SORTING PROBLEM IN FULLY HOMOMORPHIC ENCRYPTED DATA

by
Gizem Selcan Çetin

Submitted to the Graduate School of Engineering and
Natural Sciences in partial fulfillment of the requirements
for the degree of Master of Science

Sabancı University

August, 2014

SORTING PROBLEM IN FULLY HOMOMORPHIC ENCRYPTED DATA

Approved by:

Assoc. Prof. Dr. Erkay Savaş (Thesis Supervisor)	.....
Assoc. Prof. Dr. Yücel Saygın	.....
Assoc. Prof. Dr. Cem Güneri	.....

Date of Approval: .....

SORTING PROBLEM IN FULLY HOMOMORPHIC ENCRYPTED DATA

Gizem Selcan Çetin

Computer Science and Engineering, Master's Thesis, 2014

Thesis Supervisor: Erkay Savaş

Abstract

Fully Homomorphic Encryption (FHE) schemes allow users to perform computations over encrypted data without decrypting the ciphertext. This is possible via two operations which are bitwise addition and multiplication, namely logical XOR and logical AND operations, which can be applied over the bits individually encrypted under the fully homomorphic encryption scheme. Since any Boolean circuit can be realized using only AND and XOR gates, they can be used to build circuits for the computation of even more complicated operations over encrypted data. This property of FHE cryptosystems is especially useful in cloud computing applications, since data owners who use cloud computing for storage and computation, usually tend not to trust servers and for security reasons, they prefer storing their data in encrypted form. By using FHE cryptographic primitives, now servers are allowed to perform any desired task over the encrypted user data without the knowledge of secret key or plaintext. In this thesis, we focus on solving one such task that cloud server performs over encrypted data; sorting the elements of an integer array. We introduce two sorting schemes, both of which are capable of efficiently sorting data in fully homomorphic encrypted form. The technique is obtained by focusing on the minimization of the depth of the sorting circuit in addition to more traditional metrics such as the number of comparisons. The reduced

depth of the sorting network allows a slower growth in the noise of encrypted bits and thereby makes it possible to select smaller parameter sizes for the underlying homomorphic encryption scheme resulting in much faster computation of homomorphic sorting. We present a leveled/batched implementation for the proposed sorting algorithms, using an NTRU based homomorphic encryption library, which yields significant improvements over classical sorting algorithms.

TAM HOMOMORFK ŞİFRELENMİŞ VERİLER ÜZERİNDE SIRALAMA PROBLEMİ

Gizem Selcan Çetin

Bilgisayar Bilimleri ve Mühendisliği, Yükseklisans Tezi, 2014

Tez Danışmanı: Erkay Savaş

Özet

Tam Homomorfik Şifreleme (THS) programları, kullanıcıların şifrelenmiş veri üzerinde her türlü işlemi yapmasına olanak verir. Bu, şifrelenmiş veri bitleri üzerinde uygulanan çarpma ve toplama, bir diğer deyişle mantıksal VE veya ÖZELVEYA işlemleri sayesinde mümkün olur. Her türlü mantıksal devre sadece ÖZELVEYA ve VE mantıksal işlemlerini gerçekleştiren mantıksal kapılar kullanılarak oluşturulabildiği için, bu iki temel THS işlemi, şifreli metinler üzerinde daha karmaşık operasyonların da hesaplanabilmesini sağlar. Bulut bilişim kullanıcıları çoğunlukla bulut sunucularına güvenmemeye meyilli olduklarından, güvenlikleri gereği, bilgilerini şifreleyerek saklama yoluna giderler. Dolayısıyla şifreli veriler üzerinde işlem yapabilmeyi olanaklı kılan homomorfik şifreleme sistemleri, özellikle bulut bilişim uygulamalarında yaygın kullanım alanı bulacaktır. THS sayesinde, bulut sunucuları artık istenilen herhangi bir işlemi, kullanıcının gizli şifresini veya açık veriyi görmeden, THS yapıtaşlarını kullanarak gerçekleyebilir. Bu tez kapsamında, bir sunucunun uygulamak isteyebileceği bu tür işlemlerden biri olan sıralama problemine odaklanılmıştır. Bu amaçla, tam homomorfik şifreleme sistemi ile şifrelenmiş veriyi verimli bir şekilde sıralamaya yarayacak iki yeni sıralama algoritması sunulmuştur. Bu algoritmalar karşılaştırma sayısı gibi geleneksel ölçütlerin yanısıra, oluşacak sıralama devresinin derinliğinin en aza indirgenmesine

odaklanarak tasarlanmışlardır. Derinliğin azaltılması, operasyonlar sırasında şifrelenmiş veri bitlerinde oluşan ve şifre çözümünü olanaksız kılan gürültünün daha yavaş bir şekilde artmasını, dolayısıyla daha küçük güvenlik parametreleriyle çalışılabilmesini sağlamış ve bu da verimin artmasını mümkün kılmıştır. Önerilen sıralama algoritmaları, NTRU temelli THS sistemi icin geliştirilmiş bir yazılım kütüphanesi kullanılarak gerçeklenmiş ve klasik sıralama algoritmalarına göre çok daha iyi sonuçlar verdiği gösterilmiştir.

to all the squirrels who shared my life...

Acknowledgements

First of all, I would like to thank my supervisor Assoc. Prof. Dr. Erkay Savaş for his guidance, patience and motivation throughout my academic life. He, with the experience of many years of academic teaching and advising, perceived that this topic would attract my full attention and introduced me the perfect thesis subject. Without his support and mentoring, this thesis would not have been completed. I am also grateful to members of my thesis defense comittee: Assoc. Prof. Dr. Yücel Saygın and Assoc. Prof. Dr. Cem Güneri for their valuable time.

I would like to express my gratitude to Assoc. Prof. Dr. Berk Sunar and Yarkın Doröz for giving me the opportunity of working with their group and sharing their project with me. I will always remember and appreciate their help.

My labmate, classmate, even once my teaching assistant, but above all, my precious friend Ecem Ünal, my childhood friend, my best friend, my sister -not by blood but from the heart- Duhan Torlak, I cannot thank these people enough for being there for me when I need them. My labmate Alperen Pulur, I would like to thank him for inspiring me with an idea during our braingstorming sessions. I am grateful to all my collegues from our Cryptography and Information Security Laboratory FENS2001, for their priceless friendship.

My special thanks to The Scientific and Technological Research Council of Turkey, TÜBİTAK for financially supporting my graduate study under BİDEB program.

Finally, I would like to thank my family to whom I owe everything. I am beyond lucky to have such an amazing pair of parents Nurten and İbrahim Çetin, a caring sister İrem Tekin, an aunt Nurşen Akın who is always there for me. I have been and always will be grateful for their endless love and support.

Contents

List of Figures

List of Tables

1 Introduction

The idea of performing operations over encrypted data without ever decrypting it, was firstly proposed in [1], and recently became *theoretically* possible due to the *fully homomorphic encryption* (FHE) scheme introduced by Gentry in [2, 3]. The motivation behind the idea is that when users encrypt their data and save them in an untrusted server, and afterwards when they need to perform a computation over the encrypted data, they do not want to go with the trivial solution; namely download the ciphertexts from the server, decrypt them with their secret keys, perform the intended computation on the plaintext data, and possibly encrypt the data and/or results and send them back to the server. Due to its impracticality and/or infeasibility, this is obviously not a convenient way of managing data; since even for a simple operation, many encryption/decryption operations are necessary and the network traffic is increased due to the huge amount of data exchanged between the user and the server. In particular, if the client is using the server in order to reduce his computational workload and storage requirements, for example by outsourcing them to a cloud service, then he will definitely prefer that the server performs the actual operations, and minimize any local computations on client side without sacrificing security and privacy of data involved.

The first fully homomorphic encryption scheme [2, 3] is far from practical and more of a theoretical interest due to its excessive amount of computation and memory requirements. In a short amount of time after the introduction of the first FHE, however, more practical schemes were proposed due to the popularity and relevancy of the subject, especially in cloud computing applications. Consequently, the scientific community started to focus on some practical operations that can be homomorphically performed over the encrypted data.

When managing, storing, and processing confidential information, such as the amount of

financial assets in banking accounts, salary, age, and other sensitive demographic employee information or any other personal data, security and privacy concerns immediately follow. FHE scheme can be profitably used to alleviate the aforementioned concerns. For instance, when a manager at a company wants to take the average of the age or the salaries of the staff of the company, which are private data on personal basis, using FHE she can ask the cloud server to take their arithmetic mean over the encrypted data and return only the encryption of the mean value. Another application would be finding the minimum or maximum values from a set of numbers. More challenging task, for instance, would be sorting an array of encrypted integers homomorphically.

Our goal in this thesis is proposing new sorting schemes that will be advantageous in homomorphic setting since well known sorting algorithms turn out to be not efficient when applied over the encrypted data. In particular, we draw attention that many classical algorithms in computer science may have to be re-designed for efficient homomorphic computation. In the particular case of sorting, we inspect the best known sorting algorithms in the literature, propose new algorithms and compare them in terms of computational complexity.

Since the best FHE schemes are not sufficiently fast yet, we work with relatively small sets of unsorted integers. Moreover, the achieved execution time results for homomorphic computations are much higher than those for plaintext data. However, FHE is a rapidly developing area and as new FHE schemes are likely to appear in the near feature, the sorting of encrypted data will be practical. All the same, our quest for sorting algorithms that are designed to perform better in homomorphic setting will remain a relevant research area.

The organization of the thesis can be outlined as follows

- We take a closer look at the FHE algorithms that can be used for homomorphic computations in Section 2.

- In order to give an idea of the operations that can be computed over homomorphically encrypted ciphertexts, we will briefly go over a few simple boolean circuits which are built using only AND and XOR gates, also known as algebraic normal form, in Section 3. The idea is that these two logical operations can be performed homomorphically. In general, we will see that converting any Boolean function into a special form,

Algebraic Normal Form (ANF), is possible.

- Then, in Section 4, several classical sorting algorithms are analyzed, and we show that some are more suitable than others for leveled homomorphic evaluation. Specifically, we characterize them with respect to a new metric, i.e. the circuit depth. As it turns out, the existing sorting schemes are simply not suitable for homomorphic evaluation.

- In Section 4, we introduce two new depth optimized sorting schemes which lend themselves to shallow circuit evaluation of depths of only $O(\log(N) + \log(\ell))$ and $O(\log_{3/2}(N) + \log(\ell))$ respectively, for sorting N elements, where ℓ represents the size of the array elements in number of bits. Furthermore, we instantiate a somewhat homomorphic encryption scheme (SWHE) based on NTRU, and present implementations of the proposed sorting algorithms using this SWHE scheme in the following section, namely in Section 5. Our results confirm our theoretical analysis, i.e. that the performance of the proposed sorting algorithm scales favorably as N increases. Although the results are still not practical from the time and efficiency point of views, they are promising considering that the overall FHE concept is relatively new, and there is a long way from the start with an almost infeasible solution to a scheme which is practically acceptable. Our work is one step to achieve this goal.

- Finally, in Section 6, we conclude the thesis and outline the possible future work ideas on the subject.

2 Literature Review and Background

An encryption scheme is *fully homomorphic* (FHE scheme) if it permits the efficient evaluation of any boolean circuit or arithmetic function on ciphertexts [1]. Gentry introduced the first FHE scheme [2, 3] based on lattices that supports the efficient evaluation for arbitrary depth circuits. This was followed by a rapid progression on new FHE schemes. van Dijk, et al., proposed a FHE scheme based on ideals defined over integers [4]. In 2010, Gentry and Halevi [5] presented the first actual FHE implementation along with a wide array of optimizations to tackle the infamous efficiency bottleneck of FHEs. Further optimizations for FHE, which also apply to somewhat homomorphic encryption (SWHE) schemes followed including batching and SIMD optimizations, e.g. see [6, 7, 10].

Several newer SWHE & FHE schemes appeared in the literature in recent years. Brakerski, Gentry and Vaikuntanathan proposed a new FHE scheme (BGV) based on the learning with errors (LWE) problem [11]. To cope with noise the authors propose efficient techniques for noise reduction. While not as effective as Gentry's recryption operation, these lightweight techniques limit the noise growth enabling the evaluation of much deeper circuits using only a depth restricted SWHE scheme. The costly recryption primitive is only used to evaluate extremely complicated circuits. In [10] Gentry, Halevi and Smart introduced a LWE-based FHE scheme customized to achieve efficient evaluation of the AES cipher without bootstrapping. Their implementation is highly optimized to for efficient AES evaluation using key and modulus switching techniques [11], batching and SIMD optimizations [7]. Their byte-sliced homomorphic AES implementation takes about 5 minutes to evaluate an AES block.

More recently, Alt-López, Tromer and Vaikuntanathan (ATV) proposed SWHE and FHE schemes based on Stehlé and Steinfeld's generalization of the NTRU scheme [13] that sup-

ports inputs from multiple public keys [12]. Bos et al. [14] introduced a variant of the NTRU FHE scheme along with an implementation. The authors modify the NTRU scheme by adopting a tensor product technique introduced earlier by Brakerski [15] such that the security depends only on standard lattice assumptions. The authors advocate use of the Chinese Remainder Theorem on the message space to improve the flexibility of the scheme. Also, modulus switching is no longer needed due to the reduced noise growth. Doröz, Hu and Sunar propose another variant based on the NTRU scheme in [16]. The implementation is batched, bit-sliced and features modulus switching techniques. The authors also specialize the modulus to reduce the public key size. The authors report an AES implementation which achieves one minute evaluation time per AES block [10]. More recent FHE schemes displayed significant improvements over earlier constructions in both time complexity and in ciphertext size. Nevertheless, both latency and message expansion rates remain roughly two orders of magnitude higher than those of traditional public-key schemes. Bootstrapping [2], relinearization [17], and modulus reduction [11, 17] are indispensable tools for FHEs. In [17, Sec. 1.1], the *relinearization* technique was proposed as a way to re-encrypt quadratic polynomials as linear polynomials under a new key, thereby making their security argument independent of lattice assumptions and dependent only on a standard LWE hardness assumption.

Homomorphic encryption schemes have been used to build a variety of higher level security applications. Lagendijk et al. [8] 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. Using these key operations it becomes possible to achieve more sophisticated privacy-protected DSP heavy services such as face recognition, user clustering, and content recommendation. Cryptographic tools permitting restricted homomorphic evaluation, e.g. Paillier's scheme, and more powerful techniques such as Yao's garbled circuit [22] have been around sufficiently long to be used in a diverse set of applications.

Homomorphic encryption schemes are often used in privacy-preserving data mining applications. Vaidya and Clifton [23] propose to use Yao's circuit evaluation [22] for the comparisons in their k-means clustering algorithm in privacy-preserving case. The secure comparison protocol by Fischlin [24] uses the GM-homomorphic encryption scheme [26] and the

method by Sander et al. [25] to convert the XOR homomorphic encryption in GM scheme into AND homomorphic encryption. The privacy-preserving clustering algorithm for vertically partitioned (distributed) spatio-temporal data [27] uses the Fischlin formulation based on XOR homomorphic secret sharing primitive instead of costly encryption operations.

The tools for somewhat homomorphic encryption developed to achieve fully homomorphic evaluation have only been considered for a few years now for use in applications. For instance, in [18] Lauter et al. consider the problems of evaluating averages, standard deviations, and logistical regressions which provide basic tools for a number of real-world applications in medical, financial, and the advertising domains. The same work also presents a proof-of-concept Magma implementation of a SWHE for the basic operations. The SWHE scheme is based on the ring learning with errors (RLWE) problem proposed earlier by Brakerski and Vaikuntanathan. Cheon et al. [9] present 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. The authors design circuits to compute the distances between two genomic strings. The work designs circuits meticulously to reduce their depths to permit efficient evaluation using BGV-type leveled SWHE schemes. In this work, we follow a route very similar to that given in [9] for sorting.

In [19], Doröz et al. use an NTRU based SWHE scheme to construct a bandwidth efficient private information retrieval (PIR) scheme. Due to the multiplicative evaluation capabilities of the SWHE, the query and response sizes are significantly reduced compared to earlier PIR constructions. The PIR construction is generic and therefore any SWHE, which supports a few multiplicative levels (and many additions), could be used to implement the PIR. The authors also give a leveled and batched reference implementation of their PIR construction including performance figures.

The only homomorphic sorting result we are aware of was reported by Chatterjee et al. in [20]. In this work, for the first time, the authors considered the problem of homomorphically sorting an array using the recently proposed `hcrypt` FHE library [21]. The authors define a number of FHE elements to realize basic homomorphic comparison and swapping operations and then implement the classical Bubble and Insertion sort algorithms using these

homomorphic functions. Noting the exponential rise of evaluation time with the array size, the authors introduce a new approach dubbed **Lazy Sort** which removes the Recrypt operation after additions allowing occasional comparison errors in Bubble Sort. While the array is not perfectly sorted the sorting time is significantly reduced. After Bubble sort the nearly sorted array is then sorted again with a homomorphically evaluated Insertion sort - this time with all Recrypt operations in place. The authors report implementation results with arrays of 5-40 elements (32-bits) which show significant reduction in the evaluation time over direct fully homomorphic evaluation. In the best case, the authors report a 1399 second evaluation time in contrast to 21565 seconds in the fully homomorphic case for an array of size 40. Despite the impressive speed gains, the work opts to alleviate the efficiency bottleneck by relaxing noise management, and by combining classical sorting algorithms instead of targeting the circuit depth of the sorting algorithm. Furthermore, it suffers from the fundamental limitations of the `hcrypt` library:

- Noise management is achieved by recrypting partial results after every major operation. Recrypt is extremely costly and is considered inferior to more modern noise management techniques such as the **modulus reduction** [11] that yield exponential gains in leveled implementations.

- `hcrypt` does not take advantage of **batching** or SIMD techniques [7] which greatly improve homomorphic evaluation performance.

In subsequent sections, we provide a brief summary of the multi-key NTRU-FHE scheme and give a slight explanation on primitive functions that is proposed by Alt-López, Tromer and Vaikuntanathan. Later, we give details of the DHS FHE library, that is used in the implementation, based on a specialized NTRU-FHE version.

2.1 The NTRU-FHE Scheme

In 2012 Alt-López, Tromer and Vaikuntanathan proposed a leveled multi-key FHE scheme (ATV) [12]. The scheme based on a variant of NTRU encryption scheme proposed by Stehlé and Steinfeld [13]. The introduced scheme uses a new operation called relinearization and

existing techniques such as modulus switching for noise control.

Doröz, Hu and Sunar use the same construction in [16] which is a single key version of ATV with reduced key size technique. The operations are performed in the ring, $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, 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 consist of four primitive functions **KeyGen**, **Encrypt**, **Decrypt** and **Eval**. A brief detail of the primitives is as follows:

KeyGen. We choose sequence of primes $q_0 > q_1 > \cdots > q_d$ to use a different q_i in each level. And for each $i = 0, \ldots, d$, at first we sample $u^{(i)}$ and $g^{(i)}$ from χ distribution, then a public and secret key pair is computed for each level as:

$$h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$$

and

$$f^{(i)} = 2u^{(i)} + 1$$

in $R_{q_i} = \mathbb{Z}_{q_i}[x]/\langle x^n + 1\rangle$. And if $f^{(i)}$ is not invertible in this ring, then it needs to be sampled again. Later we create evaluation keys for each level

$$\zeta_\tau^{(i)}(x) = h^{(i)}s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau(f^{(i-1)})^2$$

in $R_{q_{i-1}}$, 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 operation can be applied as,

$$c_1^{(i)} + c_2^{(i)} = \mathsf{Encrypt}(b_1 + b_2)$$

and, AND can be applied similarly,

$$c_1^{(i)} \cdot c_2^{(i)} = \mathsf{Encrypt}(b_1 \cdot b_2)$$

Multiplication operation 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

$$\tilde{c}^{(i-1)}(x) = \sum_{\tau} 2^{\tau} \tilde{c}_{\tau}^{(i-1)}(x)$$

Than 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 $(f^{(i)})^2$. The reason, why this holds, is given in [16]. Later, the modulus switch

$$\tilde{c}^{(i)}(x) = \lfloor q_i / q_{i-1} \tilde{c}^{(i)}(x) \rceil_2$$

decreases the noise by $\log(q_i/q_{i-1})$ bits by diving and multiplying. The operation $\lfloor \cdot \rceil_2$ refers

to rounding and matching the parity bits after worth.

2.2 The DHS FHE Library

A customized version of the NTRU-FHE Scheme that is previously proposed in [16] by Doröz, Hu and Sunar (DHS) is used for the encryption part. 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 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 depth plus 1.

- 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 render possible to evaluated higher depth circuits.

- The special 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 [6, 7] (see also [10]). 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 that it is generally the bottleneck of the running algorithms.

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

3 FHE Instructions

Since we are working on FHE data, in order to build any circuit, we will need bitwise operations and equations in Algebraic Normal Form (ANF) in which we use two fundamental binary operations; multiplication (" $\cdot$ ") and addition (" $\oplus$ "). Both of these operations take two 1-bit inputs and the result is again a 1-bit value. In digital logic, these operations are implemented by AND and XOR gates.

If we perform a simple task such as comparing two numbers of ℓ-bit, we will need two operations; IsEqual and LessThan. The comparison circuit takes two ℓ-bit operands, and the output is only 1 bit. Another task is summing ℓ bits, which is basically computing Hamming Weight of an ℓ-bit number. The output is $\lceil \log(\ell) \rceil$-bit long in this case, since the maximum Hamming Weight value is when all input bits are 1 and sum would be ℓ which is a $\lceil \log(\ell) \rceil$-bit number.

Even though there are some software tools which deal with ANF conversion, they do not consider circuit depth so they are not useful for our main goal which is keeping the circuit as shallow as possible.

3.1 Equality Circuit C_{EQUAL}

The C_{EQUAL} circuit simply compares two ℓ-bit integers X and Y, and outputs 1 if X equals Y, otherwise it outputs 0. We can start by solving the problem verbally. In other words, one can claim that if all bit values in X are the same with corresponding bit values in Y, then the two numbers are equal to each other. We visualize it as a pseudocode as follows,

Input Words: Two ℓ-bit numbers with the following bit representation $X = \langle x_{\ell-1}, \ldots, x_1, x_0 \rangle$ and $Y = \langle y_{\ell-1}, \ldots, y_1, y_0 \rangle$.

Output value: if $(X = Y)$ $z = 1$ else $z = 0$.

if $(x_0 == y_0) \ \wedge \ (x_1 == y_1) \ \wedge \ \ldots \ \wedge \ (x_{\ell-1} < y_{\ell-1})$ **then**

$z = 1$

else

$z = 0$

end if

In Boolean algebra, if we need to check if two bits are identical we can simply use an XOR gate. XOR outputs 0 for the identical bit values and 1 for different bits. Hence, we can formalize the comparison circuit for ℓ-bit numbers as follows:

$$z = (X = Y) = \prod_{i \in [\ell]} (x_i = y_i) = \prod_{i \in [\ell]} (x_i \oplus y_i \oplus 1)$$

Notice that, for FHE computations, multiplication take 2 inputs, so that we are using 2 input AND gates. As a result, the product chain of ℓ elements may be evaluated using a binary tree of depth $\lceil \log(\ell) \rceil$. An example circuit for $\ell = 4$ is given in Figure 1. As seen in the figure, multiplicative depth is $\log(4) = 2$ for $\ell = 4$.

3.2 Less Than Circuit $C_{LESS-THAN}$

In a similar manner, the $C_{LESS-THAN}$ circuit compares two ℓ-bit integers X and Y, and outputs 1 if X is smaller than Y else it outputs 0. The formalization of the operation is given in the following.

Input Words: Two ℓ-bit numbers with the following bit representation $X = \langle x_{\ell-1}, \ldots, x_1, x_0 \rangle$ and $Y = \langle y_{\ell-1}, \ldots, y_1, y_0 \rangle$.

Output value: if $(X < Y)$ $z = 1$ else $z = 0$.

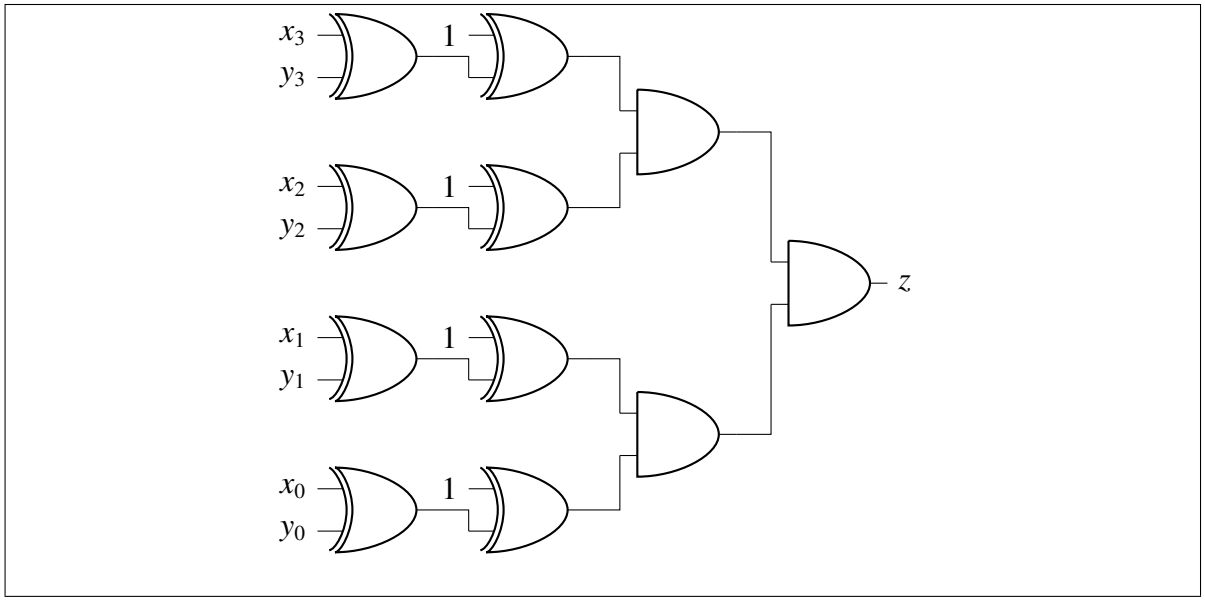


Figure 1: C_{EQUAL} for $\ell = 4$

if $[(x_0 < y_0) \wedge (x_1 == y_1) \wedge \ldots \wedge (x_{\ell-1} == y_{\ell-1})] \vee \ldots \vee [(x_1 < y_1) \wedge (x_2 == y_2) \wedge \ldots \wedge (x_{\ell-1} == y_{\ell-1})] \vee \ldots \vee [(x_{\ell-1} < y_{\ell-1})]$ **then**

$z = 1$

else

$z = 0$

end if

In condition evaluations we can convert the OR (logical disjunction $\vee$) gates to XOR ($\oplus$) gates. To see why this works, first note that $a + b = a \oplus b \oplus (a \cdot b)$ where a and b are bit values. If $a \cdot b = 0$ then $a + b = a \oplus b$. Then, we can make the following proposition for the conjunction cases of the above conditional expressions:

Proposition 1 *In the expression for condition of above IF statements, any two distinct conjunctions ρ and ρ' it holds that $\rho\rho' = 0$.*

Proof Find two distinct conjunctions ρ and ρ' where $(x_k < y_k) \in \rho$ and $(x_l < y_l) \in \rho'$, $k \neq l$. Then if $k < l$, we will have $(x_l == y_l) \in \rho$ and as a result we will have $(x_l < y_l)(x_l == y_l) \in \rho\rho'$. Since $(x_l < y_l)(x_l == y_l) = 0$, $\rho\rho' = 0$. Otherwise, if $k > l$, then we will have $(x_k == y_k) \in \rho'$ and as a result we will have $(x_k < y_k)(x_k == y_k) \in \rho\rho'$. Since $(x_k < y_k)(x_k == y_k) = 0$, $\rho\rho' = 0$. □

According to above proposition, we can convert all OR occurrences to $\oplus$, for which we use the symbol $\sum$ in accumulative cases. We can formalize the comparison circuit as follows:

$$z = (X < Y) = \sum_{i \in [\ell]} \left[(x_i < y_i) \prod_{i<j<\ell} (x_j = y_j) \right]$$

where $(x_i < y_i) = y_i \cdot (x_i \oplus 1)$ and $(x_j = y_j) = y_j \oplus x_j \oplus 1$.
Here, the equality $(x_i < y_i) = y_i \cdot (x_i \oplus 1)$ can be obtained from the truth table for $(x_i < y_i)$ below.

x	y	$(x < y)$
0	0	0
0	1	1
1	0	0
1	1	0

The expansion of the formula gives a sum of products expression where the product with the maximum number of elements occurs when $i = 0$. The product chain contains $\ell + 1$ elements where 2 bits are contributed by the $(x_0 < y_0)$ term and the rest are from the $(y_j \oplus x_j \oplus 1)$ terms. The product of $\ell + 1$ elements may be evaluated using a binary tree, in which case we achieve the minimum depth of $\lceil \log(\ell + 1) \rceil$. An example circuit for LessThan operation is illustrated in Figure 2 for $\ell = 4$.

3.3 Hamming Weight Circuit C_{HW}

Different from the first two instructions, C_{HW} does not have a general structure for different ℓ-bit inputs. In general, an half-adder is used to sum two bits while a full-adder is used for three bits. So, for optimization purposes different number and different type of adders are used for different ℓ values.

A half-adder computes the sum and the carry for the input bits x and y,

$$s = x \oplus y$$

$$c = x \cdot y$$

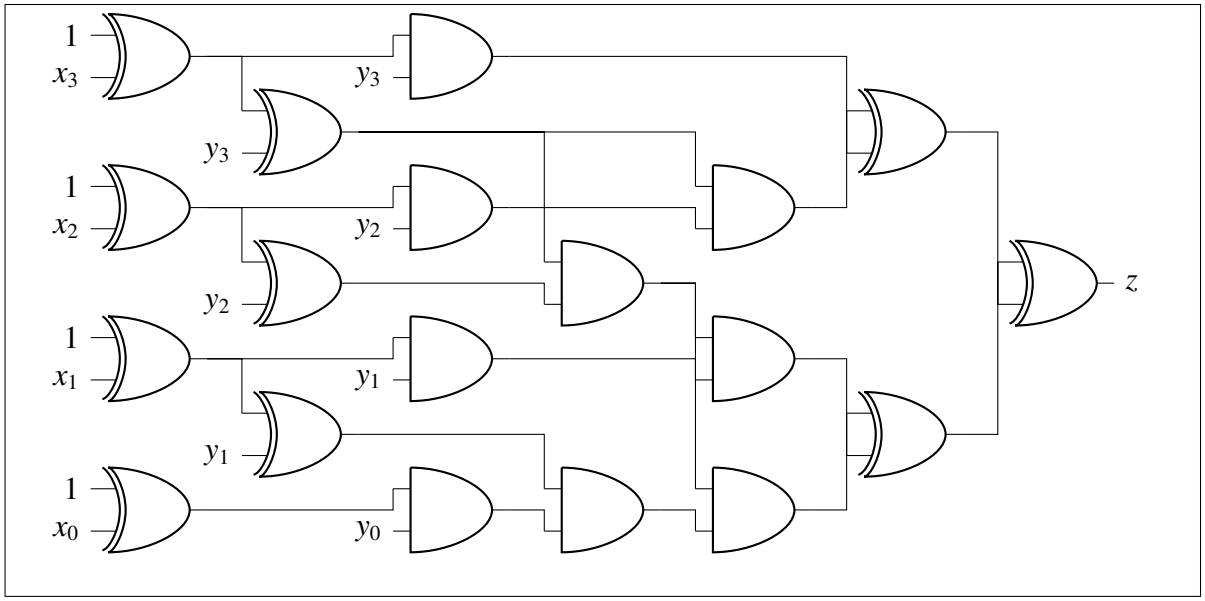


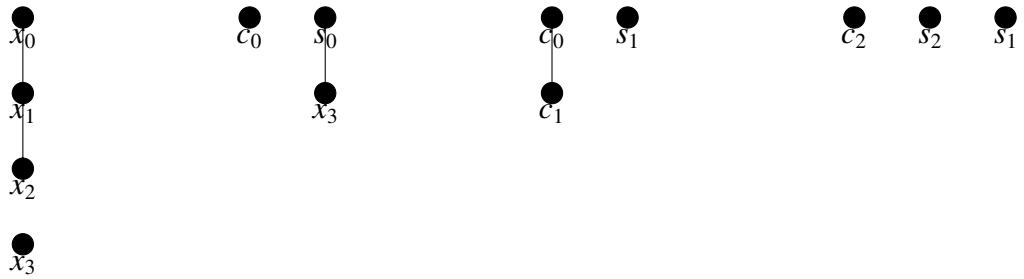
Figure 2: $C_{LESS-THAN}$ for $\ell = 4$

A full-adder computes the sum and the carry for the input bits x, y and z as,

$$s = x \oplus y \oplus z$$

$$c = (x \cdot y) \oplus (x \cdot z) \oplus (y \cdot z)$$

As seen above, both adders take 1 multiplicative depth. For instance, if $\ell = 4$ then we can group the first three bits, and use a full Adder, then continue with a half adder in the second level, and so on. Similar approach is applied for larger ℓ values. As a rough approximation total depth becomes $O(\log(\ell))$. An illustration of the steps is given below.



Here, first a full adder sums the first three bits, x_0, x_1, and x_2, resulting in two bits, namely c_0 and s_0. Since s_0 is aligned with x_3, they are added using a half adder, which produces c_1 and s_1. A final addition of c_0 and c_1 will complete the operation.

4 Sorting Algorithms

Sorting is an old problem in the history of computing. Even though the main idea behind the task is simple, it has been an attractive subject because the solution to this problem has different complexity measures and since it is a simple problem, it has to be solved with the least number of operations/the shortest amount of time/the smallest memory etc. There are numerous sorting algorithms proposed, some are better known and widely used while the others are optimized in the aspect of a specific complexity measure and none of them can be labeled as the best. For the purpose of this thesis, we will focus on comparison based sorting algorithms and the property which we want to optimize will be the multiplicative depth of the sorting circuit.

Sorting network is a comparison based model, which consists of comparator circuits and swapping operations. The difference between classical comparison-based sorting algorithms such as Quick Sort and sorting networks, is that all operations are set in advance, which means that there is no data dependency and additionally sorting networks are built for fixed input size. For instance if an array is reversely ordered which is the worst case, Quick Sort complexity becomes $O(n^2)$, but in the average, complexity of Quick Sort is $O(n\log(n))$ and this is due to the occasional skipping of some steps of the algorithm, depending on the data which can be partially sorted.

On the other hand, in sorting networks, algorithm steps are applied exactly in the same manner for any input data. All the same, sorting networks, despite the impossibility of early termination, are useful for parallel computation. This is because suboperations in each stage of the algorithm are independent from each other, and there is input/output data dependency only between consecutive stages. Since we are trying to sort encrypted inputs we are some-

how blind in each step of the algorithm. As a result, even though data dependent algorithms may be faster, being independent from the input makes sorting networks only candidates for FHE Sorting.

Even though there are some algorithms which are especially desinged as a sorting network, some classical sorting algorithms can also be represented as a network, which FHE properties require. Firstly, we will go over some well known algorithms and then give an analyze for sorting networks. In the figures, the horizontal wires represent the elements of an array to be sorted, vertical lines stand for compare and swap operations, and the black dots are the inputs of the comparison block. After a comparison and swapping operation are applied, the outputs are placed as; the smaller element goes to the upper wire and the larger element is placed on the other. For simplicity of the figures, in this section we used $N = 8$ for the input array size, that is to say, we provide visualization for sorting network of 8 numbers.

4.1 Bubble Sort

Bubble Sort is one of the simplest sorting techniques that permits a rather straightforward implementation using only primitive comparison and swap operations. Chatarjee et al. [20] 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. In each pass the elements are pairwise compared and according to the result, they are swapped to move the smaller element to the left (in case of a horizontal array). The average and worst case performances for an array of N elements are the same: $O(N^2)$. An illustration of a simple application of the algorithm is given in Figure 3.

During homomorphic evaluation since we have no way of knowing when the array is sorted for early termination, we need to make $N - 1$ passes over the array, thus always suffer the worst case complexity. Since after each pass another element in the rightmost portion is sorted the passes decrease by one in number of elements compared and swapped after each pass. Each comparison can be evaluated using a depth $O(\log(\ell))$ circuit for an ℓ-bit wide array elements. The swap only adds one multiplication. Therefore the depth of the Bubble

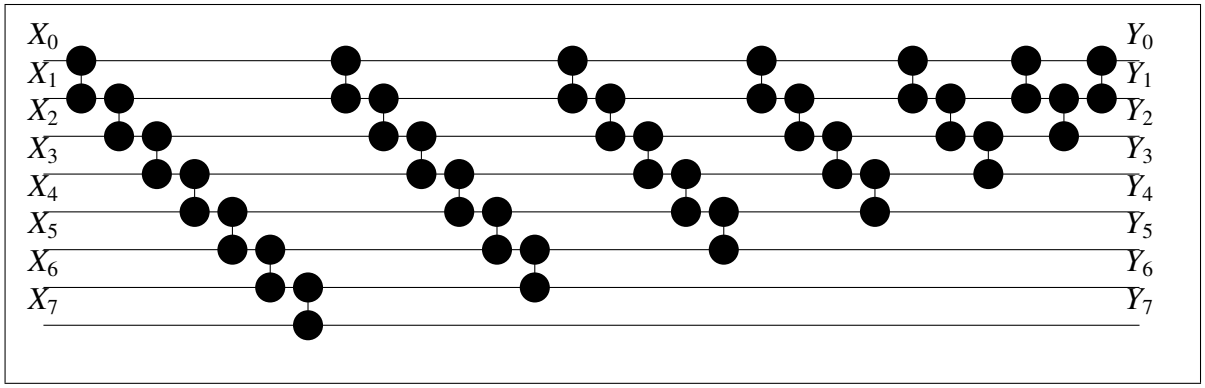


Figure 3: Bubble Sort

Sort circuit will be

$$\begin{aligned} d(C_{B-SORT}) &= [(N-1)+(N-2)+\ldots+1][\log(\ell)+1] \\ &= \frac{N^2-N}{2}[\log(\ell)+1]. \end{aligned}$$

Now we can make some economy by not waiting until a pass is finished to start the next pass. We can *overlap* the passes except with one comparator delay due to the delays we suffer in the very first comparison. A diagram showing the overlapped Bubble Sort circuit is shown in Figure 4. Each node represents a conditional swap operation where the lesser of the input values is moved up and the other down. The number of comparison and swap operations is $N(N-1)/2$. The first pass takes $N-1$ comparator delays, but each additional pass takes only one extra delay, accounting to a total of $N-2$ delays. Therefore overall complexity of this new circuit becomes,

$$\begin{aligned} d(C_{B-SORT}) &= [(N-1)+(N-2)][\log(\ell)+1] \\ &= (2N-3)[\log(\ell)+1] \end{aligned}$$

Note that in their implementation Chatarjee et al. [20] perform the comparison using a carry propagate adder based subtraction circuit resulting in a circuit depth $d(C_{B-SORT}) = (N^2-N)(\ell+1)/2$ instead. While the computational complexity of the scheme is low, the $O(N^2)$ circuit depth is prohibitive.

In the next section, we will give an alternative way of Bubble Sort, known as Odd Even

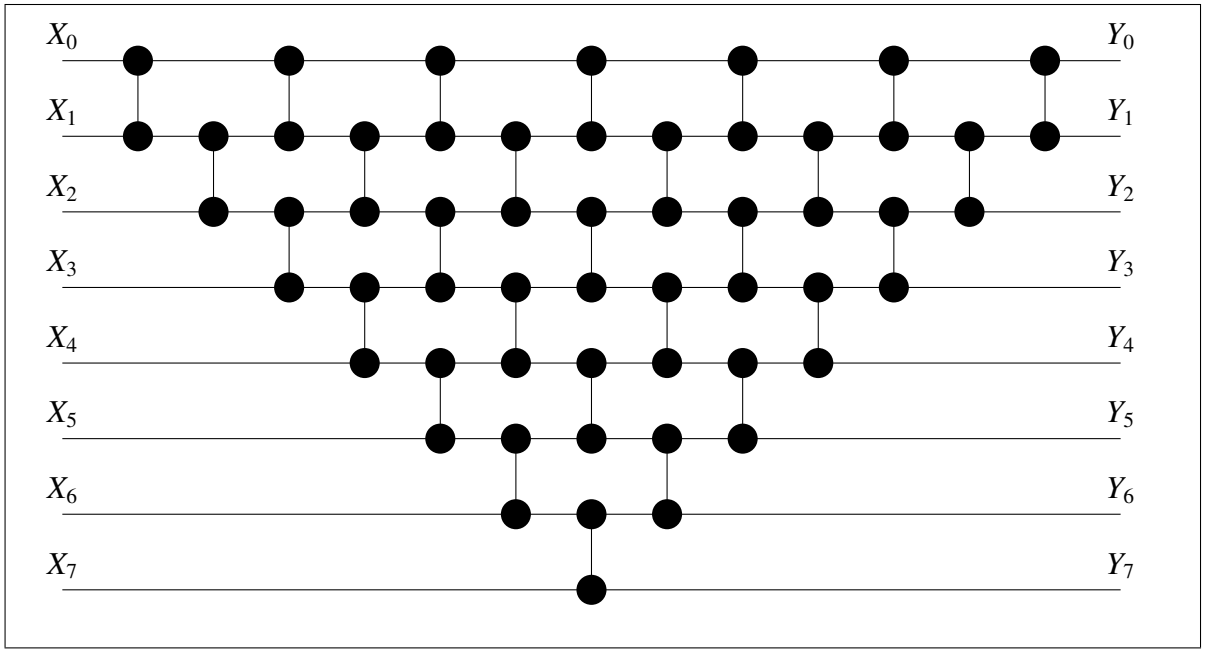


Figure 4: Bubble sort circuit with overlaps

Transposition Sort, with less depth, which is more suitable for parallel programming.

4.2 Odd Even Transposition Sort

A trellis shaped circuit arrangement of Bubble sort network is known as Odd Even Transposition Sort. The method is illustrated in Figure 5. The circuit admits N inputs, and computes the N sorted output values after N passes. The total number of comparisons is $N - 1$ in each two consecutive stage, so overall, there are $N(N - 1)/2$ comparators. And the depth of the circuit is,

$$d(C_{TR-SORT}) = N[\log(\ell) + 1]$$

4.3 Insertion Sort

Insertion sort is a simple sorting algorithm that iteratively builds a sorted array from an unsorted one. The sorted array initially holds only the first element. Then each element is one by one added to the sorted list by comparing it from right to left with the elements in the sorted list until a smaller element is encountered. The new element is then inserted into the sorted array next to the first smaller element when scanning right to left. The average case

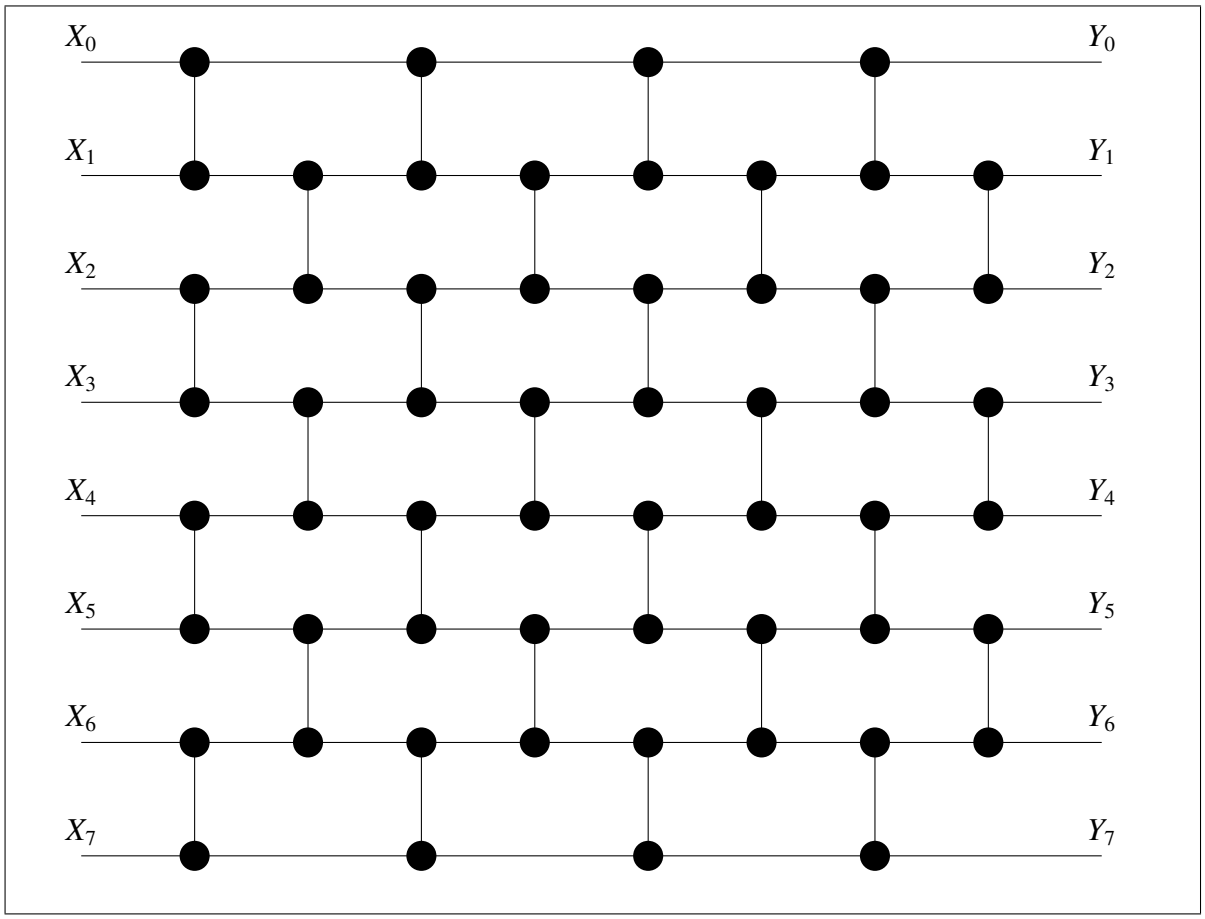


Figure 5: Bubble Sort circuit arranged into a trellis structure, known as Odd Even Transposition Sort

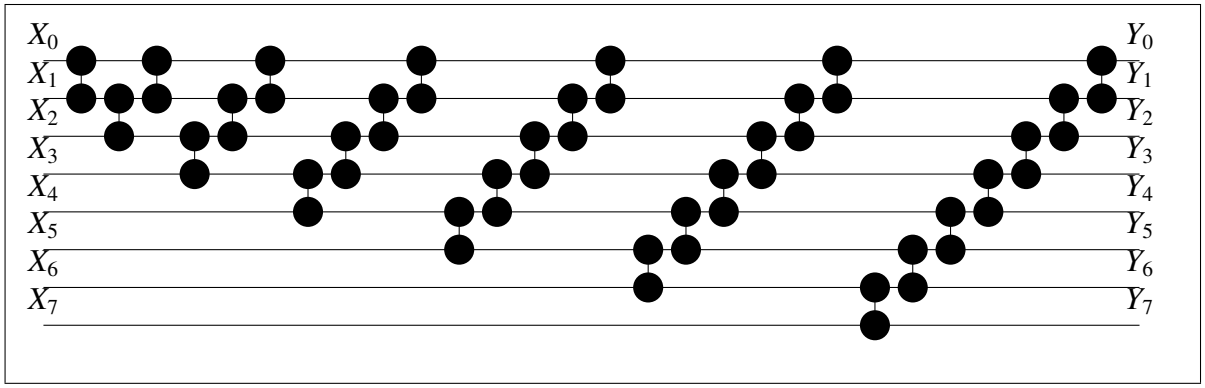


Figure 6: Insertion Sort

and the worst case complexities of the algorithm are $O(N^2)$ while the best case is only $O(N)$. The circuit for conventional Insertion Sort is given in Figure 6.

When considered as a circuit for homomorphic evaluation we need to run the algorithm with the worst case complexity, without making early decisions as in Bubble Sort. We build up the sorted array one by one making increasing number of comparison and conditional swaps. We obtain a circuit depth of

$$\begin{aligned} d(C_{I-SORT}) &= [1 + 2 + \ldots + N - 1][\log(\ell) + 1] \\ &= \frac{N^2 - N}{2}[\log(\ell) + 1]. \end{aligned}$$

Now, when we consider the comparison network C_{I-SORT} in Figure 6 in light of parallel computing, this circuit can be used in a more efficient way by overlapping some comparisons, similar to that we did for C_{B-SORT}. Then, notice that if we compress the circuit in Figure 6 horizontally, we will actually get the same circuit of Figure 4. Consequently, one can claim that, considering sorting networks and FHE sorting, Insertion Sort and Bubble Sort are reduced to the identical algorithm and implementation.

In [20] Chatarjee et al. rely on the fact that after the *imperfect* application of Bubble Sort that the array is *nearly* sorted. Therefore Insertion Sort performs nearly in linear time.

4.4 Merge Sort

Merge Sort is an asymptotically faster algorithm and allows early termination in normal execution, which reduces the complexity. The algorithm is recursively applied by splitting arrays

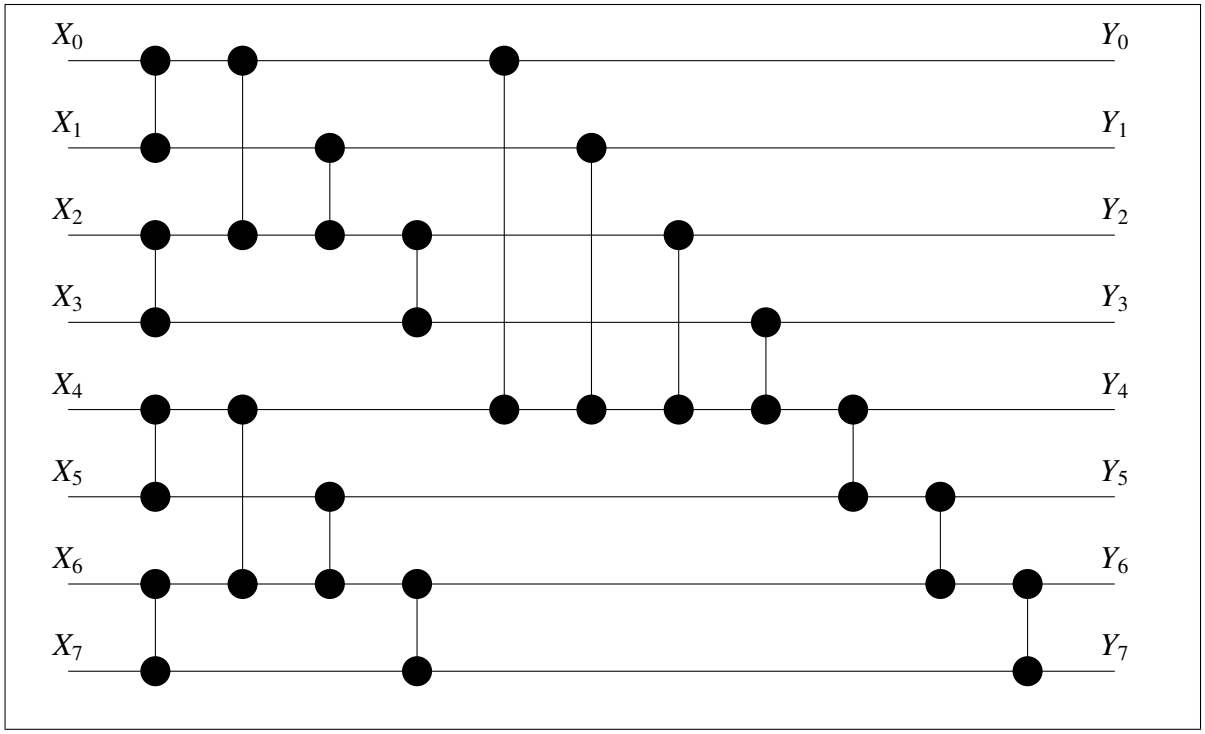


Figure 7: Merge Sort

into smaller ones. In the innermost recursion, arrays of two elements are sorted, where only one comparison is needed in each case. The merge step is started, which combines two individually sorted arrays into a single sorted array. The operation continues until all the array is sorted. The algorithm is highly parallelizable since different parts of the array can be sorted independently until higher levels are reached. In addition, with best, average, and worst case performances of $O(N \log(N))$, Merge Sort is a popular choice for sorting big data. A sorting network representing Merge Sort is illustrated in Figure 7.

The parallel nature of the algorithm makes it an interesting candidate for homomorphic evaluation. However, since early termination is not possible in homomorphic evaluation, an analysis for the depth of the circuit is necessary to assess its efficiency. The number of comparisons is the same as the Bubble Sort algorithm, which is $(N^2 - N)/2$.

Since analyzing the depth of the circuit for the Merge Sort algorithm is different in fully homomorphic computation, an analysis requires in depth treatment, we provide an explanation for the simple case where the number of elements in the array is a power of two. In the innermost part of the recursion, we compare two elements, A_i and A_{i+1}. Consequently, the

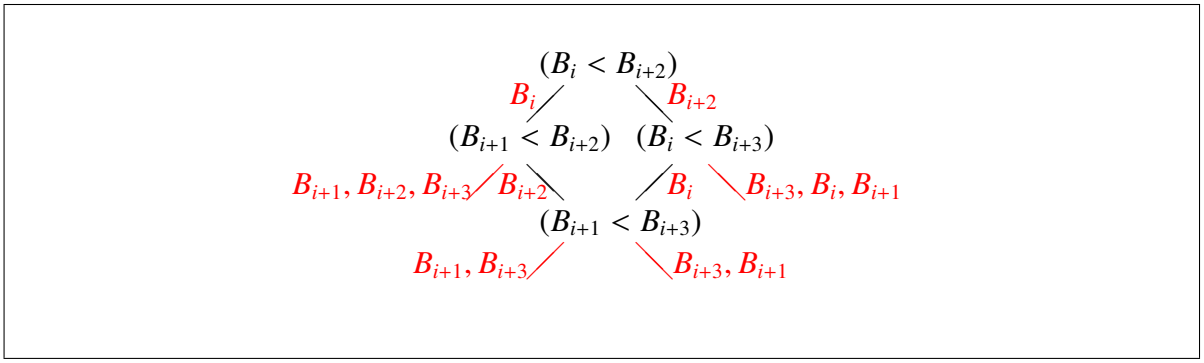


Figure 8: Merging two individually sorted arrays

algebraic normal form for the circuit for each comparison can be derived as follows:

$$\begin{aligned} B_i &= A_i(A_i < A_{i+1}) \oplus A_{i+1}(A_i < A_{i+1})' \\ B_{i+1} &= A_i(A_i < A_{i+1})' \oplus A_{i+1}(A_i < A_{i+1}) \end{aligned}$$

This equations results in circuit of depth $\log \ell + 1$, where ℓ is the bit length of array elements.

Next, we combine two sorted arrays, namely (B_i, B_{i+1}) and (B_{i+2}, B_{i+3}) into a sorted array of $(C_i, C_{i+1}, C_{i+2}, C_{i+3})$. We can illustrate the merge step as in Figure 8.

In Figure 8, the left side of every comparison operation implies the comparison returns true, otherwise it returns false. Depending on the comparison results, we can sort array elements. The sorted array can be traced from top to bottom in the tree in Figure 8. As can be observed from the figure, early termination is possible in normal computation, therefore not all comparisons have to be performed. However, the homomorphic evaluation of sorting requires that all four comparisons need to be performed. The algebraic normal form of the Boolean expressions for the circuit outputs contain product terms with up to four inputs. For example, the formula for C_{i+3} contains the product term

$$B_{i+3}(B_i < B_{i+2})(B_{i+1} < B_{i+2})'(B_{i+1} < B_{i+3})$$

which requires a comparison network with depth 2. This, in turn, results in a circuit with depth $2 \cdot (\log(\ell) + 1)$. Given that there are $log(N)$ levels in the Merge Sort algorithm, the depth

of the circuit can be calculated as

$$\begin{aligned} d(C_{M-SORT}) &= [1 + 2 + \ldots + \log(N)][\log(\ell) + 1] \\ &= \frac{\log^2(N) + \log(N)}{2}[\log(\ell) + 1] \end{aligned}$$

Consequently, we can conclude that asymptotic complexity for the overall depth is found as

$$d(C_{M-SORT}) = O(\log^2(N)\log(\ell))$$

Since in each step, no more than N comparisons are done, number of comparisons is $O(N\log^2(N))$.

In the homomorphic case, the given analysis would imply a better strategy for sorting algorithms where all comparisons can be done first in parallel to decrease the circuit depth. In what follows we introduce a new sorting circuit inspired from this merge sort circuit that achieves depth $O(\log(N) + \log(\ell))$.

4.5 Odd-Even Merge Sort

It has a similar recursive structure to Merge Sort. The algorithm considers two already sorted half-lists, at first sorting odd and even indexed elements seperately and then merging them. Final step is to compare and swap inner adjacent elements. We can illustrate this algorithm as in Figure 9.

Here, let each recursive step of the algorithm be a stage and in a stage let there be k numbers to be sorted in parallel. In order to sort k numbers, we will need $\log(k)$ passes in that stage. In the outermost stage, it is $\log(N)$ passes and in the innermost stage, it will be only 1. So the overall depth can be calculated as;

$$\begin{aligned} d(C_{OEM-SORT}) &= [1 + 2 + \ldots + \log(N)] \\ &= \frac{\log^2(N) + \log(N)}{2} \end{aligned}$$

This is the depth in terms of comparisons, but since we are interested in the depth in term

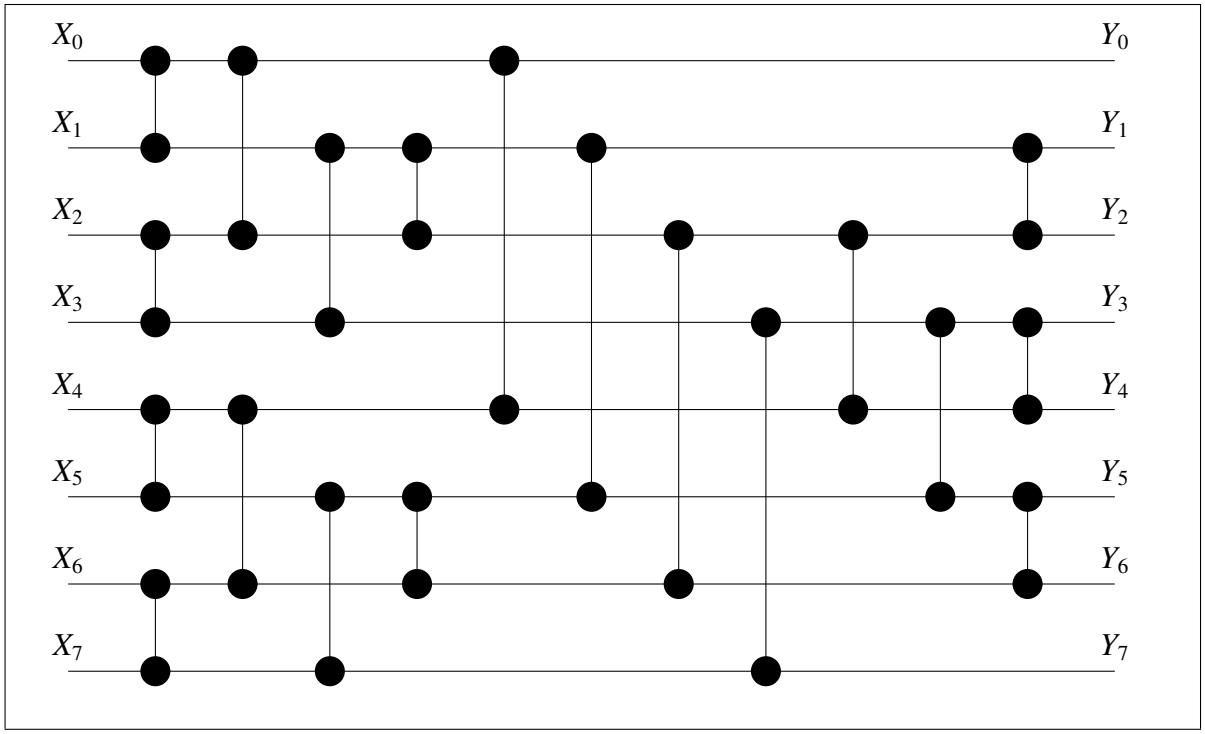


Figure 9: Odd-Even Merge Sort

of multiplication operation we have to consider the depth of one comparison operation, so that the overall depth will be

$$d(C_{OEM-SORT}) = \frac{\log^2(N) + \log(N)}{2}[\log(\ell) + 1]$$

The overall depth complexity is same with classical Merge Sort, with $O(\log^2(N)\log(\ell))$ and the total number of comparisons can be computed as $O(N\log^2(N))$.

4.6 Bitonic Sort

It is a parallelizable algorithm for sorting. It has similar complexity measures with Odd-Even Merge Sort, but with slightly fewer number of comparisons. The sorting network is shown in Figure 10. The depth is computed as,

$$\begin{aligned} d(C_{OEM-SORT}) &= [1 + 2 + \ldots + \log(N)][\log(\ell) + 1] \\ &= \frac{\log^2(N) + \log(N)}{2}[\log(\ell) + 1] \end{aligned}$$

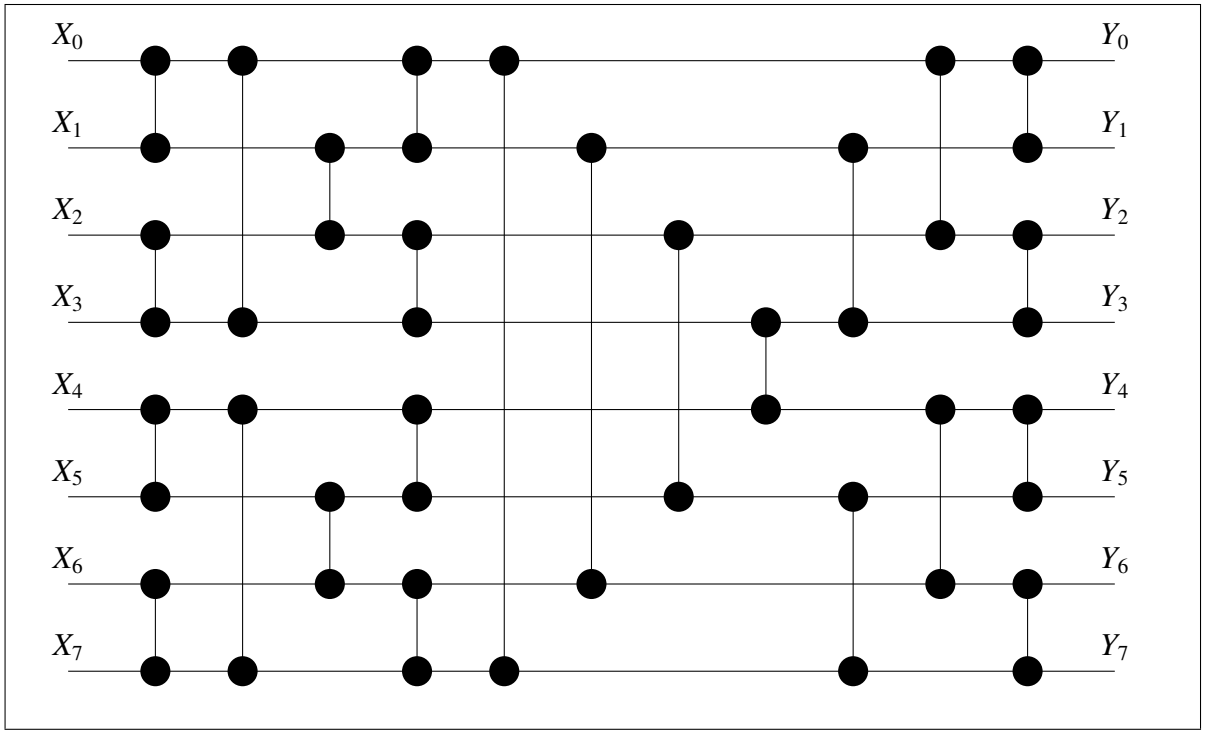


Figure 10: Bitonic Sort

Similarly, the depth is again in the same order with $O(\log^2(N)\log(\ell))$ and as show in Figure 10, in each stage, there are $N/2$ comparisons, which lead to a total of $O(N\log^2(N))$ comparison operations.

4.7 Proposed Depth Optimized Sorting Algorithms

Here we propose two sorting algorithms which are optimized to achieve the shallowest, in terms of multiplicative depth, circuit possible. The algorithm takes an array of elements which are fed to the sorting circuit as an input and gives the ordered elements as the output vector. For these two proposed circuits, we will use the notation C_{EQUAL} and $C_{LESS-THAN}$ introduced in Section 3 where necessary. The algorithms for these circuits is given in the following sections.

For both of these sorting algorithms, we will use a comparison matrix M, which can be described as follows:

The Comparison Matrix

Input vector: $X = \langle X_0, X_1, \ldots, X_{N-1}\rangle$

Output vector: $Y = \langle Y_0, Y_1, \ldots, Y_{N-1} \rangle$

We construct the comparison matrix M as:

$$M = \begin{pmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,N-1} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{N-1,0} & m_{N-1,1} & \cdots & m_{N-1,N-1} \end{pmatrix}.$$

Each $m_{i,j}$ is computed as follows[1]:

$$m_{ij} = \begin{cases} 1 & \text{if } X_i < X_j \\ 0 & \text{else} \end{cases}$$

where $i, j \in N$ and $i < j$. The diagonal elements are self comparisons, i.e. $X_i < X_i$, so we set diagonal values $m_{i,i} = 0$ without any computation. The remaining entries in the lower triangular part of the M, whose indices satisfy $i > j$, are computed as $m_{ji} = m_{ij} \oplus 1$. Note that the lower triangular part corresponds to the comparisons in the form $m_{ji} = (X_i \geq X_j)$.

Notice that, this is a straightforward approach since we are simply comparing every element to every other elemen in the input array. But in terms of depth, it has a significant advantage, since doing all comparisons beforehand (and most importantly in parallel) spares us $d(C_{LESS-THAN})$ depth in each comparison level. In the construction of M we need $N(N-1)/2$ parallel $C_{LESS-THAN}$ operations. This means the depth of this initial step will be 1 in terms of comparison and $\log(\ell + 1)$ in terms of multiplication as stated earlier. By creating this M initially, we will simply be able to evade further $C_{LESS-THAN}$ computations during the execution of later steps and multiplicative depth will be minimized with this approach. We can illustrate this as a sorting network as in Figure 11.

4.7.1 Direct Sort

First proposed method is based on finding the rankings of the input elements. This means that for each element of the input vector we will find an index which corresponds to the order

[1]Note that when there is no ambiguity we will drop the comma, i.e. write $m_{i,j}$ as m_{ij} in the indices for brevity.

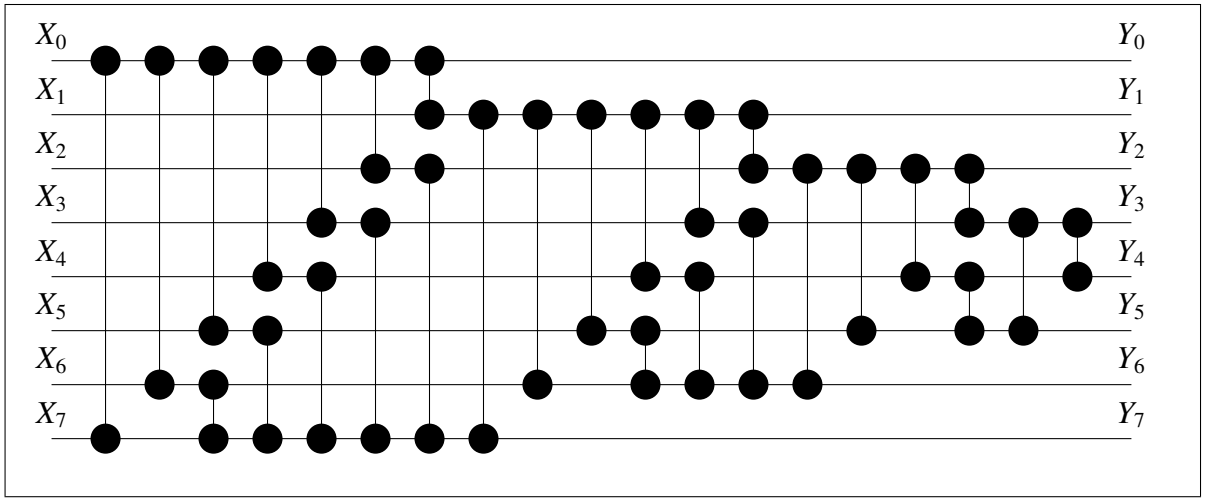


Figure 11: A Sorting Network that compares all pairs in a set - without swapping

of that element in the sorted output vector. For example; for an input vector $X = \langle 2, 4, 3, 1 \rangle$, the rankings would be as $\sigma = (1, 3, 2, 0)$. That is to say, the last element 1 will have index 0 in the output vector, the first element 2 will have index 1 and so on.

In order to retrieve these ranking values we will make use of the comparison matrix M that we have already defined. And σ, the index vector, will be computed as:

$$\sigma = \begin{pmatrix} \sum_{i=0}^{N-1} m_{i,0} & \sum_{i=0}^{N-1} m_{i,1} & \cdots & \sum_{i=0}^{N-1} m_{i,N-1} \end{pmatrix}$$

Note that in M, the summation of all elements in a column, say column j, gives the number of elements, which the element X_j is larger than, because we are adding 1 to the sum for each such value. This summation gives, at the same time, the index of X_j in the sorted output vector. In other words, if an element is larger than k other elements, then this implies that it is the $k + 1^{th}$ largest element and its order is k in a zero-based output array.

For example; for an input vector $X = \langle 1, 3, 4, 3 \rangle$, the comparison matrix M and the index

vector σ will be obtained as:

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\sigma = \begin{pmatrix} 0 & 2 & 3 & 1 \end{pmatrix}$$

And so, the output vector will be $Y = \langle 1, 3, 3, 4 \rangle$.

Now, since all data is in an encrypted form, we have no knowledge of the σ contents, as a result we cannot use it directly. Hence our problem is reduced to retrieving this final output from σ vector. For this, we will make use of C_{EQUAL} such that:

$$Y_j = \sum_{i \in [N]} (\sigma_i = j) X_i \text{ for } j \in [N]$$

Here, we simply compare each element of the index vector σ with each possible index value (which is bounded by $[0, N-1]$) and if there is an equality, then we have the element for the current element of the output vector. Since C_{EQUAL} outputs 0 or 1, when there is a match $(\sigma_i = j)$ it will become 1 which will result in adding X_i to the value of Y_j, and otherwise only 0 will be summed up.

For example, considering our example above we will have for Y_0

$$\begin{aligned} Y_0 &= \sum_{i \in [N]} (\sigma_i = 0) X_i \\ &= (\sigma_0 = 0) X_0 + (\sigma_1 = 0) X_1 + (\sigma_2 = 0) X_2 + (\sigma_3 = 0) X_3 \\ &= (0 = 0) X_0 + (2 = 0) X_1 + (3 = 0) X_2 + (1 = 0) X_3 \\ &= (1) X_0 + (0) X_1 + (0) X_2 + (0) X_3 \\ &= X_0. \end{aligned}$$

4.7.2 Greedy Sort

In our second depth optimized algorithm, we again make use of the comparison matrix M. However, using σ may not be always efficient since computing σ requires homomorphic additions of the elements in the columns of M, which are followed by many multiplications and further additions as shown in the direct evaluation based sorting algorithm. Computation of homomorphic additions for σ will increase the depth of the circuit by around $\log \ell$ levels anf subsequent operations will further increase the depth of the circuit. Therefore we take a more direct approach to compute the output.

Instead, we compute every possible permutation for each index in the sorted array. For instance, to determine Y_0 we need to check if the candidate X element is smaller than all the other element in X, to be set as the smallest element of the sorted array. We can provide the predicate expression yielding the Y_0 assignment explicitly as follows.

if $(X_0 < X_1) \ \wedge \ (X_0 < X_2) \ \wedge \ \ldots \ \wedge \ (X_0 < X_{N-1})$ **then**

$Y_0 = X_0$

else if $\neg(X_0 < X_1) \ \wedge \ (X_1 < X_2) \ \wedge \ \ldots \ \wedge \ (X_1 < X_{N-1})$ **then**

$Y_0 = X_1$

else if $\ldots$ **then**

$\vdots$

end if

Similarly, for Y_1 if an element is smaller than all others except one, then we can conclude that it is the second smallest element. In this case, we compute more possibilities, namely $\binom{N-1}{1}$, in each `if-else` statement since we have the possibility of an element X_i being larger than any of the other elements. The expression for Y_1, which determines the second smallest element is given as follows.

if $[(x_0 < x_1) \ \wedge \ \ldots \ \wedge \ \neg(x_0 < x_{N-1})] \ \vee \ \ldots \ \vee \ [\neg(x_0 < x_1) \ \wedge \ \ldots \ \wedge \ (x_0 < x_{N-1})]$ **then**

$y_1 = x_0$

else if $[(x_1 < x_0) \ \wedge \ \ldots \ \wedge \ \neg(x_1 < x_{N-1})] \ \vee \ \ldots \ \vee \ [\neg(x_1 < x_0) \ \wedge \ \ldots \ \wedge \ (x_1 < x_{N-1})]$ **then**

$y_1 = x_1$

else if $\ldots$ **then**

⋮

end if

Using the comparison matrix M, we can convert the `if-else` statements into logic circuits and compute the sorted elements. The `if-else` statements give us an exact mutually exclusive partitioning in the output assignments. Therefore, we can use XOR (logical exclusive disjunction $\oplus$) gates to combine each statement. For instance, Y_0 evaluated by the following circuit

$$Y_0 = (m_{0,1} \dots m_{0,N-1})\, X_0 \oplus (m_{1,0} \dots m_{1,N-1})\, X_1 \oplus \dots \oplus (m_{N-1,0} \dots m_{N-1,N-2})\, X_{N-1}$$

We can write this equation in a more compact form, if we use a coefficient for each X_i, such as $\theta_{t,i}$, where t stands for the index of Y_t. Using $t = 0$ we have

$$\theta_{0,i} = m_{i,0} \dots m_{i,k} \dots m_{i,N-1} \quad \text{where } i \neq k$$

and the overall equation becomes

$$Y_0 = \theta_{0,0} X_0 \oplus \dots \oplus \theta_{0,N-1} X_{N-1}\ .$$

In Section 3, we give a proposition claiming that we can convert OR gates to XOR gates, when at most one conjunction outputs 1. The same rule applies here as well. We can give the following proposition for the conjunction cases of X_i, to show that it can either have only one conjunction that outputs 1 or none:

Proposition 2 *In the expression for $\theta_{t,i}$ for element X_i any two distinct conjunctions ρ and ρ' it holds that $\rho\rho' = 0$.*

Proof In order to evaluate all the combinations we always find $m_{k,l} \in \rho$ and $m_{l,k} \in \rho'$ for some $k, l \in N - 1$. Otherwise $\rho = \rho'$, a contradiction. Since $\rho\rho'$ will contain contain the conjunction $m_{k,l} m_{l,k}$ we always have $\rho\rho' = 0$ by $m_{k,l} = m_{l,k} \oplus 1$. □

Now we can freely convert all occurrences of OR's to $\oplus$. Hence, the circuit for Y_1 becomes

Sorting Circuit C_{G-SORT}
Input vector: $x = \langle x_0, x_1, \ldots, x_{N-1} \rangle$
Output vector: $y = \langle y_0, y_1, \ldots, y_{N-1} \rangle$ $y = C_{G-SORT}(x)$ is defined in three steps:

Step 1: Using $C_{LESS-THAN}$ compute $m_{i,j}$ where $i, j \in N$ and $i < j$ as

$$m_{ij} = \begin{cases} 1 & \text{if } x_i < x_j \\ 0 & \text{else} \end{cases}$$

Also set $m_{ii} = 0$ and $m_{ji} = m_{ij} \oplus 1$ for $j > i$.

Step 2: Compute $\theta_{t,i}$ for $t, i \in [N]$ as

$$\theta_{t,i} = \sum_{\substack{k_1=0 \\ k_1 \neq i}}^{N-t} m_{k_1 i} \sum_{\substack{k_2=k_1+1 \\ k_2 \neq i}}^{N-t+1} m_{k_2 i} \ldots \sum_{\substack{k_t=k_{t-1}+1 \\ k_t \neq i}}^{N-1} m_{k_t i} \prod_{\substack{j=0 \\ j \neq i \\ j \neq k_1, \ldots, k_t}}^{N-1} m_{ij}$$

Step 3: Compute the output vector y_t for $t \in [N]$ as

$$y_t = \sum_{i=0}^{N-1} x_i \theta_{t,i}$$

Figure 12: Proposed depth optimized greedy sorting circuit $y = C_{G-SORT}(x)$

$$\begin{aligned} Y_1 =& [m_{0,1} m_{0,2} \ldots m_{0,N-2} m_{N-1,0} \oplus m_{0,1} m_{0,2} \ldots m_{N-2,0} m_{0,N-1} \oplus \ldots m_{1,0} m_{0,2} \ldots m_{0,N-2} m_{0,N-1}] x_0 \\ & \oplus \ldots \oplus [m_{N-1,0} m_{N-1,1} \ldots m_{N-1,N-3} m_{N-2,N-1} \oplus m_{N-1,0} m_{N-1,1} \ldots m_{N-3,N-1} m_{N-1,N-2} \oplus \ldots \oplus \\ & m_{0,N-1} m_{N-1,1} \ldots m_{N-1,N-3} m_{N-1,N-2}] x_{N-1} \ . \end{aligned}$$

In a more general formula, the output $Y = C_{G-SORT}(x)$ is computed as;

$$
\begin{aligned}
Y_0 &= \sum_{i=0}^{N-1} X_i \prod_{\substack{j=0 \\ j\neq i}}^{N-1} m_{ij} \\
Y_1 &= \sum_{i=0}^{N-1} X_i \sum_{\substack{k_1=0 \\ k_1\neq i}}^{N-1} m_{k_1 i} \prod_{\substack{j=0 \\ j\neq i,k_1}}^{N-1} m_{ij} \\
Y_2 &= \sum_{i=0}^{N-1} X_i \sum_{\substack{k_1=0 \\ k_1\neq i}}^{N-2} m_{k_1 i} \sum_{\substack{k_2=k_1+1 \\ k_2\neq i}}^{N-1} m_{ki} \prod_{\substack{j=0 \\ j\neq i,k_1,k_2}}^{N-1} m_{ij} \\
&\vdots \qquad\qquad \vdots \\
Y_{N-1} &= \sum_{i=0}^{N-1} X_i \sum_{\substack{k_1=0 \\ k_1\neq i}}^{1} m_{k_1 i} \sum_{\substack{k_2=k_1+1 \\ k_2\neq i}}^{2} m_{k_2 i} \\
&\quad \cdots \sum_{\substack{k_{N-1}=k_{N-2}+1 \\ k_{N-1}\neq i}}^{N-1} m_{k_{N-1} i} \prod_{\substack{j=0 \\ j\neq i,k_1,\ldots k_t}}^{N-1} m_{ij}
\end{aligned}
$$

Each output of the circuit C_S computes a summation of the input values $X_0, \ldots, X_{N-1}$, where the values are weighted with $\theta_{t,i}$. Note that $\theta_{t,i}$ evaluates a logic expression that tells us whether X_i ends up in position t, i.e. $Y_t = X_i$, after sorting. For this sums over all the possible combinations that would result in i^{th} input value having order t. The sorting circuit is concisely defined in Figure 12.

In Figure 13 we give a toy example that evaluates C_{G-SORT} for an input array of size $N = 4$.

Toy Example: $N = 4$
Input vector: $x = \langle x_0, x_1, x_2, x_3 \rangle = \langle 2, 4, 1, 2 \rangle$
Output vector: $y = \langle y_0, y_1, y_2, y_3 \rangle$

The circuit $y = C_{G-SORT}(x)$ is instantiated for $N = 4$ as

$$\begin{aligned}
y_0 &= x_0(m_{01}m_{02}m_{03}) \oplus x_1(m_{10}m_{12}m_{13}) \\
&\oplus x_2(m_{20}m_{21}m_{23}) \oplus x_3(m_{30}m_{31}m_{32}) \\
y_1 &= x_0[m_{10}(m_{02}m_{03}) \oplus m_{20}(m_{01}m_{03}) \oplus m_{30}(m_{01}m_{02})] \\
&\oplus x_1[m_{01}(m_{12}m_{13}) \oplus m_{21}(m_{10}m_{13}) \oplus m_{31}(m_{10}m_{12})] \\
&\oplus x_2[m_{02}(m_{21}m_{23}) \oplus m_{12}(m_{20}m_{23}) \oplus m_{32}(m_{20}m_{21})] \\
&\oplus x_3[m_{03}(m_{31}m_{32}) \oplus m_{13}(m_{30}m_{32}) \oplus m_{23}(m_{30}m_{31})] \\
y_2 &= x_0[m_{10}(m_{20}(m_{03}) \oplus m_{30}(m_{02})) \oplus m_{20}(m_{30}m_{01})] \\
&\oplus x_1[m_{01}(m_{21}(m_{13}) \oplus m_{31}(m_{12})) \oplus m_{21}(m_{31}m_{10})] \\
&\oplus x_2[m_{02}(m_{12}(m_{23}) \oplus m_{32}(m_{21})) \oplus m_{12}(m_{32}m_{20})] \\
&\oplus x_3[m_{03}(m_{13}(m_{32}) \oplus m_{23}(m_{31})) \oplus m_{13}(m_{23}m_{30})] \\
y_3 &= x_0(m_{10}(m_{20}m_{30})) \oplus x_1(m_{01}(m_{21}m_{31})) \\
&\oplus x_2(m_{02}(m_{12}m_{32})) \oplus x_3(m_{03}(m_{13}m_{23}))
\end{aligned}$$

We evaluate the $C_{G-SORT}(x)$ in three steps as follows

Step 1: Using $C_{LESS-THAN}$ we compute m_{ij} for $i, j \in N$ and $i < j$, and then set $m_{ii} = 0$ and $m_{ji} = m_{ij} \oplus 1$ for $j > i$ obtaining

$$\begin{array}{llll}
m_{00} = 0 & m_{01} = 1 & m_{02} = 0 & m_{03} = 1 \\
m_{10} = 0 & m_{11} = 0 & m_{12} = 0 & m_{13} = 0 \\
m_{20} = 1 & m_{21} = 1 & m_{22} = 0 & m_{23} = 1 \\
m_{30} = 0 & m_{31} = 1 & m_{32} = 0 & m_{33} = 0
\end{array}$$

Step 2: Compute $\theta_{t,i}$ for $t, i \in [N]$ as (the implicants are marked in bold)

$$\begin{aligned}
\theta_{0,0} &= m_{01}m_{02}m_{03} = 0 \\
\theta_{0,1} &= m_{10}m_{12}m_{13} = 0 \\
\theta_{0,2} &= \mathbf{m_{20}m_{21}m_{23}} = 1 \\
\theta_{0,3} &= m_{30}m_{31}m_{32} = 0
\end{aligned}$$

$$\begin{aligned}
\theta_{1,0} &= m_{10}(m_{02}m_{03}) \oplus \mathbf{m_{20}(m_{01}m_{03})} \oplus m_{30}(m_{01}m_{02}) = 1 \\
\theta_{1,1} &= m_{01}(m_{12}m_{13}) \oplus m_{21}(m_{10}m_{13}) \oplus m_{31}(m_{10}m_{12}) = 0 \\
\theta_{1,2} &= m_{02}(m_{21}m_{23}) \oplus m_{12}(m_{20}m_{23}) \oplus m_{32}(m_{20}m_{21}) = 0 \\
\theta_{1,3} &= m_{03}(m_{31}m_{32}) \oplus m_{13}(m_{30}m_{32}) \oplus m_{23}(m_{30}m_{31}) = 0
\end{aligned}$$

$$
\begin{aligned}
\theta_{2,0} &= m_{10}(m_{20}(m_{03}) \oplus m_{30}(m_{02})) \oplus m_{20}(m_{30}m_{01}) = 0 \\
\theta_{2,1} &= m_{01}(m_{21}(m_{13}) \oplus m_{31}(m_{12})) \oplus m_{21}(m_{31}m_{10}) = 0 \\
\theta_{2,2} &= m_{02}(m_{12}(m_{23}) \oplus m_{32}(m_{21})) \oplus m_{12}(m_{32}m_{20}) = 0 \\
\theta_{2,3} &= \mathbf{m_{03}}(m_{13}(m_{32}) \oplus \mathbf{m_{23}}(\mathbf{m_{31}})) \oplus m_{13}(m_{23}m_{30}) = 1
\end{aligned}
$$

$$
\begin{aligned}
\theta_{3,0} &= m_{10}(m_{20}m_{30}) = 0 \\
\theta_{3,1} &= \mathbf{m_{01}}(\mathbf{m_{21}}\mathbf{m_{31}}) = 1 \\
\theta_{3,2} &= m_{02}(m_{12}m_{32}) = 0 \\
\theta_{3,3} &= m_{03}(m_{13}m_{23}) = 0
\end{aligned}
$$

Note that in each group $\theta_{t,i}$ selects only one source i value for each output position t.

Step 3: Compute the output vector $y_t = \sum_{i=0}^{N-1} x_i\theta_{t,i}$ for $t \in [N]$ as $y = \langle 1, 2, 2, 4\rangle$.

Figure 13: Toy sorting example with $N = 4$ elements.

5 Analysis of Algorithms and Implementation Details

In this chapter, we provide the analysis of the proposed algorithms for homomorphic sorting and the results of their implementations in software.

5.1 Direct Sort Circuit

Previously described C_{D-SORT} algorithm steps can be given as:

- Compute entries of the M matrix in parallel.
- Sum the columns of M using a Hamming Weight circuit and retrieve σ.
- Compare the entries of σ with all possible indices and add the elements conditionally.

The steps of C_{D-SORT} are described in Algorithm 1.

5.1.1 Complexity of C_{D-SORT}

In this section, we give the complexity of evaluating C_{D-SORT} using Algorithm 1 in terms of number of ANDs and the multiplicaive depth of the circuit.

AND Complexity. The number of ANDs used by C_{D-SORT}, can be broken down in terms of ANDs used in the comparisons (to construct M), the evaluation of the σ entries, ANDs used by C_{EQUAL} evaluations and ANDs used in the final summation. The comparison circuit $C_{LESS-THAN}$ for bitwise comparisons and then later compression to a single decision bit

Algorithm 1 Direct Sorting Algorithm

```
1:  function SORT(X, Y, N)
2:      for i ← 0 to N − 1 do                          ▷ Construct M table
3:          M[i][i] ← 0
4:          for j ← i + 1 to N − 1 do
5:              M[i][j] ← LessThan (X[i], X[j])
6:              M[j][i] ← M[j][i] + 1
7:          end for
8:      end for
9:      M ← Transpose (M)
10:     for i ← i + 1 to N − 1 do                      ▷ Construct σ vector
11:         S[i] ← HammingWeight (M[i], N)
12:     end for
13:     for i ← 0 to N − 1 do                          ▷ Construct Y, output vector
14:         Y[i] ← 0
15:         for j ← 0 to N − 1 do
16:             z ← IsEqual (i, S[j])
17:             Y[i] ← Y[i] + AND (z, X[j])
18:         end for
19:     end for
20: end function
```

consumes about

$$\#AND_{LT} \approx 3\ell$$

AND gates. For the comparisons in the lower diagonal half of M (and since computing the upper diagonal does not require any ANDs) to compute M we need

$$\#AND_{M} \approx 3(N^2 - N)/2\ell$$

AND gates. The σ computations involve the addition of N single bit entries of M resulting in $\log(N)$ size entries. This is repeated N times for each entry of σ. Assuming the maximum of $2\log(N)$ AND computations for adding two $\log(N)$ size integers then the total number of ANDs required to compute σ is found as

$$AND_{\sigma} \approx N^2 \log(N) .$$

Computation of the equality comparisons requires $\log(N)$ ANDs per comparison and in total

to compute all comparisons $\theta_{t,i}$ we need

$$AND_{\theta_{t,i}} \approx N^2 \log(N)$$

AND gates. The final sum $y_t = \sum_{i\in[N]} \theta_{t,i} x_i$ for $t \in [N]$ requires only

$$AND_{\sum} \approx N^2$$

AND gates. Therefore the total AND complexity of C_{D-SORT} comes to

$$AND_{C_{D-SORT}} \approx N^2(2 + \log(N)) \, .$$

Multiplicative Depth. In Section 3 we have already determined that $d(C_{LESS-THAN}) = \log(\ell+1)$ and $d(C_{EQUAL}) = \log(\ell)$. In the computations of the entries of σ we are adding N bits together to form a $\log(N)$-bit sum. Since we are using a Hamming Weight circuit defined in Section 3 we arrange adders into a binary tree form, but instead of reducing 2 gates into 1 in each step, we are reducing 3 to 1 by using full adders. Hence the depth complexity of the addition step is

$$d(\sigma) = \log_{3/2}(N).$$

Taking into account the parallel $C_{LESS-THAN}$ and C_{EQUAL} comparisons and single multiplication in the final summation the total depth complexity becomes

$$d(C_{D-SORT}) = (\lceil \log(\ell + 1) \rceil) + \log_{3/2}(N-1) + \log(\ell) + 1$$

5.2 Greedy Sort Circuit

In the previous section, we developed a sorting circuit C_{G-SORT} with low depth. The exact evaluation complexity depends on how the expressions are grouped together and reused. Here we further optimize the circuit

- to reduce the number of primitive operations used in evaluating $\#C_{G-SORT}$. We will present the breakdown of $\#C_{G-SORT}$ into simple operations such as comparisons, mul-

tiplications and additions.

- to reduce the *multiplicative depth* $d(C_{G-SORT})$ of the circuit. The additions have negligible effect to noise growth during homomorphic evaluation when compared to the effect of multiplications. The multiplicative depth will determine the size of the parameters in the SWHE instantiation and the application of noise reduction techniques.

Here we aim to keep the multiplicative depth of the algorithm as low as possible and to minimize the number of ANDs. For the sake of simplicity, we first focus on $i = 0$ in the toy example in Figure 13, where we have coefficients of the form

$$\begin{aligned}
\theta_{00} &= m_{01}m_{02}m_{03} \\
\theta_{10} &= m_{10}m_{02}m_{03} \oplus m_{01}m_{20}m_{03} \oplus m_{01}m_{02}m_{30} \\
\theta_{20} &= m_{10}m_{20}m_{03} \oplus m_{10}m_{02}m_{30} \oplus m_{01}m_{20}m_{30} \\
\theta_{30} &= m_{10}m_{20}m_{30}.
\end{aligned}$$

Manipulating the above equations, we obtain

$$\begin{aligned}
\theta_{00} &= (m_{01}m_{02})\, m_{03} \\
\theta_{10} &= (m_{10}m_{02} \oplus m_{01}m_{20})\, m_{03} \oplus (m_{01}m_{02})\, m_{30} \\
&= (m_{01} \oplus m_{02})\, m_{03} \oplus (m_{01}m_{02})\, m_{30} \\
\theta_{20} &= (m_{10}m_{20})\, m_{03} \oplus (m_{10}m_{02} \oplus m_{01}m_{20})\, m_{30} \\
&= [(m_{01}m_{02}) \oplus (m_{01} \oplus m_{02}) \oplus 1]\, m_{03} \oplus (m_{01} \oplus m_{02})\, m_{30} \\
\theta_{30} &= (m_{10}m_{20})\, m_{30} \\
&= [(m_{01}m_{02}) \oplus (m_{01} \oplus m_{02}) \oplus 1]\, m_{30}\ .
\end{aligned}$$

From now on, the values of the form $m_{j,i}$, i.e. $i < j$, will be labeled as complement. Also, t – complement will be used for an expression which has all the possible t number of complement values covered. For instance, $\theta_{0,i}$ is a 0 – complement expression, while $\theta_{1,i}$ is 1 – complement and $\theta_{2,i}$ is 2 – complement.

In this scheme, we always group our product terms pairwise, i.e. use two input gates. Starting from the comparisons we will gradually build a step-by-step process for the table entries eventually forming the expressions for $\theta_{t,i}$. Since we fixed $i = 0$, at first we start with a table Θ_1 given as

$m_{0,1}$	$m_{0,2}$	$m_{0,3}$	$\dots$	$m_{0,N-1}$

In order to form groups of two, we always take two consecutive column elements. And for the first step, we need three operations over each pair: 1 AND, 1 XOR and 1 AND of their inverses.

$m_{0,1}m_{0,2}$	$\dots$	$m_{0,N-2}m_{0,N-1}$
$m_{0,1}m_{2,0}$ $\oplus$ $m_{1,0}m_{0,2}$	$\dots$	$m_{0,N-2}m_{N-1,0}$ $\oplus$ $m_{N-2,0}m_{0,N-1}$
$m_{1,0}m_{2,0}$	$\dots$	$m_{N-2,0}m_{N-1,0}$

Instead of computing the third row, we can save multiplications by simply computing the XOR of the outputs of the first two operations and take the inverse obtaining Θ_2 as

$m_{0,1}m_{0,2}$	$\dots$	$m_{0,N-2}m_{0,N-1}$
$m_{0,1} \oplus m_{0,2}$	$\dots$	$m_{0,N-2} \oplus m_{0,N-1}$
$(m_{0,1}m_{0,2})$ $\oplus$ $(m_{0,1} \oplus m_{0,2})$ $\oplus$ 1	$\dots$	$(m_{0,N-2}m_{0,N-1})$ $\oplus$ $(m_{0,N-2} \oplus m_{0,N-1})$ $\oplus$ 1

The table above now has $c = \lceil (N-1)/2 \rceil$ columns, and 3 rows, and in each row there are $t-$ complement expressions, where t is the row number. In other words, in $Row = 0$ there are all $0-$ complement expressions, in $Row = 1$ there are all $1-$ complement expressions and finally in $Row = 2$ there are all $2-$ complement expressions. In each step, we will protect this property so that finally when we have the table with $t = (N-1)$ rows and 1 column, it will be our coefficient vector $\theta_{t,i}$ for input X_i.

In the next step, we again construct our new pairs from the elements of consecutive columns. But this time, each element of each row will be paired up with each element on each row of the next column. So that we will have $3^2 = 9$ such pairs for only the first two columns, since we have $c/2$ consecutive columns. The total number of pairs will be $9c/2$ in this step. We perform 1 AND operation on each pair. In order to protect the $Row = t$ has $t-$ complement property, we will always add the new AND outputs to our table, according to a new concept, namely the weight of the pair. It can be defined as the sum of the row indices of pair elements. And this weight value gives us, the number of the row, which the pair's product will be added to. That is to say, we will XOR the AND output of pairs with the same weight value. For instance, if a pair P consists of the element of $Row = 0$ and $Column = 0$ and the element of $Row = 2$ and $Column = 1$ then the pair's weight is $0 + 2 = 2$. This means that output of pair P will be XORed with the output of all other pairs with weight $= 2$.

Our new table Θ_3 will be

$m_{0,1}m_{0,2}m_{0,3}m_{0,4}$	...
$(m_{0,1} \oplus m_{0,2})m_{0,3}m_{0,4}$ $\oplus$ $m_{0,1}m_{0,2}(m_{0,3} \oplus m_{0,4})$	...
$(m_{0,1}m_{0,2})(m_{0,3}m_{0,4} \oplus m_{0,3} \oplus m_{0,4} \oplus 1)$ $\oplus$ $(m_{0,1}m_{0,2} \oplus m_{0,1} \oplus m_{0,2} \oplus 1)(m_{0,3}m_{0,4})$ $\oplus$ $(m_{0,1} \oplus m_{0,2})(m_{0,3} \oplus m_{0,4})$	...
$(m_{0,1} \oplus m_{0,2})(m_{0,3}m_{0,4} \oplus m_{0,3} \oplus m_{0,4} \oplus 1)$ $\oplus$ $(m_{0,1}m_{0,2} \oplus m_{0,1} \oplus m_{0,2} \oplus 1)(m_{0,3} \oplus m_{0,4})$	...
$(m_{0,1}m_{0,2} \oplus m_{0,1} \oplus m_{0,2} \oplus 1)$ $(m_{0,3}m_{0,4} \oplus m_{0,3} \oplus m_{0,4} \oplus 1)$	...

We will repeat the same step with 5 rows and $c/2$ columns, and then repeat the same steps until there remains only one column. So there will be a total of $k = \lceil \log(N-1) \rceil$ iterations,

and the final table will be as

$\theta_{0,0}$
$\theta_{1,0}$
...
$\theta_{N-1,0}$

Since for this example we set $i = 0$, we have the final $\theta_{t,0}$ vector, so we need to perform all of these steps $\forall i \in [N]$. Next we compute the ANDs $\theta_{t,i}X_i$, $\forall t, i \in [N]$.

$\theta_{0,0}X_0$	$\theta_{0,1}X_1$	...	$\theta_{0,N-1}X_{N-1}$
$\theta_{1,0}X_0$	$\theta_{1,1}X_1$	...	$\theta_{1,N-1}X_{N-1}$
...	...	...	...
$\theta_{N-1,0}X_0$	$\theta_{N-1,1}X_1$	...	$\theta_{N-1,N-1}X_{N-1}$

In the final step all we have to do is to compute the sum $Y_t = \sum_{i\in[N]} \theta_{t,i}X_i$, $\forall t \in [N]$. The steps of the method for efficiently evaluating C_{G-SORT} are described in Algorithm 2.

5.2.1 Complexity of C_{G-SORT}

In this section, we determine the complexity of evaluating C_{G-SORT} using Algorithm 2 in terms of number of ANDs and the circuit depth (AND levels).

AND Complexity. The total number of AND operations may be broken down into the sum of the number of ANDs used in the $C_{LESS-THAN}$ comparisons, and in the computation of the $\theta_{t,i}X_i$ products as follows

$$\#AND_{C_{G-SORT}} = \#AND_{LT} * \left[\frac{N(N-1)}{2}\right] + \#AND_{\theta x} \, .$$

The comparison circuit $C_{LESS-THAN}$ for bitwise comparisons and than later compression to a single decision bit consumes about

$$\#AND_{LT} \approx 3\ell$$

Algorithm 2 Greedy Sorting Algorithm

```
1:  function SORT(X, Y, N)
2:      for i ← 0 to N − 1 do                                   ▷ Construct M table
3:          M[i][i] ← 0
4:          for j ← i + 1 to N − 1 do
5:              M[i][j] ← LessThan (X[i], X[j])
6:              M[j][i] ← M[j][i] + 1
7:          end for
8:      end for
9:      iter ← ⌈log(N − 1)⌉
10:     Row ← 1, Col ← (N − 1)
11:     for i ← 0 to N − 1 do                                   ▷ Construct Θ2
12:         for j1 ← [(i + 1) mod N] to [(i − 1) mod N] do
13:             j2 ← (j1 + 1) mod N
14:             Term1 ← AND (M[i][j1], M[i][j2])
15:             Term2 ← (M[i][j1] + M[i][j2])
16:             j ← (j1 − i) mod N
17:             T[i][0][j] ← Term1
18:             T[i][1][j] ← Term2
19:             T[i][2][j] ← Term1 + Term2 + 1
20:             j1 ← (j1 + 1) mod N
21:         end for
22:         if N is even then
23:             j ← (N/2 − 1)
24:             T[i][0][j] ← M[i][j1]
25:             T[i][1][j] ← M[j1][i]
26:             T[i][2][j] ← 0
27:         end if
28:     end for
29:     Row ← 3, Col ← ⌈(N − 1)/2⌉
30:     for k ← 1 to iter − 1 do                                ▷ Perform Θ3 iterations
31:         Row′ ← 2^(k+1) + 1
32:         for i ← 0 to N − 1 do
33:             for j ← 0 to Col − 1 do
34:                 for r1 ← 0 to Row − 1 do
35:                     for r2 ← 0 to Row − 1 do
36:                         T[i][r1 + r2][j/2] ← T[i][r1 + r2][j/2]
37:                         +AND(C[i][r1][j], C[i][r2][j]);
38:                     end for
39:                 end for
40:                 j ← j + 1
41:             end for
```

```
42:             if Col is odd then
43:                 for r ← 0 to Row − 1 do
44:                     T[i][r][j/2] ← T[i][r][j]
45:                 end for
46:                 for r ← Row to Row′ − 1 do
47:                     T[i][r][j/2] ← 0
48:                 end for
49:             end if
50:         end for
51:         Row ← Row′
52:         Col ← Col/2
53:     end for
54:     for t ← 0 to N − 1 do                          ▷ Compute θ_{i,t}X_i products
55:         for i ← 0 to N − 1 do
56:             TX[t][i] ← AND (T[i][t][0], X[i])
57:         end for
58:     end for
59:     for t ← 0 to N − 1 do                          ▷ Sum θ_{i,t}X_i products
60:         for i ← 0 to N − 1 do
61:             Y[t] ← Y[t] + TX[t][i]
62:         end for
63:     end for
64: end function
```

AND gates. To compute the $\theta_{t,i}X_i$ table we need N^2 ANDs. Since we are applying bitwise AND operations with ℓ bit vector operands we have as a total of $N^2\ell$ ANDs. In each of the iterations of Θ_3, we are halving the width of the table starting from an initial width of N until we collapse it to a single column. Recall that $k = \lceil log(N-1) \rceil$. If we sum over the iterations and also include the prior $N^2\ell$ ANDs we obtain the total number of ANDs required for the computation of $\theta_{t,i}X_i$ products from the comparison matrix as follows:

$$\begin{aligned} \#AND_{\theta x} &= N\left(1\frac{(N-1)}{2} + \cdots + (1+2^{k-1})^2\frac{(N-1)}{2^k}\right) \\ &\quad +N^2\ell \\ &= [N(N-1)]\left[\sum_{n=1}^{k} 2^{-n}(1+2^{n-1})^2\right] + N^2\ell \\ &= \left[\frac{N(N-1)}{2}\right]\left[k(k+1) + (2^k-1) + (2-2^{1-k})\right] \\ &\quad +N^2\ell \end{aligned}$$

Therefore, $\#AND_{\theta x} = O(N^3) + O(N^2\ell)$ and the overall AND complexity of C_{G-SORT} is found as

$$\#AND_{C_{G-SORT}} = O\left(N^2\ell\right) + O\left(N^3\right)$$

Multiplicative Depth. The overall depth of C_{G-SORT} is determined by

$$d(C_{G-SORT}) = d(C_{LESS-THAN}) + d(\theta_{t,i}X_i).$$

In Section 3 we have already determined that $d(C_{LESS-THAN}) = \log(\ell+1)$. During the $\theta_{t,i}X_i$ summations we employed a circuit arranged in a binary tree of depth $d(\sum \theta_{t,i}X_i) = k+1$. Substituting $k = \lceil \log(N-1) \rceil$ the overall circuit depth is found as

$$d(C_{G-SORT}) = 1 + \lceil \log(\ell+1) \rceil + \lceil \log(N-1) \rceil .$$

5.3 Timing Results

We implemented the proposed depth optimized sorting algorithm in Algorithm 2 using the SWHE scheme of [16] and evaluated C_{G-SORT} for a number of array lengths. Here we briefly

Array Size N		4	8	16	32	64
8-bit	d	8	9	10	11	12
	$\log q$	136	153	170	187	204
	δ	1.0027	1.0031	1.0035	1.0038	1.0042
32-bit	d	10	11	12	13	14
	$\log q$	170	187	204	221	238
	δ	1.0035	1.0038	1.0042	1.0046	1.0050

Table 1: Circuit depth d, max. coefficient size $\log(q)$, and Hermite factor δ for selected ℓ and N

summarize the parameter selection process and present the simulation results.

5.3.1 Parameter Selection

According to [16] the NTRU based SWHE Scheme requires Hermite factor $\delta < 1.0066$ to achieve a security level of about 80-bit. We set the per level cutting rate $\log p = 17$ and polynomial degree $n = 8191$ which allows the message slot size of 630-bit for batching. We simulate for both $\ell = 8$-bit and $\ell = 32$-bit integer inputs and select array size N as powers of two[2]. In Table 1 we give results for circuit depths, maximum bit sizes and Hermite factors for different cases. The largest Hermite factor we have in among parameter choices is $\delta = 1.0050$ gives us 140-bit security which is the lowest security level for all cases.

5.3.2 Implementation Details

We implemented the proposed homomorphic sorting algorithm in C++ where we relied on DHS-FHE Library [16]. All simulations were performed on a 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-bit and 32-bit integers are given in Table 2. For 32-bit wide array elements with $N = 64$ our algorithm runs in about 50 hours. The amortized running time for homomorphically sorting $N = 64$ 32-bit elements 287 seconds. For $N = 4$ the sorting time takes as low as 0.57 seconds per sort. The 32-bit implementation has 86 relinearizations for each $C_{LESS-THAN}$ operation whereas 8-bit implementation has only

[2]Note that N is *not* restricted to a power of two. Also we include the $N = 40$ case in our experiments to enable comparison with [20].

19 relinearizations.

We note that the ratio of running times which is about $\sim 4.5\times$ for the 32-bit and 8-bit cases, is proportional to the ratio of the number of relinearization operations. Furthermore, we observed that %80-85 percent of time is spent on the $C_{LESS-THAN}$ operations (Step 1 of Algorithm 2) in all cases.

Array Size	8 Bit		32 Bit	
	Total	Normalized	Total	Normalized
4	66	0.10	362	0.57
8	338	0.53	1839	2.91
16	1856	2.94	8287	13.15
32	8728	13.85	41034	65.13
40	15121	24.00	69514	110.33
64	39919	63.36	180980	287.26

Table 2: Timings for Homomorphic Sorting for different Array Sizes (in seconds)

In comparison, the homomorphic Lazy Sort implementation of [20] takes about 976 and 1400 seconds for array sizes 10 and 40 respectively. Our implementation takes 13.15 and 110.33 seconds for array sizes 16 and 40. In 40 element case we are 12.7 times faster than their implementation.

We also have the experimental results for Odd Even Transposition Sort implementation and it shows us that, the depth of the sorting circuit is directly related with the total time of the computation as we claimed. Odd Even Transposition Sort takes 519, 19643 and 657682 seconds for 4, 8 and 16 number of 8-bit elements respectively.

6 Conclusion

In this thesis, we proposed 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. We proposed and motivated circuit depth as a *new* metric to be studied in the context of sorting algorithm. Existing sorting algorithms are not optimized for homomorphic evaluation and with the new age of parallel computing, there may be a going back to more simple approaches in sorting problem, as a result we presented the depth analysis for several classical sorting algorithms: Bubble sort, Insertion Sort and Merge Sort and then Sorting Networks. An overall comparison is given in Table 3. Inspired by the performance of Merge Sort, we introduced a new depth-optimized sorting algorithm which achieves an a circuit depth of $O(\log(N)+\log(\ell))$ and again inspired by this new sorting circuit we developed another ranking based algorithm which achieves $O(\log_{3/2}(N)+\log(\ell))$ but with different constants. An overall comparison with respect to the size of input arrays is given in Table 4.

To study the real-life performance of our sorting algorithm, we instantiated an NTRU based SWHE scheme in the DHS FHE library and presented simulation results for selected array lengths. The implementation performs favorably achieving one to two orders of magnitude speed up over the proposal by [20] for the same array lengths. In addition we implemented the Odd Even Transition Sort algorithm, inspired by Bubble Sort and its execution times were prohibitively high so that we were not able to run the implementation for more than 16 elements. Roughly, for 8 bit numbers, it took 520 seconds for 4 elements and 19643 seconds for 8 elements. And yet, the following issues need to be further explored:

Algorithm	Depth	#Comparisons
Bubble Sort	$O(N^2 \log(l))$	$O(N^2)$
Overlapped Bubble	$O(N \log(l))$	$O(N^2)$
Odd Even Sort	$O(N \log(l))$	$O(N^2)$
Insertion Sort	$O(N^2 \log(l))$	$O(N^2)$
Merge Sort	$O(\log^2(N) \log(l))$	$O(N \log^2(N))$
Odd-Even Merge Sort	$O(\log^2(N) \log(l))$	$O(N \log^2(N))$
Bitonic Sort	$O(\log^2(N) \log(l))$	$O(N \log^2(N))$
Direct Sort	$O(\log_{3/2}(N) + \log(l))$	$O(N^2)$
Greedy Sort	$O(\log(N) + \log(l))$	$O(N^2)$

Table 3: Comparison of different sorting algorithms in terms of multiplicative depth and number of comparisons

- transforming different problem solutions into Algebraic Normal Form equations/circuits in a multiplicative depth-optimized way, and
- trade-offs between circuit depth and other traditional metrics such as the number of additions and multiplications. For this, we plan to propose different sorting network implementations over our FHE scheme.
- a new bit-based approach can be adopted in sorting problem, since FHE ciphertext consist of bitwise encrypted data. So, instead of considering the integer comparison as a fundamental operation, whole sorting problem can be achieved over bitwise comparisons. This idea, not pursued within the scope of this thesis, can be seen as direction for future work.

Algorithm	Size=8	Size=16
Bubble Sort	196	840
Overlapped Bubble	91	203
Odd Even Sort	56	112
Insertion Sort	196	840
Merge Sort	42	70
Odd-Even Merge Sort	42	70
Direct Sort	15	17
Greedy Sort	10	11

Table 4: Comparison of different sorting algorithms in terms of multiplicative depth for different array sizes of 32-bit elements

References

[1] Rivest, R.L., Adleman, L., Dertouzos, M.L.: "On data banks and privacy homomorphisms." In: *Foundations of Secure Computation*, 1978.

[2] Gentry, C.: "Fully homomorphic encryption using ideal lattices." *Symposium on the Theory of Computing (STOC)*, 2009, pp. 169-178.

[3] Gentry, C.: *A Fully Homomorphic Encryption Scheme*. Ph.D. thesis, Department of Computer Science, Stanford University, 2009.

[4] Van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: "Fully homomorphic encryption over the integers." *Advances in Cryptology–EUROCRYPT 2010* (2010): 24-4

[5] Gentry, C., Halevi, S.: "Implementing Gentry's fully-homomorphic encryption scheme," *Advances in Cryptology–EUROCRYPT 2011*, pp. 129–148, 2011.

[6] Gentry, C., Halevi, S., Smart, N.P.: "Fully homomorphic encryption with polylog overhead." Manuscript, 2011.

[7] Smart, N.P., Vercauteren, F.: "Fully homomorphic SIMD operations." Manuscript at http://eprint.iacr.org/2011/133, 2011.

[8] Reginald L. Lagendijk, Zekeriya Erkin, and Mauro Barni: "Encrypted Signal Processing for Privacy Protection: Conveying the Utility of Homomorphic Encryption and Multiparty Computation." *IEEE Signal Process. Mag.* 30(1):82-105 (2013)

[9] Cheon, Jung Hee, Miran Kim, and Kristin Lauter. "Secure DNA-Sequence Analysis on Encrypted DNA Nucleotides." Manuscript at http://media.eurekalert.org/aaasnewsroom/MCM/-FIL_000000001439/EncryptedSW.pdf, 2014.

[10] Gentry, C., Halevi, S., Smart, N.P.: "Homomorphic evaluation of the AES circuit." *Advances in Cryptology - CRYPTO 2012*, 850-8, 2012.

[11] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: "Fully homomorphic encryption without bootstrapping." *Innovations in Theoretical Computer Science, ITCS* 309–325, 2012.

[12] Alt-López, A., Tromer E., Vaikuntanathan, V.: "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption." In: *Proc. of the 44th STOC*, pp. 1219-1234. ACM, 2012.

[13] Stehlé, D., Steinfeld, R.: "Making NTRU as secure as worst-case problems over ideal lattices." *Advances in Cryptology – EUROCRYPT '11* 27–4, 2011.

[14] Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: "Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme". In LNCS *PQCrypto 2013*. pp. 45–64. Springer, 2013.

[15] Brakerski, Z.: "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In *Advances in Cryptology – CRYPTO 2012*, Springer LNCS Volume 7417, 2012, pp 868-886.

[16] Doröz, Y., Hu, Y., Sunar, B.: "Homomorphic AES Evaluation using NTRU", IACR ePrint Archive. Technical Report 2014/039 January 2014. URL: http://eprint.iacr.org/2014/039.pdf

[17] Brakerski, Z., Vaikuntanathan, V.: "Efficient fully homomorphic encryption from (standard) LWE." *Foundations of Computer Science (FOCS), 2011* IEEE 52nd Annual Symposium on. IEEE, 2011.

[18] Lauter, K., Naehrig, M., Vaikuntanathan, V.: "Can homomorphic encryption be practical?" In: *Proceedings of the 3rd ACM CCSW (Cloud Computing Security Workshop)* ACM, 2011.

[19] Doröz, Y., Sunar B., Hammouri, G. "Bandwidth Efficient PIR from NTRU. Workshop on Applied Homomorphic Cryptography and Encrypted Computing", WHAC'14, 2014.

[20] Chatterjee, A., Kaushal, M., Sengupta, I.: "Accelerating Sorting of Fully Homomorphic Encrypted Data", G. Paul and S. Vaudenay (Eds.): INDOCRYPT 2013, LNCS 8250, pp. 262–273, 2013.

[21] Brenner M., Perl H., Smith M.: `libScarab Software Library` https://hcrypt.com/

[22] Yao, A. C.: "Protocols for secure computations", In: *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, Washington, DC, USA: IEEE Computer Society, pp. 160-164, 1982.

[23] Vaidya, J., Clifton, C.: "Privacy-preserving k-means clustering over vertically partitioned data", In: *KDD '03: Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, ACM, pp. 206-215, 2003.

[24] Fischlin, M.: "A cost-effective pay-per-multiplication comparison method for millionaires", D. Naccache (Ed.) In *CT-RSA 2001: Topics in Cryptology -The Cryptographers' Track at RSA Conference*, LNCS 2020, Berlin, Germany, Springer, pp. 457-471, 2001.

[25] Sander, T., Young, A., Yung, M.: "Non-interactive cryptocomputing for NC1", In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, IEEE Computer Society, pp. 554-566, 1999.

[26] Goldwasser, S., Micali, S: "Probabilistic encryption and how to play mental poker keeping secret all partial information", In *Proc. 14th Symposium on Theory of Computing*, pp. 365377, 1982.

[27] Yildizli, C., Pedersen, T. B., Saygin, Y., Savas, E., Levi, A.: "Distributed Privacy Preserving Clustering via Homomorphic Secret Sharing and Its Application to (Vertically) Partitioned Spatio-Temporal Data", IJDWM 7(1), pp. 46-66, 2011.