Applications of Fully Homomorphic Encryption

by

Gizem Selcan Çetin

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Electrical and Computer Engineering

by

April 2019

APPROVED:

Professor Berk Sunar,
Dissertation Advisor

Assistant Professor Andrew Clark,
Electrical and Computer Engineering

Professor William J. Martin,
Mathematical Sciences

Dr. Hao Chen,
Microsoft Research

Professor Reinhold Ludwig,
Head of Department

Abstract

Homomorphic encryption has progressed rapidly in both efficiency and versatility since its emergence in 2009. Meanwhile, a multitude of pressing privacy needs — ranging from cloud computing to healthcare management to the handling of shared databases such as those containing genomics data — call for immediate solutions that apply fully homomorpic encryption (FHE) and somewhat homomorphic encryption (SHE) technologies. Recent rapid progress in fully homomorphic encryption has catalyzed renewed efforts to develop efficient privacy preserving protocols. Several works have already appeared in the literature that provide solutions to these problems by employing leveled or somewhat homomorphic encryption techniques.

Here, we propose efficient ways of adapting the most fundamental programming problems; boolean algebra, arithmetic in binary and higher radix representation, sorting, and search to the fully homomorphic encryption domain by focusing on the multiplicative depth of the circuits alongside the more traditional metrics. The reduced depth allows much reduced noise growth and thereby makes it possible to select smaller parameter sizes in leveled FHE instantiations resulting in greater efficiency savings. We begin by exploring already existing solutions to these programming problems, and analyze them in terms of homomorphic evaluation and memory costs. Most of these algorithms appear to be not the best candidates for FHE solutions, hence we propose new methods and improvements over the existing algorithms to optimize performance.

Acknowledgements

This work was supported by the NSF awards CNS #1319130 and CNS #1561536. Throughout this dissertation, I have received a great deal of support and assistance. I would first like to thank my Ph.D. advisor Professor Berk Sunar for his support on this journey. His wisdom and guidance will always be remembered and appreciated. I would like to express my very great appreciation to my dissertation committee members Professor Andrew Clark, and Professor Reinhold Ludwig. I am grateful for their time, and valuable feedbacks. I would especially like to thank Professor William J. Martin for his enthusiasm, and unlimited moral and mathematical support.

My special thanks are extended to the Cryptography and Privacy Research group at Microsoft Research. I am indebted to Kristen Lauter for giving me an amazing opportunity to work as an intern under the SEAL homomorphic encryption project. I would like to thank Kim Laine for his supervision and mentorship. I would like to offer my special thanks to Hao Chen for all our insightful discussions during my internship and also for being a part of my dissertation committee.

The members of the Vernam Lab of WPI, you all have a special place in my heart. Gorka Irazoqui, Berk Gulmezoglu, and Koksal Mus you have not just been my colleagues or lab-mates, but my emotional support system, whenever I needed it. I would like to thank Wei Dai and Yarkin Doroz for keeping the homomorphic discussion going. I am grateful for everyone I had crossed paths with at WPI and in Worcester, MA. They will always be in my happiest memories.

Finally, I cannot put into words how grateful I am for the support of my dear family. I would like to thank my parents Nurten and Ibrahim Cetin for always believing in me, even when I have become a nightmare child under research stress. Having their full support, no matter what, has been the one strong source of energy that kept me going.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In algebra and group theory, a homomorphism is a *structure-preserving mapping* between two algebraic structures. This means a map $f : X \to Y$ between two sets X, Y equipped with the same structure such that, if $*$ is an operation of the structure (such as two groups, two rings, or two vector spaces), then

$$f(a * b) = f(a) * f(b)$$

for every pair a, b of elements of X. In cryptography, a Homomorphic Encryption (HE) method is a cryptographically secure and invertible map from the plaintext domain to the ciphertext domain with at least one operation that provides homomorphism in these two algebraic structures while protecting the privacy of the plaintexts. In particular, HE methods that have only one homomorphic operation are *partially* Homomorphic Cryptosystems, and those that have both addition and multiplication homomorphism are called *fully* Homomorphic Encryption (FHE) schemes.[1]

The question of having a secure and practical FHE scheme was first introduced

[1]There is more to what makes a real FHE scheme, but that will be described in more detail later.

by Rivest *et al.* [1] in 1978. The authors concluded this work by asking two open questions: "1. Does privacy homomorphism have enough utility to make it worthwhile in practice? 2. For what algebraic systems, does a useful privacy homomorphism exist?" Since then, the general consensus in the cryptography community has been that the killer applications of FHE — ranging from private search queries to financial analysis on the cloud to machine learning over shared databases containing sensitive information such as genome data — are in the two-party setting. Specifically, it seems to be an essential tool for outsourcing computation along with storage of sensitive data to an untrusted party. However, the practicality of using FHE in real world applications remained unexplored until after 2009, when Gentry proposed a solution [2] to the second question of Rivest *et al.* Homomorphic encryption has progressed rapidly in both efficiency and versatility since its emergence in 2009. In this work, we tackle the challenges of applying the most fundamental programming problems to Fully Homomorphic Encryption framework to establish that the solutions to real world privacy problems can indeed be made practical by utilizing efficient FHE constructions.

1.1 A History of Fully Homomorphic Encryption

Although several partially HE methods were proposed following the work of Rivest *et al.* the existence of a Fully Homomorphic Encryption stayed an open question until Gentry proposed the first plausibly secure FHE in 2009 [2]. For example, very widely used public key cryptosystem RSA (1978) [3] has multiplicative homomorphism, while Goldwasser-Micali (1982) [4], ElGamal (1985) [5] and Paillier (1999) [6] cryptosystems have additive homomorphism, therefore they are all known as partially HE methods.

Gentry's first FHE solution was based on homomorphism in ideal lattices. In his pivotal work, he proves that if we start with a *somewhat* Homomorphic Encryption (SHE or SWHE) scheme that is capable of evaluating circuits of up to depth $d+1$, and if we can implement the scheme's own decryption circuit in d homomorphic evaluation levels, then this scheme is *boostrappable*. Bootstrapping, i.e., evaluation of the decryption circuit with only homomorphic operations by using an encryption of the secret key is the crucial innovation that made the first FHE construction possible. This technique however was extremely costly and inefficient. In 2010, Gentry and Halevi [7] presented the first actual FHE implementation along with a wide array of optimizations to tackle the infamous efficiency bottleneck of FHE schemes.

We have witnessed an amazing wealth of improvements in SHE and FHE schemes over the last decade. Several newer SHE and FHE schemes appeared in the literature in the following years. In 2011, Brakerski and Vaikuntanathan presented an FHE scheme that is based on the standard learning with errors (LWE) problem [8]. The security of their scheme depends on the worst-case hardness of the short vector problem in *arbitrary lattices*, varying from Gentry's first construction that relies on the worst-case hardness of problems on *ideal lattices*. In this work, they introduce a technique called *re-linearization* to obtain a somewhat homomorphic scheme using *symbolic encryptions* of the secret key, namely *evaluation keys* and another useful technique *modulus reduction* to turn their somewhat homomorphic construction to a fully homomorphic one.

Later, Brakerski, Gentry and Vaikuntanathan proposed a new FHE scheme (BGV) based on again learning with errors (LWE) problem and its ring variant (RLWE) [9]. In this work, they deviate from Gentry's blueprint, namely costly bootstrapping operation and reliance on ideal lattices and rather use the new techniques

from [8] to achieve an efficient *leveled* FHE scheme that is capable of evaluating L-level arithmetic circuits with computation complexity that is quasi-linear in the security parameter. They also provide a bootstrapping method as an optimiziation to their RLWE based new scheme. Further optimizations for FHE which also apply to somewhat homomorphic encryption schemes followed including batching and SIMD optimizations; see, e.g., [10, 11, 12].

In 2012, a new notion of *scale invariance* was introduced by Brakerski in [13]. Instead of using modulus reduction/switching that is used in the previous leveled FHE schemes, one can use a scale-invariant scheme and do not need to reduce/switch the modulus during homomorphic evaluations. In a subsequent work, Fan and Vercauteren applied this new method to BGV scheme and proposed a scale-invariant leveled FHE scheme based on the RLWE problem in [14].

In [12] Gentry, Halevi and Smart proposed the first homomorphic evaluation of a complex circuit: a full AES block. Their implementation is highly optimized for efficient AES evaluation using key and modulus switching techniques [9], batching and SIMD optimizations [10]. Their byte-sliced AES implementation takes about 5 minutes to homomorphically evaluate an AES block encryption. In 2012, Halevi (and later Shoup) published the HElib [15], a C++ library for FHE that is based on BGV scheme from [9]. In early 2015, Gentry, Smart, Halevi (GHS) [16] published significantly improved AES performance results with 2 seconds amortized per-block runtime.

In 2012, López-Alt *et al.* [17] proposed a leveled FHE scheme (LTV) based on the Stehlé and Steinfeld variant of the NTRU scheme [18]; LTV supports inputs from multiple public keys. Later in 2013, Bos *et al.* [19] introduced a scale-invariant of the LTV scheme (YASHE) along with an implementation. The authors modify LTV by adopting the tensor product, i.e., scale-invariance, technique introduced earlier by

Brakerski [13] thereby providing a security reduction to that of standard lattice-based problems. Later, Doröz *et al.* proposed another variant of the LTV in [20] (DHS), putting forward a batched, bit-sliced implementation that features the modulus switching technique from [9] alongside a modified relinearization that reduces the size of the evaluation keys. Later in 2016, Albrecht, Bai and Ducas showed that the narrow key distribution used in LTV and its variants enables subfield attacks for poorly chosen parameters [21].

In [22] Gentry, Sahai and Waters (GSW) proposed quite a different approach based on the Learning with Errors (LWE) problem where they choose the secret key as an approximate eigenvector $\mathbf{v}$ and encode every plaintext λ as a matrix A_λ with $A_\lambda \mathbf{v} \approx \lambda \mathbf{v}$. One very attractive feature of this system is an operation called *flattening* which controls the noise growth by transforming ciphertexts into a set of binary structures, i.e., *bit decomposing* the ciphertexts. This noise control technique eliminates the need for relinearization and costly evaluation keys. In 2015, Ducas and Micciancio [23] presented the FHEW scheme that achieves bootstrapping for GSW in half a second.

Later in 2016, Doröz and Sunar adapted the flattening technique for NTRU ciphertexts, proposing F-NTRU in [24]. While eliminating the need for relinearization and evaluation keys, F-NTRU benefits from sampling its keys from a wide distribution, therefore being immune to the subfield attack against NTRU-based schemes [21]. Due to its small parameter sizes, F-NTRU achieves fast homomorphic multiplication, for instance a depth 30 multiplication circuit can be executed in ≈ 17 ms. Around the same time in 2016, Chillotti *et al.* introduced TFHE that is a variant of LWE and GSW over a *torus* representation and implemented a highly optimized bootstrapping method for this scheme [25, 26, 27]. They report a 13 milliseconds per gate which is a 50 times speed up over the bootstrapping in FHEW [23].

In 2017, Cheon et al. introduced CKKS, a homomorphic encryption scheme for approximate arithmetic [28], and in [29] the authors describe a bootstrapping method for the same scheme. Later, an improvement by two orders of magnitude in amortized bootstrapping method of CKKS is reported in [30]. In early 2018, cuFHE – a CUDA-accelerated implementation of TFHE is released [31]. cuFHE achieves 0.5 msec per gate performance – a 26 times speed up over TFHE when run on an NVIDIA Titan Xp GPU.

1.2 Why Do We Need Fully Homomorphic Encryption?

When the notion of privacy homomorphism was first introduced in [1], a small loan company with storage and computation needs was given as a sample application. In this example, a loan company uses a commercial time-sharing service to store its data. The company's database contains sensitive financial information, therefore they decide to protect their data by encrypting all of it and sharing only this encrypted database with the storage service. Basic principle is that the data can never be decrypted by the storage computer, but only by the home office of the loan company, i.e. the data owner. There is one problem with this setup however, the time-sharing service provides storage utilities to the loan company, but in order to use the computational utilities they have to compromise from the privacy of their stored data. Unfortunately, when the loan company needs information such as; the size of the average loan outstanding, or the total incoming loan payments in the next cycle, or the number of loans over $\$1,000,000$; these questions require some sort of arithmetic computation to be answered. Of course, the most trivial solution to this problem is for the home office to download the whole encrypted database,

decrypt it and perform the computations over their cleartext data.

When Gentry proposed the first FHE construction [2], he presented another example: a private search engine that enables the user to submit an encrypted query and computes an encrypted result without ever knowing any information about the query itself or the result. Similarly, it can also provide encrypted file storage on a remote server, e.g. Dropbox, Google Docs, and allow the user to search over encrypted files such that the server can retrieve the files that satisfy some boolean test without ever decrypting the files.

Another significant application of FHE is in human genome privacy studies. The growth of genome data and computational requirements overwhelm the capacity of servers. Many institutions that handle big genome data and the National Institute of Health (NIH) are considering using cloud computing services to reduce their storage and computation costs in their genome research. Privacy and security are major concerns when deploying cloud-based data analysis tools. FHE comes as a natural solution to protect sensitive genome data while providing computation utilities on remote servers. To this end, Department of Biomedical Informatics at UCSD, and School of Informatics and Computing at Indiana University co-organize iDASH an annual competition to demonstrate the state of the art privacy preserving technologies that can be used to solve biomedical challenges involving confidential big genome data.[2] Their fully homomorphic encryption competition track over the years included tasks such as building a machine learning model (i.e., logistic regression) over encrypted genotype/phenotype data to predict the disease, Genome Wide Association Studies (GWAS) based on linear or binary logistic regression to compute the p-values of different encrypted SNPs, secure genotype imputation etc.

[2] `www.humangenomeprivacy.org`

1.2.1 A History of FHE Applications

With the improved primitives as a springboard, homomorphic encryption schemes have been used to build a variety of higher level security applications. For example, Lagendijk *et al.* give a summary of homomorphic encryption and MPC techniques to realize key signal processing operations such as evaluating linear operations, inner products, distance calculation, dimension reduction, and thresholding in [32].

Meanwhile SHE tools, developed mainly to achieve FHE, have also been explored for use in applications in their own right. In [33] for instance, Lauter *et al.* consider the problems of evaluating averages, standard deviations, and logistic regression which provide basic tools for a number of real-world applications in the medical, financial, and advertising domains. Later, Lauter *et al.* show in [34] that it is possible to implement genomic data computation algorithms where the patients' data are encrypted to preserve patient privacy. The authors used a leveled SHE scheme which is a modified version of LTV where they omit the costly relinearization operation.

In [35], Bos *et al.* show how to privately perform predictive analysis tasks on encrypted medical data. The authors use the SHE implementation of [19] to provide timing results. in [36], Graepel *et al.* demonstrate that it is possible to homomorphically evaluate machine learning algorithms in a service while protecting the confidentiality of the training and test data. They also provide benchmarks for a small scale data set to show that their scheme is practical. In [37], Cheon *et al.* presented a method along with implementation results to compute encrypted dynamic programming algorithms such as Hamming distance, edit distance, and the Smith-Waterman algorithm on genomic data encrypted using a somewhat homomorphic encryption algorithm.

1.3 Challenges of Homomorphic Evaluations

Bootstrapping [38], relinearization [8], modulus reduction [9], scale-invariance [13] and flattening [22] remain as indispensable tools for FHE schemes. Having additive (+) and multiplicative homomorphism (×) along with bootstrapping allows us to perform any polynomial function ($f_{+,\times}(x_1, x_2, \cdots, x_k)$) over arbitrary number – k – of input ciphertexts – x_i –. In theory, the privacy applications are made possible with FHE, however in practice it is not always trivial to apply FHE schemes directly on existing solutions. Most homomorphic encryption schemes provide, as basic functionality, addition and multiplication of ciphertexts which encrypt elements in some ring, with the caveat that multiplication gates are considerably "more expensive" than addition gates. At face value, this equips us with the ability to evaluate multivariate polynomials on inputs with a strong preference for low degree polynomials.

1. Depending on the application, FHE plaintexts may need to store messages in many different forms, e.g., a Personal Identification number, a credit card number or an account balance can all be large integers, whereas a patient's immunization record or genome data can be long strings. One of the challenges of building FHE applications is to find the most efficient way of representing the sensitive data before encryption. NTRU and RLWE based leveled FHE schemes use a polynomial ring $\mathcal{R}_p = \mathbb{Z}_p[x]/f(x)$ as their plaintext space, where p, – the *plaintext modulus* – is a small integer. On the other hand, efficient bootstrapping methods are highly optimized for LWE-based bit encryptions, i.e., $p = 2$, but do not handle messages from a higher radix, i.e., $p = 2^k$ or a polynomial domain which is a limiting factor.

2. The encryption mapping is from a small plaintext modulus to a really large ciphertext modulus with $q >> p$ with high dimension n. This affects both

computation and *communication* complexity dramatically. In most schemes, a single bit is encrypted into a high degree polynomial with fat coefficients, which requires costly ring operations such as polynomial multiplication and reduction, to be able to compute a single logic gate. In addition to that, the size of the ciphertexts in the orders of q^n bits can be quite large in comparison to a single bit. This means, encoding or batching of input messages is essential in applications where bandwidth is limited.

3. Ciphertext multiplications corrupt the encryption mask and increases the noise which is why it is usually followed by a combination of key and noise management techniques, e.g. relinearization followed by modulus switching or bootstrapping. A ciphertext multiplication is already an expensive algebraic operation in FHE rings, and the following operations can be even more costly computationally. This requires for custom optimizations to reduce the number of multiplications and the multiplicative depth of the circuit. High level software algorithms solving fundamental programming problems are designed to optimize the complexity with respect to the most expensive block of the computation and some operations such as logical gates or division with constants are considered trivial in these programs. In the contrary, in homomorphic circuit evaluations even the simplest algorithm step can be quite challenging to implement efficiently. Therefore, we need new ways of optimizing these fundamental algorithms to reduce the complexity and analyze the cost of the overall circuit evaluation not just by the cost of the gates in the circuit, but also the cost of the method following the ciphertext multiplication.

4. Finally, sometimes there may be a trade-off in choosing the most efficient plaintext encoding and the ciphertext key/noise management technique. If, for

instance, a circuit is too deep to utilize with a leveled FHE scheme and we need to apply bootstrapping and considering the most efficient implementations of bootstrapped methods are specifically designed for bit encryptions, this might prevent us from using a special polynomial encoding to represent our input messages. Depending on the applications, design decisions regarding plaintext representation and ciphertext evaluations should be made jointly.

1.4 Our Contributions

Most homomorphic encryption schemes provide, as basic functionality, addition and multiplication of ciphertexts which encrypt elements in some ring, with the caveat that multiplication gates are considerably "more expensive" than addition gates. At face value, this equips us with the ability to evaluate multivariate polynomials on inputs with a strong preference for low degree polynomials.

In applications such as machine learning, other fundamental operations become essential: **division**, **zero test**, **thresholding** and **comparison**. Bit-level encryption excels at functions with Boolean output but incurs prohibitive cost when required to perform arithmetic even in moderate-sized message domains. Approaching this from the other end, we seek out algebraically efficient algorithms for the operations in the above list for schemes with large message domains.

Further progress towards real world FHE applications requires new ideas for the efficient implementation of algebraic operations on word-based (as opposed to bit-wise) encrypted data. Whereas handling data encrypted at the bit level leads to prohibitively slow algorithms for the arithmetic operations that are essential for cloud computing, the word-based approach hits its bottleneck when operations such as integer comparison are needed. In Chapter 5, we tackle this challenge by propos-

ing solutions to problems — including comparison and division — in word-based encryption. We present concrete performance figures for all proposed primitives via F-NTRU scheme. We present an array of solutions to improve the versatility of higher characteristic SHE/FHE schemes along with new abilities, specifically:

- We compare three approaches to field inversion, each with its advantages; these naturally lead to algorithms for division, zero test and equality checking. The first method is exact but slower; the others produce rational approximations, which we scale to integers. Our approach based on Newton-Raphson iterations also gives us an algorithm for square roots. Our convergence-based approach performs better when the characteristic is large due to its amenability to a specially chosen plaintext space and encoding technique. Particularly valuable by-products include comparison circuits and threshold functions.

- We summarize with an overall comparison of word-wise homomorphic algebraic operations vis a vis their bit-wise counterparts for a 32-bit integer domain.

- We implement the proposed methods using an F-NTRU based homomorphic encryption library and provide execution times for these implementations.

In Chapter 6, we focus a fundamental programming task, *sorting*, and analyze different sorting algorithms and compare their performances for sorting encrypted data. As of writing the first part of this chapter, other homomorphic sorting results we are aware of are Chatterjee *et al.* [39] and Emmadi *et al.* 's [40]. In [39], the authors introduce a hybrid technique, i.e. Lazy Sort. This method first uses Bubble Sort to nearly sort the input elements. Then, the list is sorted again by using Insertion Sort. The authors claim that this method has better complexity than the worst case scenario, which is disputed in both our work and [40]. Emmadi et al.

implement and compare Bubble Sort, Insertion Sort, Bitonic Sort and Odd-Even Merge Sort in [40] and their observations are consistent with the analysis provided in our work. More recently, Narumanchi et al. compared bitwise and integer-wise encryption techniques for homomorphic comparison and sorting operation in [41]. Their analysis shows that it is more efficient to use bitwise encryption in terms of performance. As all algorithms proposed in the previous works still perform poorly when sorting a large data set, the largest N used is around 64 in the experiments.

- Our survey includes well-known algorithms such as Bubble Sort, Merge Sort and two sorting networks: Bitonic Sort and Odd-Even Merge Sort [42]. Due to the high depths of the known algorithms, we proposed two new depth-optimized methods: Greedy Sort and Direct Sort in our first work. Both of these algorithms require a circuit of depth $\mathcal{O}\left(\log(N) + \log(\ell)\right)$ where N is the number of elements and ℓ is the bit-length of each element. Both algorithms improve in the circuit depth metric over classical algorithms by at least *1-3 orders of magnitude*.

- The main contribution in our follow-up work is proposing an alternative way of sorting numbers by computing the Hamming weight of N bits with only N homomorphic multiplications in the Direct Sorting method. In comparison to our proposed method with $\mathcal{O}(N)$ multiplications, Direct Sort and Greedy Sort require $\mathcal{O}(N \log N)$ and $\mathcal{O}(2^N)$ multiplications, respectively. Our algorithm implements the Direct Sort method with minimum number of operations. We also observe that our proposed method is a compact implementation of Greedy Sort. Therefore, it both minimizes the circuit depth and the number of homomorphic evaluations. Furthermore, our method for efficient evaluation of the Hamming weight computation can be used in many other homomorphic appli-

cations. In addition to performance improvements, the proposed algorithm is easier to analyze and implement in comparison to the previous methods. Even when batching is not applicable, the highly parallelizable nature of the algorithm makes it an efficient candidate for a GPU implementation. Our sorting method is generic and can be implemented with existing software libraries, e.g. HElib [15].

- Our next contribution is adapting Single-Instruction Multiple-Data (SIMD) idea from [10] and permutation technique from [11] to evaluate homomorphic comparisons in parallel to sort the elements of a single set. In previous works, batching is used to sort separate number sets simultaneously; therefore batching cannot be applied when there is only one set to be sorted. We propose placing the set elements into message slots of a single plaintext and using homomorphic rotation method from [11] when cross-slot computation is required. Gentry et al. used this technique to evaluate an AES circuit homomorphically in [16]. This method requires key switching after every rotation. However we are able to reduce the number of comparisons from N^2 to N provided that the number of slots is greater than or equal to N.

- For our first performance results, we instantiate a DHS leveled scheme based on NTRU, and present an implementation of the proposed sorting algorithms: Greedy and Direct Sort. Our results confirm our theoretical analysis, i.e. that the performance of the proposed sorting algorithm scales favorably as N increases. In the second part, we give an implementation of our newly proposed method Polynomial Ranks Sort and an updated implementation of Direct Sort using BGV based HElib software.

For our final application, we focused on Gentry's first example; a private search

engine. More specifically, we designed a completely blind web search engine. Considering the computational and bandwidth requirements of early FHE schemes, along with the fact that web search is a real-time application, the lack of practical proposals prior to ours is not surprising. There are however more recent works in the literature that tackle encrypted search problem. This topic can be categorized into three different groups:

1. In the first category, both the keyword and the database are encrypted. For example, a user looking for an information sensitive word in their private files outsourced to a remote server, e.g., Microsoft OneDrive, Google Docs, Dropbox. In this scenario, the files have to be encrypted by the owner before being uploaded to the server. The search keyword needs to be kept private, hence the owner has to encrypt it before sending the query. Lookup of the secret keyword is performed over the already encrypted files on the server side and the result is returned to the user in hidden form. For instance, [43, 44] demonstrate solutions to this type of encrypted search over patients' outsourced genomic data, and in [45, 46], Akavia *et al.* propose more generic solutions.

2. In the second category, we consider the case when only the database is encrypted. A user might have private files stored in the cloud and they might want to do a search with a not-so-sensitive keyword. Although technically this is possible, it may come with other security problems. Once the server performs a query with an open keyword, it will learn from that search and may classify the private user data with respect to queried words by using them as labels. If that does not leak any sensitive information –for example if the search keyword is a generic time-stamp and the user needs access to certain

messages from a particular date and time and the user is aware of the honest-but-curious server model– the user may choose to send the query in an open form. However in this case, there is no reason to encrypt that information in the first place. Therefore, this scenario is highly unlikely to be useful in secure applications.

3. In the last group, we have only the keyword encrypted which is the most applicable scenario to our web-search model. Since the owner of the web-index tables is the server itself (e.g. Google or Microsoft) and the content on the web is publicly available, they are not required or motivated to keep them in an encrypted form. Additionally, we have a finite number of keywords in our look-up table. Since the search engine already has the index tables set, the user can encrypt the index of the keyword instead of the keyword itself, i.e. Private Information Retrieval (PIR) [47, 48, 49]. There are more recent PIR works that use FHE [8, 50, 51]. We adapt the method from [49] and propose a hybrid homomorphic solution in this work. We also consider the multi-keyword search in this work. Since the search index tables do not take into account of any possible combination of different words, we need to be able to search for single keywords separately and find the intersection of the encrypted results. This is known as the Private Set Intersection (PSI) problem. In [52], a homomorphic PSI algorithm is described. Their method has two parties communicating where one party sends an encrypted set of values and the other party compares those with their own set. The second party returns a zero-flag placed on the index of each intersected value. This does not apply to our scenario, because sending one bit flag is not sufficient for returning the URL results back to the user.

In Chapter 7;

- We present the first end-to-end study of blind web search using homomophic encryption where the client submits encrypted keywords and the server performs a blinded lookup and returns the results again in encrypted form. We separate our problem into two parts; Comparison and Aggregation.

- We cast Comparison into a private information retrieval (PIR) problem and compare various PIR algorithms, i.e. variants of Kushilevitz-Ostrovsky PIR, for suitability in our setting.

- We present a thorough analysis of the depth and bandwidth trade-offs, as well as the number of multiplications for each proposed solution.

- We present our Aggregation step first in the single keyword scenario and then extend our construction to consider the queries with multiple keywords. To this end, we perform a homomorphic intersection by encoding URLs as zeros of polynomials. We compare our approach against other with regard to the problem of returning false positives.

- With all the pieces in place, we provide a noise analysis of the proposed methods with respect to F-NTRU parameters and finally give the implementation results of the proposed schemes using a GPU implementation of the scheme. The results show that the bandwidth overhead is in the MBytes while the query requires micro to milliseconds for processing per row to support single and multiple keyword lookups with intersection, i.e. AND operations on keywords.

The works in Chapter 5 are collaborations with Yarkın Doroz, Berk Sunar and William J. Martin, and published in ArcticCrypt 2016. An earlier version of this

work is published in IACR Cryptology ePrint Archive. Chapter 6 consists of publications with Yarkın Doroz, Berk Sunar, and Erkay Savas. Our first results are published in LatinCrypt 2015, with a follow-up work in LatinCrypt 2017. Our final work, with the batched implementation is under reviewing process with IEEE Transactions on Emerging Topics in Computing. The works in Chapter 7 are collaborations with Wei Dai, Yarkın Doröz, Berk Sunar, and William J. Martin. Our first results are published on IACR Cryptology ePrint Archive in 2016, and our follow-up work is currently under revision for submission.

Chapter 2

Background

A typical fully homomorphic encryption scheme has five primitive operations: ParamGen, KeyGen, Encrypt, Decrypt, and Eval and a typical two-party communication can be seen in Figure 2.1. In general, in a two-party FHE application, the data owner is responsible of the first four operations, and the server has two tasks storage of the encrypted data and perform the homomorphic evaluations over this encrypted database.

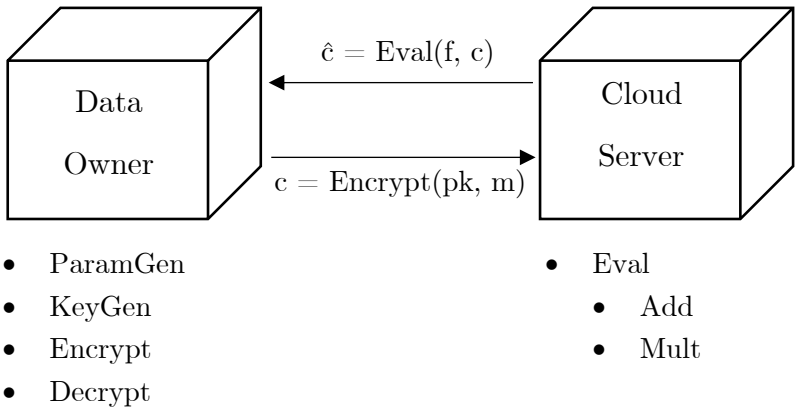


Figure 2.1: A typical two-party FHE Application.

We define these primitive operations for ring-based, fully homomorphic public-

key encryption schemes as follows:

1. ParamGen: takes the plaintext modulus p and a security parameter λ as inputs, and generates a set of parameters: an error distribution χ_{err}, a key distribution χ_{key}, ring dimension, and coefficient modulus.

2. KeyGen: takes the parameter set generated by ParamGen as inputs, and creates a random public and secret key pair (pk, sk).

3. Encrypt: maps a message m from the plaintext space to a ciphertext C. This ciphertext consists of three essential parts; a function of the message m, an *encryption mask*, i.e., a randomized function of the public key pk and a *noise mask*, i.e., a function of a uniformly sampled error e from χ_{err}. A ciphertext typically has the following form:

$$C = f_1(m) + f_2(\mathsf{pk}) + f_3(e)$$

   with specially chosen functions f_1, f_2, f_3.

4. Decrypt: is the inverse of Encrypt, therefore maps a ciphertext C to a plaintext m by first removing the encryption mask and secondly removing the noise mask. A decryption error occurs when there is an overflow caused by the noise.

5. Eval: takes a list of ciphertexts $C_1, \cdots, C_k$ and a circuit $\mathcal{C}$ that implements the polynomial function $f_{+,\times}(x_1, \cdots, x_k)$ and computes $C = \mathcal{C}(C_1, \cdots, C_k)$. The decryption of the output ciphertext gives us the output of the function f over the respective plaintexts, i.e., $\mathsf{Decrypt}(C) = f_{+,\times}(m_1, \cdots, m_k)$ where $C_i = \mathsf{Encrypt}(m_i)$ for $i = 1, \cdots, k$.

Table 2.1: Comparison of the existing ciphertext noise/key management techniques.

Technique	**Performance**	**Purpose**	**Motivation**
bootstrapping	slow	recrypts the message	unlimited operations
relinearization	moderate	corrects the encryption mask	required when there is no bootstrapping
modulus switching	fast	cuts a portion of the noise	efficient when depth is low

From the five primitives Eval is the one where actual homomorphic evaluations are computed. Here, we define two homomorphic operations: Add and Mult. Let $C_1, C_2 = \mathsf{Encrypt}(m_1), \mathsf{Encrypt}(m_2)$ and we have homomorphic properties which let us perform additions and multiplications over ciphertexts:

$$\mathsf{Decrypt}(\mathsf{Add}(C_1, C_2)) = m_1 + m_2 \tag{2.1}$$

$$\mathsf{Decrypt}\left(\mathsf{Mult}(C_1, C_2)\right) = m_1 \times m_2 \tag{2.2}$$

Among these two operations, Mult corrupts the encryption mask, and causes a significant noise growth. In order to fix these problems, FHE schemes make use of different techniques. A summary of these methods are described in Table 2.1. We provide a table of existing libraries for comparison in Table 2.2. For more details on secure parameter selection for different security levels and known attacks against FHE schemes, we refer users to the Homomorphic Encryption Standard document [53].

In the following sections, we give more details on selected FHE schemes that we used in our implementations.

Table 2.2: Comparison of the existing FHE libraries.

FHE Library	FHE Scheme	Dependencies	Bootstrapping	Batching
HElib[15]	BGV[9], CKKS[28]	NTL[54], GMP	Yes	Yes
DHS	DHS[20]	NTL[54], GMP	No	Yes
SEAL[55]	FV[14], CKKS[28]	No	No	Yes
cuHE[56]	FNTRU[24]	CUDA	No	No
cuFHE[31]	TFHE[25]	CUDA	Yes	No

2.1 LTV Based DHS Scheme

This is a summary of the multi-key LTV-FHE scheme and a brief explanation on the primitive functions that are proposed by López-Alt *et al.* Later in this section, we give details of the DHS scheme based on a single key customized LTV.

In 2012 López-Alt *et al.* proposed a leveled multi-key FHE scheme (LTV) [17]. The scheme based on a variant of NTRU encryption scheme proposed by Stehlé and Steinfeld [18]. The introduced scheme uses relinearization [8] technique to correct the encryption mask and modulus switching [9] for noise control. Doröz *et al.* proposed a single key version of LTV (DHS) in [20] with reduced key size technique. The operations are performed in $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ where n is the polynomial degree and q is the prime modulus. The scheme also defines an error distribution χ, which is a truncated discrete Gaussian distribution, for sampling random polynomials that are B-bounded. The term B-bounded means that the coefficients of the polynomial are selected in range $[-B, B]$ with χ distribution. The scheme consists of four primitive functions, namely KeyGen, Encrypt, Decrypt and Eval. A brief detail of the primitives is as follows:

KeyGen. Choose sequence of primes $q_0 > q_1 > \cdots > q_d$ to use a different q_i in each level. A public and secret key pair is computed for each level: $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ and $f^{(i)} = 2u^{(i)} + 1$, where $\{g^{(i)}, u^{(i)}\} \in \chi$. Create evaluation keys for each level:

$\zeta_\tau^{(i)}(x) = h^{(i)}s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau(f^{(i-1)})^2$, where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i \rfloor]$.

Encrypt. To encrypt a bit b for the i^{th} level we compute: $c^{(i)} = h^{(i)}s + 2e + b$, where $\{s, e\} \in \chi$.

Decrypt. In order to compute the decryption of a value for specific level i we compute: $m = c^{(i)}f^{(i)} \pmod 2$.

Eval. The gate level logic operations XOR and AND are done by computing the addition and multiplication of the ciphertexts. In case of $c_1^{(i)} = \mathsf{Encrypt}(b_1)$ and $c_2^{(i)} = \mathsf{Encrypt}(b_2)$; XOR is equal to $c_1^{(i)} + c_2^{(i)} = \mathsf{Encrypt}(b_1 + b_2)$ and, AND is equal to $c_1^{(i)} \cdot c_2^{(i)} = \mathsf{Encrypt}(b_1 \cdot b_2)$. The multiplication creates a significant noise in the ciphertext and to cope with that we apply Relinearization and modulus switch. The Relinearization computes $\tilde{c}^{(i)}(x)$ from $\tilde{c}^{(i-1)}(x)$ extending $\tilde{c}^{(i-1)}(x)$ as a linear combination of 1-bounded polynomials $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}_\tau^{(i-1)}(x)$. Then, using the evaluation keys it computes $\tilde{c}^{(i)}(x) = \sum_\tau \zeta_\tau^{(i)}(x)\tilde{c}_\tau^{(i-1)}(x)$ as the new ciphertext. The formula is actually the evaluation of homomorphic product of $c^{(i)}(x)$ and $(f^{(i)})^2$. Later, the modulus switch $\tilde{c}^{(i)}(x) = \lfloor \frac{q_i}{q_{i-1}}\tilde{c}^{(i)}(x) \rceil_2$ decreases the noise by $\log(q_i/q_{i-1})$ bits by dividing and multiplying the new ciphertext with the previous and current moduli, respectively. The operation $\lfloor \cdot \rceil_2$ refers to rounding and matching the parity bits.

2.1.1 DHS Customizations

We use a customized version of the LTV scheme that is previously proposed in [20] by Doröz, Hu and Sunar (DHS). The code is written in C++ using NTL package that is compiled with GMP library. The library contains some special customizations that improve the efficiency in running time and memory requirements. The customizations of the DHS implementation are as follows:

- We select a special m^{th} cyclotomic polynomial $\Psi_m(x)$ as our polynomial modulus. The degree of the polynomial n is equal to the euler totient function of m, i.e. $\varphi(m)$. In each level the arithmetic is performed over $\mathrm{R}_{q_i} = \mathbb{Z}_{q_i}[x]/\langle\Psi_m(x)\rangle$, where modulus q_i is equal to p^{k-i}. The value p is a prime number that cuts $(\log_p)$-bits of noise and the value k is equal to the depth plus 1.

- Due to the special structure of the moduli p^{k-i}, the evaluation keys in one level can also be promoted to the next level via modular reduction. For any level we can evaluate the evaluation key as $\zeta_\tau^{(i)}(x) = \zeta_\tau^{(0)}(x) \pmod{q_i}$. This technique reduces the memory requirement significantly and makes it possible to evaluate higher depth circuits.

- The specially selected cyclotomic polynomial $\Psi_m(x)$ is used to batch multiple message bits into the same polynomial for parallel evaluations as proposed by Smart and Vercauteren [10, 57]. The polynomial $\Psi_m(x)$ is factorized over $\mathbb{F}_2$ into equal degree polynomials $F_i(x)$ which define the message slots in which message bits are embedded using the Chinese Remainder Theorem. We can batch $\ell = n/t$ number of messages, where t is the smallest integer that satisfies $m|(2^t - 1)$.

- The DHS library can perform 5 main operations; KeyGen, Encryption, Decryption, Modulus Switch and Relinearization. The most time consuming operation is Relinearization, which is generally the bottleneck.

  Therefore, the most critical operation for circuit evaluation is Relinearization. The other operations have negligible effect on the run time.

2.1.2 Modified Relinearization

We modify previously implemented method of relinearization where it uses linear combination of 1-bounded polynomials of the ciphertext $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}_\tau^{(i-1)}(x)$. Previously, the number of evaluation key polynomials and the number of multiplications in relinearization is $\lceil \log(q) \rceil$. For deep circuits with many levels the bitsize $\lceil \log(q) \rceil$ is two/three orders of magnitude which increase the memory requirements and number of multiplications significantly. In order to achieve a speedup, we group the bits of the ciphertext and use the linear combination of word (r-bits) sized polynomials rather than binary polynomials. Setting the word size as $w = 2^r$, we implement the following changes:

- Compute the evaluation keys as: $\zeta_\tau^{(i)}(x) = h^{(i)} s_\tau^{(i)} + 2e_\tau^{(i)} + w^\tau (f^{(i-1)})^2$, where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i / r \rfloor]$.
- Divide the ciphertext into linear combinations of word sized polynomials: $\tilde{c}^{(i-1)}(x) = \sum_\tau w^\tau \tilde{c}_\tau^{(i-1)}(x)$.
- Compute the relinearization as: $\tilde{c}^{(i)}(x) = \sum_\tau \zeta_\tau^{(i)}(x) \tilde{c}_\tau^{(i-1)}(x)$

The changes above decreases the memory requirement by r times. With this change relinearization requires r times fewer multiplications. However this does not yield r times speedup. This is due to the increase of the coefficient size of the linear combination polynomials from 1 to r bits. Thus the cost of a multiplication increases.

2.2 F-NTRU

In this section, we define the leveled FHE scheme F-NTRU that we employ in our application. There are three main reasons behind why we choose F-NTRU in our

implementations:

1. F-NTRU does not use evaluation or bootstrapping keys, and can achieve high performance for low-depth circuits. Our construction has low-degree evaluations; therefore we obviate the need for bootstrapping functionality.

2. This F-NTRU variant allows us to encode large integers directly, unlike fast bootstrapped schemes which rely on encryptions at the bit level only.

3. Homomorphic evaluations in F-NTRU can be highly distributed across multiple platforms, due to scheme's core reliance on easily parallelizable matrix-vector operations. In a scenario such as web search, a server can reasonably utilize a distributed system large enough (using cheap GPUs, for example) to execute each ciphertext multiplication at the computational cost of a single polynomial multiplication.

F-NTRU Setup

A leveled FHE scheme [24] F-NTRU adopts the flattening technique proposed in GSW to derive an NTRU based scheme that (akin to GSW) does not require evaluation keys or key switching. This scheme eliminates the decision small polynomial ratio (DSPR) assumption but relies only on the standard R-LWE assumption. F-NTRU uses wide key distributions, and hence is **immune** to the subfield lattice attack [21]. The setup of F-NTRU is as follows:

- **ParamGen:** Contemporary parameters for the NTRU scheme are used in F-NTRU with the exception that the "small prime" p is not required to be an integer but rather the polynomial $p = x - b$ where, here, $b = 2$. Given security parameter λ, we choose an integer modulus $q = q(\lambda)$ and a polynomial degree $n = n(\lambda)$ and perform our computations in the ring $\mathbb{Z}_q[x]/(x^n + 1)$. We also

fix the Gaussian distributions $\chi_{\text{err}} = \chi_{\text{err}}(\lambda)$ and $\chi_{\text{key}} = \chi_{\text{key}}(\lambda)$ using the same security parameter.

- **KeyGen:** We select $g, f' \leftarrow \chi_{\text{key}}$ and compute the secret key $f = pf' + 1$ and the public key $h = pgf^{-1}$.

- **Encrypt:** The encryption function from NTRU, $\mathsf{Enc}(m) = hs + pe + m$ where the polynomial $m = \sum_i \alpha_i x^i \in \mathbb{Z}_b[x]/(x^n + 1)$ encodes the integer $\alpha = \sum_{i=0}^{n-1} \alpha_i b^i$, We must first define an operation called BitDecomp that splits a ciphertext polynomial $c(x)$, into ℓ b-ary polynomials and we show it as follows:

$$\begin{aligned}\mathsf{BitDecomp}(c(x)) &= \langle \tilde{c}_{\ell-1}(x), \cdots, \tilde{c}_1(x), \tilde{c}_0(x) \rangle \\ &= \vec{\tilde{c}},\end{aligned}$$

and given ℓ $\tilde{c}_i(x)$s, computing $c(x)$ is called the inverse bit decomposition, $\mathsf{BitDecomp}^{-1}$.

$$\begin{aligned}\mathsf{BitDecomp}^{-1}(\vec{\tilde{c}}) &= \sum_{i=0}^{\ell-1} 2^i \cdot \tilde{c}_i(x) \\ &= c(x).\end{aligned}$$

In this scheme we have a vector of NTRU encryptions, as our ciphertext of a single encrypted message μ. The length of the ciphertext vector is $\ell = \log q$ and we start by placing an encryption of zero in every element of this vector.

$$\begin{aligned}\vec{c} &= \langle \mathsf{Enc}_{\ell-1}(0), \mathsf{Enc}_{\ell-2}(0), \ldots, \mathsf{Enc}_0(0) \rangle \\ &= \langle c_{\ell-1}, c_{\ell-2}, \ldots, c_0 \rangle ,\end{aligned}$$

where $c_i = \mathsf{Enc}_i(0) = hs_i + pe_i$. By taking the transpose of $\vec{c}$, we first place each encryption of zero in a single row and then using bit decomposition over each row, we build an $\ell \times \ell$ matrix $c = \mathsf{BitDecomp}(\vec{c}^{\top})$.

Finally, using this matrix, we encrypt the message μ by computing,

$$C = \mathsf{Flatten}(I_\ell \cdot \mu + c)$$

where I_ℓ is the identity matrix of order ℓ, $\mathsf{Flatten}$ is the special technique from [22] which is an inverse bit decomposition, followed by a bit decomposition operation:

$$\mathsf{Flatten}(\vec{\tilde{c}}) = \mathsf{BitDecomp}(\mathsf{BitDecomp}^{-1})(\vec{\tilde{c}}).$$

- **Decrypt:** To decrypt a ciphertext, we take the first row of the matrix, which is the vector $\vec{\tilde{c}}_0$, and apply $\mathsf{BitDecomp}^{-1}$ to form an NTRU ciphertext $\mathsf{BitDecomp}^{-1}(\vec{\tilde{c}}_0) = c_0$. Once we compute the NTRU ciphertext, we apply the decryption method from the NTRU scheme as $\lfloor c_0 f \rceil \bmod p$ and retrieve the message μ.

- **Eval:** The homomorphic XOR and AND operations are matrix addition and multiplication operations, followed by a $\mathsf{Flatten}$ operation as below.

$$C' = \mathsf{Flatten}(C + \tilde{C}) \quad , \quad C' = \mathsf{Flatten}(C \cdot \tilde{C}).$$

Chapter 3

Operations Related to Plaintext Space

A proper representation of data is significant in homomorphic applications for both communication and computation complexity. Throughout this work, we follow a similar notation to previous works [10, 11, 12]. We define the messages with lowercase letters $a \in \mathcal{R}_p$, batched or encoded plaintexts with Greek letters $\alpha \in \mathcal{R}_p$, and the ciphertexts with uppercase letters $A \in \mathcal{R}_q$ where $\mathcal{R}_p = \mathbb{Z}_p[x]/\Phi_m(x)$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\Phi_m(x)$. We use primes p and q (often referred as plaintext and ciphtertext moduli, respectively) in our schemes and $\Phi_m(x)$ is the m^{th} cyclotomic polynomial.

3.1 Binary Representation

When we have messages in the range $\left[0, \ldots, 2^{k-1} - 1\right]$, we can use their k-bit binary representation for encryption. In this case, we set plaintext modulus $p = 2$ and define binary encryption as follows. Given a number a with $\mathsf{BitDecomp}(a) = \langle a_0, \ldots, a_{k-1} \rangle$,

i.e., $a = \sum_{i=0}^{k-1} 2^i \cdot a_i$, we can encrypt each bit separately $A_i = \mathsf{Encrypt}(a_i)$, and have a vector of ciphertexts $\vec{A} = \langle A_0, \ldots, A_{k-1} \rangle$.

Given two bits $a_1, a_2 \in \{0, 1\}$ and their encryptions A_1 and A_2, two Eval primitives are defined over modulus 2:

$$\mathsf{Decrypt}(A_1 + A_2) \equiv a_1 + a_2 \mod 2 \tag{3.1}$$

$$\mathsf{Decrypt}(A_1 \times A_2) \equiv a_1 \times a_2 \mod 2 \tag{3.2}$$

3.2 Word Representation

When we have messages in the range $\left[0, \cdots, 2^{k-1} - 1\right]$, we can use a higher radix plaintext modulus $p \geq 2^{k-1}$ and define word-wise encryption as follows. Given a number a, we encrypt the whole word and have a single ciphertext $A = \mathsf{Encrypt}(a)$. Given two numbers $a_1, a_2 \in \{0, \cdots, p-1\}$ and their encryptions A_1 and A_2, two Eval primitives are defined over modulus p:

$$\mathsf{Decrypt}(A_1 + A_2) \equiv a_1 + a_2 \mod p \tag{3.3}$$

$$\mathsf{Decrypt}(A_1 \times A_2) \equiv a_1 \times a_2 \mod p \tag{3.4}$$

3.3 Polynomial Encoding

Some schemes allow us to use the approach described in [34], where a small polynomial is used as the plaintext modulus p. In this case, we choose the plaintext modulus $p = x - b$ with a small integer b. For instance, setting $b = 2$ we can

represent a k-bit message a by using 2-based encoding as follows:

$$\alpha(x) = \sum_{i=0}^{k-1} a_i x^i, \quad \text{where } a = \sum_{i=0}^{k-1} a_i 2^i \ .$$

Upon decryption, message a can be retrieved by simply evaluating $\alpha(x)$ at $x = 2$, i.e. $\alpha(2) = a$.

Note that, provided the ring modulus exceeds L, this provides for carry-free addition of L ciphertexts; i.e., if $a_j = \sum_{i=0}^{k-1} a_{j,i} 2^i$ is encoded as $\alpha_j(x) = \sum_{i=0}^{k-1} a_{j,i} x^i$, then the polynomial $\beta(x) = \sum_{j=1}^{L} \alpha_j(x)$ satisfies $\beta(2) = \sum_j a_j$.

Also note that, provided the ring dimension exceeds $k \cdot L + L - 1$, this provides for products of L ciphertexts with no overflow; i.e., if $a_j = \sum_{i=0}^{k-1} a_{j,i} 2^i$ is encoded as $\alpha_j(x) = \sum_{i=0}^{k-1} a_{j,i} x^i$, then the polynomial $\beta(x) = \prod_{j=1}^{L} \alpha_j(x)$ satisfies $\beta(2) = \prod_j a_j$.

Finally, we should remark that in order to use b-based polynomial encoding, we do not necessarily need to use a polynomial plaintext modulus. Instead, setting $p > L$, we can compute addition of L ciphertexts without an overflow. Similarly, setting $p > \binom{L}{L/2}$, we can compute a product of L ciphertexts. However, increasing the magnitude of p has a significant negative effect on the noise growth, thus using a small polynomial as our plaintext modulus gives us an advantage over choosing a large modulus.

Example 1. *For $a_1 = 6$ and $a_2 = 5$, we have the binary encodings $\alpha_1(x) = x + x^2$ and $\alpha_2(x) = 1 + x^2$, respectively. If we set $p = (x - 2)$; then we recover the addition*

$$\begin{aligned} \beta(x) &= \alpha_1(x) + \alpha_2(x) = 1 + x + 2x^2 \\ \beta(2) &= 11 \\ &= 6 + 5 \end{aligned}$$

naturally, as a part of the decryption. Otherwise, we need to set $p > L = 2$ in this example with two ciphertexts, to prevent wrap-around over the plaintext coefficients.

Example 2. *For the numbers $a_1 = 7$ and $a_2 = 5$, binary encodings $\alpha_1(x) = 1+x+x^2$ and $\alpha_2(x) = 1 + x^2$, respectively and $p = (x - 2)$; we compute the multiplication as*

$$\begin{aligned}\beta(x) &= \alpha_1(x) \times \alpha_2(x) = 1 + x + 2x^2 + x^3 + x^4 \\ \beta(2) &= 35 \\ &= 7 \times 5\end{aligned}$$

naturally, as a part of the decryption. Otherwise, we need to set $p > \binom{L}{L/2} = 2$ in this example with two ciphertexts, to prevent wrap-around over the plaintext coefficients.

3.4 Fixed Point Encoding

Given a rational number $r = (a, b)$ where a is the integer part and b is the fractional part both written base 2, and let t, k be the bit-length of integer and fractional parts respectively. When we want to encrypt r, it is encoded into the following polynomial:

$$\rho(x) = \sum_{i=0}^{k-1} b_i x^i + \sum_{i=k}^{k+t-1} a_i x^i$$

An ordered pair (C, k), where $C = \mathsf{Encrypt}(\rho)$ is viewed as an encryption of the number r. The integer k specifies the location of the precision point. Adding or subtracting two such ciphertexts requires us to align their precision points: for (C_1, k_1) and (C_2, k_2) with $k_2 > k_1$, the sum is represented by $(C_1 x^{k_2-k_1} + C_2, k_2)$. Likewise, multiplication of ciphertexts (C_1, k_1) and (C_2, k_2) yields $(C_1 \cdot C_2, k_1 + k_2)$.

Similarly, arithmetic between a ciphertext and a cleartext follow the same fixed point rules.

For decoding the decrypted polynomial $m(x) = \mathsf{Decrypt}(C, k)$, we will compute the output value as $M = 2^{-k}m(2)$ which, by a slight abuse of notation, we identify with $\mathsf{Decrypt}(C, k)$ below. By setting $p = x - 2$ we directly evaluate $m(2)$ at the end of decryption, which includes reduction modulo $x - 2$. Therefore, the only required operation after decryption is shifting the bits to the right with respect to the binary precision point location.

3.5 Batching and SIMD Operations

Given two ciphertexts A, B that are encrypted under an FHE scheme, computing $A + B$ and $A \times B$ in $\mathcal{R}_q$ gives us encryptions of $a + b$ and $a \times b$ in $\mathcal{R}_p$. We use $[\![\cdot]\!]$ to represent an encryption of a constant value.

From [10], we know that with specific parameters we can enable batching, a technique that is used for concurrent evaluations in the message slots. For example, when we choose the cyclotomic polynomial with m that divides $p^d - 1$ with smallest such d, we have the factorization,

$$\Phi_m(X) = \prod_{i=1}^{\ell} F_i(X) \mod p$$

where the F_i's are $\ell = \phi(m)/d$ irreducible polynomials of degree d. Then we employ the isomorphism between the plaintext space and the ℓ copies of $\mathbb{F}_{p^d}$:

$$\mathcal{R}_p \cong \frac{\mathbb{F}_p[X]}{F_1} \otimes \cdots \otimes \frac{\mathbb{F}_p[X]}{F_\ell}$$

We define a vector with ℓ messages $\vec{\alpha} = \langle \alpha_1, \cdots, \alpha_\ell \rangle$ with each α_i belonging to the

finite field $\mathbb{L}_i = \frac{\mathbb{F}_p[X]}{F_i}$. Applying inverse Chinese Remainder Theorem (CRT), we obtain a single message in $\mathcal{R}_p$. We write this as:

$$\alpha = \mathsf{CRT}^{-1}(\langle \alpha_1, \cdots, \alpha_\ell \rangle)$$

with $\alpha_i \in \mathbb{L}_i$ and $\alpha \in \mathcal{R}_p$. We say the plaintext has ℓ message slots and each message is packed in a single slot. Additions and multiplications over CRT plaintexts will be evaluated in each slot due to the natural isomorphism. For example given another plaintext $\beta = \mathsf{CRT}^{-1}(\vec{\beta})$, we have

$$\begin{aligned} \mathsf{CRT}(\alpha + \beta) &= \langle \alpha_1 + \beta_1, \cdots, \alpha_\ell + \beta_\ell \rangle \\ \mathsf{CRT}(\alpha \times \beta) &= \langle \alpha_1 \times \beta_1, \cdots, \alpha_\ell \times \beta_\ell \rangle \end{aligned}$$

with $\alpha_i \star \beta_i \in \mathbb{L}_i$ and $\star \in \{+, \times\}$. In some applications, we need to perform computations across message slots, i.e., $\alpha_i \star \beta_j$, where $i \neq j$. To this end, we permute the message slots so that message α_i and message β_j align. Due to the relation between the factors of the cyclotomic polynomial, the automorphism

$$\kappa_g : \alpha(X) \longmapsto \alpha(x^g) \mod \Phi_m(X)$$

for a g that is not a power of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$ with order ℓ affects just such a permutation. To see why this works and for the underlying Galois field theory we refer readers to [11].

Chapter 4

Binary Arithmetic and Logic Operations

Even though high-level computer programs deal with high-radix numbers, modern computers are still based on the binary number system, i.e., 0–1 bits, boolean true–false values or electronic on–off switches. One of the fundamental building blocks of a central processing unit (CPU) in a computer is an arithmetic logic unit (ALU) that is designed to perform arithmetic and bitwise logic operations on binary numbers. Although an ALU can be designed to perform complex functions, the resulting higher circuit complexity, cost, power consumption and larger size makes this impractical in many cases. Consequently, ALUs are often limited to simple functions that can be executed at very high speeds (i.e., very short propagation delays), and the external processor circuitry is responsible for performing complex functions by orchestrating a sequence of simpler ALU operations.

Many fully homomorphic encryption schemes define a plaintext space over modulo 2 by default. Especially efficient bootstrapping techniques require messages to be single bits.

4.1 Boolean Gates : AND, XOR, NOT

By setting plaintext modulus $p = 2$, we can compute boolean XOR and AND gates naturally by evaluating homomorphic Add and Mult operations, respectively. The truth tables for all boolean gates can be found in Appendix A.1.

Definition 1. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $B = \langle B_0, \ldots, B_{k-1} \rangle$*, respectively; we define bitwise XOR "* $\oplus$ *" as follows:*

$$C = \langle A_0 + B_0, \ldots, A_{k-1} + B_{k-1} \rangle$$
$$\mathit{Decrypt}(C) = \langle a_0 \oplus b_0, \ldots, a_{k-1} \oplus b_{k-1} \rangle$$

The multiplicative **depth** of XOR evaluations is 0, because there is no ciphertext multiplication, only ciphertext additions.

Definition 2. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $B = \langle B_0, \ldots, B_{k-1} \rangle$*, respectively; we define bitwise AND "* $\wedge$ *" as follows:*

$$C = \langle A_0 \times B_0, \ldots, A_{k-1} \times B_{k-1} \rangle$$
$$\mathit{Decrypt}(C) = \langle a_0 \wedge b_0, \ldots, a_{k-1} \wedge b_{k-1} \rangle$$

The multiplicative **depth** of bitwise AND evaluations is 1, because there is only one level of multiplication for each ciphertext pair.

Definition 3. *Given a bit list* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and their encryptions with plaintext*

modulus $p = 2$ under an FHE scheme, $A = \langle A_0, \ldots, A_{k-1} \rangle$; we define a chain AND *product as follows:*

$$C = \prod_{i=0}^{k-1} A_i$$

$$\mathit{Decrypt}(C) = \langle a_0 \wedge \ldots \wedge a_{k-1} \rangle$$

The multiplicative **depth** of a chain of AND gates is $\log k$, because we can compute this product in a reverse binary tree structure, where each pair is multiplied on the same level as can be seen in Figure 4.1.

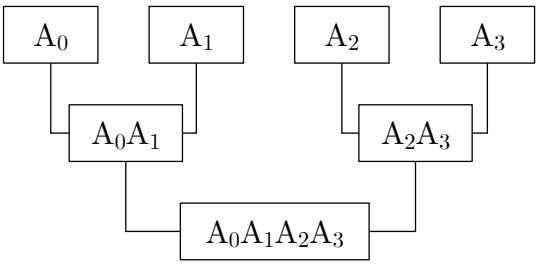


Figure 4.1: Product of 4 ciphertexts in a reverse binary tree structure.

Another useful gate is NOT which flips a bit value. We define it as follows.

Definition 4. *Given binary representation $a \in \{0, 1\}$, and its encryption A with plaintext modulus $p = 2$ under an FHE scheme, we define the* NOT *gate as follows:*

$$C = 1 - A$$

$$\mathit{Decrypt}(C) = \neg a$$

The multiplicative **depth** of NOT evaluation is 0, because there is no ciphertext

multiplication, only a scalar subtraction.

4.2 Binary Addition Circuit

Definition 5. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $B = \langle B_0, \ldots, B_{k-1} \rangle$*, respectively; we define* k*-bit addition as follows:*

$$C = \mathcal{C}_{+,k}(A, B)$$
$$\mathit{Decrypt}(C) = \langle c_0, \ldots, c_k \rangle$$
$$a + b = \sum_{i=0}^{k} 2^i c_i$$

For our binary addition circuit we can use a carry-lookahead adder (CLA) such as Kogge-Stone adder (KSA) that has a $(1 + \log k)$ **depth**.

4.3 Sideways Sum or Hamming Weight

Given k-bits $A_0, \ldots, A_{k-1}$, the integer sum $\sum_{i=0}^{k-1} A_i$ is known as Hamming Weight. Hamming Weight simply counts the number of ones in a bit stream. The base case for Hamming Weight computation is the cases with 2 and 3 inputs, known as Half Adder and Full Adder, respectively. These two circuits are shown in Figure 4.2 and we give the definitions below.

Definition 6. *Given two bits* $a_0, a_1 \in \{0, 1\}$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* A_0, A_1*; we define Half Adder circuit (*HA*) as*

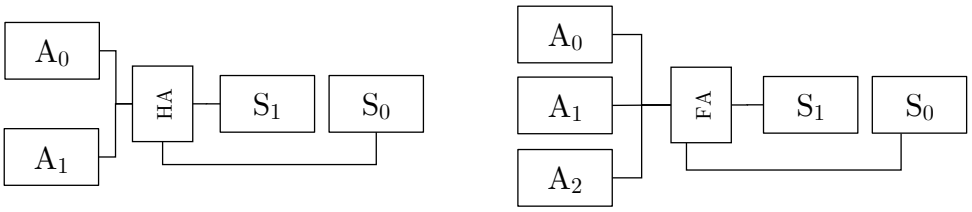


Figure 4.2: Half adder and full adder.

follows:

$$\begin{aligned} S_1, S_0 &= \mathcal{C}_{HA}(A_0, A_1) \\ &= A_0 \times A_1,\ A_0 + A_1 \\ \mathit{Decrypt}(S_1, S_0) &= a_0 \wedge a_1,\ a_0 \oplus a_1 \end{aligned}$$

Definition 7. *Given three bits $a_0, a_1, a_2 \in \{0, 1\}$, and their encryptions with plaintext modulus $p = 2$ under an FHE scheme, A_0, A_1, A_2; we define Full Adder circuit (FA) as follows:*

$$\begin{aligned} S_1, S_0 &= \mathcal{C}_{FA}(A_0, A_1, A_2) \\ &= A_0 \times A_1 + A_0 \times A_2 + A_1 \times A_2,\ A_0 + A_1 + A_2 \\ \mathit{Decrypt}(S_1, S_0) &= (a_0 \wedge a_1) \oplus (a_0 \wedge a_2) \oplus (a_1 \wedge a_2),\ a_0 \oplus a_1 \oplus a_2 \end{aligned}$$

Both FA and HA have multiplicative **depth** one due to the same level ciphertext multiplications. On the other hand, HA has one AND and one XOR gate in total and FA has three AND and four XOR gates. Next, we define the Hamming Weight (HW)

circuit for more than three inputs. In order to build this circuit for different sizes, we make use of HA and FA blocks. An example for four input Hamming Weight circuit can be seen in Figure 4.3.

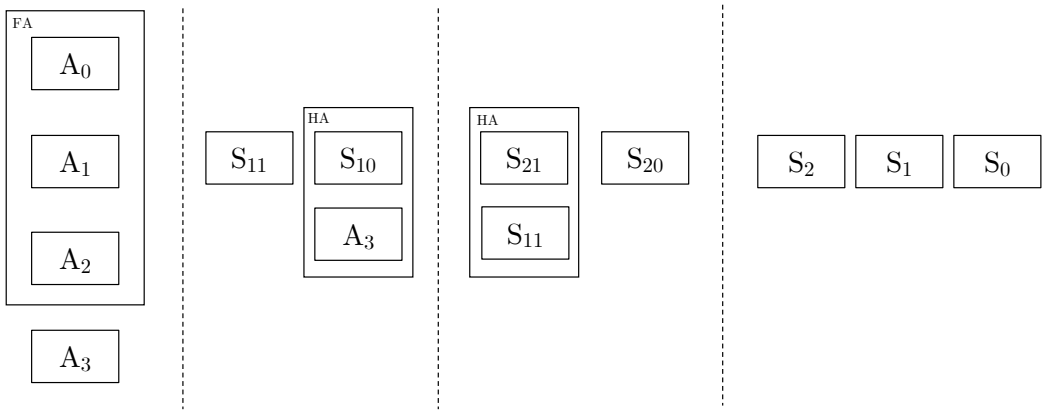


Figure 4.3: Hamming weight computation with 4 inputs.

Definition 8. *Given a bit list* $a = \langle a_0, \ldots, a_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$*; we define* k*-bit sideways sum or Hamming weight as follows:*

$$S = \mathcal{C}_{HW,k}(A)$$

$$\mathit{Decrypt}(S) = \langle s_0, \ldots, s_\ell \rangle, \quad \textit{where } \ell = \lceil \log(k+1) \rceil$$

$$\sum_{i=0}^{k-1} a_i = \sum_{i=0}^{\ell} 2^i s_i$$

Note that, the rule of thumb is to reduce the cost by using the minimum amount of adders, while maintaining the minimum depth. In Figure 4.3, every dotted line represents the next level of computation in the circuit. As can be seen in the figure, a HW circuit with 4 inputs has multiplicative depth three, due to serial FA and HA evaluations. In general this circuit has $\log_{3/2} k$ multiplicative **depth**, since a full adder reduces the number of inputs from 3 to 1, and a half adder reduces the

number of inputs from 2 to 1 in each level.

4.4 Binary Multiplication Circuit

Definition 9. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $B = \langle B_0, \ldots, B_{k-1} \rangle$*, respectively; we define* k*-bit multiplier as follows:*

$$C = \mathcal{C}_{\times,k}(A, B)$$

$$\mathit{Decrypt}(C) = \langle c_0, \ldots, c_k \rangle$$

$$a \times b = \sum_{i=0}^{k} 2^i c_i$$

For multiplication we can build a Wallace tree multiplier [58] using full and half adders and the circuit has at least $\left(1 + \log_{3/2} k/2\right)$ multiplicative **depth**.

4.5 Binary Equality Check

An equality check is a function that tests if two elements are equal to each other. It takes two inputs a and b and returns a one bit output as in Equation 4.1. It is a very fundamental block in ALU as it is very widely used in conditional branches, e.g., *beq – branch on equal* MIPS instruction.

$$f_{\mathsf{EQ}}(a, b) = \begin{cases} 1, & \text{if } a = b; \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Definition 10. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*,*

and their encryptions with plaintext modulus $p = 2$ under an FHE scheme, $A = \langle A_0, \ldots, A_{k-1} \rangle$ and $B = \langle B_0, \ldots, B_{k-1} \rangle$, respectively; we define k-bit equality check circuit as follows:

$$C = \mathcal{C}_{EQ,k}(A, B) = \prod_{i=0}^{k-1} (A_i + B_i + 1)$$

$$\mathit{Decrypt}(C) = c = \prod_{i=0}^{k-1} \neg(a_i \oplus b_i)$$

The product, i.e. the chain of k AND gates may be evaluated in a binary tree structure as in Figure 4.1 which creates a circuit with $\lceil \log k \rceil$ multiplicative **depth**.

4.6 Binary Zero Test

Another useful function, zero test, checks if a given element is equal to zero. It takes bit representation of input a and returns a one-bit output as in Equation 4.2. Note that it is actually equivalent to evaluating the equality check function with inputs a and 0, i.e., $f_{\mathsf{EQ}}(a, 0)$.

$$f_{\mathsf{ZT}}(a) = \begin{cases} 1, & \text{if } a = 0; \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

Definition 11. *Given binary representation of $a = \langle a_0, \ldots, a_{k-1} \rangle$, and its bit encryptions with plaintext modulus $p = 2$ under an FHE scheme, $A = \langle A_0, \ldots, A_{k-1} \rangle$;*

we define k-bit zero test circuit as follows:

$$\begin{aligned} C &= \mathcal{C}_{ZT,k}(A) \\ &= \prod_{i=0}^{k-1}(A_i + 1) \\ \mathsf{Decrypt}(C) = c &= \prod_{i=0}^{k-1} \neg a_i \end{aligned}$$

The multiplicative **depth** of zero test circuit is $\lceil \log k \rceil$.

4.7 Binary Comparison

Similarly, binary comparison is a function which, given two elements, tests if given two elements the first is smaller than the second. It takes two inputs a and b and returns a one-bit output as in Equation 4.3. It is another fundamental ALU block as it is also used in conditional branches, e.g., the *slt – set on less than* MIPS instruction.

$$f_{\mathsf{LT}}(a, b) = \begin{cases} 1, & \text{if } a < b; \\ 0, & \text{otherwise.} \end{cases} \tag{4.3}$$

Definition 12. *Given binary representations* $a = \langle a_0, \ldots, a_{k-1} \rangle$ *and* $b = \langle b_0, \ldots, b_{k-1} \rangle$*, and their encryptions with plaintext modulus* $p = 2$ *under an FHE scheme,* $A = \langle A_0, \ldots, A_{k-1} \rangle$ *and* $B = \langle B_0, \ldots, B_{k-1} \rangle$*, respectively; we define k-bit less than cir-*

cuit as follows:

$$\begin{aligned} C &= \mathcal{C}_{LT,k}(A, B) \\ &= \sum_{i=0}^{k-1} (A_i + 1) \times B_i \times \prod_{j=i+1}^{k-1} (A_j + B_j + 1) \\ \mathit{Decrypt}(C) = c &= \prod_{i=0}^{k-1} \neg a_i \wedge b_i \wedge \prod_{j=i+1}^{k-1} \neg(a_i \oplus b_i) \end{aligned}$$

The multiplicative **depth** of comparison circuit is $\lceil \log(k+1) \rceil$ because the longest multiplication chain, i.e., when $i = 0$ and the product is from $j = 1, \ldots, k-1$, has $k+1$ terms and it can be evaluated in a binary tree structure as in Figure 4.1. Truth tables for comparison and equality check can be found in Appendix A.2.

4.8 Multiplexer or Branching

Multiplexer is another block that is useful for conditional branching. Given a selection bit s and two statements a and b, it makes the decision based on the value of s and assigns the output as in Equation (4.4).

$$f_{\mathsf{MUX}}(s, a, b) = \begin{cases} a, & \text{if } s = 0 \\ b, & \text{otherwise.} \end{cases} \tag{4.4}$$

Definition 13. *Given a select bit $s \in \{0, 1\}$ and any representation of two numbers a and b and their encryptions with plaintext modulus $p = 2$ under an FHE scheme,*

S, A and B, respectively; we define the multiplexer circuit as follows:

$$\begin{aligned} C &= \mathcal{C}_{\mathit{MUX}}(S, A, B) \\ &= (1 - S) \times A + S \times B \\ \mathit{Decrypt}(C) &= \neg s \wedge a \oplus s \wedge b \end{aligned}$$

For instance, we can optimize this multiplexer circuit, by computing

$$\begin{aligned} C &= (1 - S) \times A + S \times B \\ &= A + (B - A) \times S \\ \mathsf{Decrypt}(C) &= a \oplus [(b \oplus \neg a) \wedge s] \ . \end{aligned}$$

With this optimization, we reduce the number of multiplications from two to one.

Chapter 5

Word Arithmetic and Numerical Approximations

With word size message domains we gain the ability to homomorphically multiply and add integers via simple ciphertext multiplications and additions, respectively. This significant gain comes at a severe price. We can no longer homomorphically compute a zero test via direct evaluation of a standard boolean comparator circuit, since the input bits are no longer accessible via our homomorphic evaluation operations. The same applies to more complex operations such as comparison evaluations, thresholding and division. Division, in particular, requires heavy computations and is challenging to evaluate in either bit or higher characteristic encryption. Therefore, it is commonly avoided by selecting division free algorithms or by postponing the computation to the client side after decryption whenever possible.

The focus of this chapter is finding low degree polynomials for functional representations of key primitives using numerical methods. Most of these are approximation algorithms where the precision and number of iterations determine the propagation error. Due to the fact that our message space is limited with respect

to the chosen FHE parameters, especially plaintext modulus, we will use the polynomial encoding technique from Section 3.3 that will provide us a larger plaintext space and allow us to achieve homomorphic fixed-point arithmetic. The key to this approach is the underlying representation, and to this end we use the fixed-point number representation from Section 3.4.

5.1 Multiplicative Inverse and Division

One of the most difficult, and currently open, questions is how to implement homomorphic division efficiently. With bit-level encryption, one could implement a parallel division circuit by unrolling the shift and subtract operations. However the depth of this division circuit would be very high; the best we can do is to use a costly carry-lookahead subtraction circuit and emulate a serial shift division algorithm with depth complexity $\mathcal{O}(n \log(n))$. In the case of higher characteristic, we run into the aforementioned comparison and sign-detection problems.

Our goal is to find a low degree polynomial say $P(x)$ for the multiplicative inverse function $f(x) = x^{-1}$. If we have such a $P(x)$, we also have a division polynomial $D(x, y) = xP(y)$ to compute the ratio x/y. We can construct such a polynomial using three different methods: The first one gives the exact algebraic solution in the ring $\mathbb{Z}_p$, the next two use numerical methods, therefore they output approximated results depending on a fixed precision and preinitialized number of iterations.

5.1.1 Fermat's Little Theorem

Our fist solution is to use Fermat's Little Theorem to compute the multiplicative inverse M^{-1} of a number M, in $\mathbb{Z}_p$. We have $M^{\alpha} \equiv M^{\beta} \mod p$ when $\alpha \equiv \beta \mod \varphi(p)$, and M and p are coprime. If we pick p a prime, then $\varphi(p) = p - 1$

and we can have $M^{-1} \mod p = M^{p-2} \mod p$ for any nonzero[1] $M \in \mathbb{Z}_p$. Thus, we define our first polynomials as:

$$P(x) = x^{p-2} \text{ in } \mathbb{Z}_p \tag{5.1}$$

$$D(x, y) = xy^{p-2} \text{ in } \mathbb{Z}_p \tag{5.2}$$

Lemma 1 (Homomorphic evaluation of division using Fermat's Little Theorem)**.** *Let* $c_1 = \mathsf{Encrypt}(M_1)$*,* $c_2 = \mathsf{Encrypt}(M_2)$ *be two ciphertexts, where* $M_1, M_2 \in \mathbb{Z}_p$ *with* p *prime. If we evaluate* $c = D(c_1, c_2) = c_1 c_2^{p-2}$*, we will have* $M = \mathsf{Decrypt}(c)$ *where* $M = M_1/M_2 \mod p$*.*

As we have a polynomial of degree $p-1$, this method is not very efficient due to the fact that we have to compute a homomorphic exponentiation of multiplicative depth $\mathcal{O}(\log(p))$. Unless p is small, without further customization, this approach will not be very practical. Additionally note that this method does not provide a multiplicative inverse over real numbers but rather in modular arithmetic. On the bright side, the output is an exact arithmetic solution, i.e., there is no approximation, no fractions or levels if precision to estimate. In the next approach we will find the reciprocal, not simply modulo p, but as a real number using a root finding algorithm.

5.1.2 Newton's Root Finding Method

For this approach, we are going to convert our problem into a root-finding problem and use the Newton-Rhapson method to solve it. We define the function $g(z) = 1/z - x$ that has a zero at $z = 1/x$. Thus, we need to find the zero of $g(z)$ to compute the multiplicative inverse function $f(x)$. Newton-Rhapson iterations start

[1]Note that, in case $M = 0$, we will have $P(0) = 0^{p-2} = 0 \mod p$.

with an initial guess z_0 and apply the following update rule:

$$z_{i+1} = z_i - \frac{g(z_i)}{g'(z_i)} = 2z_i - xz_i^2.$$

In Newton's method, the choice of the initial value z_0 is important for the success rate of the algorithm. Unfortunately, in case of homomorphic evaluations where the input is encrypted, we are forced to use a constant as the initial guess. The iterations usually stop when an error criteria is met. However, during homomorphic evaluations the iteration results will still be encrypted. Computing the error and comparing it with the tolerance would require further homomorphic evaluations. Thus, we fix the number of iterations in the initialization step according to the input range and the desired precision. When the input range is $[0, 2^k]$ we set the initial value to the mid-point, $z_0 = 2^{1-k}$. The approximated values can be seen in Figure 5.1 with a constant number of iterations $\eta = 5$. The resulting polynomials for the same fixed parameters can be seen in Equation 5.3 and Equation 5.4.

$$P(x) = -6.84 \times 10^{-49}x^{31} + 7.01 \times 10^{-46}x^{30} + \cdots - 0.48x + 1 \tag{5.3}$$

$$D(x, y) = -6.84 \times 10^{-49}xy^{31} + 7.01 \times 10^{-46}xy^{30} + \cdots - 0.48xy + x \tag{5.4}$$

Since this approximation technique uses rational numbers, we need to apply the encoding technique from Section 3.3 to the polynomial coefficients. The arithmetic is performed following the fixed-point rules.

Lemma 2 (Homomorphic evaluation of division using Newton-Rhapson root finding algorithm)**.** *Let* $(c_1, 0) = \mathsf{Encrypt}(m_1)$ *and* $(c_2, 0) = \mathsf{Encrypt}(m_2)$ *be two ciphertexts with* m_1, m_2 *binary encodings of two integers* M_1, M_2 *in the range* $[0, 2^k]$*. Given the polynomial* $D(x, y)$ *of degree* $2^\eta - 1$ *constructed from Newton-Rhapson iterations*

where η is the number of iterations, we have $\mathsf{Decrypt}(D((c_1, 0), (c_2, 0))) = M_1/M_2$.

The depth of this approximation depends on the number of iterations η, i.e., it is independent of plaintext modulus p. Consider the equation $z_{i+1} = 2z_i - z_i^2 x$, the depth of the circuit for $P(x)$ comes from the product $z_i^2 x$. Initially z_0 is a constant, hence the exponent of x in z_1 becomes 1. In the next iterations, the exponent of x will be $3, 7, \ldots$. Thus, after η iterations, the exponent will be $2^\eta - 1$ and the circuit depth is η. This gives a great advantage over the approach based on Fermat's Little Theorem, when the inputs come from a small subset of the plaintext space (assuming $\eta < \log(p)$). Note that the algorithm is flexible in the sense that we can keep iterating to increase the precision, or terminate early if less precision suffices for the application. Once the iterations have been completed, the precision has changed where the most significant $\log(k^{2^\eta})$ bits of the result represent the desired reciprocal. This means that any further computation requires other operands that will interact with the reciprocal need to be shifted to align with the segment representing the fractional part.

5.1.3 Goldschmidt's Convergence Method

We briefly and informally describe how to find the inverse by convergence as follows: Assume we want to compute the reciprocal $1/M$. The algorithm works by multiplying both the numerator and denominator by a series of values $R_1, R_2, \ldots$ so as to make the denominator converge to 1. Thus, at the end of the computation the numerator yields the desired division result:

$$\frac{1}{M} = \frac{1}{M} \cdot \frac{R_1}{R_1} \cdot \frac{R_2}{R_2} \cdots \frac{R_\eta}{R_\eta}, \qquad M R_1 \cdots R_\eta \to 1 \ .$$

The simplified binomial approach starts by scaling M to a fraction in the unit interval; i.e., to $M/2^k \in (\frac{1}{2}, 1]$. Let $\bar{M} = M/2^k$. Then we choose $z = 1 - \bar{M}$ and set $R_i = 1 + z^{2^{i-1}}$. This will yield the desired result.

$$\begin{aligned}
\frac{1}{\bar{M}} &= \frac{1}{\bar{M}} \cdot \frac{R_1}{R_1} \cdot \frac{R_2}{R_2} \cdots \frac{R_\eta}{R_\eta} \\
&= \frac{1}{1-z} \cdot \frac{R_1}{1+z} \cdot \frac{R_2}{1+z^2} \cdots \frac{R_\eta}{1+z^{2^{\eta-1}}} \\
&= \frac{R_1 \cdots R_\eta}{1 - z^{2^\eta} \approx 1} \\
&\approx R_1 \cdots R_\eta \\
&= \prod_{i=1}^{\eta} \left(1 + z^{2^i}\right) = \sum_{i=1}^{2^\eta} z^{i-1} = \sum_{i=1}^{2^\eta} \left(1 - \bar{M}\right)^{i-1} \\
\frac{1}{M} &= \frac{1}{2^k} \sum_{i=1}^{2^\eta} \left(1 - \bar{M}\right)^{i-1}
\end{aligned}$$

Then this convergence method gives us the inverse polynomial in Equation (5.5) and division polynomial in Equation (5.6):

$$P(x) = 2^{-k} \sum_{i=0}^{2^\eta - 1} (1-x)^i \tag{5.5}$$

$$D(x, y) = 2^{-k} x \sum_{i=0}^{2^\eta - 1} (1-y)^i \tag{5.6}$$

We can show that $M \cdot R_1 \in [1 - 2^{-2}, 1]$, $M \cdot R_1 R_2 \in [1 - 2^{-4}, 1]$, $M \cdot R_1 R_2 R_3 \in [1 - 2^{-8}, 1]$, etc., with products $M \cdot R_1 \cdots R_\eta$ converging to one. The approximated inverse values for a chosen η can be seen in Figure 5.1. The resulting polynomial for the same parameters is given in Equation (5.7).

$$P(x) = 0.49x^{31} - 7.75x^{30} + \cdots - 7.75x + 0.5 \tag{5.7}$$

We can mimic the inversion by convergence algorithm to effect a homomorphic di-

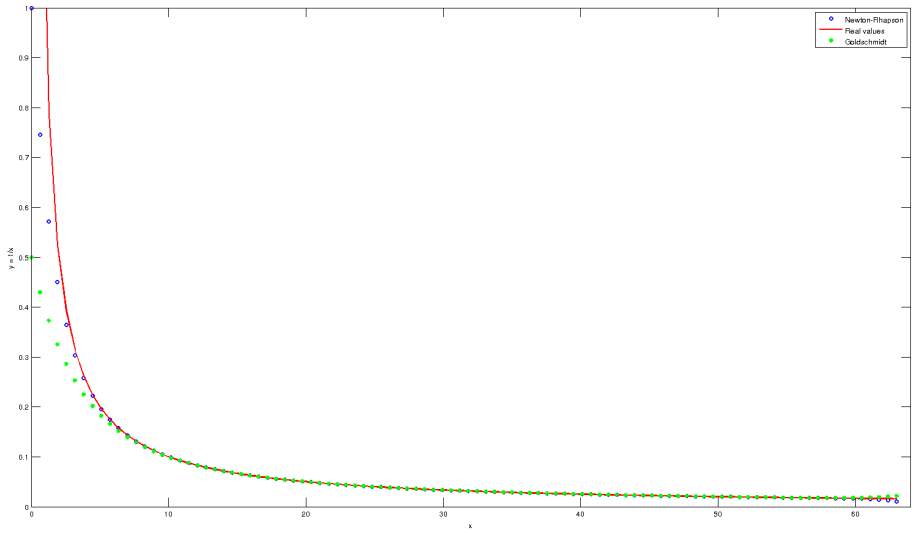


Figure 5.1: Computing inverse function by using: Newton-Rhapson root finding algorithm, with inputs $M \in [0, 2^6]$, initial guess $z_0 = 2^{-5}$, number of iterations $\eta = 5$ and Goldschmidt convergence method with inputs $M \in [0, 2^6]$ and number of factors $\eta = 5$.

vision operation. Our variable precision encoding associates the ordered pair (M, k) to scaled value $\bar{M} = M/2^k$.

Lemma 3 (Homomorphic evaluation of division using Goldschmidt's convergence method)**.** *Let* $(c_1, 0) = \mathsf{Encrypt}(m_1)$ *and* $(c_2, k) = \mathsf{Encrypt}(m_2)$ *be two ciphertexts with* m_1, m_2 *binary encodings of two integers* $M_1 \in [0, 2^k], M_2 \in [2^{k-1}, 2^k]$*. Given the polynomial* D *constructed using Goldschmidt's convergence method,* $D(x, y)$ *of degree* $2^\eta - 1$ *in* y *where* η *is the number of factors, we have*

$$\mathsf{Decrypt}(D((c_1, 0), (c_2, k))) = M_1/M_2 \ .$$

The polynomial $P(x)$ has degree $2^\eta - 1$ in x requiring a circuit of depth η. As in the previous approximation method, this is also independent of the plaintext mod-

ulus p. Both algorithms suffer from the growth in the fractions; that is, if we do not use an encoding technique, p should be chosen large enough to cover the magnitude of the end result in order to avoid overflows. Even with small precision, after a few iteration steps we end up with a large fraction. This is a generic problem in any approximation-based algorithm where we have to use real numbers. This signifies the importance of using the encoding technique in our approximation methods.[2]

5.2 Zero Test and Equality Check

We can now obtain a polynomial function that permits homomorphic evaluation of a zero test. The test returns a zero or one depending on whether or not the ciphertext is (or rounds to) an encryption of zero. Let this polynomial be $Z(x)$. Then we want to have $Z(a) = 0$ if a is equal to zero, $Z(a) = 1$ otherwise. We can retrieve this functionality using Fermat's Little Theorem by computing x^{p-1} mod p. This can be interpreted as multiplying the input x with its inverse x^{p-2} mod p or dividing a number by itself. In standard arithmetic division by zero is undefined. However, in our approximation methods, multiplicative inverse of zero is defined and to be precise it is 1 or 1/2 depending on the algorithm. Therefore, we can create a zero test polynomial by using the division polynomial as follows: $Z(x) = D(x, x) = xP(x)$. The output is 0 or 1 with some error towards the ends of the range, depending on the chosen inverse finding method. An approximation to zero test is given in Figure 5.2.

The zero test may be used trivially to homomorphically perform an equality check on two messages M_1, M_2 by computing $Z(M_1 - M_2)$. So we can define an equality check polynomial as $E(x, y) = Z(x - y)$. Note that this is a much simpler

[2]We could use RNS method for algorithms that require large p, but that would time and space complexity of the algorithm.

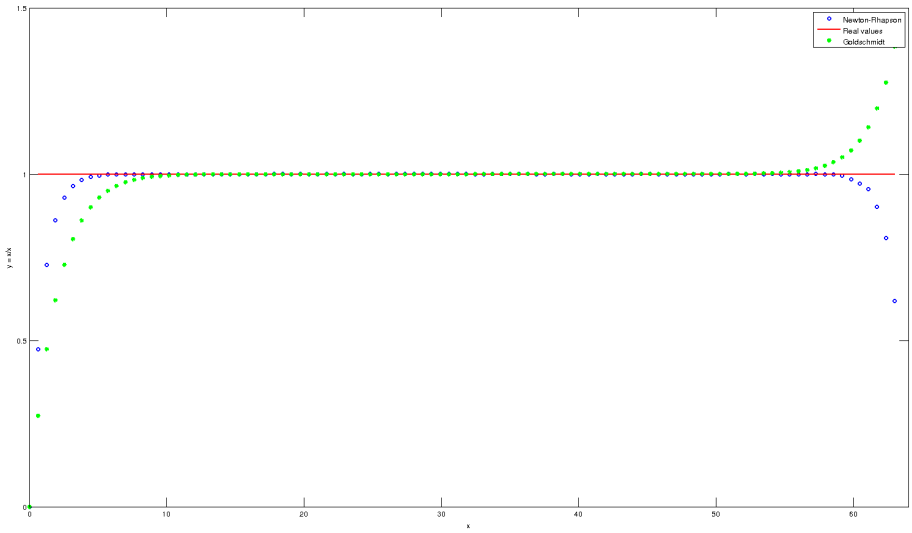


Figure 5.2: Checking if a value is zero by using the division polynomial with: Newton-Rhapson root finding algorithm, with inputs $M \in [0, 2^6]$, initial guess $z_0 = 2^{-5}$, number of iterations $\eta = 5$ and Goldschmidt convergence method with inputs $M \in [0, 2^6]$ and number of factors $\eta = 5$.

operation than magnitude comparison which we will address later in Section 5.3.

Lemma 4 (Zero Test and Equality Check)**.** *Let* $(c, 0) = \mathsf{Encrypt}(m)$ *be a ciphertext with* m *as the binary encoding of an integer* $M \in [0, 2^k]$*. Then computing* $\mathsf{Decrypt}(Z((c, 0)))$ *will gives us* ≈ 1 *if* $m \neq 0$ *and* 0 *otherwise.*

The degree of $Z(x)$ is equal to the degree of division polynomial $D(x, y)$. Thus, the complexity of a zero test depends on the chosen inverse polynomial.

5.3 Comparison and Thresholding

Since a truly homomorphic computation of a function $f(x_1, \ldots, x_n)$ can reveal no information about the operands $x_1, \ldots, x_n$, we cannot expect fast implementations of any sort of branching in our computation: all computational paths must be

indistinguishable to the party carrying out the computation. Yet, we must find efficiencies wherever we can.

Using the zero test we can compute thresholding operations easily albeit inefficiently. Assume we want to homomorphically evaluate the check $b \leq t$ for some data b and threshold $t \in \mathbb{Z}_p$. As earlier we are given the encryption of b while t is presumed available as cleartext and again we are seeking a polynomial to represent this operation. Let it be $T(x,t)$, then we want $T(a,t) = 0$ when $a < t$ whereas $T(a,t) = 1$ otherwise. We can devise this algorithm by testing the equality over the range of integers $i = 0, \ldots, t-1$ and aggregate[3] the result as $T(x,t) = \sum_{i\in[t]} (\omega - Z(x-i))$ where $Z(x)$ is a zero test polynomial as described in the previous section. Clearly, we can instead compute the complement if t is closer to p than to 0.

If we compute it using Fermat's Little Theorem, although it is not efficient, this presents a viable and exact technique for evaluating thresholds. A significant positive aspect of the formulation is that the multiplicative depth of the threshold computation is independent of the threshold constant t and is the same as the depth of an equality check: $\mathcal{O}(\log(p))$. On the other hand, the summation becomes computationally expensive – with complexity $\mathcal{O}(t\log(p))$ – as p and the range of t grow. Lookup tables and selection of special moduli can be used to increase the efficiency.

We return to the instantiation with an integer modulus p for a moment. Unless p is small without further customization this approach will not be very practical. To gain some economy over the prime p case, we may chose p to be highly composite $p = \prod_{i\in[k]} p_i$ in such a way that the zero test simply becomes $c^{\varphi(p)} = c^{\prod \varphi(p_i)}$. Then the multiplicative depth complexity of a zero test (or comparison) becomes $\sum_{i\in[k]} \log(p_i - 1)$.

[3]Since the zero tests are exclusive, we may aggregate the result using a standard homomorphic addition operation instead of a boolean OR.

Approximation Methods

In order to retrieve a threshold polynomial $T(x)$, we will make use of the Unit Step Function, i.e $H(x) = 0$ when $x < 0$ and $H(x) = 1$ when $x > 0$, then we can just compute $T(x) = H(x - t)$ where t is a fixed cleartext threshold. Furthermore, the same polynomial can be used to compare two encrypted values a, b by computing $H(a - b)$. We propose two different methods to create such a step function.

For the first approach we will make use of logistic function and the equation is given as follows, $H(x) = \lim_{k \mapsto \infty} \frac{1}{1+e^{-2kx}} \approxeq (e^x)^{2k} \left(1 + (e^x)^{2k}\right)^{-1}$. By limiting k to a small constant, we can get a smooth approximation and we can in turn use a Taylor Series approximation for the exponential function $e^x \approxeq \sum_{i=1}^{\infty} \frac{x^i}{i!}$. We can also use one of the inverse functions P that we found in Section 5.1 to handle the denominator so that $H(x)$ becomes: $H(x) \approxeq \left(\sum_{i=1}^{\infty} \frac{x^i}{i!}\right)^{2\kappa} P\left(1 + \left(\sum_{i=1}^{\infty} \frac{x^i}{i!}\right)^{2\kappa}\right)$. Even though we have obtained a threshold polynomial using this approach, it is computationally expensive considering the input to the inverse function has already a large exponent. Therefore, we use another approach: approximating a square wave using sine waves. The standard square wave function $S(x)$ can be approximated as $S(x) \approxeq \sum_{i=1}^{\infty} \frac{\sin((2i-1)x)}{(2i-1)}$ For sinus values we can use the approximation $\sin(x) \approxeq \sum_{j=1}^{\infty} \frac{(-1)^{j-1} x^{2j-1}}{(2j-1)!}$. Embedding this in the previous equation we have $S(x) \approxeq \sum_{j=1}^{\infty} \sum_{i=1}^{\infty} \frac{(-1)^{j-1}(2i-1)^{2j-2}}{(2j-1)!} x^{2j-1}$.

The output of the square wave function is in the range of $[-0.8, 0.8]$ in a period, thus we compute $H(x)$ as: $\frac{S(x)+0.8}{1.6}$. The degree of $H(x)$ depends on the upper limit for j. If we define $i \in [1, \alpha]$ and $j \in [1, \beta]$, then the largest exponent of input x (i.e., the degree of H) becomes $2\beta - 1$. Consequently, the depth of the approximation algorithm becomes $\lceil \log(2\beta - 1) \rceil = \lceil \log \beta \rceil + 1$. For different values of α and β the unit step approximation can be seen in Figure 5.3.

To make use of this approximation algorithm, we also need to associate message

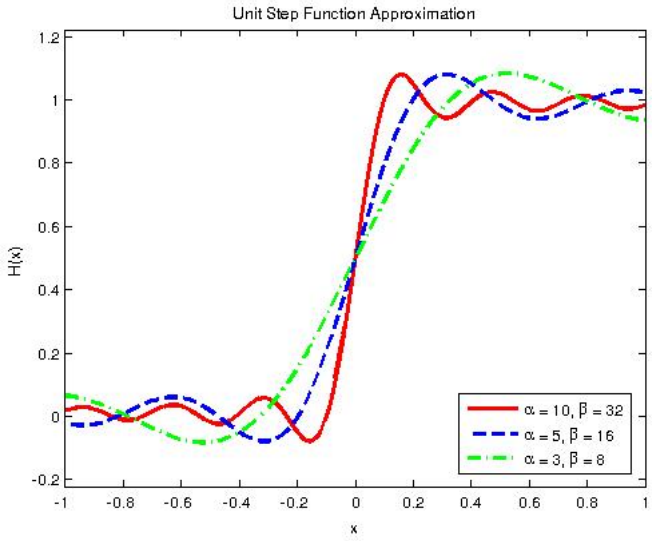


Figure 5.3: Unit step function $H(x)$ for various approximation degrees.

space elements to discrete samples of the input range $[-1, 1]$ of $H(x)$. Assume we handle elements of precision ℓ bits and we want to find $H(b-t)$, where $b, t \in [0, 2^\ell)$. Then we have an input $x = b - t \in \left(-2^\ell, 2^\ell\right)$ and we have to normalize it by a factor of $\omega = 2^\ell$ so that the normalized value lies in the input range: i.e. $x/\omega \in (-1, 1)$. As in the previous approximation methods, we need to represent ℓ fractional bits with a binary point placed right after leaving a single bit for the integer part. During evaluation we need to keep track of the precision point which moves to the left with each multiplication by x. Once the evaluation is completed, the approximation result resides in the most significant precision bit(s) ready to be used for subsequent evaluation and the maximum number of fraction bits can be found in the term with the highest exponent, $\omega^{2\beta-1}$.

5.4 Square Root

We can find an approximation to the square root of a number by using a root finding algorithm. As before, we seek a polynomial, $R(x)$ say, such that $R(b) = \sqrt{b}$. The

function $f(y) = y^2 - b$ has a root at $y = \sqrt{b}$, hence if we can approximate the root of $f(y)$, we obtain an estimate for the square root of b. If we use Newton's Root Finding method as in Section 5.1, we can iterate through the values $y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} = \frac{1}{2}\left(y_i + \frac{b}{y_i}\right)$ with an initial guess of y_0. For the inverse computation $\frac{b}{y_i}$, we can use the inverse approximation polynomial that we retrieved before, now writing $y_{i+1} = \frac{1}{2}\left(y_i + bP(y_i)\right)$. In order to handle fractions, again we need to consider a hypothetical precision point. The depth of the algorithm depends on the number of iterations, κ say; then total depth will be κ times the depth of the inverse computation $P(x)$. Thus this is a much more costly operation relative to inversion.

5.5 Comparison with Binary Artihmetic

In this section, we review of all the methods proposed above, with comparison to their binary equivalents. These operations include *addition* $(+)$, *multiplication* $(*)$, *division* $(/)$, *equality check* $(=)$ and *comparison* $(<)$.

For binary addition we can use a parallel prefix adder such as Kogge-Stone that has a $(1 + \log k)$ depth where k is the bit size of the inputs. For multiplication we can build a Wallace tree multiplier using full and half adders and the circuit has at least $\left(1 + \log_{3/2} k/2\right)$ multiplicative depth. Both addition and multiplication are trivial operations in the word domain. Ciphertext addition does not increase the noise significantly, thus it does not have an effect on the circuit depth. Multiplication increases the circuit depth by adding only one level. Division is by far the most costly of the four arithmetical operations on binary domains. In order to divide a $2k$ bit number by a k bit divisor, we can build a binary division circuit that involves k cycles of conditional k-bit subtractions. For subtraction, we can use the parallel prefix adders with a delay of $1 + \log k$. The condition statement adds one level in

each step, thus resulting in an overall circuit depth of $k\,(2 + \log k)$. For the last two operations we use simple boolean circuits from Chapter 4, where the equality check has $\lceil \log(k) \rceil$ and the less than check has $\lceil \log(k+1) \rceil$ depth.

For 32-bit integer inputs, the parameters that are used in F-NTRU setup can be seen in Table 5.1.

	Binary					**Wordwise**				
Parameters	$+$	$*$	$=$	$<$	$/$	$+$	$*$	$=$	$<$	$/$
p	2	2	2	2	2	$x-2$	$x-2$	$x-2$	$x-2$	$x-2$
$\boldsymbol{d}$	**6**	**10**	**5**	**6**	**96**	**–**	**1**	**6**	**5**	**5**
$\log q$	23	24	23	23	28	190	190	190	190	190
n	8190	8190	8190	8190	131070	4096	4096	4096	4096	4096

Table 5.1: F-NTRU parameters for bit-wise and word-wise encryption. Key to parameters: p: plaintext modulus; d: multiplicative depth of the circuit; $\log q$: bit size of the coefficient modulus; n: degree of the polynomial ring; δ: Hermite factor with respect to q and n.

5.6 Implementation Results

We implemented the proposed division, zero test, thresholding and comparison algorithms using the F-NTRU scheme using Shoup's NTL library version 9.6.3 [54] compiled with the GMP 6.1 package. Our simulations are performed on an Intel(R) Xeon(R) server with CPU E5-2637 v2 @ 3.50GHz running Fedora release 21 with 4 cores and 8 threads. Note that the proposed homomorphic algebraic operations in this chapter are generic, i.e., they can be implemented using any FHE scheme that supports word size encryption. For parameter selection, we followed the security analysis from [24] and utilized 80-bit security. We used modulus polynomial $\Phi(x) = x^n + 1$ with $n = 4192$ and our coefficient modulus size is $\ell = \log q = 190$. For the details of noise and security analysis and parameter selection, we refer the reader to [24].

For the first test, we evaluated word-wise addition and multiplications. A single 32-bit addition takes 0.25 milliseconds and a single 32-bit multiplication runs around 33 milliseconds. Note that when we evaluate multiplication, we take advantage of the one-sided multiplication of the F-NTRU scheme. Secondly, we evaluated the division circuits for the two proposed methods. For Newton's method with $\eta = 5$ and inputs in the range $[0, 64]$, we used the polynomial from Equation (5.4). In this test, we have a total execution time of 1.05 seconds. Next, we evaluated Goldschmidt's division by convergence algorithm for $\eta = 5$ and inputs in the range $[0, 64]$, we used the polynomial from Equation (5.6). In this test, we have a similar total execution time of 1.03 seconds, but the error rate for Newton's method is smaller. We also evaluated the equality checks (and/or zero check) using both Fermat's Little Theorem and division by Newton's root finding method. For Fermat's method, we set plaintext modulus to two different values $p = 17$ and $p = 257$. For Newton's method, we used the same division polynomial. For the last test, we computed a comparison with inputs in the range $[0, 32]$ and $\alpha = 5$, $\beta = 16$. Total execution times can be seen in Table 5.2.

Operation	**Algorithm**	$(d, \log(q), n)$	**Total Time**
Addition	-	$(0, 190, 4096)$	0.25 ms
Multiplication	-	$(1, 190, 4096)$	33 ms
Division	Newton-Rhapson	$(5, 190, 4096)$	1.05 sec
	Goldschmidt	$(5, 190, 4096)$	1.03 sec
Equality Check	Newton-Rhapson	$(5, 190, 4096)$	1.13 sec
	Goldschmidt	$(5, 190, 4096)$	1.11 sec

Table 5.2: Parameters and timings for: **Division** first using Newton-Rhapson root finding, then Goldschmidt convergence algorithm for encoded data using $p = x - 2$; **Equality Check** using Fermat's Little Theorem with a single message no encoding with $p = 17$ and $p = 257$; then using Newton-Rhapson root finding and Goldschmidt convergence algorithm for encoded data using $p = x - 2$; **Comparison** using Square Wave approximation for encoded data using $p = x - 2$.

Chapter 6

Sorting

Sorting is one of the most natural and crucial tasks in computing. Numerous sorting algorithms have been proposed in the literature [42]. These algorithms have been heavily investigated and characterized according to their time and space requirements, as well as to the degree of their suitability for parallelization. As far as homomorphic evaluation is concerned we have another requirement. Since most of the FHE and SWHE schemes are designed to evaluate circuits, and do not scale well when the multiplicative depth of the circuit is high, we need to add another metric — namely multiplicative circuit depth — in our selection of a homomorphic sorting scheme. For this we need to first convert the serial sorting algorithm into a circuit by unrolling loops and eliminating conditional assignments by *arithmetization*. In this chapter, the term "circuit depth" is used in lieu of multiplicative depth of the circuit and it should not be confused with "comparison depth", i.e. depth of the circuit measured in terms of comparative levels, which is used in the analysis of classical sorting algorithms in the literature.

A sorting network is a circuit which consists of comparators and swapping operations. The difference between classical comparison-based sorting algorithms and

sorting networks is that all operations are set in advance, which means that there is no data dependency in the flow of the algorithm steps in sorting networks. Since we are trying to sort encrypted inputs, we are, in a way, blind in each step of the algorithm. As a result, even though data dependent algorithms may be faster and more efficient over raw data, being independent from the input makes sorting networks the only candidates for FHE sorting. While there are some algorithms specifically designed as a sorting network, some classical sorting algorithms can also be represented as a network as FHE properties require. Hence we will go over some well known algorithms and give the depth complexity of the corresponding sorting networks.

Homomorphic sorting is an operation that blindly sorts a given set of encrypted numbers without decrypting them (thus, there is no need for the secret key). In this chapter, we propose two new algorithms specifically designed for the homomorphic sorting operation, and explain how they evolve from classical sorting networks: *direct and greedy sort* algorithms. Later, we define another efficient, and scalable method for homomorphic sorting of numbers: the *polynomial rank sort* algorithm. To put the new algorithms in a comparative perspective, we provide an extensive survey of classical sorting algorithms and networks that are not directly suitable for homomorphic computations. We also include, in our discussions, that the new algorithm is superior in terms of multiplicative depth and the number of multiplications, when compared with all other algorithms. When batched implementation is used, the number of comparisons is reduced from $\mathcal{O}(N^2)$ to a constant multiple of N provided that the number of slots is larger than or equal to the number of elements in the set. Our software implementation results confirm that the new algorithm is several orders of magnitude faster than many methods in the literature. Also, the polynomial sort algorithm scales better than the fastest algorithm in the literature

to the best our knowledge although for small sets the execution times are comparable. The proposed algorithm is amenable to parallel implementation as most time consuming operations in the algorithm can naturally be performed concurrently.

6.1 Problem Definition

Homomorphic Sorting Problem: Given an unordered set of encrypted elements $\{A_0, A_1, \cdots, A_{N-1}\}$, where A_i is an encryption of plaintext a_i for each i, we want to find an ordered set of encrypted elements
$\{B_0, B_1 \cdots, B_{N-1}\}$, provided that the decrypted list $\{b_0, \cdots, b_{N-1}\}$ is a permutation of $\{a_0, \cdots, a_{N-1}\}$ with nondecreasing order, i.e. $b_0 \leq b_1 \leq \cdots \leq b_{N-1}$.

Here we define a crucial building block that is used in most sorting methods; Compare and Swap (CAS). Given a pair (a, b), a CAS block compares the values and reorders them so that the smaller element in the pair always comes first as in Equation 6.1.

$$f_{\mathsf{CAS}}(a, b) = \begin{cases} (a, b), & \text{if } a < b \\ (b, a), & \text{otherwise.} \end{cases} \tag{6.1}$$

We can define a homomorphic evaluation circuit for CAS by using a less than comparator and sending its output to a multiplexer as the selection bit. This would give us the maximum of the two in the pair. If we use the same select bit, but with swapped elements this would give us the minimum of the two in the pair. Therefore, we define $\mathcal{C}_{\mathsf{CAS}}$ as follows:

Definition 14. *Given binary representation of two numbers a and b and their encryptions with plaintext modulus $p = 2$ under an FHE scheme, A and B, respectively;*

we define the compare and swap circuit as follows:

$$
\begin{aligned}
(C_1, C_2) &= \mathcal{C}_{\mathsf{CAS}}(A, B) \\
&= (\mathcal{C}_{\mathsf{MUX}}(S, B, A), \mathcal{C}_{\mathsf{MUX}}(S, A, B)) \text{ , where } S = \mathcal{C}_{\mathsf{LT}}(A, B) \\
(\mathsf{Decrypt}(C_1), \mathsf{Decrypt}(C_2)) &= (\neg s \wedge b) \oplus (s \wedge a) \, , (\neg s \wedge a) \oplus (s \wedge b) \text{ , where } s = (a < b), \\
&\quad \text{and } \neg s = (a \geq b)
\end{aligned}
$$

For the rest of this chapter, we use the symbol $\lessdot$ to represent encrypted less than circuit, and $\asymp$ to represent homomorphic equality check circuit for better readability, i.e. $A \lessdot B = \mathcal{C}_{\mathsf{LT}}(A, B)$ and $A \asymp B = \mathcal{C}_{\mathsf{EQ}}(A, B)$.

6.2 Known Sorting Algorithms

6.2.1 Bubble Sort

The Bubble Sort algorithm is one of the simplest sorting techniques that permits a rather straightforward implementation using only primitive comparison and swap operations. Chatterjee et al. [39] design homomorphic conditional swap circuits to facilitate homomorphic evaluation of the Bubble Sort algorithm. Very briefly, the sorting algorithm works by making passes over the array of numbers. In each pass the elements are pairwise compared and swapped to move the smaller element to the left (in case of a horizontal array). The average and worst case performance for an array of N elements are the same: $\mathcal{O}(N^2)$. Since we have no way of knowing when the array is sorted for a possible early termination during homomorphic evaluation, we need to make $N - 1$ passes over the array always arriving at the worst case complexity. Since one more element in the rightmost part of the array is sorted in every pass, the number of elements to be compared decreases by one in the next

pass. Thus, overall we will have $[(N-1)+(N-2)+\ldots+1] = (N^2-N)/2$ CAS blocks and the depth of the Bubble Sort circuit will be $[(N^2-N)/2]d_{\mathsf{CAS}}$. Considering ℓ-bit wide array elements, we have $d_{\text{BS}} = \mathcal{O}(N^2 \log(\ell))$ for the depth of the homomorphic Bubble Sort algorithm. We can gain some economy by not waiting to start the next pass until a pass is finished. We can *overlap* the passes which creates a network version of Bubble Sort, known as Odd Even Sort, detailed in the next section. [1]

6.2.2 Odd-Even Sort

A trellis shaped circuit arrangement of Bubble Sort is known as Odd-Even Sort. The circuit admits N inputs and computes the N sorted output values after N passes. In the first pass, a zero-indexed array being adopted, every even indexed element is compared and swapped with its right neighbor. In the second pass, every odd indexed element is compared and swapped with its right neighbor. Considering these two steps as a round, the identical operations are applied in each round. The total number of comparisons is $N-1$ in each round, there are N passes which means $N/2$ rounds and so overall, there are $N(N-1)/2$ comparators. And the depth of the circuit is Nd_{CAS}. Therefore $d_{\text{OES}} = \mathcal{O}(N \log(\ell))$. For more details about this sorting network, we refer readers to [42].

6.2.3 Insertion Sort

Insertion sort is a simple algorithm that iteratively builds a sorted array from an unsorted one. The sorted array initially holds only the first element of the unsorted array. Then the remaining elements of the unsorted array are added one by one to the sorted array by comparing the element from right to left with the elements in

[1] Note that in their implementation Chatterjee et al. [39] perform the comparison using a carry propagate adder based subtraction circuit, resulting in a circuit depth of $(N^2-N)(\ell+1)/2$. While the computational complexity of the scheme is low, the $\mathcal{O}(N^2)$ circuit depth is prohibitive.

the sorted array until an element smaller is encountered. The new element is then inserted into the sorted array next to the first smaller element. The average case and the worst case complexity of the algorithm is $\mathcal{O}(N^2)$ while the best case is only $\mathcal{O}(N)$. When considered as a circuit for homomorphic evaluation we need to run the algorithm with the worst case complexity, with no possible early termination as in the case of Bubble Sort. As we build up the sorted array one by one, the number of comparisons and swap operations increment by one for the next element. We obtain a circuit depth of $[1 + 2 + \ldots + N - 1]d_{\mathsf{CAS}} = (N^2 - N)/(2)d_{\mathsf{CAS}}$. Therefore, we have $d_{\mathrm{INS}} = \mathcal{O}(N^2 \log(\ell))$. This circuit can be used in a more efficient way by overlapping some comparisons, similar to Bubble Sort. Consequently we can see that, Insertion Sort and Bubble Sort end up with the same construction, when they are formulated as sorting networks.

In [39], Chatterjee et al. rely on the fact that after the *imperfect* application of Bubble Sort the array is *nearly* sorted. Thus Insertion Sort performs nearly in linear time. But even if the array is *nearly* sorted, the algorithm should run as in the worst case, since we do not have any knowledge of the misplaced elements.

6.2.4 Merge Sort

Merge Sort is an asymptotically faster algorithm and allows early termination in normal execution, which reduces its complexity. The algorithm is recursively applied by partitioning arrays into smaller ones. In the innermost recursion, arrays of two elements are sorted, where only one comparison is needed in one sub-array. In the merging step, which combines two individually sorted arrays into a single sorted array, at most three comparisons are applied in each partition. This eventually leads to $\mathcal{O}(N \log(N))$ comparisons in the worst case. But in our case, the merging step requires many more comparisons, due to algorithm's input dependent nature

and our lack of input knowledge. For instance, in the classic Merge Sort, to merge two sub-arrays each of size two, we follow one of the paths until all the elements are placed in the sorted sub-array of size four. Let our output array be B in a merge step. Then, if the Boolean expression $(A_0 \lessdot A_1)$ returns 1; we can conclude that $B_0 = A_0$, otherwise $B_0 = A_1$. But in homomorphic sorting, we cannot follow any specific path as the output of each $A_i \lessdot A_j$ block is also encrypted. Hence, we need to consider every single possible outcome of all comparison operations, i.e. every single path, which eventually necessitates comparing every possible pair.

In summary, we need to perform $(N^2 - N)/2$ comparisons to sort an array of N elements. On the other hand, since there is no swapping, i.e. no data dependency, during the execution of a single merge step, we can compute all of the comparisons in parallel at the beginning of each merge step. Consequently, applying all comparison operations before every merge step simply alters the algorithm and we end up with a totally different scheme from the classical Merge Sort algorithm. Inspired from the analysis of Merge Sort, we introduced two new sorting circuits, with the same number of comparators $\mathcal{O}(N^2)$ and the total comparison depth of $\mathcal{O}(1)$.

6.3 Proposed Sorting Algorithms

Given the inadequacies of existing sorting algorithms in permitting shallow circuit evaluation, we develop three new sorting algorithms, Greedy Sort, Direct Sort and Polynomial Rank Sort, optimized for this purpose.

6.3.1 Greedy Sort

We propose the Greedy Sort algorithm as an alternative method to classical sorting algorithms, due to its low circuit depth. However, due to the algorithm's exhaus-

tive nature in terms of number of comparisons, the method is not efficient when implemented without optimizations.

Greedy Sort works by performing every possible pairwise comparison operations at once for a given input set of elements. In order to find the minimum element b_0, one needs to compare each a_i to every other element in the set. If it is smaller than all a_js where $i \neq j$, then we can conclude it is the smallest element and set $b_0 = a_i$. Similarly, in order to find the next smallest element b_1, we need to compare every a_i to every other element and if it is smaller than all but one, then we know that it is the second minimum and set $b_1 = a_i$. We can follow the same idea until the last element of the sorted array b_{n-1}, the maximum element, is found.

We can express the conditions yielding the minimum element explicitly as in Algorithm 1, and the test conditions for finding the second smallest element is given in Algorithm 2. The if-else statements give us an exact mutually exclusive partitioning in the output assignments, hence we can use XOR ($\oplus$) gates to combine each statement.

Algorithm 1 Finding the minimum element in a set.

1: **if** $(a_0 < a_1) \ \wedge \ (a_0 < a_2) \ \wedge \ \ldots \ \wedge \ (a_0 < a_{N-1})$ **then**
2: $\quad b_0 = a_0$
3: **else if** $\neg(a_0 < a_1) \ \wedge \ (a_1 < a_2) \ \wedge \ \ldots \ \wedge \ (a_1 < a_{N-1})$ **then**
4: $\quad b_0 = a_1$
5: **else if** $\ldots$ **then**
$\vdots$
6:
7: **else if** $\neg(a_0 < a_1) \ \wedge \ \neg(a_1 < a_2) \ \wedge \ \ldots \ \wedge \ \neg(a_1 < a_{N-1})$ **then**
8: $\quad b_0 = a_{N-1}$
9: **end if**

This method requires comparison of every pair in the set, thus the initial step is to build a matrix M that holds the comparison results. Entries of this matrix, denoted as $M_{i,j}$, are evaluated using the binary circuit given in Equation 4.3 such

Algorithm 2 Finding the second minimum element in a set.

1: **if** $[(a_0 < a_1) \wedge \ \ldots \ \wedge \ \neg(a_0 < a_{N-1})] \vee \ldots \vee [\neg(a_0 < a_1) \wedge \ \ldots \ \wedge (a_0 < a_{N-1})]$ **then**
2: $\quad b_1 = a_0$
3: **else if** $[(a_1 < a_0) \wedge \ \ldots \ \wedge \ \neg(a_1 < a_{N-1})] \vee \ldots \vee [\neg(a_1 < a_0) \wedge \ \ldots \ \wedge (a_1 < a_{N-1})]$ **then**
4: $\quad b_1 = a_1$
5: **else if** $\ \ldots$ **then**
$\vdots$
6:
7: **end if**

that $M_{i,j} = A_i \lessdot A_j$. Here, note that $M_{i,j} = 1 - M_{j,i}$ (or $M_{j,i} = M_{i,j} \oplus 1$) and $M_{i,i} = 0$ for every i and j. Therefore $M_{i,j}$ is only evaluated for $i < j$. Given the comparison matrix, the ordered elements are computed as follows:

$$B_r = \theta_{r,0} A_0 + \cdots + \theta_{r,N-1} A_{N-1} = \sum_{i=0}^{N-1} \theta_{r,i} A_i,$$

where

$$\theta_{r,i} = \sum_{\substack{k_1=0 \\ k_1 \neq i}}^{N-r-1} M_{k_1,i} \ \cdots \sum_{\substack{k_r=k_{r-1}+1 \\ k_r \neq i}}^{N-1} M_{k_r,i} \prod_{\substack{j=0 \\ j \neq i \\ j \neq k_1, \cdots, k_r}}^{N-1} M_{i,j} \ . \tag{6.2}$$

The $\theta_{r,i}$ values can be seen as the binary place indicators or a decision flag that indicates whether the input A_i will be mapped to output B_r. In other words, if the input A_i has the *rank* r, which is the position of A_i in the sorted array from the left, $\theta_{r,i}$ is an encryption of 1 (naturally it is the encryption of 0, otherwise). This requires the computation of all $\theta_{r,i}$ for $i, r = 0, \cdots, N-1$, thus making the method inefficient. The Greedy Sort example can be found in Appendix A.3.

6.3.2 Direct Sort

The first step is constructing the same comparison matrix M as in Greedy Sort. Then, it computes the ranks by performing a column-wise summation of the entries of M. Since the elements of M are bits, the summation result gives the Hamming weights of the columns of M. The summation operation is implemented using a Wallace Tree of depth $\mathcal{O}(\log_{3/2} N)$. The challenge is that the rank values are encrypted, which requires an additional homomorphic equality check operation to place the elements in their correct order in the sorted array. This equality check is performed on rank values which are $\log N$ bit numbers, hence requires a homomorphic evaluation of a circuit of depth $\log \log N$. The steps of the method are given in Algorithm 3, and an example can be found in Appendix A.4.

Algorithm 3 Direct Sort

Require: A, N
Ensure: B
  for all $A_i \in A$ **do**
    for all $j > i$ **do**
      $M_{ij} \leftarrow A_i \preccurlyeq A_j$
      $M_{ji} \leftarrow 1 - M_{ij}$
    end for
  end for
  $M \leftarrow$ Transpose (M)
  for all i **do**
    $S_i \leftarrow$ HammingWeight (M_i)
  end for
  for all i **do**
    $B_i \leftarrow 0$
    for all j **do**
      $B_i \leftarrow B_i + (S_j \asymp i) \cdot A_j$
    end for
  end for

6.3.3 Polynomial Rank Sort

In this new method, the fundamental idea is to represent the rank of an array element as the degree of a monomial — which we call a *rank monomial* — and position the input element in the coefficient of the monomial.

Definition. Given an input unsorted array with N elements $\{a_0, a_1, \cdots, a_{N-1}\}$, let r_i be the rank of a_i in the array, we denote $\rho_i(x) = x^{r_i}$ the *rank monomial* of integer a_i.

Proposition. If we can find the rank monomials corresponding to the array elements $\{a_0, a_1, \cdots, a_{N-1}\}$, namely $\{x^{r_0}, x^{r_1}, \cdots, x^{r_{N-1}}\}$, then we can obtain an output polynomial, where elements of the input array are its coefficients whose weigths match the ranks of the elements:

$$\begin{aligned} b(x) = \sum_{i=0}^{N-1} a_i \rho_i(x) &= \sum_{i=0}^{N-1} a_i x^{r_i} \\ &= b_0 + b_1 x + \cdots + b_{N-1} x^{N-1} \end{aligned}$$

with $b_0 \leq b_1 \leq \cdots \leq b_{N-1}$.

Proof Sketch. To see why this works, note that the ranks of the inputs are permutations of natural numbers in the range $[0, N-1]$. Thus, there is a bijective mapping in between the i and r_i values. The same mapping gives us the ordered permutation of the input elements, i.e. $b_{r_i} = a_i$. This bijection ensures the exclusive positions of each input element in the output polynomial.

Challenges. Implementing this method on private data has two challenges: i)

finding the ranks of encrypted inputs and ii) placing the encrypted rank values in the exponent. In the following section, we propose a solution to this problem.

Finding Rank Monomials

In order to show how our algorithm works, first we assume that the inputs are not encrypted. After demonstrating the steps of the method, we show how to implement it for encrypted data, i.e. by using only homomorphic operations.

We shortly describe the method as follows: For an N-element input set, we start by finding the zero-based rankings in each 2-subset, which is an element of pairwise combinations of the input set; e.g., $\{a_i, a_j\}$, where $i, j \in [0, N-1]$ and $i \neq j$. As it contains only two elements, the ranks of its elements are either 0 or 1. Then, we construct *surrogate* monomials using these ranks and merge the subsets by performing multiplication of the surrogate polynomials. Consequently, the result gives us the rank monomial of each input element.

Initially, we consider the simplest case: an input set with only two elements $\{a, b\}$. We say the ranks of a and b are r_a and r_b, respectively with the rank monomials defined as

$$\rho_a(x) = x^{r_a} \quad \text{and} \quad \rho_b(x) = x^{r_b} \ .$$

If, for instance, the input elements have the relation $a < b$, then we can write $r_a = 0$ and $r_b = 1$, and therefore $\rho_a = 1$, $\rho_b = x$, respectively.

Now, we include a third element c in the input set. In this case, the first step is to find the ranks in each 2-subset. Namely, we consider all pairwise combinations of

the input set; $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$ and define the following surrogate monomials

$$\begin{aligned} \rho_{ab}(x) &= x^{r_{ab}} & \rho_{ba}(x) &= x^{r_{ba}} \\ \rho_{ac}(x) &= x^{r_{ac}} & \rho_{ca}(x) &= x^{r_{ca}} \\ \rho_{bc}(x) &= x^{r_{bc}} & \rho_{cb}(x) &= x^{r_{cb}} \end{aligned}$$

where r_{ij} is the rank of input i in the 2-subset $\{i, j\}$ and ρ_{ij} is the rank monomial of the same element in the same subset. The next step is merging two subsets to find the final rank monomials of all elements. Multiplying two surrogates adjusts the degree of the rank monomial for the set of three elements, namely $\{a, b, c\}$. To find the rank monomial for a particular input, we multiply all surrogate polynomials that are pertinent to that element, namely

$$\begin{aligned} \rho_a &= \rho_{ab} \cdot \rho_{ac} = x^{r_{ab}+r_{ac}} \\ \rho_b &= \rho_{ba} \cdot \rho_{bc} = x^{r_{ba}+r_{bc}} \\ \rho_c &= \rho_{ca} \cdot \rho_{cb} = x^{r_{ca}+r_{cb}}. \end{aligned}$$

Following the previous example with $a < b$, we fix the following relations and the surrogate rank monomials for the set where $c < a < b$

$$\begin{aligned} a < b &\Leftrightarrow \rho_{ab}(x) = 1,\ \rho_{ba}(x) = x \\ c < a &\Leftrightarrow \rho_{ac}(x) = x,\ \rho_{ca}(x) = 1 \\ c < b &\Leftrightarrow \rho_{bc}(x) = x,\ \rho_{cb}(x) = 1. \end{aligned}$$

Then the rank monomials will be

$$\begin{aligned}\rho_a &= \rho_{ab} \cdot \rho_{ac} = x \\ \rho_b &= \rho_{ba} \cdot \rho_{bc} = x^2 \\ \rho_c &= \rho_{ca} \cdot \rho_{cb} = 1.\end{aligned}$$

If we examine the degrees of the monomials, we find that the ranks are $r_a = 1$, $r_b = 2$ and $r_c = 0$, which are consistent with the given relation $c < a < b$.

We now generalize this method to an N-element input set by defining the following surrogate monomials for each input element a_i

$$\rho_{ij}(x) = x^{r_{ij}} \tag{6.3}$$

for all 2-subsets $\{a_i, a_j\}$ that contain a_i and a_j, $\forall j \in [0, N-1]$ and $j \neq i$. Then, computing the product of all its surrogates will yield the final rank of a_i as the power of monomial

$$\rho_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} \rho_{ij}(x). \tag{6.4}$$

The degree of $\rho_i(x)$ is the rank of a_i in the N-element set. During the process of computing rank monomial of an element a_i, its degree is incremented by one whenever a_i is larger than another element. In other words, the degree of rank monomial of an element a_i is the number of smaller elements than a_i.

Connection to Direct Sort This method is equivalent to summing the column entries of the comparison matrix M, i.e. computing the Hamming weights, as performed in Direct Sort (see Algorithm 3). In order to see this connection, we must first show that comparison matrix elements M_{ij} and 2-subset ranks R_{ij} are com-

plements of each other; i.e., $R_{ij} = 1 - M_{ij}$. This is because M_{ij} is the Boolean output of the comparison $A_i < A_j$, which is an encryption of 1 if and only if A_i is smaller than A_j. However R_{ij} is an encryption of 0 in this case, since it is the smallest element in the same 2-subset. Also, we can easily see that $R_{ij} = M_{ji}$. We formulate the product of all surrogate monomials as in Equation 6.5 and notice that the summation in the exponent is equivalent to computing the Hamming weight of the columns of M

$$\rho_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} x^{r_{ij}} = x^{\sum_{\substack{j=0 \\ j \neq i}}^{N-1} r_{ij}} = x^{\sum_{\substack{j=0 \\ j \neq i}}^{N-1} m_{ji}} . \tag{6.5}$$

Until now, we have shown that we can find the rank monomials given the surrogate monomials of element by applying polynomial multiplication. In the following section, we describe how to apply this method to encrypted input sets; i.e. computing rank monomials homomorphically.

Finding Rank Monomials in the Encrypted Domain

In the encrypted domain, we have a set of N encrypted numbers $\{A_0, A_1, \cdots, A_{N-1}\}$ by a HE scheme. Recall that the first step of the proposed algorithm constructs the surrogates for all 2-subsets as in Equation 6.3. This requires finding the encrypted rank R_{ij} in set $\{A_i, A_j\}$. Hence we use the comparison circuit from Equation 4.3 and compute the encrypted rank of a_i as

$$R_{ij} = 1 - (A_i \lessdot A_j) .$$

Using this value, we can set the surrogate monomials for element a_i using the following operation that requires arithmetic suitable for homomorphic evaluation

$$P_{ij}(x) = 1 - R_{ij} + R_{ij} \cdot x, \tag{6.6}$$

for $j = 0, \ldots, N-1$ and $j \neq i$. Rechecking the base case example with $a < b$, we can confirm that;

$$\begin{aligned} A \lessdot B &= [\![1]\!] & B \lessdot A &= [\![0]\!] \\ R_{ab} &= [\![0]\!] & R_{ba} &= [\![1]\!] \\ P_{ab}(x) &= 1 - [\![0]\!] + [\![0]\!]x & P_{ba}(x) &= 1 - [\![1]\!] + [\![1]\!]x \\ &= [\![1]\!] & &= [\![x]\!] \end{aligned}$$

What happens when two elements are equal to each other? Then, both $a \lessdot b$ and $b \lessdot a$ are expected to output a zero. To address this problem, we always perform only the first comparison and fix the second comparison to its complement. In other words, computing first one of the ranks $R_{ji} = A_i \lessdot A_j$ and setting the other $R_{ij} = 1 - R_{ji}$ solves the equality problem by making sure that the ranks of two elements are always complement of each other. Thereby, we also avoid redundant homomorphic comparisons. Note that, in the previous section, only the upper half of the comparison matrix M is computed and the lower half entries are set to the complements of the elements in the symmetric positions in the upper half of the matrix. This is identical to computing R_{ij} for half of the element pairs.

The next step is to compute the final rank monomials by using Equation 6.4. This operation which only includes polynomial multiplication, can simply be evaluated

using homomorphic evaluations

$$P_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} P_{ij}(x) \ .$$

If we multiply each P_i with the corresponding input A_i, i.e. computing $P_i A_i$, we can place the element in the rank coefficient and since every rank r_i is exclusive to an element, each element will be placed in a distinct coefficient. Thus, we can have an output polynomial B with the ordered elements in its coefficients

$$B(x) = \sum_{i=0}^{N-1} A_i x^{r_i} = \sum_{j=0}^{N-1} B_j x^j,$$

where $b_0 \leq b_1 \leq \cdots \leq b_{N-1}$. The overall method is summarized in the steps of Algorithm 4 with given encrypted input set and the set size N. In comparison to intricate Greedy Sort Algorithm, the simplicity and elegance of Algorithm 4 makes it more favorable and straightforward to implement and less complicated to analyze. As we shall see in the next section, we also gain further in efficiency.

Remark. To see the connection to Greedy Sort in more detail, note that the coefficients of the product polynomial are the binary place indicator $\theta_{r,i}$ values:

$$\begin{aligned} \rho_i(x) \quad &= \prod_{\substack{j=0 \\ j \neq i}}^{N-1} \rho_{ij}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} (m_{i,j} + m_{j,i} x) \\ &= \prod_{\substack{j=0 \\ j \neq i}}^{N-1} m_{i,j} + \left[\sum_{\substack{k=0 \\ k \neq i}}^{N-1} m_{k,i} \prod_{\substack{j=0 \\ j \neq i \\ j \neq k}}^{N-1} m_{i,j} \right] x \\ &\quad + \cdots + \left[\prod_{\substack{j=0 \\ j \neq i}}^{N-1} m_{j,i} \right] x^{N-1} \end{aligned} \tag{6.7}$$

Algorithm 4 Polynomial Rank Sort

Require: A, N
Ensure: $B(x)$
  $B(x) \leftarrow 0$
  for all $A_i \in A$ **do**
    for all $j > i$ **do**
      $M_{ij} \leftarrow A_i \lessdot A_j$
      $M_{ji} \leftarrow 1 - M_{ij}$
    end for
  end for
  for all $A_i \in A$ **do**
    $P_i(x) \leftarrow 1$
    for all $j \neq i$ **do**
      $P_i(x) \leftarrow P_i(x) \times (M_{ij} + M_{ji}x)$
    end for
    $B(x) \leftarrow B(x) + A_i \cdot P_i(x)$
  end for

Batching Input Elements

The Polynomial Rank Sort algorithm that is optimized in terms of both the circuit depth and the number of multiplications still requires $\frac{N(N-1)}{2}$ comparisons. In this section we show how we can reduce it to $N-1$ by enabling batching. When we encrypt the k-bit input elements whereby the corresponding bits of input elements batched into same ciphertexts, we only have k encryptions instead of Nk. As a result, we have both a memory and performance gain by a factor of N. However, there are a few additional factors that we need to take into account in this method. First, rotations are followed by key-switching operations which cause a slight increase in the ciphertext noise. Secondly, the number of slots must be greater or equal to the number of elements. These two requirements lead to additional constraints in parameter selection of the underlying HE scheme.

For our application, we use the plaintext modulus $p = 2$. Choosing Φ_m with $m|2^d - 1$, we utilize ℓ message slots defined over $\mathbb{F}_{2^d}$.

If we have a set of k-bit elements $\{a_0, a_1, \cdots, a_{N-1}\}$, we declare a vector $\vec{\alpha}_i$ consisting of i^{th} bits of the elements,

$$\begin{bmatrix} \vec{\alpha}_0 \\ \vec{\alpha}_1 \\ \vdots \\ \vec{\alpha}_{k-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{1,0} & \cdots & a_{N-1,0} \\ a_{0,1} & a_{1,1} & \cdots & a_{N-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,k-1} & a_{1,k-1} & \cdots & a_{N-1,k-1} \end{bmatrix} \tag{6.8}$$

$$= \begin{bmatrix} A_0 & A_1 & \cdots & A_{N-1} \end{bmatrix}, \tag{6.9}$$

where the j^{th} slot in $\vec{\alpha}_i$ is reserved for the j^{th} element's corresponding bit. A vector $\vec{\alpha}$ can be viewed as holding different elements in each slot for simplicity and we switch to uppercase letters to represent the encrypted values in the slots. In matrix representation, every row is a different ciphertext and the columns are the data slots within that ciphertext.

The first step of Algorithm 4 is to compute the comparisons $M_{ij} = A_i \lessdot A_j$ for all $i < j$. We first apply rotation and masking on the input vector to obtain the rotated slots. The following row vectors represent the left-shifted ciphertexts

$$\begin{bmatrix} \vec{\alpha}^0 \\ \vec{\alpha}^1 \\ \vec{\alpha}^2 \\ \vdots \\ \vec{\alpha}^{N-1} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & \cdots & A_{N-2} & A_{N-1} \\ A_1 & A_2 & \cdots & A_{N-1} & 0 \\ A_2 & A_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{N-1} & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

We perform comparisons in between the original input ciphertext (the first row) and each shifted ciphertext (the rest of the rows). Let the resulting ciphertexts be M_i where $M_i = \vec{\alpha}^0 \lessdot \vec{\alpha}^i$, then we have the following comparison results in the message

slots

$$\begin{bmatrix} A_0 \lessdot A_1 & A_1 \lessdot A_2 & \cdots & A_{N-2} \lessdot A_{N-1} & 0 \\ A_0 \lessdot A_2 & A_1 \lessdot A_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ A_0 \lessdot A_{N-1} & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Following the computation from Equation 6.6, we compute $M_i + (1 - M_i)x$ for each ciphertext (row). The surrogate monomials are computed in the data slots as follows

$$\begin{bmatrix} P_{0,1}(x) & P_{1,2}(x) & \cdots & P_{N-2,N-1}(x) & 0 \\ P_{0,2}(x) & P_{1,3}(x) & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ P_{0,N-1}(x) & 0 & \cdots & 0 & 0 \end{bmatrix}. \tag{6.10}$$

The above matrix holds the surrogate monomials for all i, j pairs for $i < j$. In order to have the rest of the surrogates, i.e. $P_{j,i}$ for $i < j$ we similarly compute the surrogates for the complements of M_i, i.e. $1 - M_i + M_i x$. This gives us the following message slots

$$\begin{bmatrix} P_{1,0}(x) & P_{2,1}(x) & \cdots & P_{N-1,N-2}(x) & 0 \\ P_{2,0}(x) & P_{3,1}(x) & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ P_{N-1,0}(x) & 0 & \cdots & 0 & 0 \end{bmatrix}. \tag{6.11}$$

By rotating, masking and reordering the ciphertexts in (6.11), we can have the

ciphertexts shown in (6.12).

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & P_{N-1,0}(x) \\ 0 & 0 & \cdots & P_{N-2,0}(x) & P_{N-1,1}(x) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & P_{1,0}(x) & \cdots & P_{N-2,N-3}(x) & P_{N-1,N-2}(x) \end{bmatrix} \tag{6.12}$$

Finally, adding the ciphertexts in (6.10) and (6.12) we can have the corresponding surrogates aligned in separate message slots as can be observed in (6.13).

$$\begin{bmatrix} P_{0,1}(x) & P_{1,2}(x) & \cdots & P_{N-2,N-1}(x) & P_{N-1,0}(x) \\ P_{0,2}(x) & P_{1,3}(x) & \cdots & P_{N-2,0}(x) & P_{N-1,1}(x) \\ \vdots & \vdots & \ddots & \vdots & \\ P_{0,N-1}(x) & P_{1,0}(x) & \cdots & P_{N-2,N-3}(x) & P_{N-1,N-2}(x) \end{bmatrix} \tag{6.13}$$

The last step of the algorithm is to take the products of the surrogate monomials. Therefore, if we multiply the rows of (6.13) we will compute all products in parallel message slots.

$$\begin{bmatrix} P_0(x) & P_1(x) & \cdots & P_{N-2}(x) & P_{N-1}(x) \end{bmatrix} \tag{6.14}$$

The resulting ciphertext in (6.14) has rank monomials of all the input elements in respective slots. If we multiply it with the original input vector, we can have the inputs placed in the coefficients of their rank monomial as follows

$$\begin{bmatrix} A_0P_0(x) & A_1P_1(x) & \cdots & A_{N-1}P_{N-1}(x) \end{bmatrix}$$

By shifting all data into the first slot, we can sum all of the rank monomials and

Bit Size ℓ			8					32		
Array Size N	4	8	16	32	64	4	8	16	32	64
$\mathcal{C}_{\text{INS}} \backslash \mathcal{C}_{\text{BUBS}}$	30	140	600	2480	10080	42	196	840	3472	14112
$\mathcal{C}_{\text{OES}}$	20	40	80	160	320	28	56	112	224	448
$\mathcal{C}_{\text{OEMS}} \backslash \mathcal{C}_{\text{BITS}}$	15	30	50	75	105	21	42	70	105	147
$\boldsymbol{\mathcal{C}_{\text{DS}}}$ **(Ours)**	**9**	**10**	**11**	**12**	**13**	**11**	**12**	**13**	**14**	**15**
$\boldsymbol{\mathcal{C}_{\text{GS}}}$ **(Ours)**	**7**	**8**	**9**	**10**	**11**	**9**	**10**	**11**	**12**	**13**

Table 6.1: The multiplicative depth of different sorting circuits given size N and ℓ

find the output polynomial.

$$\left[\begin{array}{cccc} A_0 P_0(x) & 0 & \cdots & 0 \\ A_1 P_1(x) & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A_{N-1} P_{N-1}(x) & 0 & \cdots & 0 \\ \hline \underbrace{\sum_{0}^{N-1} A_i P_i(x)}_{B(x)} & 0 & \cdots & 0 \end{array}\right].$$

6.4 Comparison with the Previous Methods

In Figure 6.1 and Table 6.1, we show how the circuit depths increase with different sorting algorithms. We can achieve the minimum homomorphic evaluation depth with Greedy Sort and Polynomial Rank Sort methods. Direct Sort has slightly more levels than both. In order to compare their performances, we also take the number of costly ciphertext multiplications into consideration. All three algorithms work by computing the comparison matrix M. Since that is a common step for all of the methods, we disregard the cost of constructing M in the following analysis. The complexities of the rest of the operations are summarized for each method in Table 6.2.

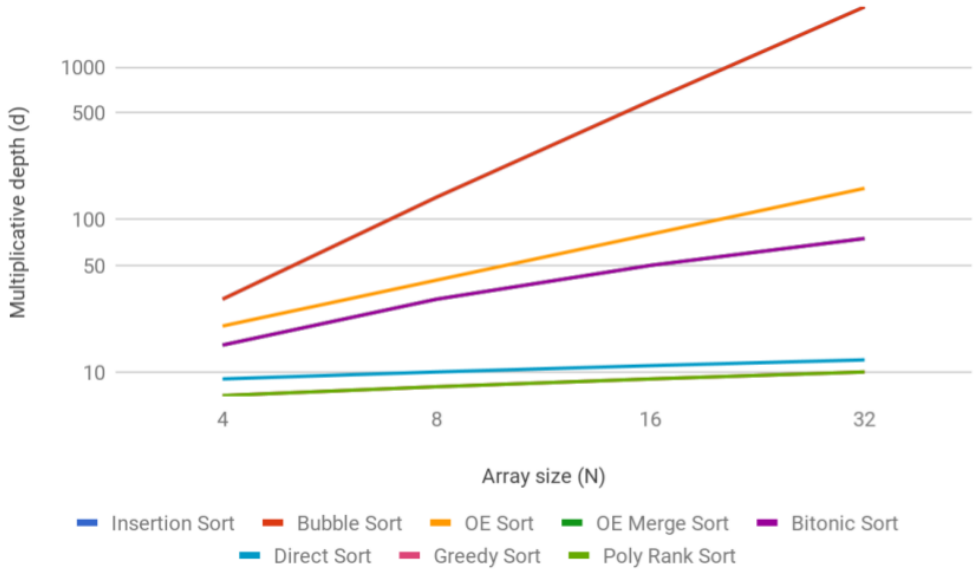


Figure 6.1: Multiplicative depths of different sorting algorithms

Table 6.2: Comparison of the proposed algorithm with the previous methods in terms of number of ciphertext multiplications, multiplicative depth and output size.

	Naïve Greedy	Direct	**Proposed Method**
Ciphertext Multiplications	$\mathcal{O}(N2^N)$	$\mathcal{O}(N^2 \log N)$	$\mathcal{O}(N^2)$
Multiplicative Depth	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$

The Naïve Greedy method implements Greedy Sort in a straightforward manner, hence we need to count the number of multiplications to compute $\theta_{r,i}$ for $i, r = 0, \ldots, N-1$. In order to find the binary place indicators for an arbitrary element A_i, we need to compute $\theta_{0,i}, \theta_{1,i}, \cdots, \theta_{N-1,i}$ as in Equation 6.2 with each one contributing

$$\sum_{k=0}^{N-1} \binom{N-1}{k} = 2^{N-1}$$

multiplications.

Direct Sort has two steps that involve ciphertext multiplications: summation of columns of M, i.e. Hamming Weight computation and the equality check of the rank values. The former is computed by using a Wallace Tree of N bits for each column. The latter is computed by using the bit-wise equality check circuit for $\log N$ bits for each possible rank, i.e. N times. Therefore, the total number of multiplications to find the rank of one element becomes $\mathcal{O}(\log N) + \mathcal{O}(N \log N)$.

Finally, our most recently proposed method Polynomial Rank Sort requires only the product of $N - 1$ ciphertexts, in order to find a single rank.

6.5 First Implementation

In our first implementation, we used DHS scheme of [20] and evaluated $\mathcal{C}_{\text{DS}}$ for a number of array lengths. Here, we briefly summarize the parameter selection process and present the simulation results.

Parameter Selection. According to [20] the NTRU based SWHE Scheme requires Hermite factor $\delta < 1.0066$ to achieve a security level of 80-bit. We set the per level cutting rate $\log p$ depending both on the circuit itself and its total depth, similarly we choose a polynomial degree n according to security threshold and maximum coefficient modulus size. We implemented $\mathcal{C}_{\text{OES}}$, $\mathcal{C}_{\text{OEMS}}$, $\mathcal{C}_{\text{DS}}$ and $\mathcal{C}_{\text{GS}}$ circuits, simulated them for both $\ell = 8$-bit and $\ell = 32$-bit integer inputs and selected array size N. [2] In Table 6.3, we enumerate the parameters which we used in our experiments for various circuit depths. The largest Hermite factor among our parameter choices is $\delta = 1.0063$, ensuring a security level of 99-bits, which is the lowest security level for all cases.

Performance Results. We implemented homomorphic Odd Even Sort, Batcher's

[2] Note that N is *not* restricted to a power of two.

Depth d	9	12	15	21	28	42	56
$\log p$	20	20	22	25	25	25	30
$\log q_0$	200	260	352	550	725	1075	1710
n	8190	8190	16384	16384	27000	32768	46656
S	630	630	1024	1024	1800	2048	2592
δ	1.0041	1.0054	1.0037	1.0057	1.0046	1.0056	1.0063

Table 6.3: Cutting size $\log p$, maximum coefficient size $\log q_0$, Polynomial degree n, message batching slot size S and Hermite Factor δ for different depths d

Bit Size ℓ			8					32		
Array Size N	4	8	16	32	64	4	8	16	32	64
$\mathcal{C}_{\text{OES}}$	400ms	3.45s	n/a	n/a	n/a	2.4s	n/a	n/a	n/a	n/a
$\mathcal{C}_{\text{OEMS}}$	270ms	3.30s	n/a	n/a	n/a	530ms	5.8s	31s	n/a	n/a
$\boldsymbol{\mathcal{C}_{\text{DS}}}$ **(Ours)**	140ms	690ms	3.14s	13.9s	1m	200ms	944ms	4.28s	18.6s	49.7s
$\boldsymbol{\mathcal{C}_{\text{GS}}}$ **(Ours)**	90ms	470ms	2.8s	13.10s	52.2s	500ms	2.4s	10.8s	49.2s	2.2m

Table 6.4: Amortized execution time of circuits for different array sizes N and input bit sizes ℓ

Odd Even Merge Sort and both of the proposed algorithms in C++ using DHS Library [20]. All simulations were performed on an Intel Xeon @ 2.9 GHz server running Ubuntu Linux 13.10. We compiled our code using Shoup's NTL library version 6.0 and with GMP version 5.1.3. The sorting times for 8 and 32 bit integers are given in Table 6.4. For $N = 64$ our algorithm runs in about 14.15 hours whereas the amortized running time, where we use batching with slot size 630, is about 1.35 minutes per sort. For $N = 4$ the sorting takes as low as 0.20 seconds per sort. In comparison, the homomorphic Lazy Sort implementation of [39] takes about 976 and 1400 seconds for array sizes of 10 and 40, respectively. For array sizes $N = 16$ and $N = 64$ our implementation takes 4.28 and 50 seconds, respectively.

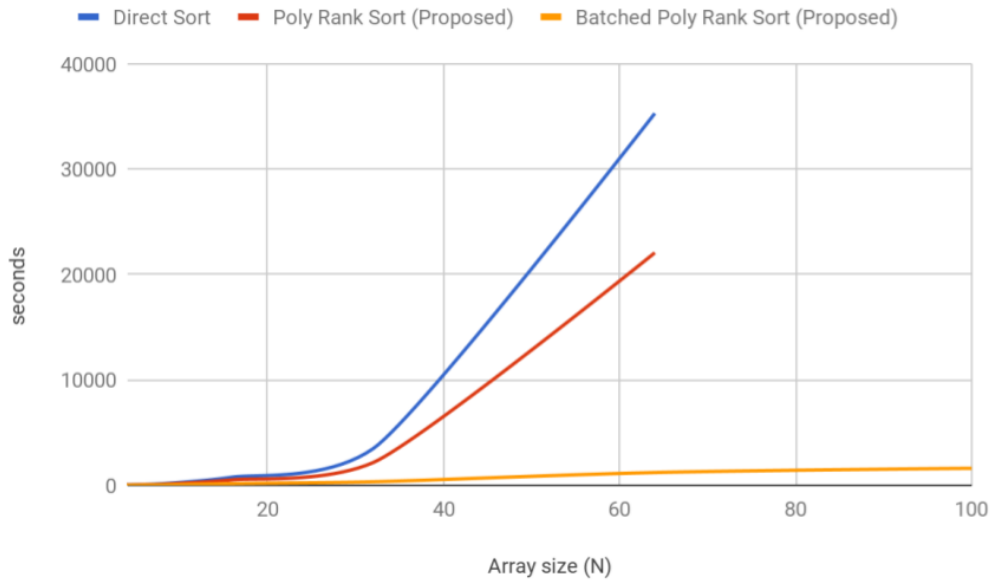


Figure 6.2: Execution times of two proposed sorting methods for different set sizes

6.6 Second Implementation

In our second implementation, we used HElib [15] that is a C++ library that implements the BGV [9] scheme. It provides homomorphic evaluations, batching and message slot rotation functionalities that our algorithm requires. We chose an NTRU-based scheme in our early work, because of its efficiency in leveled applications, compact ciphertext structure and low evaluation key sizes, specifically with DHS customizations. It was not until later in 2016, that the DSPR assumption in NTRU-variants was shown to have a security weakness in [21], hence in this implementation we choose to implement our solutions by utilizing HElib.

We first implemented Direct Sort and Polynomial Rank Sort Algorithms using HElib without packing the input elements. In that implementation we have a different ciphertext for each bit of each number, i.e. kN bit-encryptions in total. In the second implementation, we use the batching technique we described in Section 6.3.3 to pack the same indexed bit of each number in a single polynomial. By using the

default parameter setup of HElib we measured the execution times of Direct Sort, Polynomial Rank Sort and batched version of Polynomial Rank Sort. Experiment results align with the complexity improvements and we conclude that the batched Polynomial Rank Sort achieves the best results in practice. The comparison of three methods can be seen in Figure 6.2.

There are, however a number of restrictions that come with the batched approach.

- Number of data slots (ℓ) must be greater or equal to the number of input elements (N). Although, this looks like an issue at first sight, it usually is not difficult to find a parameter set that provides this constraint. That is mainly due to the fact that with larger array sizes we need more levels of homomorphic evaluations. More levels imply larger ring dimensions for security, and that gives more room to find at least the required slot count.

- The order (d) of the plaintext modulus (p) in the ring must be greater or equal to the number of input elements (N), because the final output is an $N-1$ degree polynomial into which is encoded in a single data slot an element of $\mathbb{F}_{2^d}$. This $d \geq N$ requirement is only necessary if Polynomial Rank Sort is combined with batching. In other words, any other algorithm that does not output a polynomial, such as Direct Sort, does not need to have this property. Similarly, if Polynomial Rank Sort is applied without batching then the output can lay in the full plaintext polynomial and it has room for up to encryption ring degree n which is usually quite large compared to the array size N.

In [59], Basilakis et al. also use batching to reduce the number of comparisons. In what follows we highlight the differences between our work and that of [59].

- Our sorting methods are different. They use Direct Sort (Matrix Sort), whereas

Table 6.5: Parameters and Execution Times for Different Input Sizes

Array Size	4	8	16	32	60
Circuit Depth	11	12	13	14	14
Ring Degree (n)	13200	14400	15840	15840	15840
Order (d)	60	60	60	60	60
Slot Count (ℓ)	220	240	264	264	264
Time (sec)	30	103	206	438	865

we use Polynomial Rank Sort in this implementation. Polynomial Rank Sort achieves both the minimum homomorphic evaluation depth and the minimum homomorphic multiplications.

- They use Parallel Primitive Circuits (PPC) blocks they define in the paper to do comparisons, instead of binary circuits that we use. This choice combined with the sorting algorithm has an affect on the overall depth of the circuit. Our levels are in between $11 - 14$ with respective array sizes, theirs are in the $17 - 22$ range and these depths are only for sorting a maximum of 16 input elements.

- They batch all N^2 comparisons into one or more ciphertexts by serially packing multiple copies of the input element bits into the message slots. Thus require roughly $\lceil kN^2/\ell \rceil$ ciphertexts. We pack each integer once into a message slot and have a separate batched ciphertext for each bit index. So our implementation requires $k\lceil N/\ell \rceil$ ciphertexts.

In Table 6.5, we show our batched Polynomial Rank Sort execution results for different set sizes. We used 120-bit security level for all experiments. [59] gives performance results for only 4, 8 and 16 inputs. Their implementation takes 12, 55 and 248 seconds with single-threading and 11, 49 and 220 seconds with multi-

threads for respective input sizes. In comparison, our implementation takes 30, 103 and 206 seconds for the same number of input elements, respectively. As there is no implementation with larger set sizes in the other work, we cannot compare the timings of 32 and larger sets. We can however conclude that our method scales better with the input size, as the increase in timing is only linear in our results.

Chapter 7

Search

The Electronic Frontier Foundation writes "Anonymous communications have an important place in our political and social discourse. The [US] Supreme Court has ruled repeatedly that the right to anonymous free speech is protected by the [US] First Amendment"[1]. The contents of our web searches give a glimpse into our personal lives and the information harvested from a log of such activity has, in several well-publicized instances, led to clear violations of privacy. It is not only the keywords themselves that leak information. When a user chooses from among query responses and chooses to be directed to a particular URL on the list, the contents of the query are often shared with that website. Browsing history can be tracked and shared, and with data mining techniques, the user can end up revealing much more than keywords, such as their personally identifiable or sensitive personal information. Search engines are also able to infer one's geographic location through one's IP address. When search and other such data is stored, a user profile can be created and this can be shared with third parties such as governments, marketers, and even cyber-criminals. One can easily imagine, for instance, how knowledge of

[1] `https://www.eff.org/issues/anonymity`

recent queries regarding financial instruments could assist a hacker in composing a more credible phishing email that purports to originate at a bank at which the user is a customer.

To protect users from such malicious players, search engines have introduced encrypted search traffic (keywords and results) over the past few years, employing secure connections and creating an encrypted channel between the user device and the search engine server. This option prevents third parties from spoofing responses, and also ends the practice of passing keyword history to the chosen URL (except when that URL is selected through Adwords). Solutions of this sort put the privacy barrier between the search engine and the websites who are their customers, but make no guarantees of private queries. The search engines can still form user profiles and can still share them with third parties for a price. Moreover, such keyword histories can be released by the search engine unintentionally, for example by court order or data breach. Standard search services such as Google require the cleartext query to be handled on the server side revealing to the search company a wealth of information to mine. When mined along with other sources of private information, e.g. e-mail or cloud storage, the search provider can distill a wealth of sensitive information at an unprecedented level of detail. To counter this trend, privacy friendly search services have emerged in the last few years, e.g. DuckDuckGo and StartPage by ixquick. DuckDuckGo, for instance, has rapidly gained customers, recently reaching 25M searches per day. The standard approach taken by these companies is to **promise** to respect the privacy of their customers. While there has been no incident to suspect these products, here too privacy hangs by a thread, i.e. a fragile trust mechanism.

Recent rapid progress in fully homomorphic encryption (FHE) has catalyzed renewed efforts to develop efficient privacy preserving protocols. Several works have

already appeared in the literature that provide solutions to these problems by employing leveled or somewhat homomorphic encryption techniques. Here, we focus on a natural application where privacy is a major concern: web search. An estimated 5 billion web queries are processed by the world's leading search engines each day. It is no surprise, then, that privacy-preserving web search was proposed as the paragon FHE application in Gentry's seminal FHE paper. Indeed, numerous proposals have emerged in the intervening years that address various privatized search problems over encrypted user data, e.g. private information retrieval (PIR). Yet, there is no known work that focuses on implementing a truly blind search engine utilizing an FHE construction. In this work, we focus first on single keyword queries with exact matches, aiming toward real-world viability. We then discuss multiple-keyword searches and tackle a number of issues currently hindering practical implementation, such as communication and computational efficiency.

7.1 Our Search Model

In this section, we will define a privacy-preserving web search engine that evaluates encrypted search queries.

To fully execute a typical web search, a server carries out four fundamental tasks: web crawling, indexing, ranking and retrieval. For an encrypted web search engine, the first three tasks are unchanged as these do not involve encryption; it is only the task of retrieval in response to an encrypted query from the user that we need to address here. Modern search engines have become quite sophisticated, accounting for spelling errors, synonyms or word meanings, sometimes applying ranking after retrieval. But this is beyond the scope of the present work: we deal only with exact matches here. Thus we assume that pre-processing on the server side provides us

with a rank-ordered output list of URLs attached to each keyword.

In our construction, we have two parties: a user $\mathcal{U}$ who submits the (encrypted) search query and a server $\mathcal{S}$ who is the owner of the database and the entity that performs the retrieval look up. Server $\mathcal{S}$ has a table consisting of the dictionary words and their corresponding rank-ordered search results. In a regular search engine, $\mathcal{S}$ has to know the user keyword κ in order to perform the look up. In order to be able to process encrypted keywords, we convert this standard comparison/aggregation model into a homomorphic circuit so that it can be evaluated using encrypted input(s) and it can output encrypted results.

In a simple scenario, when $\mathcal{U}$ submits a query, the input keyword κ is first encrypted on the client side under client's own encryption key, then the ciphertext(s) are sent to the server and the server performs the retrieval step obliviously using homomorphic circuits over the encrypted user input and the server's own database which is in cleartext form. For instance, in order to search for the keyword "insomnia", $\mathcal{U}$ encrypts $\kappa =$ "insomnia" using his own encryption key and sends the encrypted data[2] $[\![\text{insomnia}]\!]$ to the server. $\mathcal{S}$ then evaluates the homomorphic circuit using only the encrypted input(s) and the index table it owns. During the homomorphic evaluation, all intermediate results will still be encrypted under the client's encryption key. After the circuit evaluation, $\mathcal{S}$ returns the corresponding output ciphertext(s) to $\mathcal{U}$. Finally, $\mathcal{U}$ decrypts the resulting ciphertext(s) using its own decryption key and gets the matched query results (if any). Note that user-specific filtering, ranking and formatting of results can then be performed on the client side working with plaintexts obviating the need to share certain user preferences with an untrusted server.

In reality, there are around 60 trillion web pages on the web and the number of

[2]Throughout this chapter, we will use $[\![x]\!]$ to denote an encrpytion of x.

pages indexed by Google is around 54 billion and by Bing it is around 650 million[3]. These numbers change every hour of every day. However, they can be much smaller for a custom search engine that is designed for a specific target group of clients, for example a medical search engine would only index the health care related pages. Another example can be an internal search engine for a company that only crawls the subnet of that company. Hence, in practice the indexed pages may contain a small subset of the overall web, especially for a proof-of-concept design. We denote the set of all indexed URLs under consideration by $\mathcal{L}$ and denote the size of $\mathcal{L}$ by T. We will use a short URL representation where each element $u \in \mathcal{L}$ is a $\lceil \log T \rceil$ bit integer. In case of a universal search engine such as Google, this number can be as high as 36 bits. This forces us to make a sacrifice by placing an upper limit t on the number of "hits" for any given keyword; that is, we assume that our Search Engine Results Page (SERP) contains only the t top-ranked URLs for each simple query. This is obviously quite a bit simpler than modern search engines. However, since our outputs will be encrypted under an FHE scheme, our plaintext space -total number of bits that we can decrypt at the end of homomorphic computations- is limited due to chosen FHE parameters. Therefore the number of output URLs t depends on several factors like plaintext space, bit-size of a single URL and the number of search keywords and the details of this relation will be given in Section 7.1.3.

The other component of the database is the keyword list. The Oxford English Dictionary includes over a half million English words whereas a target-oriented vocabulary can have fewer number of items, for example Stedman's medical dictionary has around a hundred thousand medical terms. Thus, the size of the keyword list changes according to the search engine functionality. To accommodate a realistic scenario where the database can be fairly large, we choose N as one million, which

[3] `http://www.worldwidewebsize.com/`

is our bound on the number of entries in the lookup table. We will make use of a local dictionary to encode each keyword κ as a 20-bit integer ($\log N$) using their row index in the table and reserving index 0 for the keywords not found in the dictionary. These parameters determining the number of possible input/output values to be transferred between the server and the client have an effect on both the circuit size and the bandwidth, thus they have significant bearing on the runtime of the application. We will design our circuits for generic values of these parameters and return to specific values only later, in Section 7.2.3, where we observe their effect on bandwidth and execution time.

7.1.1 Associative Array and Search Algorithms

The dictionary we defined in Section 7.1 consists of a pair of objects — a *key* from the alphabetically sorted keyword list; and a list of *values*, each from the URL list — in each row. The association between the two objects will be referred to as binding. Whenever we want to find the value which is bound to a given keyword, we will apply an operation called Aggregation. This Aggregation operation requires a Comparison operation to find the keyword location in the array. To this end, there are a number of search algorithms that one can normally use such as serial search, binary search and search by hashing. From these three serial search and hash methods have a worst case complexity of $\Theta(N)$ whereas binary search has $\Theta(\log(N))$. However logarithmic complexity for binary search is not achievable in a blind search. Since homomorphic execution requires the method to explore all paths regardless of outcome, even the binary search has linear complexity in this setting.

7.1.2 The Construction for Comparison

In encrypted search schemes, expensive computations and massive user-server bandwidth requirements remain as serious obstacles to achieving real-time FHE-based applications. In this section, we present a homomorphic construction that can handle practical scenarios. We start with a primitive scheme with low circuit depth and then provide optimizations and propose a lightweight algorithm, to push our design closer to a real-life application. This improvement comes with a price: the more we decrease bandwidth, the deeper the circuit required.

The first step of the blind search is encrypting the user input κ. Hence the first decision to make is about the representation of the keywords. As the keywords are arbitrary length strings and we will be working in an homomorphic setting, binary representation seems like a natural choice. But binary representation of a string would leak information about its length. And particularly too short or too long keywords can be easier to guess. The second option is to encrypt the string as a word by setting a large plaintext domain. This requires a word-wise homomorphic comparison method which is possible, but costly. Lastly, we can use the index w of the input keyword in the dictionary. We consider the following four algorithms for encrypted keyword search. In the following algorithms, all w are $\log N$ bits, where N is the dictionary size.

- **Standard Comparison Algorithm.** In this method, the bits of the index w are encrypted. Using an equality check circuit $\prod_{j=1}^{\log N}(w_j \oplus i_j \oplus 1)$, we can compare the input to every single possible index value $i = 1, \cdots, N$ with bits $i_1 i_2 \cdots$. The bandwidth of this approach is equal to the number of bits of the index w, hence it is bounded by $s = \log N$. The number of multiplications will be $s - 1$ for each i, thus in total there are $N(s-1)$ ciphertext multiplications in this method.

- **Kushilevitz-Ostrovsky (KO) Algorithm [49].** In this case the input index w is divided into two parts, $w = (w_1, w_2)$ where $w = w_1\sqrt{N} + w_2$ and both w_1 and w_2 are one-hot encoded using the above approach. This reduces the bandwidth to $2^{\log(N)/2+1} = 2\sqrt{N}$, and the depth becomes 1. This method can be applied recursively on the new index values w_1, w_2 and we can have partitions into four subwords $w = (w_1, w_2, w_3, w_4)$ which would reduce the bandwidth to $4N^{1/4}$ and increase the depth to 2. If we partition into k pieces, we end up with a bandwidth of $kN^{1/k}$. In the case of multiplications, we are not able to set up a regular multiplication tree which is optimized for KO constructions because of the limitations of the F-NTRU scheme. In order to compute the comparison, we perform k-dimensional multiplication in a serial manner. First, we multiply along two axes, contributing $N^{1/k} \cdot N^{1/k}$ multiplications. Next, the results are multiplied with values along a third axis which results in $N^{1/k} \cdot N^{2/k}$ multiplications. After $k-1$ iterations, the total number of multiplications is $N^{1/k} \sum_{i=1}^{k-1} N^{i/k}$.

- **Hybrid Algorithm.** We will divide the input index w into two parts $w = (w_1, w_2)$ where $w = w_1 N/2^s + w_2$, i.e. w_1 is the first s bits of the index. We will perform the standard comparison on the first part w_1 of length s and encode the second part w_2 of length $\log N - s$ using the KO algorithm. Then, the bandwidth of scheme is summarized as $s + k(N/2^s)^{1/k}$, where s is coming from the first part and the rest is coming from the KO algorithm. The number of multiplications are $2^s(s-1)$ and $\sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}}$ for the first and second parts, respectively. Since we need to multiply the results of first and seconds parts with each other to form the decisions, the total number of multiplications results in $2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$.

Table 7.1: Comparison of bandwidth requirements and number of multiplications for different Comparison algorithms

Algorithms	**Bandwidth**	**Multiplications**
Standard Comparison	$\log N$	$N(\log N - 1)$
KO Construction	$kN^{1/k}$	$\sum_{i=1}^{k-1} N^{\frac{i+1}{k}}$
Hybrid Method	$s + k(N/2^s)^{1/k}$	$2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$

The overall bandwidth and multiplicative complexity comparison is given in Table 7.1. We quickly observe that the optimal choice of encoding depends heavily on the properties of the FHE scheme in use.

7.1.3 Construction for Aggregation

After the Comparison step is completed for a user input, we have the intermediate values which we call the *decision vector* and denote by $\vec{d}$. This vector has N encrypted bit values; $\vec{d}[i] = [\![1]\!]$ if $i = w$ and $\vec{d}[i] = [\![0]\!]$ otherwise. After constructing our circuits, we will see that we do not need the Comparison results all at once, but instead we can have one result at a time. This means that we do not have to store N encrypted values, but we will compute the $\vec{d}[i]$ values in an iterative way.

The second step of our application is the actual Aggregation step, where we will compute our final output(s). Note that for a more practical engine, we consider the multi keyword search (e.g., a conjunction of m terms). This means that whenever there are two or more input keywords, we will perform the Comparison multiple times. As a result, we will have one decision vector corresponding to each input keyword. For an m-keyword search, we will have decision vectors: $\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_m$ corresponding to the inputs $w_1, w_2, \ldots, w_m$.

The size of each URL list L_i that is bound to the i^{th} keyword in the database is $|L_i| = t_i$, for $i \in [1, N]$. Since the server is oblivious to the input value(s) w, it is also oblivious to the list size t_w. This means that the output size $|L_{\text{out}}|$ will be

determined by the longest list in the database, given by $t = \max\{t_i | i \in [1, N]\}$. In the next part of this section, we will describe the algorithm for a single keyword query (i.e., $m = 1$). In the following part, we will give the details for our multi-keyword aggregation problem ($m > 1$).

Single Keyword

First let us assume that we have a single input keyword with dictionary index w. We have the URL list[4] $L_i = [u_{i,1}, u_{i,2}, \ldots, u_{i,t}]$ for each keyword index i. After computing the decision vector $\vec{d}$, we can find the final outputs by simply computing

$$\begin{aligned} L &= \sum_{i=1}^{N} \vec{d}[i] L_i \\ &= \left[\sum_{i=1}^{N} \vec{d}[i] u_{i,1}, \sum_{i=1}^{N} \vec{d}[i] u_{i,2}, \ldots, \sum_{i=1}^{N} \vec{d}[i] u_{i,t} \right] \\ &= [[\![u_{w,1}]\!], [\![u_{w,2}]\!], \ldots, [\![u_{w,t}]\!]] \,. \end{aligned}$$

If each URL index $u_{i,j}$ is $\lceil \log T \rceil$ bits in length as we defined in Section 7.1, at the end the server will have $t\lceil \log T \rceil$ bits to return to the client. By using the large integer encoding technique from Section 3.3, we can decode a single ciphertext and retrieve n bits where n is the FHE ring degree. This means that we can have $\lceil t\lceil \log T \rceil / n \rceil$ output ciphertexts or alternatively, if we limit the t value such that the inequality $t\lceil \log T \rceil \leq n$ holds, then all outputs can be encoded into a single n degree polynomial. It means that the server will send back only a single ciphertext.

A further economy can be found by noting that we do not need to compute all $\vec{d}[i]$ values in advance. Instead, in the first iteration we can compute $\vec{d}[1]$ and after initializing L to $\vec{d}[1]L_1$, we can then compute $\vec{d}[2]$ and update $L = L + \vec{d}[2]L_2$.

[4] In order to maintain uniformity, we define $u_{i,j} = 0$ for $t_i < j \leq t$.

Iterating in this way, we only need to store $t+1$ encrypted polynomial coefficients at a time.

For reasons that will become clear shortly, we propose an alternative way to encode the list $L_i = [u_{i,1}, u_{i,2}, \ldots, u_{i,t}]$ as a polynomial

$$\ell_i(x) = \prod_{j=1}^{t} (u_{i,j} - x).$$

After computing the decision vector $\vec{d}$, we can find the final outputs by simply computing

$$\ell(x) = \sum_{i=1}^{N} \vec{d}[i]\ell_i(x).$$

If each URL index $u_{i,j}$ is again $\lceil \log T \rceil$ bits in length, hence the coefficient of x^h in $\ell_i(x)$ is about $\lceil \log T \rceil (t-h)$ bits. Total number of bits to be returned to the user will be $\lceil \log T \rceil \sum_{h=0}^{t} (t-h) = \frac{(t^2+t)}{2} \lceil \log T \rceil$ which is more compared to the first approach where we only have $t\lceil \log T \rceil$ bits. Hence, we will use this polynomial representation of the URL lists for multiple keyword construction in the next section.

Multiple Keywords

We discuss here only conjunctive queries; while the product of two polynomials gives a natural solution to the disjunction of keywords (union of URL lists), this changes polynomial degrees and, since disjunctive queries are uncommon, we choose not to address this added level of complexity here. In Aggregation, if a user wants to find all URL values that are bound to multiple keys, we are faced with a set intersection problem [60]. Suppose, for simplicity the query takes the form $w \wedge w'$ and the above procedures generate corresponding polynomials $\ell_w(x)$ and $\ell_{w'}(x)$, respectively. The roots of $\ell_w(x)$ (resp., $\ell_{w'}(x)$) encode the top-ranked URLs for keyword w (resp., w')

so the conjunction would naturally correspond to the gcd of the two polynomials. But this is difficult to compute homomorphically. So, as previously noted by [60], we can instead take a random polynomial in the ideal generated by $\ell_w(x)$ and $\ell_{w'}(x)$. Clearly, if $d = \gcd(\ell_w, \ell_{w'})$ and

$$f(x) = g(x)\ell_w(x) + g'(x)\ell_{w'}(x),$$

then $d(x)$ divides $f(x)$ and the probability that $f(x)$ has any additional "spurious" roots is negligible assuming reasonable parameters. In practice, it may even suffice to take $f(x) = \ell_w(x) + \ell_{w'}(x)$. But we prefer to take a more careful approach.

We can, in fact, reduce this probability of spurious roots to zero with careful choice of coefficients. All valid URLs are known to lie in the range $[1, T]$. If r and r' are primes just a bit larger than T, then it is impossible for

$$f(x) = r\ell_w(x) - r'\ell_{w'}(x)$$

to have any root in range $[1, T]$ which is not a common root of ℓ_w and $\ell_{w'}$. Indeed, if $f(u) = 0$, then

$$r\ell_w(u) = r'\ell_{w'}(u).$$

But $\ell_w(x) = \prod(u_j - x)$ for some roots u_j all lying in $[1, T]$. So $\ell_w(u) = \prod(u_j - u)$ has no prime factor exceeding T. Therefore $f(u) = 0$ for $u \in [1, T]$ forces both $\ell_w(u) = 0$ and $\ell_{w'}(u) = 0$. This naturally generalizes to a conjunction of m keywords, as we outline below. But we point out here that, with $m = 2$ being the most common conjunction, we see some economy by choosing r and r' close together — twin

primes, for example, with $r = r' + 2$ — and noting

$$\begin{aligned} f(x) &= r\ell_w(x) - r'\ell_{w'}(x) \\ &= r' \left[\ell_w(x) - \ell_{w'}(x)\right] + 2\ell_w(x) \end{aligned}$$

sometimes allows us to control growth of coefficients.

Lemma 5. *Let $w_1, \ldots, w_m$ be m distinct keywords with corresponding polynomials $\ell_1(x), \ldots, \ell_m(x)$ where the roots of $\ell_i(x)$ encode the top-ranked URLs bound to keyword w_i. Let $s_1, \ldots, s_m$ be m distinct primes with $s_i > T$ for all i and, for $1 \leq i \leq m$, define $r_i = \frac{1}{s_i} \prod_{h=1}^{m} s_h$. Then, for $f(x) = \sum_{i=1}^{m} r_i \ell_i(x)$, $\gcd\left(f(x), \prod_{u=1}^{T}(x-u)\right) = \gcd\left(\ell_1(x), \ldots, \ell_m(x)\right)$.*

Proof. If $u \in [1, T]$ and $f(u) = 0$, then reduction modulo s_i gives $\ell_i(u) = 0 \pmod{s_i}$. But $\ell_i(u)$ has no large prime factors, so $\ell_i(u) = 0$ and, since this holds for all i, u is a common root. □

The second problem we must deal with is the representation of the polynomials. Assuming the polynomials ℓ_i have degree t, we have $\|\ell_i\|_\infty = t \log T$. Therefore the large integer encoding technique from Section 3.3 turns these $(t \log T)$-bit numbers into polynomials of degree $t \log T$ with 0/1 coefficients. The computations we must perform on the ℓ_i are polynomial additions and constant multiplications; so we can perform the same operations over each coefficient separately, send the results to the user without further processing, and the user can reconstruct the polynomial encoding the conjunction of m keywords. If we write

$$\ell_i(x) = \sum_{k=0}^{t} \alpha_{i_k} x^k,$$

then constant multiplication by integer b is computed

$$b\ell_i(x) = \sum_{j=0}^{t} b\alpha_{i_k} x^k$$

and similarly, the sum of m polynomials is computed coefficient by coefficient:

$$\sum_{i=1}^{m} \ell_i(x) = \sum_{j=0}^{t} \left(\sum_{i=1}^{m} \alpha_{i_k} \right) x^k.$$

We have $(t \log T)$-bit coefficients and the constant multiplication with the integers r_i in the above lemma will add $(m-1)(\log T + 1)$ bits to the end result, so we will have approximately $(m+t)\log T$ bit values to be encoded/decoded. If we have an FHE ring of degree n, we can afford $t < \lfloor n/\log T \rfloor - m$.

We have $\ell_i(x) = \sum_{k=0}^{t} \alpha_{i_k} x^k$ and it will be convenient to likewise write $\ell_{w_j}(x) = \sum_{k=0}^{t} \alpha_{(w_j)_k} x^k$. These are stored as lists of coefficients and when the server processes a decision vector $\vec{d_j}$ encrypting the one-hot encoding of the j^{th} keyword of the query, it may compute $\sum_{i=1}^{N} \vec{d_j}[i]\ell_i(x)$ one coefficient at a time. But we can do better. The user needs access to the polynomial $\ell(x) = \sum_{j=1}^{M} r_j \ell_{w_j}(x)$ and, if we write $\ell(x) = \sum_{k=0}^{t} \beta_k x^k$, we have

$$\begin{aligned} [\![\ell(x)]\!] &= \sum_{j=1}^{m} r_j [\![\ell_{w_j}(x)]\!] \\ &= \sum_{j=1}^{m} r_j \sum_{i=1}^{N} \vec{d_j}[i] [\![\ell_i(x)]\!] \end{aligned}$$

so that,

$$[\![\beta_k]\!] = \sum_{j=1}^{m} r_j \sum_{i=1}^{N} \vec{d_j}[i] [\![\alpha_{i_k}]\!] \ .$$

This list $\{[\![\beta_k]\!]\}_{k=0}^{t}$ is passed to the user who, upon decryption, recovers all β_k, reconstructs $\ell(x)$ and applies standard root-finding techniques to obtain the desired list of URLs arising from the conjunctive query. Note that the constant term of $\ell(x)$ has no large prime factors, so its roots lying within $[1, T]$ can be recovered rather efficiently by standard techniques.

7.1.4 Noise Analysis

We take same approach as authors did in [24], since we are using the F-NTRUscheme with a small modification. Our scheme changes only the message space from 2 to $x - 2$. Using $||g||_\infty = ||f'||_\infty = B_{\text{key}}$, $||s||_\infty = ||e||_\infty = B_{\text{err}}$, a fresh ciphertext has the following noise bound:

$$\begin{aligned} ||y_{(0)}f||_\infty &\leq 4nB_{\text{key}}B_{\text{err}} + 4nB_{\text{err}}(4B_{\text{key}} + 1) \\ &\leq 4nB_{\text{err}}(5B_{\text{key}} + 1). \end{aligned}$$

At each multiplication, using the single sided multiplication approach as in [24], the noise bound is equal to

$$\begin{aligned} B_i = ||fy_i||_\infty \;\leq\; & [4n^2B_{\text{key}}B_{\text{err}}(2^w - 1)\ell \\ + \;& 4n^2B_{\text{err}}(4B_{\text{key}} + 1)(2^w - 1)\ell] \\ + \;& [4nB_{\text{err}}B_{\text{key}} + 4nB_{\text{err}}(4B_{\text{key}} + 1)] \\ + \;& [B_{i-1}] + [(4B_{\text{key}} + 1)] \end{aligned}$$

The number of multiplications for the decision depends on the Comparison algo-

Table 7.2: The values of index i in noise bound B_i to compute decisions for each entry in Comparison algorithms.

Algorithms	**# Multiplications (i)**
Standard Comparison	$s-1$
KO Construction	$k-1$
Hybrid Method	$s+k-1$

rithm. Thus, it changes the required bitsize for each algorithm. In Table 7.2 we give the maximum length of multiplicative chain to compute a decision for an entry, i.e. it is the index i used in noise bound B_i.

In addition to the base noise which is occurring from the decisions, we have additional noise occurring from the latter operations. We can list the additional noise occurrences as follows

- multiplication of decision with the message (encoded as binary polynomial): $\log\log n$
- summation of all the entries: $\log N$
- multiplication with prime number to eliminate spurious roots (encoded as binary polynomial): $\log\log r$
- number of search results which we add together (number of keywords): $\log m$

These additional noise should be added to the base noise bound so that the modulus is large enough to support the operations.

7.2 Implementation and Performance

Either single-keyword or multi-keyword search requires a tremendous amount of computation power. Taking advantage of previous research in [61, 62, 63], we believe

Table 7.3: Testing Environment

Item	Specification
CPU	Intel Core i7-3770K
CPU Freq.	3.50 GHz
System Memory	32 GB DDR3
GPU	NVIDIA GeForce Titan X
GPU Core Freq.	1.20 GHz
GPU Memory	12 GB
# of CUDA Cores	3072

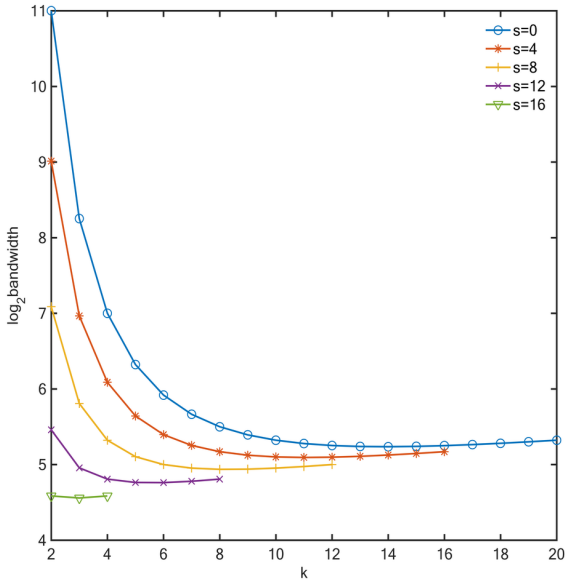
Table 7.4: A comparison of hybrid and KO algorithms with different parameters. Database has $N = 2^{20}$ entries.The bandwidth is calculated for 576 KB input ciphertexts and 48 KB output ciphertext. The first column gives the number of input keywords in each scenario. Time includes the latencies of Comparison and Aggregation, and is normalized per database entry.

#	Algorithm	s	k	Bandwidth		Time (μs)
				Input	Output	
1	KO	0	2	1,152 MB	48 KB	304
1	KO	0	9	24 MB	48 KB	341
1	Hybrid	8	8	17 MB	48 KB	168
1	Hybrid	12	5	15 MB	48 KB	173
2	KO	0	9	48 MB	2.40 MB	3,544
2	Hybrid	8	8	34 MB	2.40 MB	3,198
2	Hybrid	12	5	30 MB	2.40 MB	3,207
3	Hybrid	8	8	51 MB	2.35 MB	4,722

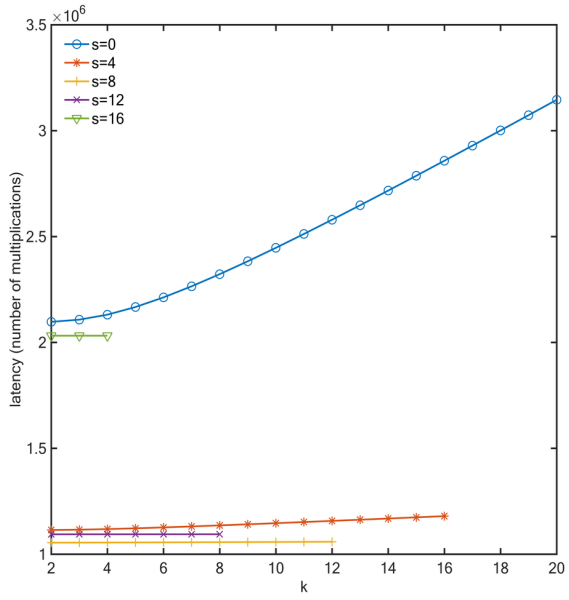
an implementation on CUDA-enabled GPUs offers high efficiency. We compared our proposed hybrid algorithm to a recursive KO algorithm in terms of bandwidth requirements and computation time. All the timing results in this paper are measured on the machine described in Table 7.3.

7.2.1 Polynomial Multiplications

During the evaluation steps, the performance of ciphertext multiplications dominates the total execution time. Our implementation focuses on optimizing a ciphertext multiplication (recall Section 2.2) which is the product of a vector of polyno-



(a) Bandwidth (base 2 logorithm)



(b) Latency (number of multiplications)

Figure 7.1: A comparison of hybrid and KO algorithms in bandwidth and latency with respect to parameter choices. The database has $N = 2^{20}$ entries. Standard comparison is applied on the first s bits of input index. A KO algorithm with k iterations is applied on the rest. KO algorithm is adopted when $s = 0$.

mials and a matrix of polynomials: $R_{2^w}^{1\times l} \times R_{2^w}^{l\times l}$. Note that we perform ciphertext multiplications in a chain and keep only the last row of the left-hand multiplier all the time, which explains why one of the multiplier is in vector form.

Starting with polynomial multiplications, we compare the efficiency of Karatsuba algorithm and the Number-theoretic transform (NTT) based algorithm proposed in [61]. We ignore overhead caused by additions or binary operations such as shifting/or/and/xor to draw a simpler yet fair comparison. Assume polynomials have degree slightly smaller than 2048. We need to perform 4096-sample NTT conversions.

In [61] a special finite field $\mathbb{F}_P$ where $P = \texttt{0xffffffff00000001}$ is chosen. Also the NTT or inverse-NTT (INTT) conversions of 4096 samples can be factorized into smaller size (e.x. 64 sample) following the Cooley–Tukey FFT algorithm [64]. Conversions of no larger than 64 samples are implemented with shifting, addition and fast modular reduction over P, which takes advantage of properties of P. The NTT-based algorithm only requires 4096 integer multiplications per conversion when multiplying twiddle factors. One polynomial multiplication requires two NTT conversion, a coefficient-wise multiplication of two NTT domain vectors (4096 integer multiplications) and one INTT conversion, which add up to a total of 16384 integer multiplications. However, the Karatsuba algorithm that requires $3^{\log 2048} = 177147$ multiplications would be much slower.

Plus, the NTT-based algorithm is highly parallelizable hence more suitable for a GPU implementation. In conclusion, the NTT-based algorithm outcomes the Karatsuba algorithm on a GPU. Following the ideas in [61], we developed 4096-point NTT/INTT conversions for a GPU. Each NTT conversion takes 0.75 μs and each INTT conversion takes 1.45 μs. Every ciphertext multiplication contains $l^2 + l$ NTT converions and l INTT conversions. The overhead of multiplications and additions

in NTT domain is negligible.

7.2.2 Modulus Selection For Efficient Flattening

Ciphertext multiplications, although taking 16-bit norm polynomials as input, produce polynomials with ln times larger norm (less than 64-bit) as output. A Flatten operation is expected to reformat the ciphertext coefficients back to 16-bit. The Flatten includes (e.x. 64-bit) integer additions with shifting and reductions modulo q. We choose the coefficient modulus q as a power of 2 for efficient modulo reduction and a prime polynomial degree n as in the classic NTRU cryptosystem.

A ciphertext multiplication takes $l + 1$ polynomials in R_q as input. The BitDecomp takes no time. Then the multiplication algorithm produces l polynomials with 64-bit coefficients. $\mathsf{BitDecomp}^{-1}()$ or $\mathsf{Flatten}() = \mathsf{BitDecomp}(\mathsf{BitDecomp}^{-1}())$ is achieved with a sequence of 16-bit additions with carries, which adds 2.66 μs.

7.2.3 Performance of the Proposed Methods

In our experiments, we first fix the dictionary size $N = 2^{20}$ and the FHE parameters according to the noise analysis given in Section 7.1.4 and the security parameter of F-NTRU scheme. We use the ring $\mathbb{Z}_q/\left(x^n + 1\right)$ where the coefficient modulus is $q = 2^{192}$ and the degree is $n = 2039$. Following the security analysis from [20], the Hermite factor for the chosen values is 1.00525 which provides 128-bit security.

Based on previous analysis of bandwidth requirements and computation time, Fig. 7.1 illustrates a comparison of several algorithms with selected yet typical parameters. The hybrid algorithm becomes a KO construction when $s = 0$. And when $k = 2$, it is a basic (non-recursive) KO construction. With the help of those figures, we may find optimal parameters that has low bandwidth and latency.

As shown in Table 7.1, a standard comparison algorithm has minimum bandwidth requirement. The same fact is reveal in Fig. 7.1a: a hybrid algorithm, when applying standard comparison on more bits, i.e. when choosing a larger s, has a lower bandwidth requirement. Then as we apply more iterations of KO, i.e. as k increases from 2, bandwidth requirement drops significantly. The value k, although affects latency of KO, has insignificant influence on hybrid schemes. Hence, a optimal hybrid scheme with a certain s would choose k with minimal bandwidth.

The number of multiplications in a hybrid method is expressed as a formula in Table 7.1. The fact that one part of the formula involves s while another part involves k makes Fig. 7.1b more complicated (rather than increasing/decreasing along with s). We can see that when $s = 8$, the latency is lower than all other cases. And choosing $s = 4, 8, 12$ does not gives a critical difference in latency. However, when comparing $s = 4, 8, 12$ in Fig. 7.1a, one may easily notice the remarkable difference in bandwidth.

The hybrid methods outcome the (recursive) KO construction on bandwidth requirements and computation time. Within the hybrid methods, if the priority is to reduce bandwidth requirement, select $s = 12$ and $k = 5$; if the priority is efficiency, select $s = 8$ and $k = 8$. These two parameter sets are adopted in Table 7.4. For single-keyword search, the hybrid scheme with $s = k = 8$ requires 37.7% less bandwidth and costs half the time, comparing to those of the KO construction with $k = 9$ iterations. However, for multi-keyword scenarios, the Aggregation weights most of the latency. Therefore the advantage of hybrid methods is clear.

Finally, the size of the output ciphertexts depends on the number of URLs that we return t in the multikeyword scenario with respect to the inequality from Section 7.1.3, $t < \lfloor n/\log T \rfloor - m$ where n is 2039. We set $\lceil \log T \rceil = 40$ so that the indexed URL set $\mathcal{L}$ can have up to a billion URLs. Therefore for 2 keywords, we

afford sending back at most $t = 49$ URLs and for 3 keywords, $t = 48$ at most. In each case the server sends back $t + 1$ encrypted coefficients of the resulting t-degree URL polynomial and each coefficient is represented in a separate ciphertext. In order to increase t, we have to choose a larger degree for the FHE setup, which would end up increasing both the bandwidth and the computation time. In the single keyword scenario, we choose to limit the number of URL outputs $t = 50$, so that the result fits into a single ciphertext following the relation $t\lceil \log T \rceil \leq n$ from Section 7.1.3. Note that in this case, if we want to increase the number of output URLs t, we can do so by increasing the bandwidth and sending back $\tilde{n}$ ciphertexts each carrying n bits, as long as $t \leq \frac{\tilde{n}n}{\lceil \log T \rceil}$ holds.

Chapter 8

Conclusion

In this dissertation, we adapted the most fundamental programming problems to be used in real world fully homomorphic encryption applications. To this end, we analyzed the existing methods, proposed new algorithms and optimized them with respect to the cost of homomorphic evaluations. We also improved our solutions by utilizing different encoding and packing methods to achieve the best communication and computation efficiency.

This work first explores advances in word-based homomorphic encryption. Directly addressing the weakest points of the current word-based approach, we propose an assortment of solutions to challenging algorithmic bottlenecks that have hampered existing systems from exploiting the full utility of ring operations in large characteristic. As our starting point, we have proposed three distinct approaches to inversion. These lead to efficient algorithms for division, zero test, equality check, thresholding, comparison, and square root, mostly in terms of approximation-based algorithms. While many of these operations involve unsurprisingly high degree polynomials (hence require evaluation of deep circuits), our implementation experiments give reasonable timings for limited applications. The most practical use of these

techniques remains in applications where all but a small number of gates are addition and multiplication gates, with approximation-based algorithms applied only just before decryption. While we have focused on the F-NTRU variant of the Stehlé and Steinfeld scheme, much of what we explore here is system agnostic and can be adapted to any word-based FHE.

We next tackled the encrypted sorting problem and proposed two depth optimized sorting algorithms for efficient homomorphic evaluation. Circuit depth is intimately related to the parameter sizes in leveled homomorphic encryption implementations and therefore directly affect the overall performance of the homomorphic circuit evaluation. Existing sorting algorithms are not optimized for homomorphic evaluation. To close this gap we presented the depth analysis for several classical sorting algorithms: Bubble sort, Insertion Sort, Odd Even Sort, Odd Even Merge Sort, Merge Sort, and Bitonic Sort. Inspired by the performance of Merge Sort we introduced two new depth-optimized sorting algorithms which achieve a circuit depth of $\mathcal{O}(\log(N) + \log(\ell))$. To study the real-life performance of our sorting algorithms, we instantiated an NTRU based SHE scheme in the DHS library and presented simulation results for selected array lengths. For this we determined the ideal parameter choices, e.g. modulus cutting levels to cope with noise growth and Hermite work factor estimates to ensure reasonable security margins. The implementation performs favorably achieving significant speedup over the proposal in [39] for similar array lengths.

We proposed another method, polynomial rank sort, which performs significantly better than previous algorithms and provides a depth and cost-optimized circuit for homomorphic sorting. It reduces the number of ciphertext multiplications to $\mathcal{O}(N^2)$ for sorting an array of N elements without packing. Furthermore, the algorithm is suitable for parallel implementation as the algorithm steps are designed to take ad-

vantage of multiple functional units. When batching is enabled, we sort the whole list with only $N/2$ comparisons followed by only N multiplications. Proposed batching method also reduces the ciphertext size from kN to N where k is the bit-length of the input elements. Although costly key switching operations are required to enable batching, the timing results demonstrate that its overall advantage proves to be much more significant than its overhead in BGV homomorphic encryption scheme. The new algorithm (along with its two precursor algorithms in our earlier work, namely direct and greedy sort algorithms) is generic and can be used with other recent FHE schemes. Although we expect a performance gain when the new algorithm is implemented using other FHE schemes, this, however, depends on the choice of the scheme and the trade-off between the costs of key switching and ciphertext multiplication, which is left as future work.

This work is also the first to present a blind web search engine prototype in which both keywords and responses are encrypted in an end-to-end fashion. This model can be be used in specialized indexes of the web that is focused on specific content and can be eventually extended to a generic web search engine. We used a hybrid adaptation of KO PIR for search look-ups. We also explored the realm of multi-keyword search and combined our homomorphic search with a PSI solution that uses the roots of a polynomial to embed information. This allows user to submit search queries with multiple keywords. We implemented our search model using leveled FHE scheme F-NTRU and by enabling an NVIDIA Titan Xp with 3840 CUDA cores. Our experiments show that what was possible in theory, is also possible in practice if we index a targeted portion of the web as we can retrieve results in under 0.2 milliseconds for a single keyword and around 3.2 milliseconds for a 2-keyword search.

Appendix A

Appendix

A.1 Truth tables for boolean gates.

a	b	$a \oplus b$	$a \wedge b$	$a \vee b$	$\neg a$	$\neg b$	$\neg(a \oplus b)$	$\neg a \wedge b$
0	0	0	0	0	1	1	1	0
0	1	1	0	1	1	0	0	1
1	0	1	0	1	0	1	0	0
1	1	0	1	1	0	0	1	0

A.2 Truth tables for comparison and equality check.

a	b	$a < b$	$a = b$
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1

a_1	a_0	b_1	b_0	$(a_1a_0) < (b_1b_0)$	$(a_1a_0) = (b_1b_0)$
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	0	1

A.3 Greedy sort example.

We demonstrate a simple example for an input vector with four elements $A = \langle [\![1]\!], [\![3]\!], [\![4]\!], [\![3]\!] \rangle$. The comparison matrix M will be evaluated as

$$M = \begin{pmatrix} 0 & [\![1]\!] & [\![1]\!] & [\![1]\!] \\ [\![0]\!] & 0 & [\![1]\!] & [\![0]\!] \\ [\![0]\!] & [\![0]\!] & 0 & [\![0]\!] \\ [\![0]\!] & [\![1]\!] & [\![1]\!] & 0 \end{pmatrix}.$$

The outputs for $N = 4$ can be computed as follows. Note that in each group $\theta_{r,i}$ selects only one index i value for each output position r. At the end we have the output vector $B = \langle [\![1]\!], [\![3]\!], [\![3]\!], [\![4]\!] \rangle$ i.e. encryptions of the ordered input numbers

$\langle 1, 3, 3, 4 \rangle$.

$$
\begin{aligned}
B_0 &= A_0\theta_{0,0} \oplus A_1\theta_{0,1} \oplus A_2\theta_{0,2} \oplus A_3\theta_{0,3} \\
&= A_0(M_{01}M_{02}M_{03}) \oplus A_1(M_{10}M_{12}M_{13}) \\
&\oplus A_2(M_{20}M_{21}M_{23}) \oplus A_3(M_{30}M_{31}M_{32}) \\
&= A_0 \cdot [\![1]\!] \oplus A_1 \cdot [\![0]\!] \oplus A_2 \cdot [\![0]\!] \oplus A_3 \cdot [\![0]\!] \\
&= A_0 \\
B_1 &= A_0\theta_{1,0} \oplus A_1\theta_{1,1} \oplus A_2\theta_{1,2} \oplus A_3\theta_{1,3} \\
&= A_0[M_{10}(M_{02}M_{03}) \oplus M_{20}(M_{01}M_{03}) \oplus M_{30}(M_{01}M_{02})] \\
&\oplus A_1[M_{01}(M_{12}M_{13}) \oplus M_{21}(M_{10}M_{13}) \oplus M_{31}(M_{10}M_{12})] \\
&\oplus A_2[M_{02}(M_{21}M_{23}) \oplus M_{12}(M_{20}M_{23}) \oplus M_{32}(M_{20}M_{21})] \\
&\oplus A_3[M_{03}(M_{31}M_{32}) \oplus M_{13}(M_{30}M_{32}) \oplus M_{23}(M_{30}M_{31})] \\
&= A_0 \cdot [\![0]\!] \oplus A_1 \cdot [\![0]\!] \oplus A_2 \cdot [\![0]\!] \oplus A_3 \cdot [\![1]\!] \\
&= A_3 \\
B_2 &= A_0\theta_{2,0} \oplus A_1\theta_{2,1} \oplus A_2\theta_{2,2} \oplus A_3\theta_{2,3} \\
&= A_0[M_{10}(M_{20}(M_{03}) \oplus M_{30}(M_{02})) \oplus M_{20}(M_{30}M_{01})] \\
&\oplus A_1[M_{01}(M_{21}(M_{13}) \oplus M_{31}(M_{12})) \oplus M_{21}(M_{31}M_{10})] \\
&\oplus A_2[M_{02}(M_{12}(M_{23}) \oplus M_{32}(M_{21})) \oplus M_{12}(M_{32}M_{20})] \\
&\oplus A_3[M_{03}(M_{13}(M_{32}) \oplus M_{23}(M_{31})) \oplus M_{13}(M_{23}M_{30})] \\
&= A_0 \cdot [\![0]\!] \oplus A_1 \cdot [\![1]\!] \oplus A_2 \cdot [\![0]\!] \oplus A_3 \cdot [\![0]\!] \\
&= A_1 \\
B_3 &= A_0\theta_{3,0} \oplus A_1\theta_{3,1} \oplus A_2\theta_{3,2} \oplus A_3\theta_{3,3} \\
&= A_0(M_{10}(M_{20}M_{30})) \oplus A_1(M_{01}(M_{21}M_{31})) \\
&\oplus A_2(M_{02}(M_{12}M_{32})) \oplus A_3(M_{03}(M_{13}M_{23})) \\
&= A_0 \cdot [\![0]\!] \oplus A_1 \cdot [\![0]\!] \oplus A_2 \cdot [\![1]\!] \oplus A_3 \cdot [\![0]\!] \\
&= A_2
\end{aligned}
$$

A.4 Direct sort example.

If we demonstrate an example for Direct Sort using the same input vector

$$A = \langle [\![1]\!], [\![3]\!], [\![4]\!], [\![3]\!] \rangle$$

, the comparison matrix M and the index vector σ will be obtained as

$$\begin{aligned} M &= \begin{pmatrix} 0 & [\![1]\!] & [\![1]\!] & [\![1]\!] \\ [\![0]\!] & 0 & [\![1]\!] & [\![0]\!] \\ [\![0]\!] & [\![0]\!] & 0 & [\![0]\!] \\ [\![0]\!] & [\![1]\!] & [\![1]\!] & 0 \end{pmatrix} \\ \sigma &= \begin{pmatrix} [\![0]\!] & [\![2]\!] & [\![3]\!] & [\![1]\!] \end{pmatrix}. \end{aligned}$$

Then, the elements of the sorted set are obtained as follows

$$
\begin{aligned}
B_0 &= (\sigma_0 \asymp 0) \cdot A_0 \oplus (\sigma_1 \asymp 0) \cdot A_1 \oplus (\sigma_2 \asymp 0) \cdot A_2 \oplus (\sigma_3 \asymp 0) \cdot A_3 \\
&= [\![1]\!] \cdot A_0 + [\![0]\!] \cdot A_1 + [\![0]\!] \cdot A_2 + [\![0]\!] \cdot A_3 \\
&= A_0 \\
B_1 &= (\sigma_0 \asymp 1) \cdot A_0 \oplus (\sigma_1 \asymp 1) \cdot A_1 \oplus (\sigma_2 \asymp 1) \cdot A_2 \oplus (\sigma_3 \asymp 1) \cdot A_3 \\
&= [\![0]\!] \cdot A_0 + [\![0]\!] \cdot A_1 + [\![0]\!] \cdot A_2 + [\![1]\!] \cdot A_3 \\
&= A_3 \\
B_2 &= (\sigma_0 \asymp 2) \cdot A_0 \oplus (\sigma_1 \asymp 2) \cdot A_1 \oplus (\sigma_2 \asymp 2) \cdot A_2 \oplus (\sigma_3 \asymp 2) \cdot A_3 \\
&= [\![0]\!] \cdot A_0 + [\![1]\!] \cdot A_1 + [\![0]\!] \cdot A_2 + [\![0]\!] \cdot A_3 \\
&= A_1 \\
B_3 &= (\sigma_0 \asymp 3) \cdot A_0 \oplus (\sigma_1 \asymp 3) \cdot A_1 \oplus (\sigma_2 \asymp 3) \cdot A_2 \oplus (\sigma_3 \asymp 3) \cdot A_3 \\
&= [\![0]\!] \cdot A_0 + [\![0]\!] \cdot A_1 + [\![1]\!] \cdot A_2 + [\![0]\!] \cdot A_3 \\
&= A_2
\end{aligned}
$$

As expected, the output vector is $B = \langle A_0, A_3, A_1, A_2 \rangle$ or an encrypted $\langle [\![1]\!], [\![3]\!], [\![3]\!], [\![4]\!] \rangle$.

A.5 Polynomial rank sort example.

Similar to Greedy and Direct Sort, we demonstrate the working of the new method in an example with the same encrypted input set $A = \langle [\![1]\!], [\![3]\!], [\![4]\!], [\![3]\!] \rangle$ and the

comparison matrix:

$$M = \begin{pmatrix} 0 & [\![1]\!] & [\![1]\!] & [\![1]\!] \\ [\![0]\!] & 0 & [\![1]\!] & [\![0]\!] \\ [\![0]\!] & [\![0]\!] & 0 & [\![0]\!] \\ [\![0]\!] & [\![1]\!] & [\![1]\!] & 0 \end{pmatrix}.$$

The surrogates are evaluated as

$$\begin{aligned} P_0(x) &= (M_{01} + M_{10}x) \cdot (M_{02} + M_{20}x) \cdot (M_{03} + M_{30}x) \\ &= [\![1]\!] \cdot [\![1]\!] \cdot [\![1]\!] \\ &= [\![1]\!] \\ P_1(x) &= (M_{10} + M_{01}x) \cdot (M_{12} + M_{21}x) \cdot (M_{13} + M_{31}x) \\ &= [\![x]\!] \cdot [\![1]\!] \cdot [\![x]\!] \\ &= [\![x^2]\!] \\ P_2(x) &= (M_{20} + M_{02}x) \cdot (M_{21} + M_{12}x) \cdot (M_{23} + M_{32}x) \\ &= [\![x]\!] \cdot [\![x]\!] \cdot [\![x]\!] \\ &= [\![x^3]\!] \\ P_3(x) &= (M_{30} + M_{03}x) \cdot (M_{31} + M_{13}x) \cdot (M_{32} + M_{23}x) \\ &= [\![x]\!] \cdot [\![1]\!] \cdot [\![1]\!] \\ &= [\![x]\!] \end{aligned}$$

and finally, the output can be seen as a polynomial, whose coefficients are the

elements of the input set ordered in terms of their magnitudes

$$\begin{aligned} B(x) &= A_0 \cdot P_0(x) + A_1 \cdot P_1(x) + A_2 \cdot P_2(x) + A_3 \cdot P_3(x) \\ &= A_0 \cdot [\![1]\!] + A_1 \cdot [\![x^2]\!] + A_2 \cdot [\![x^3]\!] + A_3 \cdot [\![x]\!] \\ &= A_0 + A_3 x + A_1 x^2 + A_3 x^3 \\ &= [\![1 + 3x + 3x^2 + 4x^3]\!]. \end{aligned}$$

Bibliography

[1] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[2] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[3] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[4] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 365–377, New York, NY, USA, 1982. ACM.

[5] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[6] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[7] Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*, pages 129–148, 2011.

[8] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, pages 97–106, 2011.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 2011.

[10] N.P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Cryptology ePrint Archive, Report 2011/133, 2011. https://eprint.iacr.org/2011/133.

[11] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 850–867, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[15] Shai Halevi and Victor Shoup. HElib, homomorphic encryption library. Internet Source, 2012. https://github.com/shaih/HElib.

[16] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit (updated implementation). Cryptology ePrint Archive, Report 2012/099, 2015. https://eprint.iacr.org/2012/099.

[17] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1219–1234, New York, NY, USA, 2012. ACM.

[18] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT'11, pages 27–47, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding*, pages 45–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] Yarkın Doröz, Yin Hu, and Berk Sunar. Homomorphic AES evaluation using the modified ltv scheme. *Des. Codes Cryptography*, 80(2):333–358, August 2016.

[21] Martin Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched ntru assumptions. In *Proceedings, Part I, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9814*, pages 153–178, Berlin, Heidelberg, 2016. Springer-Verlag.

[22] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[23] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[24] Yarkın Doröz and Berk Sunar. Flattening ntru for evaluation key free homomorphic encryption. Cryptology ePrint Archive, Report 2016/315, 2016. `http://eprint.iacr.org/2016/315`.

[25] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT (1)*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016.

[26] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

[27] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018. `https://eprint.iacr.org/2018/421`.

[28] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[29] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 360–384, Cham, 2018. Springer International Publishing.

[30] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/1043, 2018. `https://eprint.iacr.org/2018/1043`.

[31] Wei Dai and Berk Sunar. CUDA-accelerated fully homomorphic encryption library. `https://github.com/vernamlab/cuFHE`, 2018.

[32] R.L. Lagendijk, Z. Erkin, and M. Barni. Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. *Signal Processing Magazine, IEEE*, 30(1):82–105, Jan 2013.

[33] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124. ACM, 2011.

[34] Kristin Lauter, Adriana Lopez-Alt, and Michael Naehrig. Private computation on encrypted genomic data. Technical Report MSR-TR-2014-93, Microsoft Research, June 2014.

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

[36] T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. *Cryptology ePrint Archive: Report 2012/323*, June 2012.

[37] JungHee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, volume 8976 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 2014.

[38] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178. ACM, 2009.

[39] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In Goutam Paul and Serge Vaudenay, editors, *Progress in Cryptology INDOCRYPT 2013*, volume 8250 of *Lecture Notes in Computer Science*, pages 262–273. Springer International Publishing, 2013.

[40] N. Emmadi, P. Gauravaram, H. Narumanchi, and H. Syed. Updates on sorting of fully homomorphic encrypted data. In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 19–24, Oct 2015.

[41] H. Narumanchi, D. Goyal, N. Emmadi, and P. Gauravaram. Performance analysis of sorting of fhe data: Integer-wise comparison vs bit-wise comparison. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 902–908, March 2017.

[42] D.E. Knuth. *The Art of Computer Programming*, volume 1-3. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[43] Gizem S. Çetin, Hao Chen, Kim Laine, Kristin Lauter, Peter Rindal, and Yuhou Xia. Private queries on encrypted genomic data. *BMC Medical Genomics*, 10(2):45, Jul 2017.

[44] Miran Kim, Yongsoo Song, and Jung Hee Cheon. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC Medical Genomics*, 10(2):42, Jul 2017.

[45] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure search on encrypted data via multi-ring sketch. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 985–1001, New York, NY, USA, 2018. ACM.

[46] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. Setup-free secure search on encrypted data: Faster and post-processing free. Cryptology ePrint Archive, Report 2018/1235, 2018.

[47] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 41–, Washington, DC, USA, 1995. IEEE Computer Society.

[48] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[49] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373, Oct 1997.

[50] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient pir from ntru. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, volume 8438 of *Lecture Notes in Computer Science*, pages 195–207. Springer Berlin Heidelberg, 2014.

[51] S. Angel, H. Chen, K. Laine, and S. Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, May 2018.

[52] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1243–1255, New York, NY, USA, 2017. ACM.

[53] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[54] Victor Shoup. NTL: A Library for doing Number Theory.

[55] Microsoft SEAL (release 3.2). `https://github.com/Microsoft/SEAL`, February 2019. Microsoft Research, Redmond, WA.

[56] Wei Dai and Berk Sunar. CUDA homomorphic encryption library. `https://github.com/vernamlab/cuHE`, 2017.

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

[58] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, Feb 1964.

[59] Jim Basilakis and Bahman Javadi. Efficient parallel binary operations on homomorphic encrypted real numbers. Cryptology ePrint Archive, Report 2018/201, 2018. `https://eprint.iacr.org/2018/201`.

[60] Lea Kissner and Dawn Song. Privacy-preserving set operations. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 241–257, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[61] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating NTRU based homomorphic encryption using GPUs. *2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*, 2014.

[62] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating SWHE based PIRs using GPUs. In *Workshop on Applied Homomorphic Computing – WAHC 2015*, 2015.

[63] Wei Dai and Berk Sunar. cuHE: A homomorphic encryption accelerator library, 2015.

[64] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput*, 19(90):297–301, 1965.