13], after which the scheme is named, produced an alternative method of noise management that mitigated the need for Gentry's expensive bootstrapping method. The scheme utilised two new techniques, key-switching and modulus switching introduced by Brakerski and Vaikuntanathan [15, 16]. Key-switching allows the user to reduce the dimension of the ciphertext and modulus switching is used to decrease the magnitude of the noise element. Thus using these techniques mitigates the need for Gentry's bootstrapping method as well as the requirement to "squash" the decryption circuit in order to reduce its circuit complexity.

These innovative techniques produced what are known as leveled fully homomorphic encryption schemes. Briefly, the scheme contains a chain of moduli relative to a ciphertext, the number of moduli in the chain corresponds to the number of levels in the scheme. Given a ciphertext $c \in \mathbb{R}_q$ that is an element of a modulo ring $\mathbb{R}_q$ for modulus q, one can convert the ciphertext into $c' \in \mathbb{R}_q'$ where $q' < q$, such that c and c' encrypt the same plaintext message but relative to a different ciphertext modulus. The key difference between the two ciphertexts is the noise level has been approximately reduced by a factor of $\frac{q}{q'}$. This scheme will be described in more detail in Chapter 3.

2.2.3 The FV Scheme

Following the work of Brakerski [12], Fan and Vercauteren [38] developed an FHE scheme that is similar to the BGV scheme conceptually. The scheme differs from the previously described BGV scheme in that it obtains high levels of security through the introduction of additional "noise" elements thus allowing for the use of smaller parameters and thus a smaller algebra. The benefit of this is utilising a smaller algebra for the homomorphic operations increases the speed of the processes without sacrificing the level of security. Fan and Vercauteren achieve this by introducing a scaling factor that is present within the ciphertext throughout the algorithm up to and including the decryption phase defined as follows:

$$m = \left[\left\lfloor \frac{p}{q}[\langle \vec{c}, \vec{s} \rangle]_q \right\rceil\right]_p \tag{2.1}$$

where p is the plaintext prime and p/q is the scaling factor. The idea for preventing the scaling factor from affecting the correctness of the decryption procedure is to multiply the message polynomial by a large factor in order to make to the scale negligible and easily removed through rounding. This FHE variant results in slower noise growth.

2.2.4 The GSW Scheme

Gentry et al. [41] proposed an alternative LWE-based FHE scheme that made the homomorphic operations easier to compute. Previous schemes relied on key-switching in order to reduce the dimension of the ciphertext after each homomorphic multiplication because in previous schemes multiplication is the tensor product of two ciphertexts.

The GSW scheme proposed a FHE scheme where the plaintext messages are integers instead of polynomials and the data is encrypted into matrices as opposed to vectors as in the BGV scheme. This mitigates the need for both modulus switching and key-switching as a result, which although creates a conceptually simpler FHE scheme arguably decreases the efficiency as opposed to previous FHE schemes as additions and multiplications are now performed using matrix additions and multiplications.

Khedr et al. [54] ported the GSW scheme to the RWLE setting in order to improve the scheme's efficiency. However the scheme continues to have a higher complexity than previous schemes.

2.2.5 Non-LWE-Based FHE Schemes

In addition to the LWE-based schemes that have been described previously, Gentry's framework for building an SHE scheme that can evaluate its own decryption circuit homomorphically was explored using alternative cryptosystems. Two of note are AGCD and NTRU-based [49] schemes.

Such a scheme of note is the one proposed by van Dijk et al. [32]. Using Gentry's framework [39], van Dijk et al. made it conceptually simpler by reducing the security problem of their scheme from the LWE problem to the Approximate Greatest Common Divisor (AGCD) problem. As a result of this the homomorphic operations are performed on integers rather than operations on ring elements in the LWE-based scheme. However to ensure a practical level of security within this scheme the size of the public key is $O(\lambda^{10})$ for security parameter λ which is too large for practical purposes.

It is also noted that NTRU-based HE schemes are conceptually similar to LWE-based schemes however ciphertexts can be represented using a single polynomial element as opposed to the two polynomial elements required to represent ciphertexts in LWE-based schemes. This allows for faster homomorphic operations although the noise growth rate is considerably higher in NTRU-based schemes as shown by Lepoint and Naehrig [57].

2.3 Summary

As it has been presented in this section, the concept of achieving a fully homomorphic encryption scheme had been an open problem first mentioned in 1978 [68]. It was not until the 2009 breakthrough by Gentry [39] that such a scheme was first shown to be achievable. Although lacking in practical efficiency the concept of building a SHE scheme that can evaluate its own decryption circuit was further explored using the various cryptosystems LWE, AGCD and NTRU. This eventually led to new techniques being proposed that improved the overall efficiency of FHE schemes such as modulus switching and key-switching.

Gentry's work has led to numerous FHE variants that have in turn produced implementations such as HElib [48] which is based on the BGV scheme [14] and S.E.A.L. [18] which is based on the FV scheme [38] as well as TFHE [24] which is based on the GSW scheme [41]. In addition to this there is an encryption library called PALISADE [66] which supports multiple FHE protocols including the FV, BGV and various other schemes.

Chapter 3

Mathematics of FHE

This chapter will introduce important mathematical concepts that will be referenced throughout the duration of the thesis. We begin with a brief introduction to the concept of algebraic lattices and rings which are integral to the learning with errors problem. Naturally this leads into a description of the mathematical problem with which the FHE schemes used in this thesis are based upon. Finally, the remaining chapters of this thesis describe applications that are based upon a BGV variant of FHE. Due to this an introduction and detailed description of the BGV scheme will be provided in section 3.3, along with descriptions of the algorithms which make up the scheme.

3.1 Introduction To Lattices and Rings

This section will introduce the theoretical concepts on which the LWE-based schemes are based upon.

Given an m-dimensional Euclidean space $\mathbb{R}^m$, a lattice is the vector space

$$\mathcal{L}(\vec{b}_1, ..., \vec{b}_n) = \left\{ \sum_{i=1}^{n} x_i \vec{b}_i : x_i \in \mathbb{Z} \right\} \tag{3.1}$$

defined by all integral linear combinations of n linearly independent vectors $\vec{b}_1, ..., \vec{b}_n \in \mathbb{R}^m$ for $m \geq n$. The set of vectors $\vec{b}_1, ..., \vec{b}_n$, or lattice basis can also be represented as a matrix

$$B = [b_1, ..., b_n] \in \mathbb{R}^{m \times n} \tag{3.2}$$

where each basis vector is a column of the matrix. Thus equation 3.1 can be rewritten using matrix notation as $\mathcal{L}(B) = \{Bx : x \in \mathbb{Z}^n\}$ where Bx is a matrix-vector multiplication. The integers m and n are called the *dimension* and *rank* of the lattice respectively.

Recall that a ring R is a set of elements with two binary operations, addition and multiplication. We define an ideal as follows

Definition 3.1.1. Given a ring R, the ideal I is a subset of R that forms an additive group and is a left ideal if for all $a \in I$ and $r \in R$ there exists $ra \in I$. Likewise a I is a right ideal if for all $a \in I$ and $r \in R$ there exists $ar \in I$. A two-sided ideal is an ideal that is both a left and right ideal.

We consider two-sided ideals and refer to these as simply ideals. The rings we use in this thesis are quotient rings defined below.

Definition 3.1.2. Given a ring R and a two-sided ideal I, the quotient ring of R mod I is the group of cosets R/I with the operations of coset addition and coset multiplication.

We will be mostly working with integer polynomial rings of the form $R = \mathbb{Z}[X]/(\Phi_m(X))$ where $\Phi_m(X)$ is the m'th cyclotomic polynomial. Furthermore the quotient ring $R_p = R/pR$ is the ring of integer polynomials modulo $p \in \mathbb{Z}$.

3.2 The Learning with Errors Problem

Cryptographic schemes are commonly based upon hard mathematical problems in terms of security. Examples of these types of problems would be the discrete logarithm problem and integer factorisation, both of which will be described in the following section. The security of fully homomorphic encryption is similarly based upon the hardness of a mathematical problem, namely, the learning with errors (LWE) problem [67] which will now be introduced.

Prior to introducing the mathematical problems of which the security of FHE schemes are based upon, we introduce the inner product, also known as the dot product or scalar product, between two vectors

Definition 3.2.1 (Inner product)**.** Given two vectors $\vec{a} = (a_1, a_2, \ldots, a_n)$ and $\vec{b} = (b_1, b_2, \ldots, b_n)$ we define the inner product as

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

where n denotes the dimension of the vector space. Note that the dimension of the two vectors must always be equal.

For the duration of this thesis we denote the inner product of two vectors $\vec{a}$ and $\vec{b}$ as $\langle \vec{a}, \vec{b} \rangle$.

The LWE problem is based on the "learning from parity with error" problem defined as follows:

Definition 3.2.2 (Learning from parity with error)**.** Given a system of equations with "errors"

$$\begin{aligned} \langle s, a_1 \rangle &\approx_\epsilon b_1 (\text{mod } 2) \\ \langle s, a_2 \rangle &\approx_\epsilon b_2 (\text{mod } 2) \\ &\vdots \end{aligned}$$

where the a_i's are chosen independently from a uniform distribution over $\mathbb{Z}_2^n$. The problem is finding the value of $s \in \mathbb{Z}_2^n$ when the b_i's are correct with probability $1 - \epsilon$ for integer $n \geq 1$ and real number $1 \geq \epsilon \geq 0$.

Note that when $\epsilon = 0$ the problem is trivial to solve via Gaussian elimination. This is the standard "learning from parity" problem. Blum et al. [5] contributed the current best known sub-exponential algorithm for solving the LWE problem.

The Learning with Errors (LWE) problem is an extension of the learning from parity with error problem defined above. The LWE problem introduced by Regev [67] is defined as follows:

Definition 3.2.3 (LWE)**.** For security parameter λ, let $n = n(\lambda)$ be an integer dimension, let $q = q(\lambda) \leq 2$ be an integer, and let $\chi = \chi(\lambda)$ be a distribution over $\mathbb{Z}$. The $LWE_{n,q,\chi}$ problem is to distinguish the following two distribution: In the first distribution, one samples $(\vec{a_i}, b_i)$ uniformly from $\mathbb{Z}_q^{n+1}$. In the second distribution, one first draws $\vec{s} \leftarrow \mathbb{Z}_q^n$ uniformly and then samples $(\vec{a_i}, b_i) \in \mathbb{Z}_q^{n+1}$ by sampling $\vec{a_i} \leftarrow \mathbb{Z}_q^n$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = \langle \vec{a}, \vec{s} \rangle + e_i$. The $LWE_{n,q,\chi}$ assumption is that the $LWE_{n,q,\chi}$ problem is infeasible.

It was proven by Regev [67] that for specific Gaussian error distributions χ and moduli q that the $LWE_{n,q,\chi}$ assumption is true if and only if certain worst-case lattive problems are hard to solve using a quantum algorithm.

Brakerski et al. [14] state a definition of Regev's result using B-bounded distributions, which is a distribution over the integers where the magnitude of a sample is bounded with high probability. Before presenting the definition for B-bounded distributions, we introduce the definition of a negligible function

Definition 3.2.4 (Negligible function)**.** A function μ is negligible if and only if for all $c \in \mathbb{N}$, there exists $n_0 \in \mathbb{N}$, such that for all $n \geq n_0$, we have $\mu(n) \geq n^{-c}$.

Using definition 3.2.4 we can now define B-bounded distributions

Definition 3.2.5 (B-bounded distributions)**.** A distribution sample $\{\chi_n\}_{n \in \mathbb{N}}$ supported over the integers, is called B-bounded if

$$e \overset{\Pr}{\leftarrow} \chi_n[|e| > B] = \mathsf{negl}(n),$$

where $\mathsf{negl}(n)$ is a negligible function.

Regev [67] presented a worst-case to average-case reduction for the LWE problem which is captured in the following theorem:

Theorem 1 (Regev [67])**.** *For any integer dimension n, prime integer $q = q(n)$, and $B = B(n) \geq 2n$, there is a B-bounded distribution χ that can be efficiently sampled such that if there exists an efficient (possibly quantum) algorithm that solves $LWE_{n,q,\chi}$, then there is an efficient quantum algorithm for solving $\tilde{O}(qn^{1.5}/B)$-approximate worst-case SIVP.*

Recall the definition of the known lattice problem,

Definition 3.2.6 (Shortest Independent Vectors Problem (SIVP))**.** Given a lattice L with rank n, find n linearly independent vectors $v_1, \ldots, v_n \in L$ such that

$$\|v_i\|_p \leq c\lambda_n^{(p)}(L)$$

for all $i = 1, \ldots, n$.

Theorem 1 shows that LWE is as hard as the well known lattice problem SIVP. This problem was subsequently extended to a ring variant known as the ring learning with errors.

3.2.1 The Ring Learning with Errors Problem

First introduced by Lyubashevsky et al. [60], the ring learning with errors (RLWE) problem is defined as follows

Definition 3.2.7 (RLWE)**.** For security parameter λ, let $f(x) = x^d + 1$ where $d = d(\lambda)$ is a power of 2. Let $q = q(\lambda) \geq 2$ be an integer. Let $R = \mathbb{Z}[x]/(f(x))$ and let $R_q = R/(qR)$. Let $\chi = \chi(\lambda)$ be a distribution over R. The $RLWE_{d,q,\chi}$ problem is to distinguish the following two distributions: In the first distribution, one samples (a_i, b_i) uniformly from R_q^2. In the second distribution, one first draws $s \leftarrow R_q$ uniformly and then samples

$(a_i, b_i) \in R_q^2$ by sampling $a_i \leftarrow \mathbb{R}_q$ uniformly, $e_i \leftarrow \chi$, and setting $b_i = a_i \cdot s + e_i$. The $RLWE_{d,q,\chi}$ assumption is that the $RLWE_{d,q,\chi}$ problem is infeasible.

The RLWE problem is known to be hard as the extensively studied shortest vector problem (SVP) over ideal lattices can be reduced to it [60]. The shortest vector problem is defined as follows:

Definition 3.2.8 (SVP)**.** Given a basis $B \in \mathbb{Z}^{m \times n}$, find a non-zero lattice vector Bx for $x \in \mathbb{Z}^n \backslash \{0\}$ such that $\|Bx\| \leq \|By\|$ for all $y \in \mathbb{Z}^n \backslash \{0\}$.

This problem has been shown to be NP-hard [62], thus making the RLWE problem to be at least NP-hard.

3.2.2 The General Learning with Errors Problem

Brakerski et al. [14] noticed that the LWE and RLWE problems were identical syntactically, aside from using difference rings and different vector dimensions. This led to their definition of the general learning with errors (GLWE) problem

Definition 3.2.9 (GLWE)**.** Let $n, m, q \in \mathbb{Z}$; let $R = \mathbb{Z}[X]/(\Phi_m(X)), R_q = R/(qR)$; and let χ be a distribution over R. Given arbitrarily many samples $(\vec{x}, y_i) \in R_q^{n+1}$, where $y_i = \langle \vec{x}_i, \vec{s} \rangle + e_i$, with $\vec{x}_i, \vec{s} \leftarrow R_q^n$ sampled uniformly and $e_i \leftarrow \chi$, find $\vec{s}$.

The specific LWE variant of the GLWE problem is when $m = 1$ in definition 3.2.9. In other words R is the ring of integers $\mathbb{Z}$. The ring learning with errors (RLWE) problem uses a polynomial ring $m \geq 1$ as opposed using $\mathbb{Z}$ and the dimension is different, namely $n = 1$ in definition 3.2.9 for the RLWE case.

Using the GLWE problem, Brakerski, Gentry and Vaikuntanathan [14] defined an FHE scheme known as the BGV scheme, named after the authors.

3.3 The BGV Scheme

The BGV scheme [14] is based upon the GLWE problem and is the scheme which the remaining sections of the thesis will be based upon. This section discusses the algorithmic overview of the BGV scheme followed by a more in-depth description of the two techniques that were introduced to replace Gentry's bootstrapping algorithm.

Given a polynomial ring modulo a cyclotomic polynomial $\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X))$ we define the plaintext space $\mathbb{A}_p = \mathbb{A}/p\mathbb{A}$ to the polynomial ring modulo plaintext prime p. In addition to this, the corresponding ciphertext space is defined to be $\mathbb{A}_q = \mathbb{A}/(q\mathbb{A})$ where $q \gg p$ for odd modulus q. Since all FHE data is contained within polynomial rings, plaintext messages need to be encoded into polynomials $m \in \mathbb{A}_p$ before they are encrypted into ciphertext polynomials $c \in \mathbb{A}_q$.

The BGV scheme is a public key encryption scheme comprising of the following main algorithms:

- *Setup*(λ, L): Given a security parameter λ and the number of levels L, generates a ladder of decreasing moduli from q_L to q_0 of size $(L+1) \cdot \mu$ bits and μ bits respectively. The parameter μ is chosen alongside other parameters to ensure the scheme is based on a General Learning With Errors (GLWE) instance which provides 2^λ bit security against known attacks. Let the ring $\mathbb{A} = \mathbb{Z}[x]/(x^d + 1)$ and let the set $params_j = (q_j, d_j, n_j, N_j, \chi_j)$ be the set of parameters defining the scheme at a specific level $0 \leq j < L$. Thus this routine generates L parameter sets.

- *KeyGen(params)*: Given the security parameter λ, generates the public encryption key pk and a corresponding secret decryption key sk.

- *Encrypt*(m, pk): Given a plaintext message m and public key pk, encrypts the message into a ciphertext c.

- *Decrypt*(c, sk): Given a valid ciphertext c and a secret key sk, decrypts the ciphertext to obtain the plaintext message m.

A mathematical description of these algorithms will be provided in the following subsections. Note that the above list of algorithms describes a plain GLWE-based encryption scheme without homomorphic operations. The following detailed descriptions will be followed with a description of the homomorphic additions and multiplications. Finally the section will be concluded with a description of the two novel techniques that were introduced to the BGV scheme to mitigate Gentry's bootstrapping operation, namely modulus switching and key-switching.

3.3.1 Setup

This section provides a description of the setup phase of the BGV scheme given a security parameter λ, representing 2^λ security against known attacks. This value is typically chosen to be $\lambda = 100$ [14].

Firstly we determine if we are setting parameters for a LWE-based scheme or a RLWE-based scheme. Recall from the description of the GLWE problem 3.2.9 that LWE is GLWE with $d = 1$ which defines the ring $\mathbb{A} = \mathbb{Z}[X]/(x + 1) = \mathbb{Z}$. Alternatively, the RLWE problem is GLWE with $n = 1$ which defines the vector dimensions over the polynomial ring $\mathbb{A}^n$. Note that when selecting a RLWE-based scheme, the degree d of the polynomial $x^d + 1$ is chosen to be a power of 2.

To create the scheme we select the vector dimension n, an odd modulus q, a *noise* distribution χ over the ring $\mathbb{A}$, which is typically chosen to be a Gaussian distribution, and $N = \lceil (2n + 1) \log q \rceil$. Note, given a ciphertext space $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$ we can assume the plaintext space is $\mathbb{A}_2 = \mathbb{A}/2\mathbb{A}$ for simplicity. However larger plaintext spaces are possible of the form $\mathbb{A}_p = \mathbb{A}/p\mathbb{A}$ for prime number p where $p \ll q$.

It must be noted that the noise distribution χ is chosen to be as small as possible. The reason for this will become clearer after the introduction of homomorphic operations to the scheme which will be described in subsequent subsections. As mentioned in [14, p.8], to achieve 2^λ security against known lattice attacks, one must have $n \cdot d = \Omega(\lambda \cdot \log(q/B))$ where B is the bound on the length of the noise.

Thus, prior to the setup phase, we should obtain a set of parameters $params = (q, d, n, N, \chi)$ defining the scheme that shall be used. However, looking back at the definition of *Setup(*λ, L*)* notice that there are multiple of levels L and the set of parameters, $params_j$, define the ladder of decreasing moduli from q_L to q_0. It is noted that the ring dimension d_j and noise distribution χ_j are independent of L and can be denoted as $d = d_L$ and $\chi = \chi_L$ respectively. The vector dimension n_j may depend on the circuit level.

3.3.2 Key generation

Following the description provided in [14, p.11], we proceed to provide a description of the generation of both the secret and public keys of an FHE scheme. Given the set of parameters $params_j$ we create a secret $\vec{s_j} \in \mathbb{A}_q^{n+1}$ and public counterpart $A_j \in \mathbb{A}_q^{n+1 \times 1}$ for each circuit level $0 \leq j \leq L$ that make up the secret key sk and public key pk respectively.

More specifically, the secret key sk is a collection of randomly chosen vectors $\vec{s_j}$ where each $\vec{s} \leftarrow (1, s'_1, s'_2, \ldots, s'_n) \in \mathbb{A}_q^{n+1}$ and each $s' \leftarrow \chi^n$. Namely, each s' is an n-dimensional vector chosen randomly from a Gaussian distribution χ. For simplicity we shall denote the secret key element $\vec{s_j} = (1, s')$ where $s' \in \mathbb{A}_q^n$.

Using the previously generated secret key sk and the set of parameters $params_j$ we generate a uniform matrix $A' \leftarrow \mathbb{A}_q^{N \times n}$ and choose an N-dimensional "noise" vector $\vec{e} \leftarrow \chi^N$. We then set $\vec{b} \leftarrow A's' + 2\vec{e}$. Now we can define our public key as the $(n+1)$-column matrix consisting of $\vec{b}$ followed by the n columns of $-A'$. Thus our public key is the collection of $A'_j s$. Note prior to the key generation steps we can now perform the operation $A \cdot \vec{s} = 2e$, which is an important property that will be useful for the decryption phase.

As mentioned previously the public and secret keys are a collection of $A'_j s$ and $\vec{s_j}' s$ respectively, with each element corresponding to a specific level $0 \leq j \leq L$. Therefore it is necessary to provide a method of switching between each of the keys. This is achieved through the novel method introduced in [14] called key switching which is described in more detail in section 3.3.6, thus concluding the description of the key generation phase.

3.3.3 Encryption and Decryption

Having generated the necessary secret and public keys of the FHE scheme it is now possible to encrypt and decrypt messages. This section will describe this stage of the scheme.

Encryption

Let the message $m \in \mathbb{A}_2$ be the message that we want to encrypt. Given a previously generated public key A as described in the previous section and message m we set the message vector $\vec{m} \leftarrow (m, 0, \ldots, 0) \in \mathbb{A}_q^{n+1}$ for vector dimension n and modulus q as defined in the parameter set $params_j$ in the previous sections.

We then randomly choose a vector $\vec{r} \leftarrow \mathbb{A}_2^N$ typically from a uniform distribution. Note that the coefficients of the vector are from the range $[0, 2)$ where $[\cdot, \cdot)$ denotes a half-open interval that only includes the endpoint of the left side of the interval. Specifically in our case the coefficients are integers chosen to be greater than or equal to 0 and less than 2. This is to ensure the vector has a low norm, where the norm we use is the Euclidean norm or *L2-norm*. Given an n-dimensional Euclidean space $\mathbb{R}^n$ the Euclidean norm of a vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ is defined as $\|\vec{v}\|_2 := \sqrt{v_1^2 + \cdots + v_n^2}$. Choosing $\vec{r}$ to have low norm reduces the size of the "noise" which is important to keep small, the reason for this will become apparent in the following subsection describing the homomorphic arithmetic operations.

Having obtained the message vector $\vec{m}$, the randomly chosen vector $\vec{r}$ and given the public key A we define the ciphertext $\vec{c} \leftarrow \vec{m} + A^T r \in \mathbb{A}_q^{n+1}$, where A^T denotes the transpose of matrix A.

Note that when encrypting, the resultant ciphertext $\vec{c}$ always begins at the highest circuit level L. Thus the public key received as an input to the encryption method is typically A_L, which in turn produces a ciphertext $\vec{c} \in \mathbb{A}_{q_L}^{n+1}$.

Decryption

Suppose we have a ciphertext that is encrypted under some secret key $\vec{s_j}$. In [14, p.11] it is noted that the ciphertext could be augmented with an index which indicates which level it belongs to. In order to decrypt the ciphertext we perform an inner product, as defined in definition 3.2.1. This is then followed by two modulo operations, first with the ciphertext modulus q and then with the plaintext modulus p. We use the notation $[e]_q$ to denote the modulo operation e mod q for some element e and modulus q.

The decryption formula as stated on [14, p.8] produces a plaintext message $m \leftarrow [[\langle \vec{c}, \vec{s_j} \rangle]_q]_2$. This decryption formula involves three steps. Firstly the inner product between the input ciphertext $\vec{c}$ with its corresponding secret key $\vec{s_j}$. This is then followed by the reduction of the output with q, note that this modulus will also be relative to the same level as the secret key, namely q_j. Finally reducing the output again with the plaintext modulus, which in our case is chosen to be 2, produces the desired plaintext message m.

3.3.4 Arithmetic operations

Subsequent to obtaining the ability to decrypt and encrypt homomorphically, we can now explore how this enables us to perform arithmetic operations between ciphertexts, namely

- *Add(pk, c_1, c_2)*: Given two ciphertexts $c_1 = E(m_1)$ and $c_2 = E(m_2)$ perform an addition operation resulting in a ciphertext c_3 containing the sum of the two ciphertexts $c_3 = E(m_1 + m_2) = E(m_1) + E(m_2) = c_1 + c_2$.

- *Mult(pk, c_1, c_2)*: Given two ciphertexts $c_1 = E(m_1)$ and $c_2 = E(m_2)$ perform a multiplication operation resulting in a ciphertext c_3 containing the product of the two ciphertexts $c_3 = E(m_1 \times m_2) = E(m_1) \times E(m_2) = c_1 \times c_2$.

Notice that each method also takes the set of public keys pk as an input. This is necessary for performing any key switching of modulus switching as pk contains every public key as well information on how to switch between them.

Prior to describing these arithmetic operations, we remind the reader that we encrypt our messages with an added "noise" element, previously defined as $\vec{r} \leftarrow \mathbb{A}_2^N$ in the encryption phase and $\vec{e} \leftarrow \chi^N$ in the public key generation phase. This is in order to ensure the security of our homomorphic encryption scheme based on the GLWE problem. However a byproduct of this method is the noise element grows when homomorphic operations are performed on a ciphertext. This is the reason why it is important to keep the noise elements as small as is feasible without jeopardising the security. Due to all elements existing within a modulo ring $\mathbb{A}_q$, the coefficients of the noise polynomial must be kept smaller than q_j as to avoid wraparound. If wraparound occurs, in other words the noise grows larger than q and is reduced to a smaller element, then the decryption process will not successfully remove all of the noise.

This in turn means there are a limited number of operations that one can perform on a ciphertext before the noise element grows too large and no further operations can be performed while continuing to preserve the correctness of the message when decrypted. Gentry introduced the computationally expensive operation known as bootstrapping [39] however the BGV scheme introduced modulus switching as a computationally cheaper alternative.

Brakerski et al. begin by defining an arithmetic circuit of depth L and generating the necessary parameters. This is the reason for generating a chain of moduli, each corresponding to a specific level of the arithmetic circuit that we want our FHE scheme to be capable of evaluating. A more detailed description of this method is provided in section 3.3.5.

It is therefore preferable to have a ciphertext which has been freshly encrypted to have a minimal amount of noise so as to maximise the number of operations that can be performed before the requirement of modulus switching.

Addition

The addition of two ciphertexts produces another ciphertext containing an encryption of the sum. A byproduct of this is the noise, n, grows at most linearly at a rate of $2n$ in the number of addition operations.

Given two ciphertexts $\vec{c_1}$ and $\vec{c_2}$, first ensure they are both encrypted under the same secret key $\vec{s_j}$ as the BGV scheme does not support operations between two ciphertexts encrypted under different keys. If they are not initially encrypted under the same key then perform a "FHE.Refresh" method [14, p.12] which involves a modulus switch operation followed by a key switch operation so that the ciphertext corresponding to the higher circuit level is brought down to the same level as the ciphertext which to be operated with.

Now that the two ciphertexts correspond to the same secret key $\vec{s_j}$, set $\vec{c_3} \leftarrow \vec{c_1} + \vec{c_2} \bmod q_j$. This is a vector addition, which is an entry-wise addition operation. We then interpret $\vec{c_3}$ as a ciphertext under the secret key $\vec{s_j}' = \vec{s_j} \otimes \vec{s_j}$, the tensor of the secret key $\vec{s_j}$.

We then "refresh" the ciphertext $\vec{c_3}$ by modulus switching from q_j to q_{j-1} and then key switching from $\vec{s_j}'$ to $\vec{s}_{j-1}$. This produces another "refreshed" ciphertext $\vec{c_4}$ corresponding to the secret key $\vec{s}_{j-1}$ and is output by the addition method.

Multiplication

The addition of two ciphertexts produces another ciphertext containing an encryption of the product. As with addition, the noise grows with each operation. However, the noise, n, grows at a faster rate of at most n^2 in the number of multiplications.

Given two ciphertexts $\vec{c_1}$ and $\vec{c_2}$, first ensure they are both encrypted under the same secret key $\vec{s_j}$ as in the addition routine. If not then perform the necessary modulus and key switching operations to bring the ciphertext corresponding to the higher level down to the level of the other ciphertext.

Subsequent to obtaining two ciphertexts correspond to the same secret key $\vec{s_j}$, set $\vec{c_3} \leftarrow \vec{c_1} \otimes \vec{c_2} \bmod q_j$. The multiplication of two ciphertexts is a vector tensor product, also known as the outer product, operation defined as follows:

Definition 3.3.1 (Tensor product)**.** Given two vectors $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_m)$, we can form a tensor denoted by $v \otimes w$ which is equivalent to vw^T. This produces a matrix

$$v \otimes w = \begin{bmatrix} v_1w_1 & v_1w_2 & \ldots & v_1w_m \\ v_2w_1 & v_2w_2 & \ldots & v_2w_m \\ \vdots & \vdots & \ddots & \vdots \\ v_nw_1 & v_nw_2 & \ldots & v_nw_m \end{bmatrix}$$

The resultant ciphertext $\vec{c_3}$ representing the product is encrypted under the secret key $\vec{s_j}' = \vec{s_j} \otimes \vec{s_j}$ and thus needs to be "refreshed" as in the addition phase via modulus switching from q_j to q_{j-1} and then key switching from $\vec{s_j}'$ to $\vec{s}_{j-1}$. This produces another "refreshed" ciphertext $\vec{c_4}$ corresponding to the secret key $\vec{s}_{j-1}$ and is output by the multiplication method.

It is remarked that since addition increases the noise considerably slower than multiplication, it is not necessary to refresh the ciphertext after each addition operation.

3.3.5 Modulus Switching

Modulus switching is the novel technique proposed by Brakerski et al. [14][Section 3.3] that introduced the concept of leveled FHE schemes. The concept involves a "chain" of moduli $q_0 < q_1 < ... < q_L$ where L is the number of levels in the scheme and newly encrypted ciphertexts begin being defined in $\mathbb{A}_{q_L}$ and the lowest level ciphertext defined in $\mathbb{A}_{q_0}$. Ciphertexts at the i'th level are vectors of two dimensions $\vec{c} = (c_0, c_1) \in (\mathbb{A}_{q_i})^2$.

Given a ciphertext $\vec{c}$ that encrypts a plaintext message $\mathfrak{m}$, suppose the noise element is reaching the upper threshold and we require to reduce its relative magnitude without altering the encrypted message. This is achieved by scaling the ciphertext down by a factor $\Delta = q/q'$ where q is the current ciphertext prime and q' is the ciphertext prime that we are modulus switching down to. We require the ciphertext $\vec{c'}$ that is the $\mathbb{A}$-vector closest to $\Delta \cdot \vec{c}$ such that $\vec{c'} \equiv (q/q')\vec{c}(\text{mod } p)$.

It is formally shown in [13, Lemma 4] that it is possible to change the inner modulus of the decryption equation $m = [[\langle \vec{c}, \vec{s} \rangle]_q]_p$ while maintaining the correctness of decryption under the same secret key.

Following the optimised procedure described in [40] the method of modulus switching down is as follows:

- Let $\vec{d} = \vec{c} \bmod \Delta$
- Add or subtract multiples of Δ from the coefficients of $\vec{d}$ until $\vec{d} \bmod p \equiv 0$
- Set $\vec{c'} = \vec{c} - \vec{d}$

3.3.6 Key-Switching

Also known as dimension reduction, the key-switching method [14][Section 3.2] is used to reduce the dimension of the ciphertext after a multiplication operation. Given two ciphertexts $\vec{c}_1, \vec{c}_2$ the multiplication of two ciphertexts is the tensor product $\vec{c}_1 \otimes \vec{c}_2$. This ciphertext can still be decrypted using the tensor product of the secret key $\vec{s} \otimes \vec{s}$ using the same decryption technique, this is shown in Section 4.5.2. Naturally, the dimension of the ciphertext cannot continue to increase indefinitely if the efficiency of the scheme is to be preserved since otherwise, the ciphertext grows in size exponentially in the number of multiplications performed.

The procedure converts a tensored ciphertext $\vec{c}_1$ decryptable by the tensored secret key $\vec{s}_1 \otimes \vec{s}_1$ into a smaller two dimensional ciphertext $\vec{c}_2$ that is decryptable by a new secret key $\vec{s}_2$. The reason for switching to a different secret key is to avoid the issue of circular security.

Given the resultant ciphertext from a multiplication operation $\vec{c} = (c_0, c_1, c_2)$ which is decryptable under the secret key $\vec{s} = (1, s, s^2)$ as shown in Section 4.5.2. We want to key-switch to a canonical ciphertext $\vec{c'} = (c'_0, c'_1)$ that is decryptable under a new secret key $\vec{s'} = (1, s')$ so as to reduce the ciphertext to a 2-dimensional vector. This is achieved through defining a key-switching matrix $W = (a, b) \in \mathbb{A}^{2\times 3}$ where $a_i \in_R \mathbb{A}_q$ is a polynomial chosen randomly from the ring $\mathbb{A}_q$ and $b_i = s_i + p \cdot e - a_i \cdot s'$. Note that the i-th column of W contains an encryption of the new secret key under the i-th part of the old secret key where $\vec{s_i} = s^i$. Then we obtain our the ciphertext $\vec{c'}$ as so:

$$\vec{c'} := (c'_0, c'_1) = W\vec{c}^T = \sum_i (b_i \cdot c_i, a_i \cdot c_i) \tag{3.3}$$

This correctness of this is shown in [48][Section 3.1.6] in addition to an optimisation that reduces the total amount of noise added through this operation via decomposing the ciphertext into sections known as "digits".

3.3.7 Batching

Batching was as optimisation introduced in the BGV scheme [14, p.16] which reduced the per-gate computation from quasi-linear in the security parameter λ to poly-logarithmic by packing data into a single ciphertext.

Smart and Vercauteren [74] were the first to rigorously analyse batching in a FHE context. They observed that RLWE-based ciphertexts can have many plaintext *slots*, associated to the factorisation of the plaintext space into algebraic ideals.

For simplicity reasons the plaintext space was previously defined as $\mathbb{A}_2$, however the FHE scheme functions correctly in the same way when using a plaintext space $\mathbb{A}_p$ for some prime p. This allows for the batching of several modulo-p entries into a single ciphertext which are treated independently. Additionally, it allows the possibility of manipulating polynomials in each *slot* each of degree n, where n is the dimension defined in the setup phase described in section 3.3.1.

Given the plaintext space $\mathbb{A}_p = \mathbb{Z}_p[x]/(x^d + 1)$ for some plaintext prime p and some degree d that is a power of 2. Despite each element of $\mathbb{A}_p$ being a polynomial of degree $d - 1$, it is possible to use a specific encoding to perform operations in an entry-wise manner. This is due to evaluating an arithmetic circuit over $\mathbb{A}_p$ on input $x \in \mathbb{A}_p^n$ implicitly evaluates, for each i, the same arithmetic circuit over $\mathbb{A}_{p_i}$ on input x projected down to $\mathbb{A}_{p_i}^n$. This is formally captured in [14, Theorem 4, p.18] as follows

Theorem 2. *Let $p = 1$ mod $2d$ be a prime of size polynomial in λ. The RLWE-based instantiation of FHE using the ring $R = \mathbb{Z}[x]/(x^d+1)$ can be adapted to use the plaintext space $R_p = \otimes_{i=1}^{d} R_{p_i}$, while preserving correctness and the same asymptotic performance. For any boolean circuit f of depth L, the scheme can homomorphically evaluate f on ℓ sets of inputs with per-gate computation $\tilde{O}(\lambda \cdot L^3/min\{d, \ell\})$.*

Theorem 2 relies on a well known theorem in number theory known as the Chinese Remainder Theorem (CRT) defined as

Theorem 3 (Chinese Remainder Theorem)**.** *Let R be a commutative ring with unity, and $I_1, I_2, \ldots, I_n$ be finitely many ideals of R such that $I_i + I_j = R$ for any two distinct elements i and j of $\{1,2,\ldots,n\}$. Then, $I_1 I_2 \ldots I_n = I_1 \cap I_2 \cap \cdots \cap I_n$ and the canonical ring homomorphism $R/(I_1 I_2 \ldots I_n) \to R/I_1 \times R/I_2 \times \cdots \times R/I_n$ is an isomorphism.*

Applying this theorem to the FHE scheme, given a plaintext space $\mathbb{A}_p = \mathbb{Z}_p[x]/(x^d + 1)$, for some prime p, it is possible to break the ring down into n ideals $I_i = F_i(x)$ where $F_i(x)$ is an arbitrary factor of $x^d + 1$ modulo the p for $0 < i \leq n$. These ideals are relatively prime, thus applying the Chinese Remainder Theorem yields the isomorphism $\mathbb{A}_p \cong \mathbb{A}_p/I_1 \times \mathbb{A}_p/I_2 \times \cdots \times \mathbb{A}_p/I_n$. This result means that applying a sequence of arithmetic operations in the ring $\mathbb{A}_p$ is equivalent to performing the same operations in each $\mathbb{A}_p/I_i$ independently and then applying the isomorphism on the result.

3.3.8 Cryptographic Usage

Generally, FHE is useful in scenarios in which a client wishes to outsource computations on data without revealing the data being computed upon. This causes FHE to typically be used against an *honest-but-curious* adversary.

The *honest-but-curious* adversary is a legitimate participant in a protocol who will not deviate from the defined protocol but will attempt to learn all possible information from received messages.

Since messages received in an FHE protocol are encrypted and remain so throughout the protocol, the *honest-but-curious* adversary is unable to learn any information about the data being processed.

FHE can be of use in scenarios where data needs to be processed, however the data itself is confidential, such as in the medical sector. For example, there has been extensive work on using FHE to perform analytics, like logistic regression, on medical data [23, 9, 2, 76, 56].

There has also been various work applying FHE to private information retrieval (PIR) [78, 34, 55, 1]. PIR is when a client desires to query a server and retrieve data from its database at index i while preserving the privacy of the index from the server. This led to work on a version of PIR which had a stronger security requirement, known as single-database PIR. Single-database PIR retains the property of keeping the index secret from the server but has the added requirement of the client only receiving the entry it requested. This is also known as oblivious transfer which saw further application of FHE schemes [25, 58, 42, 7].

3.4 Summary

This section has provided a description of the underlying theoretical structure of the BGV-variant of a FHE scheme that will be used throughout the future research in this thesis. The LWE problem, which the security of FHE schemes are reliant upon was introduced by Regev and shown to be sufficiently hard. This led to the lattice-based encryption schemes that allowed the breakthrough of Gentry's FHE scheme in 2009. Having shown the community it was theoretically possible to build an FHE scheme, albeit an inefficient one, this sparked increased interest and research into building new HE schemes. This was eventually followed by the two major techniques introduced to Gentry's scheme to create the "leveled" FHE scheme. More specifically, the modulus switching and key-switching techniques that were described in this section.

Chapter 4

Anonymous Routing

This chapter will attempt to further improve the understanding of FHE, in particular the BGV scheme. This will build upon the theory introduced in section 3.3 through applying the foundational operations such as addition and multiplication to a concrete example.

As mentioned previously, a common use case for FHE is the outsourcing of computation on sensitive data. One common application of FHE is for a homomorphic comparison algorithm, often used for homomorphic database queries. This section explores and demonstrates the capability for FHE and namely a homomorphic comparison algorithm to be applied to scenarios outside of database queries. We demonstrate the possibility of using FHE to provide a different type of security than just data security but user anonymity. This section describes a routing algorithm that uses FHE to provide anonymity against a global adversary.

4.1 Introduction

In the technological era, communication has clearly become an important service whether it be over long or short distances. A desirable property for communications is the privacy of the information being sent or received. However, providing data security is sometimes not enough for clients to be confident in using such a service for communicating. Clients may often desire to remain anonymous when communicating with others, whether it be the sender, recipient, or both. This can be achieved in various ways, including encryption. This chapter is particularly interested in using Fully Homomorphic Encryption to provide communications which are not only private but also anonymous.

Private Information Retrieval (PIR) was initially introduced in 1995 by Chor et al. [26]. The concept involves a client desiring to query a database, held on some server, using an index while maintaining the privacy of the index from the server. The trivial solution to this scenario would be to simply return the entire database to the client however this would incur a large communication cost and be impractical. An extensively explored solution to this problem is a homomorphic database query [78, 34, 55, 1]. The technique enables a client to query a database without showing the intent of the query by encrypting it. This is a natural application of a homomorphic comparison algorithm. This chapter explores the versatility of FHE through the use of such an algorithm outside of the "typical" scenario. Specifically, in applying FHE to the anonymity domain.

Using the terminology proposed by Pfitzmann and Hansen [65], anonymity can be defined as when a "subject is not identifiable within a set of subjects, the *anonymity set*." Specifically, given a set of possible participants within a network it is impossible to identify the active participants at any given time. From this definition we can define specific subsets with respect to the roles of the participants. Namely, we can define *sender* and *recipient anonymity* as the ability for a particular individual to be unidentifiable within the sender and recipient anonymity sets respectively.

Anonymity schemes generally aim to provide message *unlinkability*, which is when it is not possible for an adversary to discriminate who is communicating with whom within a network where they can observe both sets of the senders and recipients.

4.2 Existing Methods for Anonymity

Due to the increase in interest for anonymity tools there has been a lot of previous research into this area. As a result of this, there exist various available services, among them are Tor [33], Freenet [28] and I2P [79]. These three are effective and readily accessible tools which have gained considerable momentum in usage and popularity. As with many schemes in security, these anonymity services are also susceptible to adversaries and a range of different attacks. Thus constant work is being performed to improve the usability and strength of these schemes.

Numerous anonymity schemes are based upon early work on a tool known as mix networks [17]. They route packets across a network via proxies known as mixes. Each mix in the path uses public key encryption, therefore an additional layer of encryption is applied proportional to the number of mixes in the path. Due to the message being relayed across the network via intermediary proxies, a passive attacker at the entry and exit point to the network is unable to determine the recipient and sender of the message respectively. In order to improve the scheme's strength against de-anonymisation, more notably correlation attacks, the mixes implement aggregation and introduce random delays upon packets received. Although these techniques improve the robustness against attacks, they in turn decrease the performance of the network. Thus mix networks are known as a high latency anonymous communication network. This type of anonymous communication network (ACN) is useful for services such as email or other services where the anonymity is more important than the speed of transmission.

As there became a higher demand for real time communication there developed a greater need for a new type of ACN, more notably, low-latency anonymous communication networks. This gave rise to an onion routing service known as Tor. Tor relies on routing packets across an overlay network know as a circuit. Circuits are paths in the network that contain numerous nodes known as onion routers. A packet that is sent across the network is encrypted with several layers of encryption. Each layer is encrypted under a different symmetric key relative to the pre-negotiated secret key held by each onion router in the designated path that the packet will take across the network. As a result of the following set-up, each node in the network only knows the immediately preceding and succeeding node in the path. Thus the exit node, i.e. the last node in the network before the final destination does not know the original sender of the packet but only the final recipient. In addition to this, since it is not necessary for users of the service to also run their own Tor onion router, they may not necessarily be a part of the Tor network. Therefore there will exist an entry node into the network from the recipient. Due to Tor being a low-latency anonymity service, the onion routers do not utilise delays or permutations of the packets. Subsequently, Tor is susceptible to attacks and more notably, like all low latency anonymous communication networks, does not protect communications against a global adversary [52].

As previously mentioned, Tor does not consider the participants of the network to individually be a part of the routing network itself. Unlike Tor there are other ACNs that regard the set of users as additional possible routing nodes within the network. Among these peer-to-peer (P2P) networks, Freenet and I2P are among the most renowned. Freenet provides an anonymous distributed file system. Each node of the network maintains a local data store and a routing table containing addresses of other available nodes within the system. This allows for the publication and retrieval of information across the network anonymously without the requirement of a centralised infrastructure.

Like Tor, I2P provides an anonymous communication service that is peer-to-peer using an overlay network. This tool also utilises ideas from early work on mix networks, end-to-end encryption and similar to Tor, provides an additional layer of encryption for each node that the packet is routed through. Packets sent across the network are routed through a user-defined set of routers known as tunnels. These tunnels also contain the node of the user as a member of the path. Other than the nodes owned by the users being directly involved in the anonymous routing procedure, unlike Tor the ACN is also fully distributed as it does not rely upon any centralised directory servers.

The aim of these types of networks is to provide greater protection against attackers, especially in regard to the previous scenario involving the exit/entry nodes in the Tor environment. More specifically each node within a P2P network is unable to discern whether the previous node in the path is the original source of the information or simply a relay node. The same applies to discerning whether the succeeding node in the path is the intended recipient of the information. This aids in protecting users from attackers but as stated before, as I2P and Tor are low-latency anonymous communication networks, they fail to protect against an adversary that has the ability to monitor the majority of the network, which the scheme in this section aims to provide.

4.3 Mathematics

Before we go into the description of the algorithm, we will provide a brief description of the mathematical structure that is required for an FHE scheme to function.

The FHE scheme used is a variant of the BGV scheme [14], described in section 3.3 which uses a polynomial ring $\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X))$ that is defined as an integer ring over a chosen m'th cyclotomic number field. Our plaintext space is defined to be the ring $\mathbb{A}_p$, where p is our prime plaintext modulus which reduces our ring $\mathbb{A}_p = \mathbb{A}/(p\mathbb{A})$.

A plaintext element $a \in \mathbb{A}_p$ is a polynomial of degree $\phi(m)$ with coefficients modulo p, where $\phi(m)$ is Euler's totient which coincides with the number of numbers smaller than m which are co-prime. As described in sections 3.3.7 and 4.5.1, we can decompose our plaintext a into a 'slot' representation to give us SIMD (Single Instruction Multiple Data) capabilities shown in figure 4.1.

Recall from section 3.3.7, we have the isomorphism $\mathbb{A}_p \cong \mathbb{A}_p/(F_1(x)) \times \mathbb{A}_p/(F_2(x)) \times \cdots \times \mathbb{A}_p/(F_l(x))$, where the number of slots is defined by $l = \phi(m)/d$, where d is the degree of $F_i(x)$. This implies that our cyclotomic polynomial decomposes into l irreducible polynomials each of degree d, namely $\Phi_m(x) = F_1(x) \cdot F_2(x) \ldots F_l(x) (\text{mod } p)$.

To maximise the efficiency of packing data into the plaintext slots, we can encode data across the coefficients of the polynomials in each slot. The specific encoding is described in section 4.5.1.

For the remainder of this chapter we will consider our plaintext encoded in 'slot' representation as defined in figure 4.1, where each slot $0 < i \leq l$ contains a degree-d polynomial R_i. Since the underlying structure of the plaintext and ciphertext are the same, we shall use the term 'plaintext slots' to describe the slot structure of both the plaintext and ciphertext. We will use ciphertext when describing something specific to the ciphertext space.

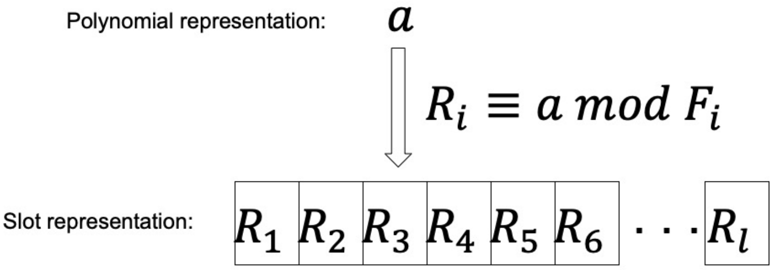


Figure 4.1: Plaintext slot representation where $a \in \mathbb{A}_p$ is a polynomial and each slot $R_i \equiv a \bmod F_i$ is a polynomial modulo a factor F_i of the cyclotomic polynomial $\Phi_m(x)$.

The ciphertext space is defined using a ciphertext modulus q to be $\mathbb{A}_q = \mathbb{A}/(q\mathbb{A})$ where the q is an odd number such that $q \gg p$. Note that q is a product of a 'chain' of odd prime numbers $q_0, \ldots, q_L$ which makes up a modulus chain as described in section 3.3.

In order to decrypt our ciphertexts we perform an inner product operation, denoted by $\langle \vec{u}, \vec{v} \rangle$ where $\vec{u}, \vec{v}$ are vectors of equal length. Therefore we have $\langle \vec{u}, \vec{v} \rangle = \sum_{i=1}^{n} u_i \cdot v_i$, where u_i is the i-th element of $\vec{u}$. This is subsequently followed by a modulus reduction function which we denote as $[a]_p$, which equates to calculating $a \bmod p$.

4.4 The Algorithm

A brief description of the algorithm will be presented below. The FHE scheme used is a variant of the BGV scheme [14] described in detail in section 3.3. We will attempt to use notation consistent with the previous chapter wherever possible.

At a high level, each message sent across the network is encrypted using FHE and contains a tag within the message, which is also encrypted into the same ciphertext. The tag is a label stating the recipient of the message and thus the route it should take across the network. Each node will contain a local list of possible labels that will be encoded using FHE. The node will then perform a homomorphic comparison operation between the tag and the locally stored labels. The node then will send a ciphertext along each outbound channel but only a single ciphertext will contain the message. Once the ciphertext has reached its destination, the recipient can decrypt the message conventionally using FHE. This removes the trust of the individual nodes in the network and provides anonymity against a global adversary through the security assumptions of FHE.

For simplicity, the algorithm will be described using a server-client model where the sever containing an arbitrary number of nodes and the sender and receiver are the clients. The algorithm consists of three major phases as follows:

1. ***Pre-Computation:*** Performed on client side, the sender encodes their message into a plaintext polynomial. The encoded plaintext will then be encrypted using the public encryption key of the intended recipient. This section of the algorithm can be broken into the following sub-algorithms:

   (a) *Encode*: Encode the plaintext message as a polynomial $a \in \mathbb{A}_p$ as depicted in figure 4.1.

   (b) *Encrypt*: Using the recipients public key, encrypt the plaintext polynomial a as a ciphertext $c := Enc(a)$.

2. ***Anonymous Routing:*** Performed on the server side, a node receives a message and a label both of which are encrypted. The node contains a list of labels corresponding to each output wire and compares the query label with each local label homomorphically. The homomorphic comparison will produce an encrypted mask. This message is multiplied with each mask and then sent to the next node in the

network. This is performed using the following sub-algorithms:

(a) *Homomorphic Comparison:* Compares the query label of the ciphertext with the labels held locally. This is carried out through the following:

   i. *Subtraction:* Compute the difference via a slotwise subtraction operation between the encrypted query label and the local labels. Produces a ciphertext with 0 in the slots that matched and non-zero otherwise.

(b) *Mask Creation:* Using the computed difference, generate a mask which will be an encryption of $\{0,1\}$, where 1 represents the matching label and 0 otherwise. This can be broken into the following:

   i. *Fermat's Little Theorem (FLT):* Given the difference c, use the result of FLT: $c^{p-1} \equiv 1 \bmod p$ to map all slots with non-zero values to 1 and 0 otherwise.

   ii. *Mask Inversion:* Invert the values of the slots such that 1 is mapped to 0 and vice versa.

   iii. *Partial Match Removal:* Remove partial matches through an inter-slot multiplication of the mask. Produces an encryption of 1 if there was an exact match and 0 otherwise.

(c) *Message Split:* Given a mask ciphertext for each output wire of the node, multiply the message ciphertext with the mask and then send the result to the next node in the network. If there was a matching label the original message is sent unmodified, otherwise the message is multiplied by an encryption of 0 and the zeroed out message is sent.

3. ***End Phase:*** The recipient receives an encrypted message from the routing server and retrieves the message.

   (a) *Decrypt:* Decrypt the received ciphertext c' using the secret key of the recipient to obtain a plaintext polynomial $a \in \mathbb{A}_p$.

(b) *Decode:* Decode the plaintext polynomial from polynomial representation to ASCII.

This algorithm can be viewed as an implementation of a variant of an anonymous communication technique known as "the mix" but in a Fully Homomorphic environment. The benefit of this over using the generic method which utilises a public encryption scheme at each node present in the network, the nodes must be trusted to not leak any information about the recipient of any messages it receives. Thus if a node in the network becomes compromised by an adversary then the anonymity of the recipient will become compromised. This scheme removes the trust required of the nodes in the communication network.

4.5 Implementation

This section contains a description of the implementation of the anonymous routing algorithm mentioned in section 4.4, providing a description of how each step is performed homomorphically.

4.5.1 Pre-Computation

The first step in using any FHE scheme is the parameter selection. This is highly dependent on the algorithm being performed as well as the data involved but most importantly on the security level as described in section 3.3.1.

In order to select suitable parameters, we must understand how data is represented in the BGV leveled homomorphic encryption scheme [14]. As described in section 3.3.7, this variant of the BGV scheme enables batching of several modulo-p entries into slots of a single plaintext which are treated independently. It also allows for the possibility of manipulating polynomials of d degree in each slot of the plaintext, where d is a value determined by the chosen parameters.

Using a plaintext space $\mathbb{A}_p := \mathbb{Z}_p[x]/(\Phi_m(x))$ for some prime number p and integer m. The elements of $\mathbb{A}_p$ are polynomials, with coefficients modulo p, of degree $\phi(m) - 1$, where $\phi(m)$ is Euler's totient that defines the number of integers smaller than m which are co-prime. We are able to encode our data in a way that enables us to perform operations in an entry-wise manner. This is possible due to the following decomposition into a direct product

$$\mathbb{A}_p \cong \bigotimes_{i=1}^{\ell} E_i \tag{4.1}$$

using the Chinese remainder theorem for rings shown in Theorem 3 where each E_i is isomorphic to $\mathbb{A}_p/(F_i(x))$, where $F_i(x)$ is an arbitrary degree-d factor of the m-th cyclotomic polynomial $\Phi_m(x)$ modulo p. The maximum number of slots is defined as $l = \phi(m)/d$, where d is the order of p in the multiplicative group $\mathbb{Z}_m^*$. This also coincides with the degree of $F_i(x)$. Since the rings E_i are isomorphic to each other we can set $E := E_1$ and thusly consider the isomorphism $\mathbb{A}_p \cong E^l$. More specifically we have $\Phi_m(X) = F_1(X) \cdot F_2(X) \dots F_l(X) \pmod{p}$. Thus a plaintext polynomial message $a \in \mathbb{A}_p$ represents an l-vector containing entries a mod F_i for i in the range $1 \leq i \leq l$. Therefore consideration must be taken to choose m and p to define the correct size and shape of the plaintexts given our data as well as the level of security.

Encoding the Message Data

In this chapter we consider messages that can be packed into a single plaintext, however if the number of characters exceeds $\phi(m)$ then we can encode the message across multiple plaintexts. Due to the ability to pack data across the slots of a plaintext, we can use logical SIMD capabilities. Therefore we encode our message across the plaintext slots.

A typical representation of data is ASCII. Assuming we want to represent all *printable* ASCII characters in decimal representation then we must choose an algebra that enables us to represent at least 95 characters. Thus we choose a plaintext prime of $p = 97$ to represent all of the necessary characters. Due to the first 32 characters being non-printable characters, when encoding the message, the printable characters will be remapped by subtracting 32 from the original decimal representation of the character (i.e. The white space character 32 is mapped to 0, the letter 'A' is mapped from 65 to 33 etc.). With this mapping, all printable characters can be represented as an integer in the range $(0, 95)$.

Given an algebra, each slot of our plaintext is a polynomial in E_i, as in equation 4.1, for slot i with coefficient vector $\vec{v} = (v_0, \ldots, v_d)$. We encode each ASCII character as a single entry of the coefficient vector $\vec{v}$, repeating the process for each slot of the plaintext. This means we can pack up to d characters per plaintext slot. As a single plaintext element is a polynomial in the ring $\mathbb{A}_p \cong E^l$, where $l = \phi(m)/d$, then we can pack up to $d \cdot l = \phi(m)$ characters across an entire plaintext. For typical levels of security $\lambda \approx 100$ we can expect $\phi(m)$ values of around 10000.

Encoding the Labels

For simplicity of describing the scheme we assume that each label is encrypted in a single plaintext. We consider a label to be a unique identifier to a single node in the network. This can be a single integer number represented in binary. Considering the current standard of Internet Protocol (IP) addresses, IPv6 uses an address size of 2^{128} which is considered to be sufficient for the foreseeable future. Therefore we require a plaintext that allows us to represent 128-bit numbers.

Since we are using a single plaintext per label we can pack the number across the coefficient vector $\vec{v} = (v_0, \ldots, v_d)$ of each slot. It is possible to pack multiple labels into a single plaintext to optimise for efficiency which will be discussed in section 4.6, however for simplicity we consider a single plaintext encoding a single label, represented as a 128-bit integer.

It is noted that the labels of each local node must be encoded in the same way as the label sent by the client to ensure the correctness of the homomorphic matching algorithm.

Encryption

Since we are using a public key encryption scheme, the encoded plaintexts are encrypted using the public key of the intended recipient. The client encrypts both the label and the message data with the same public key using the encryption routine described in section 3.3.3. The ciphertexts are then sent to recipient via the anonymous routing network.

4.5.2 Anonymous Routing

In this phase of the scheme, a node in our communication network receives two ciphertexts, one containing the label of the destination node and another containing the message data. The job of this node is to route the information received onto the correct path of the network to the intended recipient(s).

To achieve this, the node performs a homomorphic comparison of the encrypted label with each label held locally. For each output wire from the node, there will be a corresponding list of unique labels of all subsequent nodes in the network. Also each list of labels will be disjoint from each other. This means a maximum of one list can contain a matching label, furthermore there can only be a maximum of one match across all lists.

The result of the homomorphic comparison will be used to produce a single mask ciphertext for each outward path from the node which will be described in detail in section 4.5.2. The mask will be an encryption of one if the label was present in the list and a zero otherwise. Using this mask the node will multiply it with the message and send the new message ciphertext with the initially received label along each output edge. The original message will only be sent through at most one output edge. All other edges will receive a ciphertext containing an encryption of zero.

We make the assumption, without loss of generality, that the network is a directed, thus a message cannot be sent along the same wire from which it was received. In other words, each node has one input edge and there exists no loops within the network. This can be achieved through introduction of a protocol.

Homomorphic Comparison

The node performs a homomorphic comparison of the received label, which we will call the query label, with the labels held locally at the node. Each node in the network will contain a list of labels, one list for each output wire. The labels correspond to each node further along the network. Each list contains unique labels and each list is disjoint from each other. Subsequently a homomorphic comparison will only produce a maximum of one unique match across the union of lists of a single node.

For each list of labels we start by computing the difference between the query label and each node label from the list. This is achieved by subtracting the query label from each node label *slotwise.*

We note that the labels held locally are known to the node and need not be encrypted. However, the computation between the query label and local node label results in a ciphertext that is encrypted under the initial public key of the client. Thusly, each node cannot see any result subsequent to the first operation.

Subsequent to the subtraction operation, the difference of the query label and the compared node label is held in the resultant ciphertext. This ciphertext will be an encryption of zero if and only if the query label matched the node label, otherwise it will be non-zero if they did not match. Now the node will need to convert the resultant difference into a mask of ones and zeroes. Given that we have polynomials in the slots of our ciphertext as described in section 4.5.1, to achieve this we will use a variant of Fermat's Little Theorem for polynomials:

Theorem 4 (Fermat's Little Theorem)**.** *If we have a prime number p, then for any non-zero polynomial a of degree d which is not divisible by p, the number $a^{p^d-1} - 1$ is an integer multiple of p i.e.*

$$a^{p^d-1} \equiv 1 \ mod \ p$$

We adapt this result to create a generic function to achieve the behaviour desired.

This can be achieved naively through exponentiation, yet this would be expensive both computationally and in terms of noise growth due to the large number of necessary multiplications. To mitigate this we use a less noise expensive method of exponentiation [44] which is implemented in HElib [46], namely the Frobenius automorphism

$$\sigma : X \to X^p$$

Given our ciphertext c containing the difference, we must compute c^{p^d-1} and we note the result

$$p^d - 1 = (p-1)\sum_{i=0}^{d-1} p^i$$

We can then compute $c^{p^d-1} = c^{(1+p+\cdots+p^{d-1})(p-1)}$ by initially computing c^{p-1} through exponentiation and then applying the Frobenius automorphism $\sigma(c^{p-1}) = c^{(p-1)p}$. Thus we calculate the function

$$f(c) = \prod_{i=0}^{d-1} \sigma^i(c^{p-1}) \tag{4.2}$$

where $\sigma^i(c) = c^{p^i}$.

Using the function in equation 4.2 we obtain a method for computing the desired mask as follows:

$$1 - f(c) = \begin{cases} 1, & \text{iff } c = 0 \\ 0, & \text{otherwise.} \end{cases} \tag{4.3}$$

Applying equation 4.3 to the difference ciphertext c, we obtain a one in each slot where there was a match and a zero in all other slots of the ciphertext where no match was present.

Since the local list contains unique labels, there can only be at most one label which is an exact match corresponding to the query label. This matching label will produce a mask containing a one across all slots of the ciphertext. However, partial matches can occur and must be removed.

To achieve this we perform a slotwise multiplication of the mask ciphertext. This is done via creating a copy of the mask ciphertext, performing a rotation of the slots and then multiplying the rotated mask with itself. This results in a slotwise multiplication and is repeated for all permutations of the ciphertext slots which is $l = \phi(m)/d$. Therefore, if a mask ciphertext contains at least one slot containing a zero, this slot will be multiplied across the remaining slots of the ciphertext, resulting in a mask of zeroes.

Since there will always be a maximum of a single unique match, the resultant masks can be aggregated into a single ciphertext. This will produce a single mask ciphertext for each list corresponding to each output edge of the node. The mask will be an encryption of one for the correct onward path and a zero otherwise.

Routing the Message

Using the resultant masks for each output edge of the node, we perform a multiplication between the mask and the received message data. For the edge containing the matching label (producing the mask ciphertext of ones), multiplying the mask with the message data will leave the data unchanged as this is the intended route of the message. For all other routes the message data will be multiplied with a mask of zeros, thus zeroing out the data and producing a ciphertext of zeroes.

These resultant ciphertexts are then sent along the output edges to the next node where another homomorphic comparison takes place until it reaches the edge of the network. Since a ciphertext is sent along each edge of the network, it provides total anonymity against a global adversary, which will be discussed further in section 4.6.

4.5.3 Retrieving The Message

Subsequent to the receiving a message from the routing server, the recipient decrypts the encrypted message data. As described in section 3.3.3, the recipient calculates $[\langle \vec{c}, \vec{sk} \rangle]_p$. Namely, the inner product of the ciphertext with their secret key followed by a modulo operation with the plaintext prime p. This produces a plaintext containing an encoding of the original message data sent by the client.

The plaintext is a polynomial with coefficients representing ASCII characters. The recipient computes the inverse of the encoding algorithm to reproduce the original ASCII data.

4.6 Discussion

Through the use of FHE, this algorithm provides security and anonymity against a network that is an honest-but-curious adversary. Conventional routing algorithms use standard public key encryption to secure the data begin sent across the network as well as provide a level of anonymity. However, trust must still be placed in the individual nodes of the network as each node is aware of the previous node in the chain as well as the subsequent node to relay the message to. This is because the data is encrypted in layers and as each node in the chain decrypts a layer, it reveals the address of the next node in the chain.

The main disadvantage of these conventional algorithms such as Tor and I2P are they are not secure against a global adversary. Given an entity that can passively observe the network globally, it can deanonymise users of the network [52]. The algorithm described in this chapter removes this vulnerability because none of the nodes in the network need to decrypt and thus the nodes themselves do not know the route of the data across the network.

More specifically, an adversary that can observe the entire network will see a message being sent to every node of the network, thus hiding the true recipient in the entire recipient set. This makes it hard for a global adversary to deanonymise the recipient as the message path cannot simply be observed. Additionally, if an adversary intercepts a message, not only will they be required to solve the Learning With Errors (LWE) problem to decrypt the message without the secret key, there is no guarantee the ciphertext will contain the message. Thus 'zeroing' the message data at each node increases the security of the algorithm.

Additionally, with quantum computing looking to become prevalent in the future, it is advantageous that FHE is a lattice based cryptography scheme as it is assumed hard for polynomial-time quantum algorithms [61]. Due to Shor's algorithm [71], conventional public key encryption that is used in current anonymous routing algorithms may become obsolete in a future where quantum computers are prevalent. Thus, this chapter presents a scheme that is not only secure against a global adversary but also quantum resistant.

This type of algorithm also would not be possible with partially homomorphic encryption schemes as they only provide a single type of homomorphic operation, either addition or multiplication. This algorithm relies on the use of FHE to provide the operations necessary to perform a comparison algorithm homomorphically.

Nevertheless, this algorithm also introduces the natural algorithmic overhead of using FHE. This is evident in section 4.5.2, from the number of additional operations that are needed to perform the comparison and mask creation homomorphically. This is due to the inability to perform branching operations within FHE. This chapter demonstrates how when using FHE, every branch of computation must be computed, similar to how a ciphertext is sent along every edge of the network.

In order to provide total anonymity, this algorithm causes the network to be spammed with encryptions of zeroes. Every node of the network must receive a ciphertext as it is oblivious to the destination of the data. This is the main disadvantage of the algorithm presented here.

4.7 Conclusion

This chapter has shown the versatility of applying FHE to use cases within a realistic scenario. This was demonstrated by applying a homomorphic matching algorithm outside of the typical use-case of Private Information Retrieval.

It has been shown that FHE is useful for scenarios where a client is not willing to trust a server in a client-server model, the routing network being the server in this example. Despite the added algorithmic overhead, as hardware continues to increase in efficiency and speed, FHE algorithms will continue to move ever closer to feasible running times.

Nevertheless, the main drawback of FHE is the inability to efficiently compute conditional operations. FHE currently requires algorithms to compute every branch of computation. Thus it remains an open question as to how to use FHE for anonymous routing without spamming the network.

Chapter 5

Implementing Fractional Arithmetic in FHE

Previously, the use of FHE in a contrasting scenario to outsourcing computation without revealing the data was explored, this was shown to be possible however extremely impractical. Moving on from applying FHE to the field of networking, another interesting aspect to explore is applying FHE on different types of data. This section presents my work on defining and implementing rational numbers within an FHE scheme and the application of this data representation to perform simple linear regression.

5.1 Introduction

Since, Gentry's breakthrough [39] there have been numerous FHE schemes published to date [10, 32, 59, 38, 41]. However the scheme of note [14], the BGV scheme, made considerable gains in terms of efficiency and practical use of FHE.

Computing directly on encrypted data inherently comes at a cost in computational resources, namely, computation time and memory for adequate levels of security. However, with more efficient algorithms, implementations, and hardware acceleration techniques, practical work using FHE can be done [31, 47] and this cost will continue to reduce. This section does not focus on these major topics of *practical* FHE; instead, we address another major obstacle to *practical* FHE, namely that of how to represent numbers within FHE that approximate the real numbers or rationals so that useful computation in analysis can be performed on encrypted data. This has far reaching consequences in today's data driven world.

The developer implementing homomorphic algorithms has to make use of an FHE scheme's particular algebra. This algebra is mostly not flexible to represent other algebras required or data structures and care has to be taken when designing an appropriate representation. Security and algrebas in FHE schemes are based on the Learning with Errors (LWE) problem [67]. The FHE scheme that is used in this section is the BGV variant implemented in the C++ library HElib [46]. The library makes use of the ring LWE (RLWE) a variant of the LWE which uses integer polynomial rings as its algebra where the message polynomial ring is given by $\mathbb{A} \cong \mathbb{Z}[X]/(\Phi_m(X))$ which is a univariate integer ring $\mathbb{Z}[X]$ modulo the m'th cyclotomic polynomial.

A developer is challenged to understand the data and algorithm that will be used and build the FHE solution accordingly in order to accommodate the data and how it has to be computed on. This requires the user to have an in-depth understanding of the underlying structure of the FHE scheme and its algebra as it is highly dependent on numerous parameters which have a relatively complex relationship between them. Parameter selection is out of scope of this thesis. To do analysis, the problem becomes how to represent the required data in the available algebra. The issue of using the ring $\mathbb{A}$ to represent data is that it does not represent an adequate approximation of the real numbers or rationals. This limits the algorithms that can be performed as it cannot for example perform real analysis. There have been attempts to solve this issue using different representations such as fixed-point [4, 29, 11, 35] or floating-point arithmetic [22] and simpler solutions such as representing fixed precision rationals using a fixed multiplier to map rationals into integers [51, 18, 23, 19, 8]. A method of particular interest was that of representing rational numbers using continued fractions [27] which introduced new challenges in efficiency in terms of computational complexity. Moreover, there have been attempts to approximate certain continuous mathematical functions, such as logistic regression, using a polynomial function such as the Taylor approximation [9] and the minimax approximation [23].

The method of mapping the rationals into integers using a fixed scalar multiplier is viewed to be a relatively efficient solution. However, it requires the user to keep track of the scale as ciphertexts progress through an algorithm in order to ensure the messages are decoded correctly. This is because as operations are performed upon each ciphertext, not only the message within the ciphertext but the scale can also be modified via operations. These schemes tend to have an increased rate of noise growth and thus require larger values of p, the plaintext modulus, making it more challenging to select an optimal set of parameters and causing increased difficulty in evaluating deeper circuits. There has been work to improve this, such as the interesting method of changing the plaintext modulus itself from an integer p to a polynomial instead [19]. This tackles the issue of requiring larger values of p, although it creates a slightly more complicated scheme as well as introducing some overhead, most notably in the decryption phase. Furthermore, a number of previous attempts at representing rational numbers lack batching of data [22] and thus the single instruction multiple data (SIMD) properties of ciphertexts cannot be utilised.

The approach we have taken is to form objects that resemble fractions similarly to a previous work [72] which was applied to feature detection and description of images. We differ from the previous as it uses the Gentry-Sahai-Waters (GSW) scheme [41] which is an LWE variant that produces ciphertexts as matrices whereas this chapter describes the implementation of the fractions using the algebra given by the BGV variant scheme found in HElib. This chapter takes the description of the underlying theory further by elaborating upon why it is possible to form these types of fractions from the existing algebra used in FHE schemes. Moreover, our scheme is applied to linear regression and in contrast to [72] operates on the actual objects and not their components specific to a computation.

Simple linear regression is performed where the input is a set of points of a straight line with additive white Gaussian noise (AWGN). This creates a randomised "noisy" line which is encrypted before the application of linear regression is performed homomorphically. The computation returns a line of best fit which ideally should be the equation of the original line.

5.2 Implementation of Fractions

5.2.1 The Plaintext Space

The FHE scheme used will be a variant of the BGV scheme described in section 3.3. A brief overview of the mathematical structures that will be used for the field of fractions is given. The scheme has a plaintext space given by the ring $\mathbb{A}_p \cong \mathbb{Z}_p[X]/(\Phi_m(X))$ where the prime number p is our plaintext space modulus. This leads to a structure that supports SIMD operations per ciphertext where data can be packed into slots of numbers in $\mathbb{F}_{p^\lambda}$ where λ is the order of p in $\mathbb{Z}_m^*$, the multiplicative group modulo m. The slots are the residues of the polynomial from $\mathbb{A}_p$ given by the modulo of a factor of the cyclotomic polynomial $\Phi_m(X)$ mod p. The number of slots is dependent on the corresponding Galois group size given by

$$\left|\frac{\mathbb{Z}_m^*}{\langle p \rangle}\right| = \frac{\phi(m)}{\lambda}$$

where ϕ is Euler's totient function and λ is the smallest value given by $p^\lambda = 1 \bmod m$. The reader is reminded that the message or data polynomial is embedded in a larger ring during encryption and with "noise" as required by the RLWE so the slots cannot be seen without decryption.

5.2.2 The Quasi-Rationals

To make fractions, the approach makes use of a well known mathematical result, namely that every integral domain contains a field of fractions [75]. From the plaintext space described above, we have an integral domain in the slots namely $\mathbb{F}_{p^\lambda}$. This is due to p always being chosen to be prime. It is noted that every finite integral domain is a field and thus the field of fractions of $\mathbb{F}_{p^\lambda}$ is isomorphic to the field itself. At this point, the reader may wonder what advantages, if any, are there for mapping to a representation of the same object. The trade-off, which will become clearer, is that calculations are simpler to reason about and some costly operations such as finding the multiplicative inverse are now negligible. However, it should be noted that memory usage is at least doubled to store these numbers and addition is more costly. As a simplification $\lambda = 1$ in this chapter but for other cases larger λ may be required for a field of fractions from polynomials.

Recall that a field of fractions is constructed by taking an ordered pair (n, d), also written as n/d, where $n, d \in \mathbb{F}_p$ and $d \neq 0$ [75]. Therefore, using HElib's ciphertexts a field of fractions can be defined using two ciphertexts one to hold the numerator, n, and the other to hold the denominator, d, which will be called *QuasiQ*. However, due to the encryption of the plaintext to ciphertexts the *QuasiQ* objects will be slightly different from a true field of fractions.

The first thing is that the denominator cannot be guaranteed to not hold a zero value. This introduces an issue when using encrypted fractions, namely, what happens when division by zero occurs? There is no way of knowing when performing a division of two fractions whether the divisor is in fact zero and therefore producing a result that is not defined. This remains an interesting problem of what should be done or can be done in this situation. However, since the values of the ratios themselves are not computed in decimal form, the algorithms will still be carried out regardless of the values. The results when displayed after decryption however will not be defined as they will have a zero for the denominator segment of the ratio.

Another difference is that normally a field of fractions has an equivalence relation defined $(n, d) \sim (r, s) \iff ns = dr$ where $n, d, r, s \in \mathbb{F}_p$ and $d \neq 0, s \neq 0$. The fraction can be reduced to its lower representation by factoring. However, when encrypted this is not possible as the values cannot be seen to be factored in the recursion procedure that would be required. Plus, due to the integral domain used being finite, there is another equivalence relation imposed on the field of fractions due to the modulus p and this can cause issues as described in section 5.3 when evaluating the *QuasiQ* objects as well as leading to the denominator zero problem described previously.

Defining the *QuasiQ* object as an ordered pair (n, d), the fraction arithmetic can be defined as follows: addition operation: $(a, b)+(c, d) = (a\cdot d+c\cdot b, b\cdot d)$, negation operation: for fraction $r = (n, d)$ we define the additive inverse $-r = (-n, d)$, subtraction operation: $(a, b) - (c, d) = (a \cdot d - c \cdot b, b \cdot d)$, multiplication operation: $(a, b) \cdot (c, d) = (a \cdot c, b \cdot d)$, reciprocal: for fraction $r = (n, d)$ we define the multiplicative inverse $r^{-1} = (d, n)$, division operation: $(a, b) \div (c, d) = (a \cdot d, b \cdot c)$.

The ciphertext objects are operated on exactly as they normally would be in HElib, but for the *QuasiQ* objects the operations for the field of fractions are implemented, which include *negation*, *reciprocal*, *addition*, *multiplication*, as well as, *subtraction*, and *division*. All inputs a, b to the following operations are *QuasiQ* objects, namely an ordered pair of two ciphertexts (n, d) where n represents the numerator and d the denominator. The numerator and denominator of a *QuasiQ* object a can be accessed with a.`numerator` and a.`denominator` respectively.

Before we present the pseudocode, we note that as the underlying types of *QuasiQ* objects are ciphertexts, we attempt to use procedures already available within HElib [48] wherever possible. An instance of this is in the implementation of *negation*. We use the HElib operation Ctxt.negate(c) where c is the input ciphertext to be negated and the output is $-c$. This is achieved through a scalar multiplication $(-1) \cdot c$.

The quasi-rational operations are described in the following pseudocode:

```
QuasiQ.add($a, b$)
if $a$ is $b$ then return $2 \cdot a$.numerator
else
    $n \leftarrow b$.numerator
    $a$.numerator $\leftarrow a$.numerator $\cdot$ $b$.denominator
    $n \leftarrow n \cdot a$.denominator
    $a$.denominator $\leftarrow a$.denominator $\cdot$ $b$.denominator
    $a$.numerator $\leftarrow a$.numerator $+ n$
    return $a$
```

```
QuasiQ.negate($a$)
$a$.numerator $\leftarrow$ Ctxt.negate($a$.numerator)
return $a$
```

```
QuasiQ.subtract($a, b$)
$b \leftarrow$ QuasiQ.negate($b$)
$a \leftarrow$ QuasiQ.add($a, b$)
return $a$
```

```
QuasiQ.multiply($a, b$)
$a$.numerator $\leftarrow a$.numerator $\cdot$ $b$.numerator
$a$.denominator $\leftarrow a$.denominator $\cdot$ $b$.denominator
return $a$
```

```
QuasiQ.reciprocal($a$)
$n \leftarrow a$.numerator
$a$.numerator $\leftarrow a$.denominator
$a$.denominator $\leftarrow n$
return $a$
```

```
QuasiQ.divide($a, b$)
1  if $a$ is $b$ then
       $a$.numerator ← $a$.numerator · $a$.denominator
2      $a$.denominator ← $a$.numerator
3  else
       $b$ ← QuasiQ.reciprocal($b$)
4      $a$ ← QuasiQ.multiply($a, b$)
5  return $a$
```

The algorithms above which have two inputs, such as QuasiQ.subtract(a, b) for example, can be read as calculating the operation $a - b$, as opposed to calculating $b - a$, which would be QuasiQ.subtract(b, a). Subsequently, all of the necessary mathematical operations between two *QuasiQ* rationals can be performed, thus our implementation can be tested by applying it to a known mathematical algorithm.

It is noted, there is an "optimisation" in the operations QuasiQ.divide(a, b) and QuasiQ.add(a, b), specifically line 1 of the pseudocode "`if` a `is` b". The input *QuasiQ* objects a, b in the previously defined pseudocode are pairs of ciphertexts, thus we cannot tell if the inputs a and b are the same mathematically. Therefore this "optimisation" only applies for when a and b are the same object in memory (i.e. if we call the method QuasiQ.divide(a, a)).

5.2.3 Simple Linear Regression

Now that there is the ability to hold data in fractional form, this can be applied to new problems to be tackled with FHE. One problem of note is linear regression, as rational numbers can be used to accurately determine the equation of a line given a set of points. Given a set of coordinates (x_i, y_i), the desire is to calculate a line of best fit of the form $y = b_1x + b_0$ using simple linear regression. Therefore values must be calculated of the line intercept b_0 and the gradient of the line b_1 using least squares. These values are obtained via the following:

$$b_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \qquad\qquad b_0 = \bar{y} - b_1\bar{x},$$

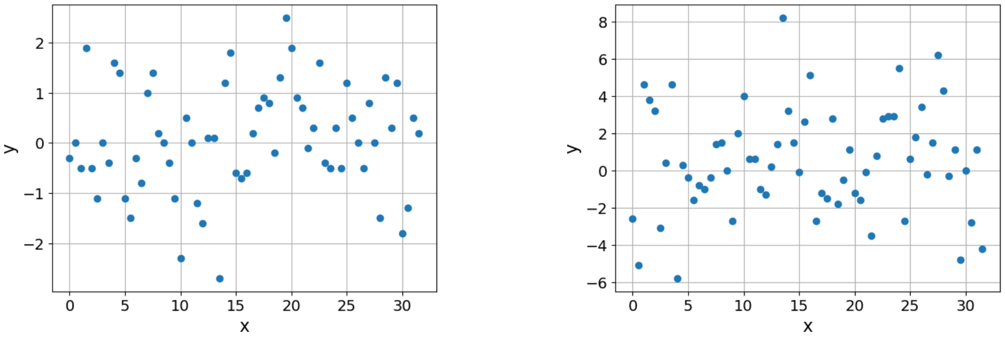
where $\bar{x}$ and $\bar{y}$ are the mean averages of the $x_i^{\prime}s$ and $y_i^{\prime}s$ respectively.

Recall from section 5.2.2 that our *QuasiQ* object is an ordered pair (n, d) where n represents the numerator and d the denominator. In this implementation, the values x_i, y_i are each encrypted as *QuasiQ* objects. Since x_i and y_i are integers, we encode each x_i as a rational of the form $x_i/1$. Similarly we encode the y_i values as a rational $y_i/1$. In other words, the input data x_i, y_i of our homomorphic regression algorithm are rationals $x_i/1$ and $y_i/1$ encoded, as *QuasiQ* objects, into pairs $(x_i, 1)$ and $(y_i, 1)$ respectively.

These values are used to encode into our fractions, the numerator and the denominator of the fraction b_1, storing these values in their respective locations in the *QuasiQ* object before using this value to calculate b_0.

5.3 Implementation of Linear Regression

The implementation of the simple linear regression discussed in section 5.2.3 is given below using the *QuasiQ* implementation of a field of fractions. The linear regression algorithm was tested on points of a straight line with added Gaussian noise. Firstly, the gradient and intercept of a straight line are selected randomly from a uniform distribution. Then we select points of this line and a random rational number r selected from a Gaussian distribution is then added to the corresponding y values to obtain a set of points $(x_i, y_i + r_i)$. The Gaussian distribution is denoted by $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean and σ is the standard deviation. The mean that is set for the Gaussian distribution in our experiments is always zero as our concern is simulating random noise on the data and not a systematic error. This new set of points is used to simulate a real-world dataset with a notable correlation. The new x and y values are then encrypted into ciphertexts and given to the linear regression algorithm to calculate the line of best fit.



(a) Noise chosen from a Gaussian Distribution $\mathcal{N}(0,1)$.

(b) Noise chosen from a Gaussian Distribution $\mathcal{N}(0,9)$.

Figure 5.1: Random noise distribution from two Gaussian distributions.

An example of noise chosen from a relatively small sample set of $\mathcal{N}(0,1)$ containing 64 (x_i, y_i) values is given in Figure 5.1a. In the experiments shown the x_i values are selected to be at intervals of 0.5, in other words, the x values are selected to be $x_i = \frac{i}{2}$ for i in the interval $[0, 63]$. The reason for choosing our data size to be in this range was to keep the total noise growth to a reasonable level so as to circumvent the need for bootstrapping.

Most of the noise variables r_i are within the interval $(-1, 1)$ with the largest offset being within the range $(-3, 3)$. This can be viewed as a "noise" vector $\vec{r} = (r_0, \ldots, r_{63})$ that is added to our straight line vector $\vec{y}$, where $y_i = mx_i + c$, to obtain the set of points in Figure 5.2. The set of x_i and corresponding $y_i + r_i$ values are encrypted and input to the linear regression algorithm, where the values b_0 and b_1 are calculated homomorphically from the linear regression equation previously stated in Section 5.2.3. However, prior to implementing the full linear regression algorithm, two subroutines are required to calculate some of the intermediary terms. Namely, an algorithm calcMean to calculate the mean of a set of given *QuasiQ* objects, which in turn requires a summation subroutine sumFrac that sums a set of *QuasiQ* objects.

Given an ordered collection of *QuasiQ* objects v, sumFrac(v) computes the sum of the collection. First we take a copy of the collection as this algorithm is destructive. The additions are treated as nodes in a binary tree and the same collection is used to store the intermediate results so memory usage is minimal. This keeps the depth of operations to a minimum, dependent only on the collection size to $\log_2 \mathsf{size}(w)$, where w is the collection of fractions and $\mathsf{size}(w)$ is the number of fractions contained in the collection. The final sum is contained in the first element of the collection which is returned. The sumFrac procedure is defined below:

```
sumFrac($v$)
$w \leftarrow v$
$e \leftarrow \mathsf{size}(w)$
$s \leftarrow 1$
while $e > 1$ do
    for $i \leftarrow 0$ to $\mathsf{size}(w)$ step $2 \cdot s$ do        # iterate through $w$ by jumps of $2 \cdot s$
        if $i + s > \mathsf{size}(w) - 1$ then break
        $w_i \leftarrow w_i + w_{i+s}$
    if $\mathsf{odd}(e) = true$ then $e \leftarrow e + 1$
    $e \leftarrow e/2$
    $s \leftarrow 2 \cdot s$
return $w_0$
```

Once sumFrac is defined, calcMean(v) is relatively simple to implement, where v is an ordered collection of *QuasiQ* objects. Now the full linear regression algorithm, linearReg(x, y), can be implemented to calculate the equations stated in Section 5.2.3, where x and y are ordered collections of the coordinates (x_i, y_i) as defined previously,

```
calcMean($v$)
$m \leftarrow \mathsf{sumFrac}(v)$
$m$.denominator $\leftarrow m$.denominator $\cdot\ \mathsf{size}(v)$
return $m$
```

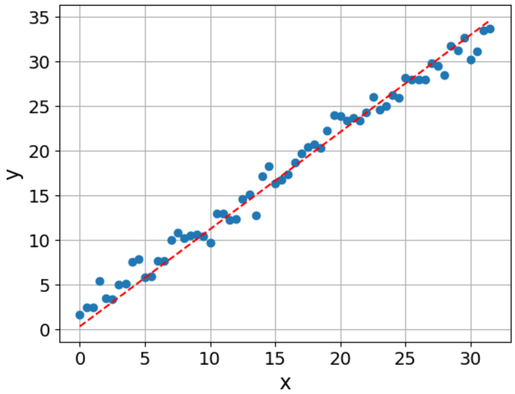


Figure 5.2: Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0,1)$.

```
linearReg($x, y$)
$xMean \leftarrow$ calcMean($x$)
$yMean \leftarrow$ calcMean($y$)
for $i \leftarrow 0$ to $n-1$ do
    $dx_i \leftarrow x_i - xMean$
    $dy_i \leftarrow y_i - yMean$
for $i \leftarrow 0$ to $n-1$ do
    $ddx_i \leftarrow dx_i \cdot dx_i$
    $ddy_i \leftarrow dy_i \cdot dx_i$
$ddxSum \leftarrow$ sumFrac($ddx$)
$ddySum \leftarrow$ sumFrac($ddy$)
$b_1 \leftarrow$ QuasiQ.divide($ddySum, ddxSum$)
$t \leftarrow$ QuasiQ.multiply($xMean, b_1$)
$b_0 \leftarrow$ QuasiQ.subtract($yMean, t$)
return $b_0, b_1$
```

The two values b_0, b_1 are returned by the linearReg as two separate *QuasiQ* objects, representing the intercept and gradient of our regression line respectively. The user can then decrypt these two values and plot the line of best fit $y = b_0 + b_1 x$, on top of the noisy data as shown in Figure 5.2.

After obtaining a graph showing the linear regression algorithm correctly fitting a line to encrypted noisy data, the variance of the Gaussian distribution used to generate the noise vector was increased. Setting the standard deviation $\sigma = 3$, produces the noise shown in Figure 5.1b. An issue appeared once the noise was increased that when generating the points for the line that the additive noise would send those points to the "wrong" equivalence class in the *QuasiQ* if the point went beyond the $[0, p-1]$ interval in the y-axis in the rationals. This causes a loss of information, as the resulting data gets allocated to an equivalence class and thus the "true" value is lost within a set of possible equivalent values.

An interesting problem to consider; however, to make further progress with the practical investigation we decided to modify the few noisy elements that fell outside the interval $[0, p-1]$. The noise selection process that tackled both the case $y_i + r_i < 0$ as well as $y_i + r_i > p - 1$. For the case of $y_i + r_i < 0$, the new data is forced to have the value $y_i + r_i = 0$. This tends to occur at the smaller x_i values as the original y_i values start close to 0 so there is a greater probability of obtaining a noise value $|-r_i| > y_i$, where $|-r_i|$ is the absolute value of $-r_i$. Examples of this can be seen in both Figure 5.3 and Figure 5.4. For the case $y_i + r_i > p - 1$, during the noise selection process, only rational numbers that can be represented using 1 decimal place are considered (note, that using the smallest precision could have also been used, $1/p - 1$). The decimal representation is converted into the *QuasiQ* rational representation by multiplying the float by 10 and then removing the remaining decimal places. Therefore always obtaining a fraction with a maximum denominator of 10. By doing this, it keeps the starting denominator relatively small compared to our plaintext modulus p to reduce the possibility of creating a numerator greater than p and thus wrapping around. Thus, as the denominator of the *QuasiQ* object increases, the maximum difference from 0 that can be represented decreases. The most important property of the input data, as in all computing approximations of the rationals, is that it exists within the bounds that accurately represents the rationals. For *QuasiQ*, the data must ideally lie with x and y values both in the

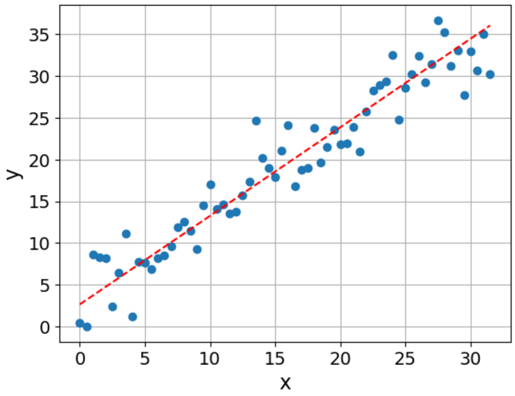


Figure 5.3: Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0, 9)$.

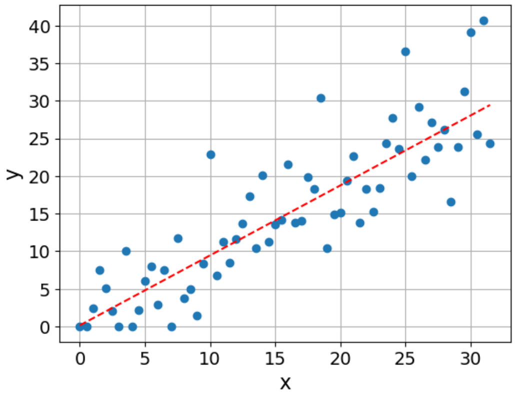
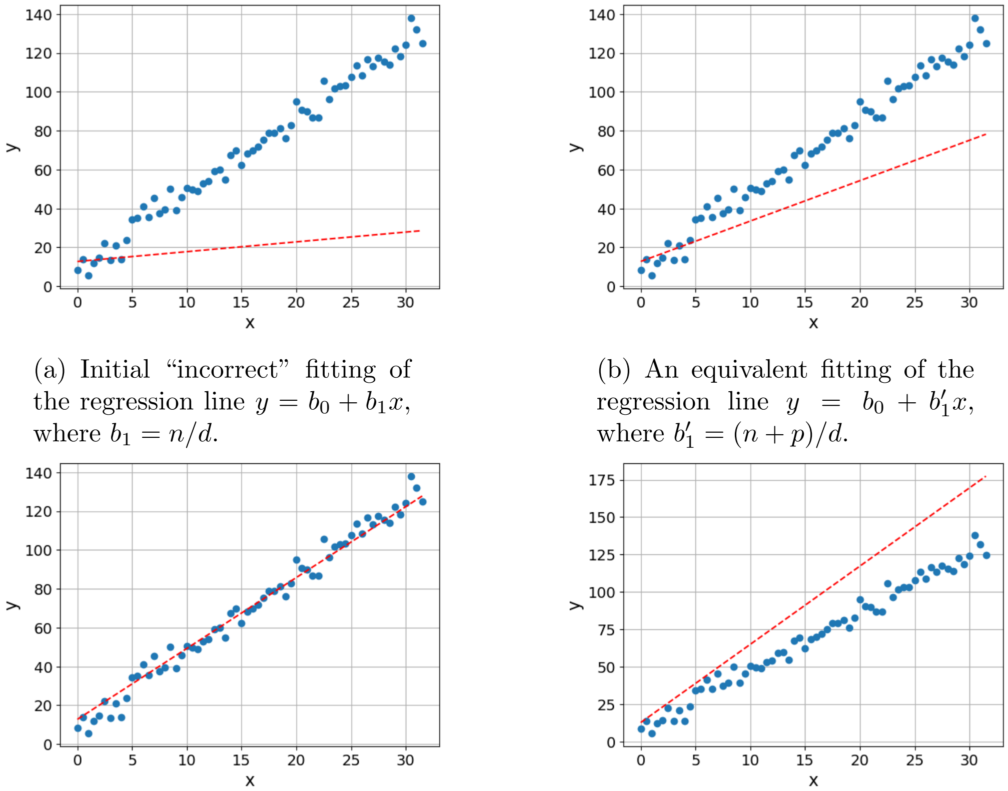


Figure 5.4: Fitting of the regression line on the input data with additive noise from $\mathcal{N}(0, 25)$.

interval $[0, p - 1]$ in the rationals.

Multiple examples have been presented where our algorithm correctly fits a line of best fit to some randomised encrypted data through simple linear regression. Unfortunately, it must be noted that occasionally the final outputs b_0, b_1 appear to give an incorrect line. Therefore upon initial inspection by the user, it looks like the algorithm has incorrectly fitted a line to the input data such as in Figure 5.5a. This is due to the fraction in the rationals being wrong, but in the correct equivalence class in $QuasiQ$, for reasons described in more detail in Section 5.4. In other words, the result the user views is the equivalence class in which the correct answer belongs to in the rationals. So it is possible to find which element of our equivalence class is the "true" result of the algorithm.



(a) Initial "incorrect" fitting of the regression line $y = b_0 + b_1x$, where $b_1 = n/d$.

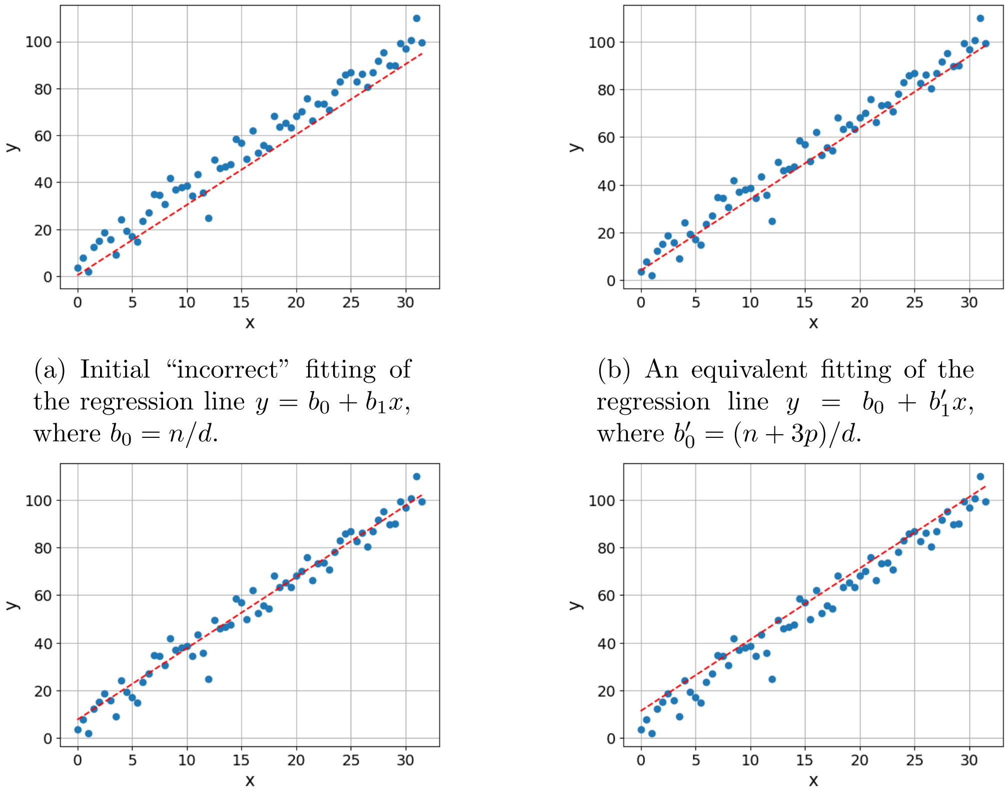
(b) An equivalent fitting of the regression line $y = b_0 + b_1'x$, where $b_1' = (n + p)/d$.

(c) An equivalent fitting of the regression line $y = b_0 + b_1''x$, where $b_1'' = (n + 2p)/d$.

(d) An equivalent fitting of the regression line $y = b_0 + b_1'''x$, where $b_1''' = (n + 3p)/d$.

Figure 5.5: Regression lines with gradients belonging to the same equivalence class.

Figure 5.5a shows the line fitted clearly is not aligned with the data. This is due to the gradient of the line being the fraction within the plaintext space or at least the representative equivalence class that contains the correct gradient. In other words, the gradient that was output from the regression algorithm is always of the form $0 \leq b_1 < p$, where p is the plaintext modulus defining the plaintext space $\mathbb{A}_p$. However, the *QuasiQ* object b_1 is a fraction $b_1 = n/d$, where $0 \leq n, d < p$. Despite this being the b_1 value received, it belongs to the equivalence class $b_1 \bmod p$. Alternatively, using the Euclidean division algorithm, given $b_1 \bmod p$ there exists a set of solutions of the form $x = k \cdot p + b_1$, for $k \in \mathbb{Z}$. This is slightly more complex given we are now dealing with a rational number $b_1 = n/d$ rather than an integer in a sense that the value belongs to the set of rationals a/b, where $a = j \cdot p + n$ and $b = k \cdot p + d$ for $j, k \in \mathbb{Z}$.



(a) Initial "incorrect" fitting of the regression line $y = b_0 + b_1 x$, where $b_0 = n/d$.

(b) An equivalent fitting of the regression line $y = b_0 + b_1' x$, where $b_0' = (n + 3p)/d$.

(c) An equivalent fitting of the regression line $y = b_0 + b_1'' x$, where $b_0'' = (n + 6p)/d$.

(d) An equivalent fitting of the regression line $y = b_0 + b_1''' x$, where $b_0''' = (n + 9p)/d$.

Figure 5.6: Regression lines with intercepts belonging to the same equivalence class.

Since the important information is the ratio between n and d, choosing to keep the denominator value constant and scanning through the elements of the equivalence class for the correct rational should obtain the correct value in the rationals. In the case in Figure 5.5, the denominator d is chosen to be kept constant. The individual graphs are obtained by incrementing through the gradients $b_1' = (n + k \cdot p)/d$ for $k = 0, 1, 2, 3$, where the case $k = 0$ is the starting line as an initial output. The correct line gradient is shown in Figure 5.5c and the line of best fit that was calculated by the linear regression algorithm. The same issue can happen to the intercept b_0 as well. An example of this is shown in Figure 5.6.

Moreover, the final case may be that neither the intercept b_0 nor the gradient b_1 have the correct rational representation when decoded. As a result, obtaining the correct rational values for the line of best fit can be challenging as there are many possible combinations that will need to be compared. In practice for simple linear regression it is not expected that wrap-around should occur very often during the computation, so scanning through the set of the equivalence class for the correct rational should be relatively quick for small k. However, for other algorithms this may not be the case.

5.4 Evaluation of the use of QuasiQs

It has been successfully shown that it is possible to not only represent fractions using an FHE scheme but also compute on such data as well. However, there are a number of new limitations, not present in previous solutions, which our solution introduces.

Arguably, the most significant limitation of this scheme is due to building a field fractions from a finite integral domain which has implications of representing the rationals, $\mathbb{Q}$. Because of the way the fractions are constructed from a finite domain, there will be rationals that are not present in an equivalence class. This is trivially seen by looking at the set of rationals that define an equivalence class in such a finite field of fractions,

$$\left[\frac{n}{d}\right]_p = \left\{\frac{up+n}{tp+d} : u,t \in \mathbb{Z}; n,d \in \mathbb{Z}_p; d \neq 0 \text{ and } p \text{ is prime}\right\}.$$

Since $d \neq 0$, it means that there is a set that is given when d is zero that is not included in the finite fractions. This is a problem because there are rational numbers that are not members of one of the equivalence classes. As these rational numbers do not exist in our field of fractions, attempting to perform operations with such rational numbers can yield somewhat unexpected results. This unfortunately is not a scenario that is easy to avoid when using the BGV scheme to compute using our *QuasiQ* objects. Detection when a denominator becomes 0 while the data is encrypted is not possible. This can be avoided at encoding of rationals by making sure that the n and d values being encoded are in the bounds $[0, p-1]$ and $[1, p-1]$ respectively. However, during computation wrap-around could occur sending d to zero and will not be noticed until the final result from the algorithm is decrypted by the user and they realise the answer is not correct.

Another issue due to the fractions being encrypted, the values cannot be observed. This is necessary for security, but has an undesired effect because in the field of fractions there is supposed to be an equivalence relation defined as discussed in Section 4.3. As a result of this, when performing an operation between two *QuasiQ* objects, a fraction may be obtained where the numerator and denominator have common factors. However, not being able to see the encrypted values, there is no way of knowing whether the result can be simplified as well as the factor in which the fraction can be simplified by. Therefore as operations are performed upon the data, the values of both the numerator and denominator increase relatively quickly and thus increases the chance of obtaining values which wrap around due to becoming larger than the plaintext modulus p. As a result of this, the scheme currently requires the use of extremely large values of p which is not a preferable parameter selection.

Moreover, another issue when using *QuasiQ* numbers is the unfortunate increase in the computation complexity of performing addition in the field of fractions. Namely, given two fractions $\frac{a}{b}$ and $\frac{c}{d}$, the resulting sum of the two fractions is $\frac{ad+bc}{bd}$. Computationally this requires us to compute three multiplications as well as an addition, which may not be too computationally costly in the plaintext domain but in the encrypted domain is an undesired implication of our form of representation. A normal homomorphic addition under the BGV scheme is not computationally expensive in terms of the amount of ciphertext noise it produces and thus usually more than one addition can be performed before a level needs to be consumed in order to reduce the ciphertext noise level. However, when performing an addition operation in a field of fractions, three homomorphic multiplications as well as the single addition are required. Moreover, due to multiplications having a greater impact on the ciphertext noise increase, it results in the approximate consumption of three levels each time that an addition operation is performed. This worsens how algorithms scale between the plaintext and encrypted domain. Performing simple linear regression on a set of 64 data points was empirically found to consume roughly 20 levels when using a plaintext prime $p = 1009$. Thus, for larger data sets it is highly likely that bootstrapping may be required to perform the algorithm.

However, despite the *QuasiQ* representation being more computationally expensive in terms of additions it is more efficient at computing the multiplicative inverse. Since the plaintext modulus p is always chosen to be prime, $\mathbb{Z}_p$ is a field and thus the field of fractions is itself $\mathbb{Z}_p$. Therefore the reason for choosing one representation over the other needs to be considered. Given an element $c \in \mathbb{Z}_p$, obtaining the multiplicative inverse c^{-1} is computationally expensive as it involves using Fermat's Little Theorem to obtain $c^{p-2} \equiv c^{-1} \bmod p$ [40, 44] and therefore many multiplications. Given a *QuasiQ* object (n, d) it is notably less expensive to compute the inverse as it involves swapping the two elements of the ordered pair to produce (d, n); albeit swapping two pointers.

Although there are limitations, such as the inability to simplify fractions while they are encrypted to mitigate the rapid increase of the numerator or denominator and limiting the chance for wrap-around to occur, there are possible future solutions to consider. In the case of simplification, one could possibly implement a method which computes the inverse of 2 in $\mathbb{A}_p$. Then after an arbitrary number of operations throughout the algorithm, an intermediary step can be performed to multiply both the numerator and denominator by the inverse of 2. This would be dividing both the numerator and denominator by 2 which is equivalent to multiplying by 1, the identity. As the algorithm is performed, a counter can record the number of occurrences of this operation and when the user decrypts the final result, the correct answer can be reconstructed by multiplying the result by the correct power of 2.

5.5 Comparison with Previous Methods

Comparing our solution with previous attempts at approximating real numbers, it is evident that our method removes the problem of added bookkeeping when it comes to keeping track of the scale in solutions using fixed scalar multipliers [51, 18, 23, 19, 8]. These previous solutions use a fixed scalar multiplier to map rationals to integers, however it is important to keep track of this scalar in order ensure the output is decoded correctly from an integer to a rational. Our solution mitigates this as we keep the ratio as a *QuasiQ* object, thus adding no extra complexity to the decoding step.

Due to the use of a polynomial for the plaintext modulus as opposed to a prime number p, [19] introduces a notable algorithmic overhead to the decryption phase. Our solution does not introduce any overhead to the encryption and decryption phase as we use standard ciphertext objects to represent both our numerator and denominator. Despite doubling the memory usage, the numerator and denominator can be computed in parallel, thus the computational overhead of encrypting and decrypting a pair of ciphertexts will be minimal.

We found that similar to previous solutions, we tend to require larger values of p, however, unlike [51, 18, 23, 19, 8], this is due to the lack of fraction simplification which described in section 5.4. Previous solutions required larger values of p due to an increased noise growth. Our solution results in an increased noise growth only during the addition of two *QuasiQ* objects but not for other operations.

Due to our solution removing the extra bookkeeping of most previous solutions in the literature, it makes our solution conceptually simpler when compared to the existing literature. However it introduces an undesired computational overhead to rational additions. Thus our solution is computationally more expensive when used for algorithms containing a large number of multiplications. Unlike previous solutions, our proposed method makes computing the reciprocal free in terms of homomorphic operations, thus division is as computationally expensive as multiplication.

The method we propose is conceptually similar to [72] which uses the Gentry-Sahai-Waters (GSW) scheme [41]. Since our method uses the BGV scheme [14], we use vector additions and multiplications as opposed to matrix multiplications and additions, thus making our method less computationally expensive.

5.6 Conclusion

The ability to preform analytics in FHE is an important area of study and no one solution has yet gained dominance. In this chapter, we have shown that a field of fractions can naturally be constructed from the plaintext space algebra from the BGV scheme. The practical implementation that we call *QuasiQ* has some limitations as opposed to an actual field of fractions due to the encryption.

As an approximation to the rationals, the *QuasiQ* objects can be put to use for analytics. This has been demonstrated with a practical example use case, namely, performing simple linear regression, implemented in FHE, upon simple generated noisy data. This opens the possibility for performing more complex algorithms in the encrypted domain with *QuasiQ* objects. There are some inherent limitations that a finite field of fractions have to represent the rationals; due to being a subfield, the “wrong” fraction can be given when decoding back to the rationals. A correction method for this has been described, something which is not possible with other alternative methods in the literature. However, the limitations are similar to other current solutions that have been proposed in the literature being limited to a bounded space and having a minimum precision and the operations between *QuasiQ* objects are simple to implement, only requiring the basic ciphertext operations. Therefore, this is a viable alternative option to perform analytics and further study is warranted.

Chapter 6

Performing Logistic Regression using FHE

Subsequent to exploring the feasibility of applying FHE to a networking environment as well as to represent rational numbers, there is another key component of FHE that is yet to be explored. Namely the feasibility of using FHE to compute complex functions that are infeasible to compute using the somewhat homomorphic variant. This section describes my experience in building a logistic regression model that worked on genomic data within an FHE environment. This project was initially aimed to solve a real-world problem proposed by the iDASH centre [50].

6.1 Introduction

6.1.1 Somewhat vs. Fully Homomorphic Encryption

Though only a theoretical plausibility result at first, the last decade saw major algorithmic improvements in FHE schemes, resulting in many research prototypes that implement this technology (e.g., [45, 36, 18, 24]) and attempt to use it in different settings (e.g., [7, 42, 54, 29, 43, 56], among others).

Nearly all contemporary FHE schemes come in two variants: The basic underlying scheme is *somewhat homomorphic* (SWHE), where the parameters are set depending on the complexity of the required homomorphic operations, and the resulting instance can only support computations up to that complexity. The reason is that ciphertexts are noisy, with the noise growing throughout the computation, and once the noise grows beyond some (parameter-dependent) threshold the ciphertext can no longer be decrypted. This can be solved using Gentry's bootstrapping technique (at the cost of relying on circular security). In this technique the scheme is augmented with a *recryption* operation to refresh the ciphertext and reduce its noise level. The augmented scheme is thus *fully homomorphic* (FHE), meaning that a single instance with fixed parameters can handle arbitrary computations. But FHE is expensive, as the computation must be peppered with expensive recryption operations. So, it is often cheaper to settle for a SWHE scheme with larger parameters (that admit larger noise) as in the previous two cases proposed in Chapter 4 and 5.

Indeed, with very few exceptions, almost all prior attempts at practical use of FHE used only the SWHE variant, fixing the target computation and then choosing parameters that can handle that computation and no more. But SWHE has its limits: as the complexity of the function grows, the SWHE parameters become prohibitively large. In this work, we set out to investigate the practical feasibility of "deep FHE", attempting to answer the fundamental question of FHE's usefulness in practice:

Can FHE realistically be used for complex functions?

It is noted that the term *recryption* will be used interchangeably with *bootstrapping* as this is the name used for HElib's [48] implementation of bootstrapping and is not to be confused with the term *re-encryption.*

6.1.2 The iDASH Competition

Over the last few years, competitions by the iDASH center [50] provided a good source of "real world problems" to grind our teeth on. iDASH promotes privacy-preserving approaches to analysis of medical data, and since 2014 they have organized yearly competitions where they present specific problems in this area and ask for solutions using technologies such as differential privacy, secure multi-party computation, and homomorphic encryption.

In the homomorphic-encryption track of the 2017 competition, the problem to be solved was to compute homomorphically the parameters of a logistic-regression model, given encrypted data. The data consists of records of the form $(\vec{x}, y)$, where $\vec{x} \in \{0,1\}^d$ is a vector of d bits, with each bit x_i representing an attribute of an individual (e.g., man or woman, over 40 or not, high blood pressure or not, etc.), and $y \in \{0,1\}$ is the target attribute that we investigate (e.g., whether or not they have a heart disease). A logistic-regression model tries to predict the probability of $y = 1$ given a particular value of $\vec{x}$, postulating that the probability $p_{\vec{x}} \overset{\text{def}}{=} \Pr[y = 1|\vec{x}]$ can be expressed as $p_{\vec{x}} = 1/(1 + \exp(-w_0 - \langle \vec{x}, \vec{w'} \rangle))$ for some fixed vector of weights $\vec{w} = (w_0, \vec{w'}) \in \mathbb{R}^{d+1}$. (The term w_0 is typically called an "offset".) Given sample data consisting of n records, our task is to find the weight vector $\vec{w} \in \mathbb{R}^{d+1}$ that best approximates the empirical probabilities (e.g., in the sense of maximum likelihood). For a more detailed exposition see Section 6.2.

In addition to presenting the problem, the iDASH organizers also provided some sample data on which to test our procedures. The data consisted of nearly 1600 records of genomic data, each with 105 attributes (but they also accepted solutions that could only handle much fewer attributes than this). With so many attributes, this appears firmly outside the scope of SWHE[1], hence we set out to design a solution to the iDASH task using FHE.[2]

[1] Some entries in the iDASH competition, including the winner, found clever ways to use SWHE for this problem, albeit only for a much smaller number of attributes. See for example [56].

[2] Unfortunately, our solution was not ready in time for the iDASH competition deadline, so we ended up not participating in the formal competition.

6.1.3 Our Logistic-Regression Procedure

The starting point for our solution is a closed-form formula that we developed for approximating the logistic-regression parameters. This formula, developed in Section 6.2 below, involved partitioning the records into "buckets", one per value of $\vec{x} \in \{0,1\}^d$, then counting the numbers of $y = 1$ and $y = 0$ in each bucket. These bucket counters are then used to derive a linear system $A\vec{w} = \vec{b}$ whose solution is the vector of weights $\vec{w}$ that we seek. As explained in Section 6.2, computing $A, \vec{b}$ from the bucket counters involve complicated functions such as rational inversion and the natural logarithm.

The first issue that we have to deal with, is that our approximation formula only yields valid results in settings where the number of records n far exceeds the number of attributes d. Specifically, it relies on the fraction of $y = 1$ records in each bucket $\vec{x}$ to roughly approximate $p_{\vec{x}}$, so in particular we must have $n \gg 2^d$ to ensure that we have sufficiently many records in each bucket. But we aim at a setting with $d > 100$, which is far outside the validity region of this approximation formula.

We thus added to our solution a "quick-n-dirty" pre-processing phase, in which we homomorphically extract from the d input attributes a set of $k \ll d$ attributes which are likely to be the most relevant ones, then apply the approximation formula only to these k attributes, and set $w_j := 0$ for all the others. Specifically, in our solution we used $k = 5$, since the sample iDASH data had very few attributes with significant w_j coefficients.

This quick-n-dirty procedure involves computing the correlation between each column (attribute) x_j and the target attribute y, this is essentially just computing linear functions. Then we find the indexes $j_1, \ldots, j_k$ of the k columns x_j that are most correlated with y, and extract only these columns from all the records. The high-level structure of our homomorphic procedure is therefore:

1. For each column j, compute $\mathsf{Corr}_j = |\mathsf{Correlation}(x_j, y)|$;

2. Compute $j_1, \ldots, j_k$, the indexes of the k columns with largest Corr_j values;

3. Extract the k columns $j_1, \ldots, j_k$, setting

$$\vec{x_i'}[1 \ldots k] := (\vec{x_i}[j_1], \ldots \vec{x_i}[j_k]);$$

4. Compute the bucket counters, for every $\vec{x} \in \{0,1\}^k$ set

$$\begin{aligned} Y_{\vec{x}} &:= \Big|\{i \le N : \vec{x_i'} = \vec{x} \text{ and } y_i = 1\}\Big|, \\ N_{\vec{x}} &:= \Big|\{i \le N : \vec{x_i'} = \vec{x} \text{ and } y_i = 0\}\Big|. \end{aligned}$$

5. Compute $A \in \mathbb{R}^{(k+1)\times(k+1)}$ and $\vec{b} \in \mathbb{R}^{k+1}$ from the $Y_{\vec{x}}$'s and $N_{\vec{x}}$'s.

6. Solve the system $A\vec{w'} = \vec{b}$ for $\vec{w'} \in \mathbb{R}^{k+1}$, then output the coefficients $w_0 := w_0'$, $w_{j_i} := w_i'$ for the columns $j_1, \ldots, j_k$, and $w_j := 0$ for all other columns j.

Jumping ahead, about 55% of the computation time is spent in the first "quick-n-dirty" phase, which is the only part of the computation that manipulates homomorphically the entire input dataset.

6.1.4 Homomorphic Computation of the Approximation Procedure

We used the HElib library [48] as our back end to evaluate our approximation procedure homomorphically. Devising a homomorphic computation of this procedure brings up many challenges. Here we briefly discuss some of them.

Implementing complex functions. Obtaining the linear system $A, \vec{b}$ from the bucket counters $Y_{\vec{x}}, N_{\vec{x}}$ involves computing functions such as rational division, or the natural logarithm. Computing these functions homomorphically, we have two potential approaches: one is to try to approximate them by low-degree polynomials (e.g., using their Taylor expansion), and the other to pre-compute them in a table and rely on homomorphic table lookup.

In this work we opted for the second approach, which is faster and shallower when applicable, but it can only be used to get a low-precision approximation of these functions. In our solution we used six or seven bits of precision, see more details in Section 6.4.

Homomorphic binary arithmetic and comparisons. Other things that we needed were the usual addition and multiplications operations, but applied to integers in binary representation (i.e., using encryption of the individual bits). Somewhat surprisingly, these basic operations were not discussed much in the literature, not in terms of proper implementations. Computing them homomorphically is mostly a matter of implementing textbook routines (e.g., carry look ahead for addition). But in this context we are extremely sensitive to the computation depth, which is not typical in other implementations. We describe our implementation of these methods and their various optimizations in Section 6.5.

Deciding on the plaintext space. HElib supports working with different plaintext-space moduli, and different calculations are easier with different moduli. In particular, the correlation computation in the first step is much easier when using a large plaintext space, as this lets us treat it as a linear operation over the native plaintext space. But most other operations above are easier when working with bits.

Here we use some features of the recryption implementation in HElib: When set to recrypt a ciphertext whose plaintext space is modulo 2, HElib uses temporary ciphertexts with plaintext space modulo 2^e for some $e > 2$ (usually $e = 7$ or $e = 8$). In particular it means that HElib can support computation with varying plaintext spaces of the form 2^e, and it also supports switching back and forth between them.

In our procedure, we used a mod-2^{11} plaintext space for computing the initial correlation, then switched to mod-2 plaintext space for everything else.

Setting the parameters. Setting the various parameters for bootstrapping is somewhat of an art form, involving many trade-offs. In our implementation we settled for using the m-th cyclotomic ring with $m = 2^{15} - 1$, corresponding to lattices of dimension $\phi(m) = 27000$. We set the number of levels in the BGV moduli-chain so that at the end of recryption we will still have nine more levels to use. Recryption itself for this value of m takes 20 levels, so we need a total of 29 levels. This means that we used a maximum ciphertext modulus q of size roughly 1030 bits, yielding a security level of just over 80 bits.

Solving linear systems. The last step in the approximation procedure above is to solve a linear system over the rational numbers. Performing this step homomorphically (with good numerical stability) is a daunting task. We considered some "pivot free" methods of doing it, but none of them seemed like it would be a good solution to what we need.

Since this is the last step of the computation, one option is to implement instead a *randomized encoding* of this step, which may be easier to compute. We discuss that option in Section 6.6, in particular describing randomized encoding of the linear-system-solver function, that may be new. However we did not implement that scheme in our solution, instead we settled for a leaky solution that simply sends the linear system to be decrypted and solved in the clear.

6.1.5 The End Result

We implemented homomorphically all aspects of the procedure above, except the final linear-system solver. The program takes a little over four and a half hours on a single core to process the dataset of about 1600 encrypted records and 105 attributes. Over two and a half hours of this time is spent on extracting the five most relevant columns, about 45 minutes are spent on computing the bucket counters, and the remaining hour and 15 minutes is spent computing A and $\vec{b}$ from these counters. We can use more cores to reduce this time, down to just under one hour when using 16 cores. See more details in Section 6.7. In terms of accuracy, our solution yields "area under curve" (AUC) of about 0.65 on the given dataset, which is in line with other solutions that were submitted to the iDASH competition.

6.1.6 Related Work

Surprisingly, not much can be found in the literature about general-purpose homomorphic implementation of basic binary operations. The first work that we found addressing this issue is by Cheon et al. [21], where they describe several optimizations for binary comparison and addition, in a setting where we can spread the bits of each integer among multiple plaintext slots of a ciphertext. They show procedures that use fewer multiplication operations, but require more rotations. These optimizations are very useful in settings where you can ensure that the bits are arranged in the right slots to begin with. But in our setting, we use these operations as a general-purpose tool, working on the result of previous computation. In this setting, the need for many rotations will typically negate the gains from saving on the number of multiplications. We thus decided to stick to bit-slice implementation throughout our solution, and try to make them use as few operations (and as small depth) as possible.

Other relevant works on homomorphic binary arithmetic are due to Xu et al. [77] and Chen et al. [20], who worked in the same bitslice model as us and used similar techniques. But they only provided partially optimized solutions, requiring deeper circuits and more multiplications than we use. (For example, they only used a size-4 carry-lookahead-adder for addition, and did not use the three-for-two procedure for multiplication.)

In terms of applying homomorphic encryption to the problem of logistic regression, the work of Aono et al. [2] described an interactive secure computation protocol for computing logistic regression, using additively-homomorphic encryption. Mohassel and Zhang [63] also described related secure-MPC protocols (but using garbled circuits, not HE). Wang et al described in [76] a system called HEALER that can compute homomorphically an exact logistic-regression model, but (essentially) only with a single attribute and only with very small number of records (up to 30).

A lot more work on the subject was done as part of the iDASH competition in 2017, but the only public report that we found on it is that of Kim et al. [56]. In this report they describe their implementation, using the somewhat-homomorphic scheme for approximate numbers due to Cheon et al. [22] to implement a homomorphic approximation of logistic regression (with a small number of attributes) using gradient-descent methods.

6.1.7 Organization

The rest of this chapter is organized as follows: In Section 6.2 we derive our closed-form approximation formula for logistic regression. Then in Section 6.3 we provide a bird-eye view of our solution, describing the individual steps and explaining how they are implemented. In Sections 6.4-6.5 we describe many of the toolboxes that we developed and used as subroutines in this solution, and in Section 6.7 we give performance results of our implementation. Our randomized encoding for the linear-system solver over the rational numbers in developed in Section 6.6, and we conclude with a short discussion in Section 6.8.

6.2 Logistic Regression and Our Approximation

Logistic regression is a technique to model dependence between related attributes. The input consists of n records (rows), each with $d+1$ attributes (columns), all of the form $(\vec{x_i}, y_i)$ with $\vec{x_i} \in \{0,1\}^d$ and $y_i \in \{0,1\}$. Below we sometimes refer to a fixed value $\vec{c} \in \{0,1\}^d$ as a *category.* (We sometimes also refer to the different values $\vec{c} \in \{0,1\}^d$ as "buckets".) The ultimate goal is to estimate the probability $p_{\vec{x}} = \Pr[y = 1|\vec{x}]$. Logistic regression is a model that postulates that this probability is determined as

$$p_{\vec{x}} = \frac{1}{1 + \exp\big(- w_0 - \sum_{i=1}^{n} x_i w_i \big)} = \frac{1}{1 + \exp\big(- \langle (1|\vec{x}), \vec{w} \rangle \big)}$$

for some fixed vector of weights $\vec{w} \in \mathbb{R}^{d+1}$. The goal of logistic regression, given all the records $\{(\vec{x_i}, y_i)\}_{i=1}^{n}$, is to find the vector $\vec{w}$ that best matches this data. Below we denote $\vec{x'} \overset{\text{def}}{=} (1|\vec{x})$, and we use the expression for $p_{\vec{x}}$ as a function of $\vec{w} \in \mathbb{R}^{d+1}$, namely we denote

$$p_{\vec{x}}(\vec{w}) \stackrel{\text{def}}{=} \frac{1}{1+\exp(-\langle\vec{x'},\vec{w}\rangle)} = \frac{\exp(\langle\vec{x'},\vec{w}\rangle)}{1+\exp(\langle\vec{x'},\vec{w}\rangle)}\ . \tag{6.1}$$

For a candidate weight-vector $\vec{w}$ and some given record $(\vec{x}, y)$, the model probability of seeing this outcome y for the attributes $\vec{x}$ is denoted $P_{\vec{x},y}(\vec{w}) \stackrel{\text{def}}{=} \{1 - p_{\vec{x}}(\vec{w}) \text{ if } y = 0,\ \ p_{\vec{x}}(\vec{w}) \text{ if } y = 1\}$. If we assume that the records are independent and use maximum-likelihood as our notion of "best match", then the goal is to find $\vec{w}^* = \text{argmax}_{\vec{w}} \left(\prod_{i=1}^{n} P_{\vec{x_i},y_i}(\vec{w})\right) = \text{argmax}_{\vec{w}} \left(\sum_{i=1}^{n} \ln(P_{\vec{x_i},y_i}(\vec{w}))\right)$.

6.2.1 A Closed-Form Approximation Formula for Logistic Regression

To get our approximation formula for logistic regression, we partition the data into the 2^d "categories" $\vec{c} \in \{0,1\}^d$. For each category $\vec{c}$, we denote the number of records in that category by $n_{\vec{c}}$, the number of records in that category with $y = 1$ by $Y_{\vec{c}}$, and the number of records with $y = 0$ by $N_{\vec{c}} = n_{\vec{c}} - Y_{\vec{c}}$ ('Y' and 'N' for YES and NO, respectively). We also partition the last sum above into the 2^{d+1} terms corresponding to all the $Y_{\vec{c}}$'s and $N_{\vec{c}}$'s,

$$\sum_{i=1}^{n} \ln(P_{\vec{x_i},y_i}(\vec{w})) = \sum_{\vec{c}\in\{0,1\}^d} Y_{\vec{c}} \cdot \ln(p_{\vec{c}}(\vec{w})) + N_{\vec{c}} \cdot \ln(1 - p_{\vec{c}}(\vec{w})).$$

Below it is convenient to do a change of variables and consider the "log odds ratio",

$$r_{\vec{c}}(\vec{w}) \stackrel{\text{def}}{=} \ln\left(\tfrac{p_{\vec{c}}(\vec{w})}{1-p_{\vec{c}}(\vec{w})}\right) = \langle\vec{c'},\vec{w}\rangle,$$

where $\vec{c'} = (1|\vec{c})$. Then, $p_{\vec{c}}(\vec{w}) = 1/\big(1 + e^{-r_{\vec{c}}(\vec{w})}\big)$ and $1 - p_{\vec{c}}(\vec{w}) = 1/\big(1 + e^{r_{\vec{c}}(\vec{w})}\big)$. With this change of variables, we now want to find

$$\vec{w}^* = \underset{\vec{w}}{\text{argmax}} \sum_{\vec{c}\in\{0,1\}^d} Y_{\vec{c}} \cdot \ln\left(\frac{1}{1+e^{-r_{\vec{c}}(\vec{w})}}\right) + N_{\vec{c}} \cdot \ln\left(\frac{1}{1+e^{r_{\vec{c}}(\vec{w})}}\right). \tag{6.2}$$

Fix some category $\vec{c} \in \{0,1\}^d$, and consider the term corresponding to $\vec{c}$ in the sum above as a function of $r = r_{\vec{c}}(\vec{w})$ (with $Y_{\vec{c}}, N_{\vec{c}}$ as parameters), namely

$$f_{Y,N}(r) \overset{\text{def}}{=} Y \ln\big(\frac{1}{1+e^{-r}}\big) + N \ln\big(\frac{1}{1+e^{r}}\big).$$

To develop our closed-form formula, we approximate $f_{Y,N}(\cdot)$ using Taylor expansion around its maximum point $r_0 = \ln(Y/N)$,

$$f_{Y,N}(r) \;\approx\; \mathsf{someConstant} \;-\; \frac{YN}{2(Y+N)} \cdot \Big(r - \ln\big(\frac{Y}{N}\big)\Big)^2 . \tag{6.3}$$

(We discuss the validity of this approximation later in this section.) Recall that we are seeking the weight-vector $\vec{w}$ that maximizes Eqn. (6.2), and hence we can ignore the $\mathsf{someConstant}$ (as well as the 1/2 factor in $\frac{YN}{2(Y+N)}$) since these do not depend on $\vec{w}$. Hence the value that we seek is

$$\begin{aligned} \vec{w^*} \;&=\; \text{argmax}_{\vec{w}} \Big\{ -\textstyle\sum_{\vec{c}} \underbrace{\frac{Y_{\vec{c}} N_{\vec{c}}}{Y_{\vec{c}} + N_{\vec{c}}}}_{\overset{\text{def}}{=} V_{\vec{c}}} \cdot \Big(r_{\vec{c}}(\vec{w}) - \underbrace{\ln(Y_{\vec{c}}/N_{\vec{c}})}_{\overset{\text{def}}{=} L_{\vec{c}}} \Big)^2 \Big\} \\ &=\; \text{argmin}_{\vec{w}} \Big\{ \textstyle\sum_{\vec{c}} V_{\vec{c}} \cdot \big(\langle \vec{c'}, \vec{w} \rangle - L_{\vec{c}} \big)^2 \Big\} . \end{aligned}$$

We continue by expressing the last expression in matrix form. Let $\vec{V}, \vec{L}$ be 2^d-dimensional column vectors consisting of all the $V_{\vec{c}}$'s and $L_{\vec{c}}$'s, respectively. Also let C_d be a $(d{+}1) \times 2^d$ 0-1 matrix whose columns are all the $\vec{c'}$ vectors (namely the m'th column is $(1|\mathsf{bin}(m))^t$). For example for $d = 3$ we have

$$C_3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} . \tag{6.4}$$

Then the expression above can be written in matrix form as

$$F(\vec{w}) \overset{\text{def}}{=} \sum_{\vec{c}} V_{\vec{c}} \cdot \big(\langle \vec{c'}, \vec{w} \rangle - L_{\vec{c}} \big)^2 \;=\; \Big(\vec{w}^T C_d - \vec{L}^T \Big) \times \mathsf{diag}(\vec{V}) \times \Big(C_d^T \vec{w} - \vec{L} \Big) .$$

$F(\vec{w})$ is a quadratic form in $\vec{w}$, and it is minimized at

$$\vec{w^*} = \operatorname*{argmin}_{\vec{w}} F(\vec{w}) = \left(C_d \times \mathsf{diag}(\vec{V}) \times C_d^T\right)^{-1} \times C_d \times \mathsf{diag}(\vec{V}) \times \vec{L}.$$

This finally gives us our closed-form approximation formula: Given all the records $(\vec{x_i}, y_i)$ we compute all the YES and NO counters for the different categories, $Y_{\vec{c}}$ and $N_{\vec{c}}$, then set $V_{\vec{c}} := Y_{\vec{c}} N_{\vec{c}}/(Y_{\vec{c}} + N_{\vec{c}})$ and $L_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$.

We then let D be the 2^d-by-2^d diagonal matrix with the $V_{\vec{c}}$'s on the diagonal and $\vec{U}$ be the 2^d vector with entries $V_{\vec{c}} \cdot L_{\vec{c}}$, and we compute the approximation $\vec{w^*} := (C_d\ D\ C_d^T)^{-1} \times C_d\ \vec{U}$.

6.2.2 Validity of the Approximation

It is clear that the approximation procedure above relies on the number of records being sufficiently larger than the number of categories, so that we have enough YES and NO instances in each category. (Indeed if any $Y_{\vec{c}}$ or $N_{\vec{c}}$ is zero then the value $L_{\vec{c}}$ becomes undefined.)

Below we therefore assume that the number of records in each category is very large (i.e., tends to infinity). This implies by the law of large numbers that $Y_{\vec{c}}$ and $N_{\vec{c}}$ (now considered as random variables) can be approximated by normal random variables. In particular they tend to their expected value $p_{\vec{c}} \cdot n_{\vec{c}}$ and $(1 - p_{\vec{c}}) \cdot n_{\vec{c}}$, and their log ratio $R_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$ tends to $\ln(p_{\vec{c}}/(1 - p_{\vec{c}}))$. Turning that around, it means that we expect $r_{\vec{c}} = \ln(p_{\vec{c}}/(1 - p_{\vec{c}}))$ to be close to its observed value $\ln(Y_{\vec{c}}/N_{\vec{c}})$, and therefore the formula from Eqn. (6.3) using the Taylor expansion of $f(r)$ around $\ln(Y_{\vec{c}}/N_{\vec{c}})$ should be a good approximation. The remaining terms in the Taylor expansion represent a residual contribution that is not explained by the model.

Remark 1. In situations where the system is over-determined, it is possible for the model to spline the data. One way to reduce the impact of this over-fitting is to include an ad hoc penalty to the likelihood against the variance in β. Commonly applied penalty functions include linear L_1, and quadratic L_2 forms. However, it should also be recognized that such penalty functions do not represent sampling variation that the binomial and multinomial distributions seek to capture when sampled from disease/exposure p_j's.

Remark 2. In the expansion, genotypes $\{0, 1, 2\}$ and other similar constructions are easily accommodated by the approximation.

6.3 Overview of Our Solution

Expanding on the description from Section 6.1.3, we now explain our approximate logistic-regression procedure in more detail. We begin in Section 6.3.1 with a description of the various functions that we want to compute, then in Section 6.3.2 we describe (still on a high level) how we implement these functions homomorphically.

6.3.1 The procedure that we implement

Input & Output. The input consists of (the encryption of) n records $(\vec{x_i}, y_i)_{i=1}^{n}$, with $\vec{x_i} \in \{0, 1\}^d$ and $y_i \in \{0, 1\}$. Below we view the input as an n-by-d binary matrix X with the i'th row equal to $\vec{x_i}$, and a column vector $\vec{y} \in \{0, 1\}^n$ containing the y_i's. The output should be a logistic regression model, consisting of $d + 1$ real-valued weights $w_0, w_1, \ldots, w_d$.

Extracting Significant Columns. We begin by considering each column of X separately, extracting the the k columns that have the strongest correlation with the target vector $\vec{y}$. Below we use $\mathsf{HW}(\vec{v})$ to denote the Hamming weight of $\vec{v}$, and denote by $X_{|j}$ the j'th column of X. We compute the following quantities:

- Let $Y := \mathsf{HW}(\vec{y})$, i.e., the number of records with $y_i = 1$;
- For $j = 1, \ldots, d$, let $\alpha_j := \mathsf{HW}(X_{|j})$, i.e., count the records with $\vec{x_i}[j] = 1$;
- For $j = 1, \ldots, d$, let $\beta_j := \mathsf{HW}(X_{|j} \wedge \vec{y})$, i.e., records with $\vec{x_i}[j] = y_i = 1$;

- For $j = 1, \ldots, d$, let $\mathsf{Corr}_j := |n \cdot \beta_j - Y \cdot \alpha_j|$, i.e., the correlation magnitude between $\vec{y}$ and $X_{|j}$.

Then we redact the input records, keeping only the k attributes with the strongest correlation to y. Namely we extract from X the submatrix $X' \in \{0,1\}^{n \times k}$, consisting of the k columns with the largest values of Corr_j. In our implementation we used in particular $k = 5$. Let $j_1, \ldots, j_k$ be the indexes of the selected columns, and denote by $(\vec{x'_i}, y_i)_{i=1}^n$ the n redacted records, i.e., the rows of $(X'|\vec{y})$.

Computing category counters, variance, and log-ratio. We accumulate the redacted records in 2^k buckets according to their $\vec{x'_i}$ values. For each category $\vec{c} \in \{0,1\}^k$, we compute $Y_{\vec{c}}$, $N_{\vec{c}}$ as the number of records in that category with $y = 1$, $y = 0$, respectively,

$$Y_{\vec{c}} := \left|\left\{(\vec{x'_i}, y_i) : \vec{x'_i} = \vec{c}, y_i = 1\right\}\right|, \quad N_{\vec{c}} := \left|\left\{(\vec{x'_i}, y_i) : \vec{x'_i} = \vec{c}, y_i = 0\right\}\right|.$$

Then we compute the variance and log-ratio, respectively, as $V_{\vec{c}} := \frac{Y_{\vec{c}} N_{\vec{c}}}{Y_{\vec{c}} + N_{\vec{c}}}$ and $L_{\vec{c}} := \ln(Y_{\vec{c}}/N_{\vec{c}})$.

Setting up the linear system. We now arrange the $V_{\vec{c}}$'s and $L_{\vec{c}}$'s in vectors $\vec{V}$, $\vec{L}$: For any $m = 0, 1, \ldots, 2^k - 1$, let $\mathsf{bin}(m) \in \{0,1\}^k$ be the binary expansion of m, we set the m'th entry in $\vec{V}$ to $V_{\mathsf{bin}(m)}$, and the m'th entry in $\vec{L}$ to $L_{\mathsf{bin}(m)}$. Next we compute the coefficients of a linear system from the vectors $\vec{L}$, $\vec{V}$ as follows:

The matrix A. Let C_k be a fixed $(k+1) \times 2^k$ binary matrix, whose m'th column is $(1|\mathsf{bin}(m))^T$. An example for $k = 3$ is illustrated in Eqn. (6.4). We compute a real $(k+1) \times (k+1)$ matrix $A := C_k \times \mathsf{diag}(\vec{V}) \times {C_k}^T$ (over $\mathbb{R}$), where $\mathsf{diag}(V)$ is the diagonal matrix with $\vec{V}$ on the diagonal.

The vector $\vec{b}$. Let $\vec{U}$ be an entry-wise product of $\vec{V}$ and $\vec{L}$, i.e., $U_m := V_m \cdot L_m$ for all $m \in [2^k]$. We compute the real $(k+1)$-vector $\vec{b} := C_k \times \vec{U}$.

Computing the output. Finally, we solve the linear system $A\vec{w'} = \vec{b}$ (over $\mathbb{R}$) for $\vec{w'} \in \mathbb{R}^{k+1}$, then set the output vector $\vec{w} \in \mathbb{R}^{d+1}$ as follows:

- The offset is $w_0 := w'_0$;
- For the selected columns $j_1, \ldots, j_k$ we set $w_{j_\ell} := w'_\ell$;
- For all other columns we set $w_j := 0$.

6.3.2 Homomorphic Evaluation

We now proceed to give more details on the various steps we used for homomorphic evaluation of the functions above. The description below is still a high-level one, with many of the details deferred to later sections. In particular, this implementation relied on many lower-level tools for homomorphic evaluation of binary arithmetic and binary comparisons that will be described in Section 6.5, homomorphic table lookup in binary representation that will be described in Section 6.4, and more.

6.3.2.1 Parameters and plaintext space

For our native plaintext space we use the cyclotomic ring $\mathbb{Z}[X]/(\Phi_m(X), 2^{11})$, with $m = 32767$ (so $\phi(m) = 27000$). This native plaintext space yields 1800 plaintext slots, each holding an element of $\mathbb{Z}[x]/(F(X), 2^{11})$ for some degree-15 polynomial $F(X)$, irreducible modulo 2^{11}. (In other words, each slot contains the Hensel lifting of $GF(2^{15})$ to a mod-2^{11} ring.)

We stress that HElib includes operations for extracting the bits of an encrypted integer in a mod-2^t plaintext space, so we can always switch to bit operations when needed. The only limitation is that it roughly takes depth t to extract t bits, and we can only use bootstrapping once we have encryption of individual bits. We therefore must ensure that we always have enough homomorphic capacity left to extract the bits of whatever integers we are manipulating.

These 1800 slots are arranged in a $30 \times 6 \times 10$ hypercube, corresponding to the generators $g_1 = 11628 \in \mathbb{Z}_m^*/(2)$ of order 30, $g_2 = 28087 \in \mathbb{Z}_m^*/(2, g_1)$ of order 6, and $g_3 = 25824 \in \mathbb{Z}_m^*/(2, g_1, g_2)$ of order 10.

We note that due to limitations of the BGV encryption scheme that we use, we cannot realistically use a larger plaintext space. Using a large plaintext modulus adds to the noise of operations in the scheme, and above 2^{11} this added noise becomes too hard to deal with. In fact a better optimized implementation would have used a smaller plaintext space of perhaps 2^8 rather than 2^{11}. This would make computing the correlation a little harder, but would reduce the noise everywhere else.

6.3.2.2 Encrypting the input

As we said in the introduction, computing the correlation is much simpler when working with a large plaintext space, but for other operations it is easier to work with bit representation. We therefore encrypt the input more than one way, as follows:

- We keep two mod-2^{11} ciphertexts as accumulators for the α and β counters, and a few other ciphertexts for packing the raw data itself. Initially all the ciphertexts are initialized to zero. The number of the raw-data ciphertexts depends on the number of records in the dataset: The packing scheme that we use allows each raw-data ciphertext to hold up to 150 records, and we "fill" these ciphertexts one at a time until we encrypt all the records.

- Given a record $(\vec{x_i}, y_i)$, we pack the bits in the next available raw-data ciphertext, using a packing scheme that considers the 27000 coefficients in the native plaintext space as arranged in a $180 \times 150 = (30 \times 6) \times (10 \times 15)$ matrix. The j'th attribute in the record is then stored in the coefficient with index (j, i) in this matrix.

  In even more detail, let us consider the $d + 1$ vector $\vec{z_i} = (y_i|\vec{x_i})$ (where we assume that $d < 180$), and the bits of this record will be stored in the raw-data ciphertext of index i div 150. We let $i' = i \bmod 150$, $i_1 := i' \bmod 15$ and $i_2 := i'$ div 15, and for every $j = 0, 1, \ldots, d$ we also also let $j_1 := j \bmod 6$ and $j_2 := j$ div 6. Then the bit $\vec{z_i}[j]$ is stored in the slot of index (j_1, j_2, i_2) in the hypercube, in the i_1'st coefficient. To encrypt this record we prepare a fresh ciphertext that encrypts all the bits from $\vec{z_i}$ in the order above (and is otherwise empty), and homomorphically add it to the appropriate raw-data ciphertext.

Then, we also add the bits $\vec{x_i}[j]$ and $y_i \cdot \vec{x_i}[j]$ to the accumulator ciphertexts α and β. Again we prepare a fresh ciphertext that has each bit $\vec{x_i}[j]$ in the j'th slot (and zero elsewhere) and add it homomorphically to the α, accumulator, and similarly we homomorphically add to the β accumulator a fresh ciphertext with $y_i \cdot \vec{x_i}[j]$ in the j'th slot (and zero elsewhere).

6.3.2.3 Computing the correlation

Once we have encrypted all the records, we have in the α, β ciphertexts all the counters α_j, β_j (which we assume are sufficiently smaller than the plaintext-space modulus 2^{11}). We also assume that we are given in the clear (a good approximation of) the value Y, i.e. the number of records with $y_i = 1$. [3] Similarly we know in the clear the number of records n, so we would like to just compute homomorphically the linear combination $n \boxtimes \beta \boxminus Y \boxtimes \alpha$. Unfortunately our plaintext space is not large enough for this computation, as we expect the result to exceed 2^{11}. Instead, what we do is use a low-resolution approximation of n, Y, namely we compute the correlation values as

$$\mathsf{Ecorr} = \lceil n/S \rfloor \boxtimes \beta \boxminus \lceil Y/S \rfloor \boxtimes \alpha$$

for an appropriate scaling factor S, chosen just large enough so we can expect the result to fit in 11 bits.

An alternative implementation (that we did not try) is to use sub-sampling. Namely instead of adding all the record data into the accumulators α, β, we can sub-sample (say) only 1/8 of the records to add. This would give us three more bits, and we can even trade

[3] This is a valid assumption in the context of medical studies, since the fraction of YES records in the overall population is always given in “Table 1” in such studies.

it off with the amount of precision in n and Y (i.e., make the scaling factor S smaller as we sub-sample less records). Yet another option would have been to extract the bits of all the integers in α, β and perform the computation using bit operations, bypassing the plaintext-space issue altogether.

Once we have the Ecorr ciphertext, we extract the bits to get ciphertexts $\mathsf{Ecorr}_1, \mathsf{Ecorr}_2$, $\mathsf{Ecorr}_3, \ldots$, where Ecorr_i encrypts the i'th bit of all the correlation numbers (represented as signed integers in 2's-complement). I.e., the j'th slot in Ecorr_i encrypts the i'th bit of the number Corr_j.

Computing the absolute value. Once we have the bits of the Corr_j's, we need to compute their absolute value. Here we simplify things by computing the 1's-complement absolute value (rather than 2's-complement). Namely, for a signed integer in binary representation $x = x_t x_{t-1} \ldots x_0$, we will set $x'_i = x_i \oplus x_t$ $(i \leq t-1)$. Note that this introduces an error if $x < 0$ (since we now have $x' = -x - 1$ rather than $x' = -x$). But we assume that the significant columns have much stronger correlation than the others, so a ±1 error should not make a difference.

6.3.2.4 Finding the k most correlated columns

We now come to the most expensive part of our procedure, where we find the indexes of the k columns with largest correlation magnitude.

We note that the correlation computations above were done on packed ciphertexts, in a SIMD manner. This means that we now have a few ciphertexts (one for each bit of precision), with the i'th bit of $|\mathsf{Corr}_j|$ stored in the j'th slot of the i'th ciphertext. We therefore find the top few values by a shift-and-MUX procedure, using a binary comparison subroutine:

- Let $\vec{C}$ be the vector of values that we have, we compare point-wise $\vec{C}$ to $\vec{C} \ll \ell/2$ (where ℓ is the number of slots), homomorphically computing in each slot $j \leq \ell/2$ a bit b_j which is zero if $C_j < C_{j_{\ell/2}}$ and one otherwise. Then we set $\vec{C'} := \vec{b} \boxtimes (\vec{C} \ll$

$\ell/2) + (1 \boxminus \vec{b}) \boxtimes \vec{C}$.

- Then we repeat the process with shifting by $\ell/4$, etc. After $\log(\ell)$ such steps we have transformed the vector into a heap with the MAX value at the root (which is at slot 0).

  We then zero-out the MAX value at slot 0, and repeat the process to get the 2nd-largest value, then the 3rd-largest value, etc. After running this procedure k times, we have our k largest values.

  We remark that this procedure that we implemented does not take any advantage of the fact that after finding the largest value we have the values in a heap rather than in arbitrary ordering. But note that in our SIMD environment we only need $\log(\ell)$ operations to extract the next largest value, the same as extracting a value from a heap. We do not know if there is a SIMD solution that uses less than $\log \ell$ operations per extracted value.

- As described above, this procedure gives the k largest Corr_j values, but our goal here is to compute the argmax, namely the *indexes* of these k largest values.

  To that end, we pack the indexes in the same order as we do the values. Namely we keep another ciphertext that packs the index j in the same slot that Ecorr packs the value Corr_j. Then we perform the comparison on Ecorr, computing the $\vec{b}$ as before, and apply the same shift-and-MUX operations to both Ecorr and the ciphertext containing the indexes. This ensures that when the MAX value arrives in slot 0 in Ecorr, the index of the corresponding column will arrive at slot 0 of the other ciphertext.

Extracting the k most correlated columns. Now that we computed the indexes of the k significant columns, we proceed to extract these columns from the raw-data ciphertexts. Note that with the packing scheme as described above, each column j is packed in all the coefficients of all the slots with hypercube indexes $(j \bmod 6, j \text{ div } 6, \star)$.

We therefore implement a homomorphic operation, similar to the shift-and-MUX from above, that given the bits of j, move each slot $(j \bmod 6, j \text{ div } 6, i)$ to position $(0, 0, i)$, then zero-out all other slots, thus extracting the raw data of column j. We repeat this for column 0 (containing the y_i's) and columns $i_1, \ldots, i_k$.

6.3.2.5 Computing the category counters

Now that we extracted the data corresponding to the relevant $k+1$ columns, we need to count for every value $\{0,1\}^{k+1}$, how many records $(y_i, \vec{x_i'})$ we have with this value. After the column extraction step above the bits of each column are packed in all the coefficients of some of the slots (namely slots of index $(0, 0, \star)$) in several ciphertexts.

As a first step in computing the counters, we distribute the bits of each column j among the slots of one ciphertext C_j, one bit per slot. This is doable since we have 1800 slots per ciphertext, and less than 1800 records in our dataset. (If we had more records we could have used more ciphertexts to pack them, this would not have made a big difference in running time.) Similar to other data movement procedures (e.g., the replicate procedures from [44]), the bit distribution can be done using a shift-and-add approach, and there are some time-vs.-noise trade-offs to be had. In our program we somewhat optimized this step, but more optimizations are certainly possible.

After we have the bits from each column j in the slots of one ciphertext C_j, we proceed to compute in a SIMD manner all the indicator bits $\chi_{i,m}$, for $i = 0, \ldots, n$ and $m \in [2^{k+1}]$, indicating whether the record $\vec{x_i'}, y_i$ belongs to category m. I.e., whether $(y_i|\vec{x_i'}) = \mathsf{bin}(m)$. This is done simply by taking all the subset products of the $k+1$ ciphertexts C_j. Namely we compute the product ciphertexts $P_0, P_1, \ldots, P_{2^{k+1}-1}$ as follows:

$$\begin{aligned}
P_0 &:= (1-C_0)\boxtimes \ \dots\ \boxtimes(1-C_{k-1})\boxtimes(1-C_k)\\
P_1 &:= (1-C_0)\boxtimes \ \dots\ \boxtimes(1-C_{k-1})\boxtimes \quad C_k\\
P_2 &:= (1-C_0)\boxtimes \ \dots\ \boxtimes \quad C_{k-1} \quad \boxtimes(1-C_k)\\
P_3 &:= (1-C_0)\boxtimes \ \dots\ \boxtimes \quad C_{k-1} \quad \boxtimes \quad C_k\\
&\vdots\\
P_{2^{k+1}-1} &:= \quad C_0 \quad \boxtimes \ \dots\ \boxtimes \quad C_{k-1} \quad \boxtimes \quad C_k
\end{aligned}$$

Computing all these 2^{k+1} products is done in depth $\lceil \log_2(k+1)\rceil$, and using not much more than 2^{k+1} multiplications, as we describe in Section 6.4. (Specifically, for our choice of $k=5$ we use 96 multiplies.)

At this point, each slot i in the ciphertext P_m contains the indicator bit $\chi_{i,m}$. All that is left is to sum up all the slots in each ciphertext P_i (as integers in binary representation), getting the bits of the corresponding counter. In our program we implemented a special-purpose accumulation procedure for this purpose, described in Section 6.5 (but that procedure is not very well optimized). The accumulation of the P_m's for $m = 0\dots, 2^{k-1}$ gives the counters N_m, and the accumulation of the P_m's for $m = 2^k, \dots, 2^{k+1}-1$ gives the counters Y_{m-2^k}.

Our accumulation routine also includes a transpose-like operation: In the input we have different categories (buckets) represented by different ciphertexts, with the different rows across the slots. In the output we have the different categories across the slots and different ciphertexts for different bit positions. (We expect seven-bit counters in the output, so we have seven ciphertexts $Q_0, \dots, Q_6$ encrypting the bits of the counters. The slots in $\sum 2^i Q_i$ give all the counter values, with the m'th counter in the m'th slot.) We therefore need to "transpose" from categories across different ciphertexts to categories across slots in the same ciphertext. This transpose-like operation is handled at the same time as the accumulation: Beginning with all the slots in the input corresponding to bits of the same counter, we gradually accumulate many bits in larger integers, thereby clearing the slots of these bits so we can pack in these slots the integers for other counters, until we have the integers of all the counters packed across the slots of the result.

6.3.2.6 Computing the variance and log-ratio

Next we need to compute from Y_m and N_m the values $V_m = \frac{Y_m N_m}{Y_m + N_m}$ and $U_m = V_m \cdot \ln(Y_m/N_m)$. Computing the variance and log-ratio is done using table lookups: As described in Section 6.4, for some function f that we want to compute, we pre-compute a table T_f such that $T[x] = f(x)$ for all x. These tables are computed with some fixed input and output precision, which means that the values there are only approximate. (In our program we use 7 input bits and 7 output bits for most tables.)

We use tables for three functions in our program, specifically $T_{inv}[x] \approx 1/x$, $T_{inv1}[x] \approx 1/(x+1)$, and $T_{ln}[x] \approx \ln(x)/(x+1)$. Then given Y_m and N_m we compute

$$\begin{aligned} r_m &:= Y_m \cdot T_{inv}[N_m] \approx Y_m/N_m, \\ V_m &:= Y_m \cdot T_{inv1}[r_m] \approx Y_m \cdot \tfrac{1}{Y_m/N_m+1} = Y_m N_m/(Y_m + N_m) \\ U_m &:= Y_m \cdot T_{ln}[r_m] \approx \ln(Y_m/N_m) \cdot Y_m N_m/(Y_m + N_m) \end{aligned}$$

Since the counters are packed in the different slots of the ciphertexts Q_i, then we only need to perform these operations once to compute in SIMD all the V_m's and U_m's.

6.3.2.7 Computing the matrix A and vector $\vec{b}$

The next step is to compute $A := C_k \times \mathsf{diag}(\vec{V}) \times C_k^T$ and $\vec{b} := C_k \times \vec{U}$ (over the rationals), using the bit representation of the V_m's and U_m's (which in our program are represented by seven bits each). Given the structure of the 0-1 matrix C_k, in our $k = 5$ implementation these computations require computing a relatively small number of subset-sums of these numbers (in binary representation). In particular, for every two bits position $\ell_1, \ell_2 \in \{0 \ldots k-1\}$, we need to sum up all the number V_m corresponding to indexes m with bits ℓ_1, ℓ_2 set (i.e, $m_{\ell_1} = m_{\ell_2} = 1$), and we also need one more subset sum of all the numbers V_m of even index ($m_0 = 0$). Similar subset sums should be computed of the numbers U_m, and we pack the numbers U_m, V_m in such a way that the sums for U_m, V_m can be computed together.

Computing all the entries of $A, \vec{b}$ for our case $k = 5$ takes only 16 subset sums. Note that since the different numbers are packed in different slots, then adding two numbers require that we rotate the ciphertext to align the slots of these numbers. Again we carefully packed the numbers in the slots to reduce the number of rotations needed, and this step in its entirety requires 50 different rotation amounts (each applied to all the seven bits of the numbers, for a total of 350 rotation operations).

6.3.2.8 Solving $A\vec{w'} = \vec{b}$

The final operation that needs to be computed is solving the linear system $A\vec{w'} = \vec{b}$ over the rationals to find $\vec{w'}$. Here, however, we have a problem: recall that the procedures above only compute an approximation of $A, \vec{b}$ (mostly due to our use of low precision in the table-lookup-based implementation of inversion and logarithm). Hence we must use a very numerically-stable method for solving this linear system in order to get a meaningful result, and such methods are expensive.

One solution (which is what we implemented) is to simply send $A, \vec{b}$ back to the client, along with the indexes $j_1, \ldots, j_k$ of the significant columns. The client then decrypts and solves in the clear to find $\vec{w'}$ and therefore $\vec{w}$. The drawback of this solution, of course, is that it leaks to the client more information than just the solution vector $\vec{w'}$. In Section 6.6 we describe a solution that prevents this extra leakage, leaking to the client only as much information as contained in $\vec{w'}$, without having to implement homomorphically expensive linear-system solvers. However we did not get around to implementing this solution in our program. (We remark that implementing it would not have added significantly to the running time.)

6.3.2.9 Bootstrapping considerations

As it turns out, most of the runtime of our program is spent in the recryption operations, between 66% and 75%. We must therefore be frugal with these operations. Some things that we did to save on them include:

- *Fully packed recryption.* HElib can bootstrap *fully packed* ciphertexts, i.e., ones that encode $\phi(m)$ coefficients in one ciphertext. The ciphertexts that we manipulate in our procedure, however, are seldom fully packed. Hence, whenever we need to perform recryption, we first pack as much data as we can in a single ciphertext, then bootstrap that ciphertext, and unpack the data back to the ciphertexts where it came from.

- *Strategic recryption.* Instead of performing recryption only at the last minute, we check the level of our ciphertexts before every big step in our program. For example, before we begin to add two numbers in binary representation, we check all the bit encryptions to ensure that we could complete the operation without needing to recrypt. If any of the input bits is at a low enough level, we pack all the input bits as above and recrypt them all. Then we unpack and perform the entire big step without any further recryption operations. This way we ensure that we never need to recrypt temporary variables that are used in internal computations, only the real data which is being manipulated.

6.4 Using Table Lookup to Compute Arbitrary Functions

Unfortunately, due to the work of this chapter being done in collaboration with IBM, the source code of the general implementation described previously is not available due to copyright. However, the low-level functions that were written, such as the table lookup of this section and the binary operations in section 6.5 were directly contributed to the open source library HElib [48]. Wherever possible we will provide a reference to the implementation of each procedure, providing the name of both the source and header files which can be found here [48].

As explained in section 6.3, we used a solution based on table lookup to implement a low-precision approximation of arbitrary functions. Namely, for a function f that we need to compute, we pre-compute in the clear a table T_f such that $T_f[x] = f(x)$ for every x in some range. Then given the encryptions of the (bits of) x, we perform homomorphic table lookup to get the (bits of) the value $T_f[x]$.

Building the table. Importantly, implementing a function using table lookup relies on fixed-point arithmetic. Namely the input and output must be encoded with a fixed precision and fixed scaling. In our implementation, we have three fixed-point parameters, precision p, scale s, and a Boolean flag ν that indicates if the numbers are to be interpreted as unsigned ($\nu = \mathsf{false}$) or as signed in 2's complement ($\nu = \mathsf{true}$). Given the parameters (p, s, ν), a p-bit string $(x_{p-1} \dots x_1 x_0)$ is interpreted as the rational number

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

In our implementation we have two such sets of parameters, (p, s, ν) for the input (i.e., indexes into T), and (p', s', ν') for the output (i.e., values in T). With these parameters, the table will have 2^p entries, each big enough to hold a $2^{p'}$-bit number. In our implementation we pack all the bits of the output *in one plaintext slot*, so we can only accommodate tables with output precision up to the size of the slots.

Preparing the table T_f with parameters $(p, s, \nu, p', s', \nu')$ for a function $f(\cdot)$, each entry in the table consists of a native plaintext element (i.e, an element in $\mathbb{Z}[X]/(\Phi_m(X), p^r)$, in our case $m = 2^{15} - 1$, $p^r = 2^{11}$). For every index $i \in [2^{p_{in}}]$, we put in $T_f[i]$, and element that has in *every plaintext slot* the bits of the integer z_i such that

$$R_{p',s',\nu'}(\mathsf{bin}(z_i)) = \left\lceil f\left(R_{p,s,\nu}(\mathsf{bin}(i))\right)\right\rfloor_{p',s'}$$

where $\lceil x \rfloor_{p',s'}$ rounds the real value x to the nearest point in the set $2^{-s'} \cdot [2^{p'}]$.

The implementation for building a lookup table was integrated into the HElib library [48]. The Application Programming Interface (API) of the procedure `buildLookupTable` can be found in `tableLookup.h` and the corresponding implementation is located in `tableLookup.cpp`.

Saturated arithmetic. When building the table, we need to handle cases where the function value is not defined at some point, or is too large to encode in p' bits. In these cases, the number that we store in the table will be either the largest or smallest number (as appropriate) that can be represented with the given parameters p', s', ν'. (For example, in the table for $f(x) = 1/x$, the entry $T_{1/x}[0]$ will have the MAXINT value $2^{p'} - 1$ encoded in all the slots.)

Computing all subset-products. The main subroutine in homomorphic table lookup is a procedure that computes all the subset products of a vector of bits. The input is an array of p encrypted bits $\sigma_{p-1}, \ldots, \sigma_1, \sigma_0$, and the output is a vector of 2^p bits ρ_m of all the subset products of the σ_i's the their negation, i.e.,

$$\begin{array}{rccccc}
\rho_0 := & (1-\sigma_i)\cdot & \ldots & \cdot(1-\sigma_{p-1}) & \cdot\,(1-\sigma_p) \\
\rho_1 := & (1-\sigma_i)\cdot & \ldots & \cdot(1-\sigma_{p-1}) & \cdot\quad \sigma_p \\
\rho_2 := & (1-\sigma_0)\cdot & \ldots & \cdot\quad \sigma_{p-1} & \cdot\,(1-\sigma_p) \\
\rho_3 := & (1-\sigma_0)\cdot & \ldots & \cdot\quad \sigma_{p-1} & \cdot\quad \sigma_p \\
\vdots & & & & \\
\rho_{2^p-1} := & \sigma_0 \quad\cdot & \ldots & \cdot\quad \sigma_{p-1} & \cdot\quad \sigma_p
\end{array}$$

Namely, for any $m \in [2^p]$, the bit ρ_m is set to $\rho_m := \prod_{m_j=1} \sigma_j \cdot \prod_{m_j=0}(1-\sigma_j)$.

To compute all these products ρ_m we use a "product tree" that on one hand ensure that the multiplication depth remains as low as possible (namely $\lceil \log_2 p \rceil$), and on the other hand tries to use as few multiplication operations as possible. For p power of two, this can be done recursively as follows:

```
ComputeAllProducts(input: $\sigma_{p-1},\dots,\sigma_0$, output: $\rho_{2^p-1},\dots,\rho_0$)
1. if $p=1$ return $\rho_0 := 1-\sigma_0$, $\rho_1 := \sigma_0$
2. else
2.     ComputeAllProducts(in: $\sigma_{p/2-1},\dots,\sigma_0$, out: $\rho'_{2^{p/2}-1},\dots,\rho'_0$)
4.     ComputeAllProducts(in: $\sigma_{p-1},\dots,\sigma_{p/2}$, out: $\rho''_{2^{p/2}-1},\dots,\rho''_0$)
5.     for $i,j$ in $0,\dots,2^{p/2}$, set $\rho_{2^{p/2}j+i} := \rho''_j \cdot \rho'_i$.
```

Essentially the same procedure applies when p is not a power of two, except that it is better to split the array so that the first part is of size power of two (i.e., size 2^ℓ for $\ell = \lceil \log_2 p \rceil - 1$) and the second part is whatever is left.

The implementation of `computeAllProducts` was integrated into the HElib library [48]. The API and corresponding implementation can be found in `tableLookup.h` and `tableLookup.cpp` respectively.

We comment that this procedure is not quite optimal in terms of the number of multiplications that it uses, but it is not too bad. Specifically the number of multiplications that it uses to compute the 2^p products is only $N(p) = 2^p + 2N(p/2) = 2^p + 2^{p/2+1} + 2^{p/4+2} + \cdots$. One optimization that we have in our program is that we stop the recursion at $p = 2$ rather than $p = 1$, and compute the four output bits using just a single multiplication (rather than four). Namely we set $\rho_3 = \sigma_1\sigma_0$, $\rho_2 = \sigma_1 - \sigma_1\sigma_0$, $\rho_1 = \sigma_0 - \sigma_1\sigma_0$, and $\rho_0 = 1 + \sigma_1\sigma_0 - \sigma_1 - \sigma_0$.

This optimization can in principle be extended to higher values of p, but it gets more complicated. The idea is that the ρ_m's can be computed in terms of the "real subset products" $\tau_m = \prod_{m_j=1} \sigma_j$. The τ_m's can be computed using a recursive formula similar to the one above, except that in the last line if $i = 0$ or $j = 0$ we do not need to multiply. (For $i = 0$ we set $\tau_{2^{p/2}j} := \tau''_j$ and for $j = 0$ we set $\tau_i := \tau'_i$.) Hence the number of products is reduced to $N'(p) = (2^{p/2} - 1)^2 + 2N'(p/2) = 2^p - p - 1$. The problem with this procedure is that recovering the ρ_m's from the τ_m's seems complicated (and the savings are not that large), so we did not attempt to implement it.

Homomorphic table lookup. Once we have a table $T[0,\dots,2^p-1]$ and an implementation of the subset-product procedure above, implementing homomorphic lookups into the table with encrypted p-bit indexes requires just a simple MUX. Namely, we are given p ciphertexts, encrypting the bits σ_i of an index into T. We apply the subset-product procedure above to get all the products ρ_m, then return $\sum_m T[m] \boxdot \rho_m$.

Note that the input ciphertext could be packed, with a different bit $\sigma_{i,j}$ in each slot j of ciphertext i. In this case our lookup procedure would return a SIMD table lookup: the coefficients of the j'th slot of the output will store the bits of $T[x_j]$, where $x_j = \sum_i 2^i \sigma_{i,j}$.

We also remark that it is possible to implement different tables in different slots, so the j'th output slot will contain $T_j[x_j]$ instead of all using the same table T. This will require only a minor change to our procedure for building the table (and no change to the homomorphic lookup procedure), but we have not yet implemented this variant.

An implementation of the table lookup procedure, `tableLookup`, can be found in the HElib library [48]. The API and corresponding implementation of this procedure can be found in `tableLookup.h` and `tableLookup.cpp` respectively.

6.5 Binary Arithmetic and Comparisons

Much of our logistic-regression procedure manipulates the various variables in their binary representation. To implement these manipulations, we rely on procedures that implement various common low-level operations, such as arithmetic and comparisons in binary representation. In this section we describe our implementation of these low-level operations, which has been integrated into the HElib library [48].

6.5.1 Adding Two Integers

One basic operation that we need is adding two integers in binary representation. The input consists of two sequences of ciphertexts, $(a_{t-1}, \ldots, a_1, a_0)$ and $(b_{t-1}, \ldots, b_1, b_0)$, encrypting the bits of two integers a, b, respectively (using padding, we can assume w.l.o.g. that the two integers have the same bit size).[4] The output is the sequence of ciphertexts $(s_{t+1}, \ldots, s_1, s_0)$, encrypting the bits of the sum $s = a + b$. Of course the hard part is to compute the carry bits in the addition, which we do as follows:

- For $i = 0, ..., t-1$ we compute "generate carry" and "propagate carry" bits, $g_i := a_i b_i$ and $p_i := a_i + b_i$. (Note that at most one of p_i, g_i can be 1.)
- We extend the generate and propagate bits to intervals, where for any $i \leq j$ we have $p_{[i,j]} = \prod_{k=i}^{j} p_k$ and $g_{[i,j]} = g_i \cdot \prod_{k=i+1}^{j} p_k$.
- The carry bit out of position j is $c_j := \sum_{i=0}^{j} g_{[i,j]}$, and the result bits are $s_i := a_i + b_i + c_{i-1}$ for $i = 0, \ldots, t$.

To get all the carry bits c_i, we therefore need to compute all the interval products $g_{[i,j]}$ for all $[i, j] \subseteq [0, t-1]$, which we do using a dynamic-programming approach. Namely we compute for all the intervals of size two, then all intervals of size up to four, etc.

To achieve this we use a directed acyclic graph (DAG) which is a finite, directed graph made of finite vertices and directed edges. Additionally if one starts at any vertex v and follows a sequence of edges which are consistently directed then it is not possible to return to vertex v. In more detail, given the inputs a_i, b_i we build an *addition DAG* (Directed Acyclic Graph) that encodes our plan for what ciphertexts to multiply in what order. This is done to ensure that we consume the smallest number of levels, and use as few multiplications as we can. Note that the input ciphertexts need not be all at the same level, and the plan may vary depending on the input levels.

[4] We can have different bits in different plaintext slots of these ciphertexts, so each slot could represent a different integer and the addition will be applied to all of them.

The DAG has two nodes for every interval $[j, i] \subseteq [0, t-1]$, representing $p_{[i,j]}$ and $g_{[i,j]}$, and each node has two parents which are the nodes that should be multiplied to form the variable of this node. We initialize the nodes in the DAG in the following order:

- First we initialize all the singleton nodes $p_{[i,i]}$, the parents are set to a_i, b_i and the level is set to $\min(\mathsf{lvl}(a_i), \mathsf{lvl}(b_i))$.
- Next we initialize all the other nodes $p_{[i,j]}$ in order of increasing interval size. To initialize $p_{[i,j+1]}$, we compute

  $$k = \arg\max_{k \in [i,j]} \left\{ \min\left(\mathsf{lvl}(p_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) \right\}$$

  (breaking ties as described later in this section). The parents of $p_{[i,j+1]}$ are set to $p_{[i,k]}$ and $p_{[k+1,j+1]}$, and its level to $\min\left(\mathsf{lvl}(p_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) - 1$.
- Next we initialize all the singleton nodes $g_{[i,i]}$, the parents are set to a_i, b_i and the level is set to $\min(\mathsf{lvl}(a_i), \mathsf{lvl}(b_i)) - 1$.
- Finally we initialize all the other nodes $g_{[i,j]}$ in order of increasing interval size. To initialize $p_{[i,j+1]}$, we compute

  $$k = \arg\max_{k \in [i,j]} \left\{ \min\left(\mathsf{lvl}(g_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) \right\}$$

  (breaking ties as described later in this section). The parents of $p_{[i,j+1]}$ are set to $g_{[i,k]}$ and $p_{[k+1,j+1]}$, and its level to $\min\left(\mathsf{lvl}(g_{[i,k]}), \mathsf{lvl}(p_{[k+1,j+1]})\right) - 1$.

This procedure ensures that each node ends up at the highest possible level (i.e. the lowest possible multiplication depth), for the given levels of the inputs a_i, b_i. When all the input bits a_i, b_i are at the same level, then the depth is $\lceil \log_2(t+2) \rceil$, since the largest term that we need to compute is the $(t+1)$-product $g_{[0,t-1]} = a_0 b_0 \cdot \prod_{i=1}^{t-1}(a_i + b_i + 1)$.

We note, however, that not all the nodes in the DAG must be computed: Only the nodes $g_{[i,j]}$ are used in the carry calculation, and not every $p_{[i,j]}$ is necessarily an ancestor of some $g_{[i',j']}$. We can hope that by breaking ties in a clever way when computing argmax above, we can minimize the number of nodes $p_{[i,j]}$ that need to be computed, hence reducing the number of multiplications that must be performed. In our implementation, we break ties heuristically by choosing among the highest-level k's the nodes that already have the largest number of children.

The homomorphic addition procedure. Given the input ciphertexts a_i and b_i, we build a DAG as above, and check that the lowest-level node in this DAG is still at a level above zero. If not, then we attempt to recrypt all the input ciphertexts, then re-build the DAG with the new input levels. Once we have a valid DAG, we compute all the $g_{[i,j]}$ nodes and from then add them as needed to compute all the carry bits, and then compute the result bits.

While computing the $g_{[i,j]}$'s, we try to compute the nodes in the DAG lazily, computing each node only when it is needed (either directly for one of the carry bits or indirectly for one of its children), and keeping the intermediate node ciphertexts around only as long as they are still needed. (I.e., as long as they still have some descendants that were not yet computed.) We also use parallelism when we can, computing different nodes using different threads (if we have them).

The implementation of this procedure, called `addTwoNumbers`, has been integrated into HElib [48]. The Application Programming Interface (API) of the procedure can be found in `binaryArith.h` along with the implementation in `binaryArith.cpp`.

6.5.2 Adding Many Integers

When we need to add many integers (all in binary representation), we use the three-for-two method (cf. [53]) to reduce their number: until we only have two integers left, then use the routine from above to add the remaining two numbers.

The three-for-two procedure. Given three integers in binary representation, $(u_{t-1}, \ldots, u_0)$, $(v_{t-1} \ldots v_0)$, $(w_{t-1} \ldots w_0)$, we can add the three bits in each position $u_i + v_i + w_i$

(over the integers), yielding a number between zero and three that can be represented in two bits. Namely $u_i + v_i + w_i = x_i + 2y_i$, where $x_i = u_i + v_i + w_i \pmod 2$ and $y_i = u_i v_i + u_i w_i + v_i w_i \pmod 2$. Adding every triple of bits u_i, v_i, w_i in this manner, we get the two integers $x = (x_{t-1} \dots x_0)$, and $y = (y_t \dots y_0 0)$, such that $x + y = u + v + w$ over the integers.

Computing the bits of x involves only additions, and each y_i can be computed using two multiplications and two additions, namely $y_i := u_i v_i + (u_i + v_i) w_i$. Hence x is at the same level as the input numbers u, v, w, and y is one level lower. (Note also that all the x_i's and y_i's can be computed in parallel.)

The add-many-numbers procedure. Given many integers in binary, we apply the three-for-two procedure to them in a tree manner, namely we partition them into groups of three, apply the three-for-two to each group separately, then collect all the resulting pairs (plus whatever leftover numbers were not part of any group) into one list, and repeat the process until only two integers are left. This yields multiplication depth $d \approx \log_{3/2}(n)$ to reduce n numbers into two, while adding at most d to the bitsize of the input integers. Once we have only two integers left, we apply the addition routine from above.

This procedure was implemented and added to the HElib library [48]. The API is located in `binaryArith.h` under `addManyNumbers` and the corresponding implementation can be found in `binaryArith.cpp`.

6.5.3 Integer Multiplication

Given two integers in binary to multiply $a = (a_{t-1} \dots a_0)$ and $b = (b_{t'-1} \dots b_0)$, we first compute all the pairwise products $b_i a_j$, and then use the add-many-numbers procedure from above to add the t' integers $2^i b_i \cdot a$. For example when multiplying a 3-bit b by a

4-bit a, we add the numbers

$$\begin{array}{rcccccc}
b_0 \cdot a = & & & b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 \\
2b_1 \cdot a = & & b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 \\
4b_2 \cdot a = & b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0
\end{array}$$

When both numbers are unsigned, we always choose $t' \leq t$, namely we let the longer integer be a and the shorter one be b.

Dealing with negative numbers. In our implementation we also implemented a multiplication of a 2s-complement number a by an unsigned number b. In that case we always use the 2s-complement number as a and the unsigned number as b, and we modify the procedure above by computing the sign extension of all the numbers $b_i \cdot a$, namely replicating the top bit in each number all the way to the largest bit position. For example, if we have a 2-bit 2s-complement number $(a_1\ a_0)$ and a three-bit unsigned number $(b_2\ b_1\ b_0)$, then we compute and add the three integers (considered as 2s-complement numbers):

$$\begin{array}{rcccc}
b_0 \cdot a = & b_0a_1 & b_0a_1 & b_0a_1 & b_0a_0 \\
2b_1 \cdot a = & b_1a_1 & b_1a_1 & b_1a_0 & 0 \\
4b_2 \cdot a = & b_2a_1 & b_2a_0 & 0 & 0\ .
\end{array}$$

An implementation of this procedure was integrated into the HElib library [48] under the name `multTwoNumbers`. The implementation and API can be found in `binaryArith.cpp` and `binaryArith.h` respectively. It is noted that we did not implement a 2s-complement by 2s-complement multiplication, since we did not need it for the current project.

6.5.4 Comparing Two Integers

The procedure for integer comparison is somewhat similar to integer addition. We have two integers in binary, $a = (a_{t-1}, \ldots, a_1, a_0)$ and $b = (b_{t-1}, \ldots, b_1, b_0)$, and we want to compute the two integers $x = \max(a, b) = (x_{t-1} \ldots x_0)$ and $y = \min(a, b) = (y_{t-1} \ldots y_0)$, as well as the two indicator bits $\mu = (a > b)$ and $\nu = (b > a)$ (note that when $a = b$, both μ, ν are zero).

We begin by computing for every $i < t$ the bits $e_i := a_i + b_i + 1$ (which is 1 iff $a_i = b_i$) and $g_i := a_i + a_i b_i$ (one iff $a_i > b_i$). We then compute the products $e_i^* = \prod_{j \geq i} e_j$ and $g_i^* = g_i \cdot \prod_{j > i} e_j$, and the bits $\tilde{g}_i = \sum_{j \geq i} g_1^*$ (one iff $a_{t-1 \ldots i} > b_{t-1 \ldots i}$). Computing the products e_i^*, g_i^* is done using a recursive procedure somewhat similar to ComputeAllProducts from Section 6.4. Finally we compute the results by setting $\mu := \tilde{g}_0$, $\nu := 1 + \tilde{g}_0 + e_0^*$, and for $i = 0, \ldots, t-1$ we set $x_i := (a_i + b_i)\tilde{g}_i + b_i$ and $y_i := x_i + a_i + b_i$.

Note that we use all the g_i^*'s but only e_0^* for computing the output results, hence we somewhat optimized our procedure for computing these products by skipping the computation of e_i^*'s that are never used.

We remark that the last product $(a_i + b_i)\tilde{g}_i$ means that our procedure may use depth one more than the minimum possible. Using the absolutely smallest possible depth is challenging, straightforward solutions would take $O(t^2)$ multiplications (vs. $O(t)$ multiplications in the procedure above). While getting minimal depth with $O(t)$ multiplications is possible in theory, the procedure for doing this is overly complex (and extremely hard to parallelize), so we opted for a simpler procedure with slightly non-optimal depth. (Also, as opposed to the addition procedure from above, the simple procedure that we implemented here does not vary depending on the level of the input ciphertexts for a_i, b_i.)

The implementation of this procedure, `compareTwoNumbers`, was integrated into the HElib library [48]. The API and corresponding implementation of this procedure can be found in `binaryCompare.h` and `binaryCompare.cpp` respectively.

6.5.5 Accumulating the bits in a ciphertext

As described in Section 6.3.2, when computing the category counters we at some point have 64 ciphertexts, with ciphertext C_m encrypting in each slot i the indicator bits $\chi_{i,m}$, and we want to sum-up these indicator bits (over the integers) and compute the

64 counters $P_m \sum_i \chi_{i,m}$. While this is theoretically just an instance of adding many numbers (these numbers being the bits $\chi_{i,m}$), there are two properties of this instance that require special optimization:

- The input bits to be added are not aligned in the same slots of different ciphertexts, but rather spread across the different slots of the same ciphertext.
- We need to perform this add-many-numbers procedure on 64 different lists in parallel, so we have an opportunity to use SIMD operations.

We therefore implemented a special-purpose shift-and-add procedure to do the accumulation, combining the addition operations that we need to make with the "matrix transpose" that transforms the input counter-per-ciphertext with different bits across the slots into a ciphertext-per-bit-position in the output with the different counters across the slots.

At every step in this procedure we keep a current list of (encrypted) arrays of integers in binary representation. Each array in the list is represented by a vector of ciphertexts $(c_0, c_1, \ldots)$, one per bit position, and the integers in the array are the different slots of $\sum_i 2^i \cdot c_i$. Initially the list consists of 64 arrays, each array corresponding to the different slots of one of the input ciphertexts, and the integers all bits (so each array is represented by length-1 vector). As the computation progresses, the integers represent partial sums (so their bitsize is getting larger), correspondingly the arrays have fewer integers in them (since the number of partial sums is getting smaller), and also the number of arrays get smaller (as we pack more counters across the slots).

In each step we perform partial addition, adding each group of r partial sums into a single larger sum. We first apply r rotations to all the ciphertexts representing all the arrays, so as to align the numbers that we need to add. These rotations mean that we are using fewer slots to hold these partial sums (since we only use one of each r slots as the "pivot" where addition is to take place). So we can pack some number $p \leq r$ of these sums and apply the add-many-numbers procedure to all of them in a SIMD manner.

This cuts the number of arrays by a p factor, and change the size of arrays by a p/r factor. (Each array is cut by a factor of r because we add r partial sums into one, but increased by a factor of p as we pack multiple arrays into one.)

The procedure that we actually implemented is slightly different, in that in the first few steps we consider the different bit positions as different arrays (so we always work with bits rather than larger integers) and just remember for each array the power of two that it should be multiplied by. Only after we complete the "transpose" part of this transformation and have just one slot per counter, we add together all the relevant integers (shifted as needed to account for the powers of two). Specifically, we begin with 64 arrays, each containing 1800 single-bit integers. Then we perform these steps: of four steps:

1. In the first step we group $r_1 = 15$ bits together for addition (yielding 4-bit numbers), and pack $p_1 = 14$ arrays together. This yields $4 \cdot \lceil 64/14 \rceil = 20$ new arrays (of bits), and the data for each category counter is spread across $1800/15 = 120$ slots.

2. In the second step we group $r_2 = 12$ bits together for addition (again melding 4-bit numbers), and pack $p_2 = 5$ them together. This yields $4 \cdot \lceil 20/5 \rceil = 16$ arrays of bits, and the data for each category counter is spread across only $120/12 = 10$ slots.

3. In the this step we group $r_2 = 10$ bits together for addition (again yielding 4-bit numbers), and no further packing is needed. This yields again $4 \cdot 16 = 64$ arrays of bits, but now the data for each category counter is all in just one slot position.

4. We note that our current 64 ciphertexts encrypt shifted bits, i.e., bits that should be multiplied by some powers of two. No shift amount corresponds to more than twelve ciphertexts, so we can re-arrange these 64 bits in just 12 integers. Then we call our add-many-numbers procedure to add these 12 integers thereby completing the accumulation of the category counters.

The specific choices of $r_1 = 15, p_1 = 14$ and $r_2 = 12, p_1 = 5$ were made so that the shift operations involved in aligning numbers before addition could be implemented with 1D rotation operations. These operations map directly to automorphisms in the underlying cryptosystem, rather than the more expensive general-purpose shifts.

6.6 Solving a Linear System

The last thing that our solution needs to do, after setting the linear system $A\vec{w'} = \vec{b}$, is to solve it and output the solution vector $\vec{w'}$. But solving a linear system homomorphically is complex, even if it is only a 6-by-6 system as in our application. Moreover, the linear system that we computed was just an approximation (due mostly to the low-precision inherent in our table-based approach to computing inversion and logarithms). Hence we must ensure that our homomorphic solver is numerically stable, making it harder still.

Instead, in our program we opted for simply sending A and $\vec{b}$ to the client, having the client decrypt and solve in the clear. This "solution", however, leaks information about the input data beyond what is implied by the vector $\vec{w'}$. This extra leakage is perhaps acceptable in the context of our application to logistic regression on medical data, but surely there are applications where such a solution will not be acceptable. So we would like to find a feasible solution that will eliminate the extra leakage, simpler than implementing a homomorphic stable linear solver.

6.6.1 Randomized Encoding with Rational Reconstruction

An appealing approach for addressing this issue is to use *randomized encoding* (cf. [3]). Namely, consider the function that we want to compute, $f(A, \vec{b}) = A^{-1}\vec{b}$, as a function over the rational numbers with bounded integer inputs. We would like to apply a randomized transformation to the input $u := \mathsf{enc}(A, \vec{b}; R)$, such that (i) it is possible to "decode" $A^{-1}\vec{b}$ from u; (ii) u does not yield any more information[5] on $A, \vec{b}$ than what is

[5] This property is formulated by requiring a simulator that can only see $\vec{w'}$ and can output the same distribution as $\mathsf{enc}(A, \vec{b}; R)$

implied by $\vec{w'} = A^{-1}\vec{b}$; and (iii) computing $\mathsf{enc}(\cdot)$ is substantially easier than computing $f(\cdot)$ itself.

If we had such randomized encoding, we could choose the randomness R and evaluate homomorphically $\mathsf{enc}(A, \vec{b}; R)$, send the encrypted u back to the client, who could decrypt u and decode $\vec{w'}$ from it. We note that the linear-system-solver function $f(\cdot)$ is in NC1, so theoretically we could apply generic randomized encoding solutions here. This solution yields a very low-depth encoding, but the size of the encoding is exponential in the depth of the circuit for f, so we do not expect them to be practical.

Below we describe a randomized encoding for the linear-system-solver function f, that uses only integer addition and multiplication. This encoding can therefore be implemented using the binary arithmetic routines that we described in Section 6.5. However we did not implement that idea in our solution, we expect it to be doable but it will add a significant overhead (see below).

Our first observation is that if we wanted to compute the linear-system-solver function modulo some prime q then it would be easy to randomize (when A is invertible): All we need is to choose a random invertible $R \in \mathbb{Z}_q^{n\times n}$ and set $A^* := RA \bmod q$ and $\vec{b^*} := R\vec{b} \bmod q$. On one hand A^* is just a random invertible function modulo q, and on the other hand $(A^*)^{-1}\vec{b^*} = A^{-1}\vec{b} \pmod{q}$.

However, in our case we want to find the solution over the rational numbers, not modulo some q. Our second observation is that we can apply here the tool of *rational reconstruction* (cf. [73, Ch 4.6]). Recall that rational reconstruction is an efficient procedure (denoted below by $\mathsf{RationalRec}(\cdot)$), such that

$$\forall a, b, q \in \mathbb{Z} \text{ s.t. } |a| \cdot |b| < q/2, \quad \mathsf{RationalRec}(q,\ ab^{-1} \bmod q) = (a, b),$$

provided that ab^{-1} is defined modulo q. In other words, the procedure gets as input a modulus q and an element $z \in \mathbb{Z}_q$, and it is guaranteed to output the unique solution

(a, b) to $az = b \pmod q$ satisfying $|a| \cdot |b| < q/2$, if such a solution exists.

In our application we are given $A, \vec{b}$ with some precision p, and the rational solution that we seek is

$$\vec{w}' = A^{-1}\vec{b} = \mathsf{adj}(A)\vec{b}/\det(A).$$

Every entry of $\vec{w}'$ is of the form x/d, where $d = \det(A)$ and x is one entry in $\mathsf{adj}(A)\vec{b}$. Since all the entries in $A, \vec{B}$ are smaller than 2^p, then d and all the entries of $\mathsf{adj}(A)\vec{b}$ are smaller in magnitude than $(\sqrt{n}2^p)^n$. If we choose $q > 2(\sqrt{n}2^p)^{2n} = 2^{2np+1}n^n$, then given the solution $A^{-1}\vec{b} \bmod q$ with entries of the form $xd^{-1} \in \mathbb{Z}_q$, we could use rational reconstruction to get the rational numbers x/d. We could therefore get our randomized encoding by randomizing $A, \vec{b}$ modulo this large q.

But this solution is still not good enough, randomizing mod q implies in particular that we need to implement a homomorphic mod-q operation, which is expensive (even when q is in the clear). Our next observation is that we can replace the reduction mod-q by adding a large enough multiple of q. Recall that for any fixed integer x, if we choose a random s from a large enough domain (relative to $|x|/q$) then the random variable $x + qs$ depends only on the value of $x \bmod q$, and is essentially independent of x div q. Specifically, if we have a bound $|x|/q < B$ and we choose s at random from $[B \cdot 2^k]$, then the result is almost independent of $x \div q$, up to statistical distance of at most 2^k.

Hence instead of computing homomorphically the reduced matrix $RA \bmod q$, we note that each entry in RA is smaller than $n2^pq$ in absolute value. We can therefore choose a random integer matrix $S \in [n2^{p+k}]$ and compute $RA+qS$ over the integers (and similarly for $\vec{b}$).

This solution is almost plausible, but it requires integer arithmetic with very large numbers, even if n and p are small. In our application we have $n = 6$ and $p = 7$, so $q = 6^6 \cdot 2^{2 \cdot 6 \cdot 7+1} \approx 2^{100}$. And even choosing a measly statistical parameter $k = 10$, the entries of $RA + qS$ would be integers with about 120 bits.

```
PARAMETERS: dimension $n$, precision $p$, security parameter $k$.
            Let $m := \max(n, \lceil \frac{2np+1}{k} \rceil)$

REncSolver (input: Invertible matrix $A \in \mathbb{Z}_{2^p}^{n \times n}$, vector $\vec{b} \in \mathbb{Z}_{2^p}^n$):
1. Choose $m$ random primes $q_1, \ldots, q_m$ of bitsize $n2^k$.
2. For each $q_i$ do
3.     Choose a random invertible matrix $R \in \mathbb{Z}_{q_1}^{n \times n}$
4.     Choose at random a matrix $S_i \in \mathbb{Z}_{n2^{p+k}}^{n \times n}$ and a vector $\vec{t_i} \in \mathbb{Z}_{n2^{p+k}}^n$
5.     Compute $A_i^* := R_i \times A' + qS_i$ and $\vec{b_i}^* := R_i \times \vec{b'} + q\vec{t_i}$.
6. Output $\{(q_i, A_i^*, \vec{b^*}_i)\}_{i=1}^m$

DecSolver(input: $\{(q_i, A_i^*, \vec{b^*}_i)\}_{i=1}^m$):
7. Solve $\vec{w_i^*} = (A_i^*)^{-1}\vec{b_i^*} \pmod{q}_i$ for all $i$
8. Use Chinese remaindering to recover $\vec{w^*}$ modulo $q = \prod_i q_i$
9. Apply rational reconstruction and output $\vec{w} :=$ RationalRec$(q, \vec{w^*})$.
```

Figure 6.1: Randomized encoding for the rational linear-system solver, $f(A, \vec{b}) = A^{-1}\vec{b}$.

Our final observation, therefore, is that we could express R and S relative to an appropriate CRT basis, thereby replacing each big-integer operation by a moderate number of operations on much smaller integers. Specifically, we choose many "smallish" primes q_i, for each one choose a random $R_i \in \mathbb{Z}_{q_i}^{n \times n}$ and $S_i \in [n2^{p+k}]^{n \times n}$ and compute homomorphically $A_i^* = R_i A + q_i S_i$ (and similarly for $\vec{b}$). As above, the client who decrypts the A_i^*'s only get $R_i A \bmod q_i$, then compute $RA \bmod q$ (where $q = \prod_{q_i}$ and $R \equiv R_q \pmod{q_i}$) and proceed as before.

One drawback with this approach is that we may end up choosing a modulus q_i that divides $\det(A)$, in which case $R_i A \bmod q_i$ will reveal mod-q_i linear correlations between the columns of A. We therefore must choose the q_i's of size slightly larger than 2^k, to ensure the same level of protection against this threat as we get against leakage from the A_i^*'s. Setting $q_i \approx n2^k$ would mean that we only need to work with integers of total size about $n^2 2^{p+2k}$. In our setting with $n = 6, p = 7$ we can choose the (admittedly weak) $k = 15$ and work with 42-bit numbers, which is expensive but doable. (With 18-bit q_i's, we would need six of them to reach the size of q that we need.) Our randomized encoding procedure is described in Figure 6.1. From the discussion above we have:

Claim 1. The function REncSolver from Figure 6.1 is a randomized encoding for the function $f(A, \vec{b}) = A^{-1}\vec{b}$ over the rational numbers, where A is invertible and $A, \vec{b}$ are bounded. □

6.6.2 Are We Still Leaking Too Much?

We end this section by pointing out that the above solution is a randomized encoding for the *exact solution function vector* $A^{-1}\vec{b}$, which by itself may already leak information. In particular it usually reveals the determinant $\det(A)$ (or a factor of it), just by taking the common denominator of all the entries in the solution vector. In the current application, what we really want to compute is the *limited precision solution* function, that rounds the exact solution to some given precision. (We can even tolerate small error in the solution.) We do not know of any feasible randomized encoding for this limited precision function.

6.7 Implementation and Performance

All of our testing was done on an Intel Xeon E5-2698 v3 (which is a Haswell processor), with two sockets and sixteen cores per socket, running at 2.30GHz. The machine has 250GB of main memory, the compiler was GCC version 4.8.5, and we used NTL version 10.5.0 and GnuMP version 6.0.

Parameters. We worked with the cyclotomic ring $\mathbb{Z}[X]/\Phi_m(X)$ with $m = 2^{15} - 1 = 32767$ (so $\phi(m) = 27000$) The largest modulus q in the moduli-chain was about 1030-bit long, corresponding to about 80 bits of security. The plaintext space was set to $2^{11} = 2048$ (but as we explained in Section 6.3, most of the computation was done with plaintext space modulo 2). This gave as a total of 1800 plaintext slots, arranged in a $30 \times 6 \times 10$ hypercube with the first two 30×6 being "good dimensions" and the last being a "bad dimension".[6] Each plaintext slot held a degree-15 extension of the ring $\mathbb{Z}_{2^{11}}$ (or an element of $GF(2^{15})$ when we used it for mod-2 computation).

[6] The distinction between "good" and "bad" dimensions in HElib is that 1D-rotations along a good dimension take a single automorphism, while along a bad dimension it takes two automorphisms and some constant multiplies to zero out some data.

This means that we could pack as many as $15 \cdot 1800 = 27000$ integers in a single ciphertext each up to 11-bit long. But in our application we only packed in each of our ciphertext either up to 27000 bits, or up to 180 11-bit integers, depending on the phase of the computation.

Unfortunately, due to copyright reasons, access to the original code is restricted. However, the core utility procedures were integrated into the open-source library HElib [48], which have been indicated where necessary in the previous sections.

6.7.1 Timing Results

Single-threaded timing. A single-thread execution of the program took just under five hours, from key-generation and encryption of the data up to and including the computation of the matrix A and vector $\vec{v}$. Homomorphic processing took just under 280 minutes of this time, and fifteen minutes were spent packing and encrypting the raw data. The program used only about 4.5GB of RAM. Only 25% of the processing time was spent on the application logic, and about 75% (210 minutes) was spent in 65 bootstrapping operations (so under 3 minutes per operation). The timing results for the different phases of the computation are described in table 6-A:

- Computing the correlations and extracting its binary representation (`corrBinary`) took almost no time, only 16 seconds;
- About 125 minutes (45% of the processing time) was spent comparing the correlation numbers and computing the indexes of the five fields most correlated to the disease (`topIndexes`);
- Once we found the indexes, it took 33 minutes (12%) to extract the actual data corresponding to these fields (`extractCols`);
- Then it took 47 minutes (17%) to compute the 64 bucket counters and their binary expansion (`bucketCounters`);

# threads:	1	2	4	8	16	30
corrBinary	.25	.18	.13	.13	.1	.1
topIndexes	125	72	42	35	24	24
extractCols	33	20	13	11	8.5	8.7
bucketCounters	47	28	18	16	12	12
compV&Y	60	35	20	16	10	10
compA&b	12	7.7	5.3	5	3.8	3.8
total	**278**	**163**	**99**	**83**	**59**	**59**
recrypt	210	119	70	56	38	38

Table 6-A: Timing results (minutes) of different phases of the logistic-regression program

- Computing the vectors $\vec{v}$ and $\vec{y}$ using table lookup operations (`compV&Y`) took another hour (22%);

- Finally, computing the matrix A and vector $\vec{b}$ from the vectors $\vec{v}, \vec{y}$ (`compA&b`) took only 12 minutes (4%).

Multi-threaded timing. Multi-threading was very effective in reducing the computation time up to eight threads, but using more threads did not help very much (and above sixteen threads the runtime leveled off completely). The processing time dropped to 83 minutes with eight threads and just under one hour with sixteen threads. The RAM consumption increased somewhat, from 4.5GB with one thread to 5.5GB with sixteen. The fraction of time spend on bootstrapping dropped slightly, from 75% with one thread to 64% with sixteen, indicating that multi-threading during bootstrapping was somewhat more effective than in other parts of the computation. The ratio between different phases of the computation did not change much when switching to multi-threaded implementation. See more details in table 6-A.

6.7.2 Is this Procedure Accurate Enough?

How good is the solution that we obtained from this procedure? As was done in the iDASH competition itself, we measured our solution using the metric of “area under curve” (AUC). The given data consisted of genomic data variables, and a target attribute representing cancer. A random model is expected to give AUC result of 0.5. One of the attributes in this data was the BRCA gene, and taking only that attribute already gives AUC result close to 0.6. On the other hand even the best plaintext-based logistic-regression model only yields AUC of about 0.7 on that dataset. Hence the game for this dataset was to get as far above 0.6 as possible.[7] We tested our solution by running it on sub-sampled data from the training dataset, and the AUC results were usually close to 0.65. This appears similar to other solutions that were submitted to the iDASH competition.

6.7.3 Timing results for the Various Components

Since our application used a large setting of parameters ($m = 2^{15-1}$) and spent most of its time bootstrapping, the performance results above do not tell the story of how the different components perform for smaller parameters or when bootstrapping is not needed. These numbers are reported in Table 6-B, with (a) reporting the number of native multiplications and circuit depth for the different operations, while in (b)-(d) we report some performance numbers in various settings.

The addition operations we tested added two n-bit numbers to get their $n+1$-bit sum, while the multiplication operations multiplied two n-bit numbers but only computed the lower n bits of the result. In the tests below we varied the security level (when processing 8-bit numbers), the number of input bits (at security level 125), and also tested the effect of multi-threading

The runtime of the operations range from a few seconds for addition and comparison in the smaller settings to about one minute for multiplication and table lookup in the larger setting. Some trends that can be seen in these numbers include the following:

- As expected, the running times of the various operations grow quasi-linearly with the cyclotomic index m (which is more or less proportional to the security level).

[7] These numbers are said to be typical for genomic data, but it probably means that this is not a good dataset on which to develop an approximation procedure. Still this is what we had, so this is what we used.

- As the input bitsize grows, the number of native multiplications (and hence the running time) is roughly quadratic for addition, linear for comparison, and roughly $n^{2.3}$ for multiplication. (For table lookup, of course the number of products grows exponentially with the bitsize, since the table itself grows exponentially with the number of bits in the index.)
- Our table lookup implementation is embarrassingly parallel, and indeed we get nearly linear speedup in the number of threads. For the other operations the speedup is less pronounced. From one to eight threads we only get more or less $3X$ speedup, and above eight threads there are almost no additional gains.

6.8 Conclusions and Discussion

In this work we investigated the question of whether full blown FHE can be used for a realistic use case. We devised a procedure to compute an approximate logistic regression model on encrypted data, and demonstrated that this can be achieved in a matter of a few hours (or even just one hour if we use multi-threading).

In the course of this work we developed many new tools for homomorphic computations. Many of these tools are general-purpose (such as binary arithmetic, table lookup, etc.), but some are specific to the current setting (e.g., specific data packing and movement schemes). Our experience in this work leads us to believe that the answer to our motivating question is “Yes, but just barely.”

We stress that *the main roadblock is not performance*: devising a logistic regression model in a matter of hours may be perfectly acceptable in many settings. (And clusters or hardware acceleration can sometimes be brought to bear as well.) The main problem was the lack of good development and support tools, developing an FHE application feels a lot like programming using only assembly language. (Indeed the reason we did not submit our work to the iDASH competition last year was because it was not debugged in time.)

(a) Number of native multiplications and circuit depth vs. bit sizes

	addition		comparison		table lookup		multiplication	
bitsize	# mults	depth	# mults	depth	# mults	depth	# mults	depth
4	12	3	17	3	18	2	14	3
8	45	4	37	4	292	3	79	6
12	96	4	58	4			205	8
16	166	5	81	5			411	10

(b) Performance (seconds) for single-threaded 8-bit operations

security	cyclotomic	addition		comparison		table lookup		multiplication	
param	m ($\phi(m)$)	time	RAM	time	RAM	time	RAM	time	RAM
70	8191 (8190)	1.8	153	1.4	142	12.8	342	2.9	180
85	11441 (10752)	3.7	277	2.9	262	28.5	568	6.2	318
210	15709 (15004)	3.8	295	3.0	275	28.5	639	6.3	343
440	32767 (27000)	9.2	610	7.2	576	68.2	1232	15.0	697

(c) Multithreaded performance for 8-bit operations $m = 15709$ (security=210)

	addition		comparison		table lookup		multiplication	
# threads	time	RAM	time	RAM	time	RAM	time	RAM
1	3.8	295	3.0	275	28.5	639	6.3	343
2	3.7	306	1.8	282	14.9	646	5.0	350
4	2.7	315	1.2	300	8.1	658	3.3	369
8	1.8	347	1.0	341	4.6	691	2.0	400
16	1.1	350	1.0	339	2.8	741	1.9	470
32	1.1	353	1.0	334	1.9	867	1.9	607

(d) Single-threaded performance for different input sizes
(encrypted at level 13, $m = 15709$, security parameter 125).

	addition		comparison		table lookup		multiplication	
bitsize	time	RAM	time	RAM	time	RAM	time	RAM
4	1.3	359	2.0	355	2.5	375	1.4	368
8	5.5	415	4.4	387	43.1	937	9.1	486
12	12.0	482	6.8	419			23.7	636
16	21.3	564	9.4	451			47.4	880

Table 6-B: Complexity measures and performance results. Time in seconds, RAM in MB.

Using FHE in real-world settings will require much more library and development support, and many more FHE toolboxes beyond the few that we implemented in this work. We believe that this is an important project, and expect to continue working along these directions.

Chapter 7

Conclusions and future work

7.1 Summary

This thesis has explored the practicality and usability of FHE schemes within a real-world environment through the study of three varying applications. A key attribute of FHE schemes is the ability to provide total security by limiting the leakage of information to either the client or the server as information can be kept encrypted throughout the process of the algorithm. The only information that is known by both parties is the algorithm that is performed, otherwise the client can be assured the server never learns any information about their data.

Firstly it was seen that FHE can be applied to an anonymous routing environment. A key component of this application is the added security gained against a global adversary that can observe the entire network. However this introduces numerous inefficiencies and reduces the practicality of the scheme

Secondly, another aspect of the practicality of FHE was explored, namely the ability to represent more than one type of number in an FHE environment. As FHE schemes are built on modulo rings it is clear to see that integer arithmetic can be performed on data in the encrypted domain. However, it is common to have data that requires a higher level of accuracy, thus a method of representing rational numbers was shown as well as the application of this FHE scheme to evaluate linear regression homomorphically.

Lastly, a notable use case for FHE schemes is performing work on confidential information such as medical data. Medical datasets can be very large and the algorithms performed on them complex. True FHE schemes are known to be inefficient which leads to most practical applications being a SWHE variant. Section 6 explored the feasibility of "deep FHE" that performed logistic regression on a set of 1600 genomic data records.

7.2 Conclusion

The key contributions of this research are exploration of three aspects of the practicality of FHE in real-world environment. One of these is a novel way of representing rational numbers within FHE which opens the scheme to be applied to numerous additional settings that require higher levels of precision without the need to increase the size of the parameters. The other major contribution was showing the practicality of performing complex functions homomorphically on real datasets.

Through these experiments it has been shown that FHE is still inefficient and may be deemed impractical in settings that require algorithms that produce results in a short time frame. However as the state-of-the-art continues to improve in addition to hardware improvements, FHE schemes will eventually become prevalent in the near future. As for the current practicality of the FHE schemes, in a setting where data security is more important than the time it takes to homomorphically compute an algorithm, then FHE can be the solution to performing such algorithms as has been shown in this thesis.

7.3 Future work

As was discussed in Section 6.8 a large problem of applying FHE to a realistic use case is the fact that FHE libraries, especially from my experience using HElib, are still in the process of maturing. A large portion of the work towards practicality is not simply building schemes from libraries but extending the capabilities of the current libraries and developing support tools for FHE first. There is a lot of scope for work in this area and as libraries continue to mature, FHE can become prevalent in the cryptographic community in the foreseeable future.

Bibliography

[1] Carlos Aguilar-Melchor et al. "XPIR: Private information retrieval for everyone". In: *Proceedings on Privacy Enhancing Technologies* 2016.2 (2016), pp. 155–174.

[2] Yoshinori AONO et al. "Privacy-Preserving Logistic Regression with Distributed Data Sources via Homomorphic Encryption". In: *IEICE Transactions on Information and Systems* E99.D.8 (2016), pp. 2079–2089. DOI: 10.1587/transinf.2015INP0020.

[3] Benny Applebaum. "Randomized Encoding of Functions". In: *Cryptography in Constant Parallel Time*. Information Security and Cryptography. Springer, 2014, pp. 19–31.

[4] Seiko Arita and Shota Nakasato. "Fully homomorphic encryption for point numbers". In: *Information Secruity and Cryptology*. Ed. by Kefei Chen, Dongdai Lin, and Moti Yung. Cham: Springer International Publishing, 2017, pp. 253–270.

[5] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant learning, the parity problem, and the statistical query model". In: *Journal of the ACM (JACM)* 50.4 (2003), pp. 506–519.

[6] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. "Evaluating 2-DNF formulas on ciphertexts". In: *Theory of Cryptography Conference*. Springer. 2005, pp. 325–341.

[7] Dan Boneh et al. "Private Database Queries Using Somewhat Homomorphic Encryption". In: *ACNS*. Vol. 7954. Lecture Notes in Computer Science. Springer, 2013, pp. 102–118.

[8] Charlotte Bonte et al. *Faster Homomorphic Function Evaluation using Non-Integral Base Encoding.* Cryptology ePrint Archive, Report 2017/333. `https://eprint.iacr.org/2017/333`. 2017.

[9] Joppe W Bos, Kristin Lauter, and Michael Naehrig. "Private predictive analysis on encrypted medical data". In: *Journal of biomedical informatics* 50 (2014), pp. 234–243.

[10] Joppe W Bos et al. "Improved security for a ring-based fully homomorphic encryption scheme". In: *Cryptography and Coding.* Ed. by Martijn Stam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 45–64.

[11] Joppe W Bos et al. "Privacy-friendly Forecasting for the Smart Grid using Homomorphic Encryption and the Group Method of Data Handling". In: *Progress in Cryptology - AFRICACRYPT 2017.* Ed. by Marc Joye and Abderrahmane Nitaj. Cham: Springer International Publishing, 2017, pp. 184–201.

[12] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: *CRYPTO.* Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 868–886. ISBN: 978-3-642-32008-8.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "Fully Homomorphic Encryption without Bootstrapping". In: *Innovations in Theoretical Computer Science (ITCS'12).* Available at `http://eprint.iacr.org/2011/277`. 2012.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *ACM Transactions on Computation Theory* 6.3 (2014), p. 13. DOI: `10.1145/2633600`.

[15] Zvika Brakerski and Vinod Vaikuntanathan. "Efficient Fully Homomorphic Encryption from (Standard) LWE". In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science.* IEEE. 2011, pp. 97–106.

[16] Zvika Brakerski and Vinod Vaikuntanathan. "Fully homomorphic encryption from ring-LWE and security for key dependent messages". In: *Annual cryptology conference.* Springer. 2011, pp. 505–524.

[17] David L Chaum. “Untraceable electronic mail, return addresses, and digital pseudonyms”. In: *Communications of the ACM* 24.2 (1981), pp. 84–90.

[18] Hao Chen, Kim Laine, and Rachel Player. “Simple Encrypted Arithmetic Library - SEAL v2.1”. In: *Financial Cryptography Workshops.* Vol. 10323. Lecture Notes in Computer Science. Springer, 2017, pp. 3–18.

[19] Hao Chen et al. *High-Precision Arithmetic in Homomorphic Encryption.* Cryptology ePrint Archive, Report 2017/809. `https://eprint.iacr.org/2017/809`. 2017.

[20] Jingwei Chen et al. “Faster Binary Arithmetic Operations on Encrypted Integers”. In: *WCSE’17, Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering.* 2017. ISBN: 978-981-11-3671-9.

[21] Jung Hee Cheon, Miran Kim, and Myungsun Kim. “Search-and-compute on encrypted data”. In: *International Conference on Financial Cryptography and Data Security.* Springer, 2015, pp. 142–159.

[22] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *ASIACRYPT (1).* Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 409–437.

[23] Jung Hee Cheon et al. “Privacy-preserving computations of predictive medical models with minimax approximation and non-adjacent form”. In: *Financial Cryptography and Data Security.* Ed. by Michael Brenner et al. Cham: Springer International Publishing, 2017, pp. 53–74.

[24] Ilaria Chillotti et al. “Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE”. In: *ASIACRYPT (1).* Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 377–408.

[25] Chongwon Cho et al. “Laconic oblivious transfer and its applications”. In: *Annual International Cryptology Conference.* Springer. 2017, pp. 33–65.

[26] Benny Chor et al. “Private information retrieval”. In: *Proceedings of IEEE 36th Annual Foundations of Computer Science.* IEEE. 1995, pp. 41–50.

[27] HeeWon Chung and Myungsun Kim. *Encoding of Rational Numbers and Their Homomorphic Computations for FHE-based Applications.* Cryptology ePrint Archive, Report 2016/344. `https://eprint.iacr.org/2016/344`. 2016.

[28] Ian Clarke et al. "Freenet: A distributed anonymous information storage and retrieval system". In: *Designing Privacy Enhancing Technologies.* Springer. 2001, pp. 46–66.

[29] Anamaria Costache et al. "Fixed-Point Arithmetic in SHE Schemes". In: *SAC.* Vol. 10532. Lecture Notes in Computer Science. Springer, 2016, pp. 401–422.

[30] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. "A secure and optimally efficient multi-authority election scheme". In: *European transactions on Telecommunications* 8.5 (1997), pp. 481–490.

[31] Jack L.H. Crawford et al. *Doing Real Work with FHE: The Case of Logistic Regression.* Cryptology ePrint Archive, Report 2018/202. `https://eprint.iacr.org/2018/202`. 2018.

[32] Marten van Dijk et al. "Fully Homomorphic Encryption over the Integers". In: *Advances in Cryptology - EUROCRYPT'10.* Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 24–43. ISBN: 978-3-642-13189-9.

[33] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router.* Tech. rep. DTIC Document, 2004.

[34] Changyu Dong and Liqun Chen. "A fast single server private information retrieval protocol with low communication cost". In: *European Symposium on Research in Computer Security.* Springer. 2014, pp. 380–399.

[35] Nathan Dowlin et al. "Manual for using homomorphic encryption for bioinformatics". In: *Proceedings of the IEEE* 105.3 (2017), pp. 552–567.

[36] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *EUROCRYPT (1).* Vol. 9056. Lecture Notes in Computer Science. Springer, 2015, pp. 617–640.

[37] Taher ElGamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[38] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption.* Cryptology ePrint Archive, Report 2012/144. `https://eprint.iacr.org/2012/144`. 2012.

[39] Craig Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, 2009.

[40] Craig Gentry, Shai Halevi, and Nigel Smart. “Homomorphic Evaluation of the AES Circuit”. In: *”Advances in Cryptology - CRYPTO 2012”.* Vol. 7417. Lecture Notes in Computer Science. Full version at `http://eprint.iacr.org/2012/099`. Springer, 2012, pp. 850–867.

[41] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *Advances in Cryptology - CRYPTO 2013, Part I.* Ed. by Ran Canetti and Juan A. Garay. Springer, 2013, pp. 75–92. DOI: `10.1007/978-3-642-40041-4_5`.

[42] Craig Gentry et al. “Private Database Access with HE-over-ORAM Architecture”. In: *ACNS.* Vol. 9092. Lecture Notes in Computer Science. Springer, 2015, pp. 172–191.

[43] Ran Gilad-Bachrach et al. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: *ICML.* Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 201–210.

[44] Shai Halevi and Victor Shoup. “Algorithms in HElib”. In: *CRYPTO (1).* Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 554–571.

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

[46] Shai Halevi and Victor Shoup. “Design and Implementation of a Homomorphic-Encryption Library”. In: (2013).

[47] Shai Halevi and Victor Shoup. *Faster Homomorphic Linear Transformations in HElib.* Cryptology ePrint Archive, Report 2018/244. `https://eprint.iacr.org/2018/244`. 2018.

[48] Shai Halevi and Victor Shoup. *HElib - An Implementation of homomorphic encryption.* `https://github.com/shaih/HElib/`. Accessed December 2017.

[49] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. "NTRU: A Ring-Based Public Key Cryptosystem". In: *ANTS.* Ed. by Joe Buhler. Vol. 1423. Lecture Notes in Computer Science. Springer, 1998, pp. 267–288. ISBN: 3-540-64657-4.

[50] *Integrating Data for Analysis, Anonymization and SHaring (iDASH).* `http://www.humangenomeprivacy.org/2017/`.

[51] Angela Jäschke and Frederik Armknecht. "Accelerating homomorphic computations on rational numbers". In: *Applied Cryptography and Network Security.* Ed. by Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider. Cham: Springer International Publishing, 2016, pp. 405–423.

[52] Aaron Johnson et al. "Users get routed: Traffic correlation on Tor by realistic adversaries". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM. 2013, pp. 337–348.

[53] Richard M. Karp and Vijaya Ramachandran. "Parallel Algorithms for Shared-memory Machines". In: *Handbook of Theoretical Computer Science (Vol. A).* Ed. by Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 869–941. ISBN: 0-444-88071-2. URL: `http://dl.acm.org/citation.cfm?id=114872.114889`.

[54] Alhassan Khedr, P. Glenn Gulak, and Vinod Vaikuntanathan. "SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers". In: *IEEE Trans. Computers* 65.9 (2016), pp. 2848–2858. DOI: `10.1109/TC.2015.2500576`. URL: `https://doi.org/10.1109/TC.2015.2500576`.

[55] Aggelos Kiayias et al. "Optimal rate private information retrieval from homomorphic encryption". In: *Proceedings on Privacy Enhancing Technologies* 2015.2 (2015), pp. 222–243.

[56] Miran Kim et al. *Secure Logistic Regression based on Homomorphic Encryption.* Cryptology ePrint Archive, Report 2018/074. `https://eprint.iacr.org/2018/074`. 2018.

[57] Tancrede Lepoint and Michael Naehrig. "A comparison of the homomorphic encryption schemes FV and YASHE". In: *International Conference on Cryptology in Africa.* Springer. 2014, pp. 318–335.

[58] Zengpeng Li, Can Xiang, and Chengyu Wang. "Oblivious transfer via lossy encryption from lattice-based cryptography". In: *Wireless Communications and Mobile Computing* 2018 (2018).

[59] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption". In: *STOC.* 2012, pp. 1219–1234.

[60] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology - EUROCRYPT'10.* Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 1–23.

[61] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Journal of the ACM (JACM)* 60.6 (2013), p. 43.

[62] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective.* Vol. 671. Springer Science & Business Media, 2012.

[63] Payman Mohassel and Yupeng Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* IEEE Computer Society, 2017, pp. 19–38. DOI: `10.1109/SP.2017.12`. URL: `https://doi.org/10.1109/SP.2017.12`.

[64] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 1999, pp. 223–238.

[65] Andreas Pfitzmann and Marit Hansen. "Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management-a consolidated proposal for terminology". In: *Version v0* 31 (2008), p. 15.

[66] Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. *PALISADE Lattice Cryptography Library User Manual (v1.2.0).* Tech. rep. 2018. URL: `https://git.njit.edu/palisade/PALISADE/blob/PALISADE-v1.2/doc/palisade_manual.pdf`.

[67] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *J. ACM* 56.6 (2009). DOI: `10.1145/1568318.1568324`.

[68] R. Rivest, L. Adleman, and M. Dertouzos. "On data banks and privacy homomorphisms". In: *Foundations of Secure Computation.* Academic Press, 1978, pp. 169–177.

[69] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[70] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[71] Peter W Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". In: *SIAM review* 41.2 (1999), pp. 303–332.

[72] Thomas Shortell and Ali Shokoufandeh. "Secure SURF with Fully Homomorphic Encryption". In: *Computing Research Repository* (2017). URL: `http://arxiv.org/abs/1707.05905`.

[73] Victor Shoup. *A computational introduction to number theory and algebra.* Cambridge University Press, 2006. ISBN: 978-0-521-85154-1.

[74] Nigel P. Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Des. Codes Cryptography* 71.1 (2014). Early verion at `http://eprint.iacr.org/2011/133`, pp. 57–81. ISSN: 0925-1022. DOI: `10.1007/s10623-012-9720-4`.

[75] Ian Stewart. *Galois Theory.* 3rd. Chapman and Hall/CRC, 2003. Chap. 16.5, pp. 171–175.

[76] Shuang Wang et al. “HEALER: homomorphic computation of ExAct Logistic rEgRession for secure rare disease variants analysis in GWAS”. In: *Bioinformatics* 32.2 (2016), pp. 211–218. DOI: `10.1093/bioinformatics/btv563`. URL: `+http://dx.doi.org/10.1093/bioinformatics/btv563`.

[77] Chen Xu et al. “Homomorphically Encrypted Arithmetic Operations Over the Integer Ring”. In: *Information Security Practice and Experience*. Ed. by Feng Bao et al. `https://ia.cr/2017/387`. Cham: Springer International Publishing, 2016, pp. 167–181.

[78] Xun Yi et al. “Single-database private information retrieval from fully homomorphic encryption”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.5 (2012), pp. 1125–1134.

[79] Bassam Zantout and Ramzi Haraty. “I2P data communication system”. In: *Proceedings of ICN*. Citeseer. 2011, pp. 401–409.