# Low Depth Circuits for Efficient Homomorphic Sorting

Gizem S. Çetin[1], Yarkın Doröz[1], Berk Sunar[1], and Erkay Savaş[2]

[1] Worcester Polytechnic Institute
{gscetin,ydoroz,sunar}@wpi.edu
[2] Sabanci University
erkays@sabanciuniv.edu

**Abstract.** We introduce a sorting scheme which is capable of efficiently sorting encrypted data without the secret key. The technique is obtained by focusing on the multiplicative depth of the sorting circuit alongside the more traditional metrics such as number of comparisons and number of iterations. The reduced depth allows much reduced noise growth and thereby makes it possible to select smaller parameter sizes in somewhat homomorphic encryption instantiations resulting in greater efficiency savings. We first consider a number of well known comparison based sorting algorithms as well as some sorting networks, and analyze their circuit implementations with respect to multiplicative depth. In what follows, we introduce a new ranking based sorting scheme and rigorously analyze the multiplicative depth complexity as $\mathcal{O}(\log(N) + \log(\ell))$, where $N$ is the size of the array to be sorted and $\ell$ is the bit size of the array elements. Finally, we simulate our sorting scheme using a leveled/batched instantiation of a SWHE library. Our sorting scheme performs favorably over the analyzed classical sorting algorithms.

**Keywords:** Homomorphic sorting, circuit depth, somewhat homomorphic encryption.

## 1 Introduction

An encryption scheme is *fully homomorphic* (FHE scheme) if it permits the efficient evaluation of any boolean circuit or arithmetic function on ciphertexts [27]. Gentry introduced the first FHE scheme [14, 15] based on lattices that supports the efficient evaluation for arbitrary depth circuits. This was followed by a rapid progression of new FHE schemes. Van Dijk et al. proposed a FHE scheme based on ideals defined over integers [10]. In 2010, Gentry and Halevi [16] 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 [17,18,29]. 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 [5]. 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 [18] 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 for efficient AES evaluation using key and modulus switching techniques [5], batching and SIMD optimizations [29]. Their byte-sliced AES implementation takes about 5 minutes to homomorphically evaluate an AES block encryption. More recently, López-Alt, Tromer and Vaikuntanathan (ATV) proposed SWHE and FHE schemes based on Stehlé and Steinfeld's generalization of the NTRU scheme [30] that supports inputs from multiple public keys [25]. Bos et al. [3] introduced a variant of the LTV FHE scheme along with an implementation. The authors modify the LTV scheme by adopting a tensor product technique introduced earlier by Brakerski [4] 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 LTV scheme in [11]. The implementation is batched, bit-sliced and features modulus switching techniques. The

authors also specialize the modulus to reduce the key size and report an AES implementation with one minute evaluation time per AES block [18]. 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 [15], relinearization [6], and modulus reduction [5,6] are indispensable tools for FHEs. In [6, Sec. 1.1], the *relinearization* technique was proposed 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. [22] 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 heavy DSP 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 [32] 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 [31] propose to use Yao's circuit evaluation [32] for the comparisons in their privacy-preserving $k$-means clustering algorithm. The secure comparison protocol by Fischlin [13] uses the GM-homomorphic encryption scheme [19] and the method by Sander et al. [28] 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 [33] uses the Fischlin formulation based on XOR homomorphic secret sharing primitive instead of costly encryption operations. The tools for SWHE developed to achieve FHE have only been around for a few years now and have not been sufficiently explored for use in applications. For instance, in [23] 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 the 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. Later in [24], Lauter et al. show that it is possible to implement genomic data computation algorithms where the patients' data are encrypted to preserve their privacy. They encrypt all the genomic data in the database and able to implement and provide performance numbers for Pearson Goodness-of-Fit test, the $D'$ and $r^2$-measures of linkage disequilibrium, the Estimation Maximization (EM) algorithm for haplotyping, and the Cochran-Armitage Test for Trend. The authors used a leveled SWHE scheme which is a modified version of [26] where they get rid of the costly relinearization operation. In [2] Bos et al. show how to privately perform predictive analysis tasks on encrypted medical data. They present an implementation of a prediction service running in the cloud. The cloud server takes private encrypted health data as input and returns the probability of cardiovascular disease in encrypted form. The authors use the SWHE implementation of [3] to provide timing results. Graepel et al. in [20] demonstrate that it is possible to execute machine learning algorithms in a service while protecting the confidentiality of the training and test data. The authors propose a confidential protocol for machine learning tasks and design confidential machine learning algorithms using leveled homomorphic encryption. More specifically they implement low-degree polynomial versions of Linear Means Classifier and Fisher's Linear Discriminant Classifier on the Wisconsin Breast Cancer Data set. Finally, they provide benchmarks for small scale data set to show that their scheme is practical. 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 [12], 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 a 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 [8]. In this work, for the first time, the authors considered the problem of homomorphically sorting an array using the recently proposed `hcrypt` FHE library [7]. The authors define a number of FHE functions 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 1,399 second evaluation time in contrast to 21,565 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** [5] that yield exponential gains in leveled implementations.
- `hcrypt` does not take advantage of **batching** or SIMD techniques [29] which greatly improve homomorphic evaluation performance.

**Our Contribution.** In this work,

- we survey a number of classical sorting algorithms, i.e. Bubble, Insertion, Odd-Even Sort, Merge, Batcher's Odd-even Merge Sort, Bitonic sort, and show that some are more suitable than others for leveled SWHE evaluation, similar to the work for distance computation presented in [9]. Specifically, we characterize them with respect to a new metric, i.e. multiplicative circuit depth. We show that the classical sorting algorithms require deep circuit evaluations and therefore are not ideal for homomorphic evaluation.
- we introduce two new depth optimized sorting schemes: Greedy Sort and Direct Sort. Both algorithms permit shallow circuit evaluation of depth only $\mathcal{O}(\log(N) + \log(\ell))$ for sorting $N$ elements, where $\ell$ represents the size of the array elements in bits. The Greedy algorithm has slightly lower depth however requires more multiplications than Direct Sort. Both algorithms improve in the circuit depth metric over classical algorithms by at least *1-3 orders of magnitude.*
- we instantiate a somewhat homomorphic encryption scheme (SWHE) based on NTRU, and present an implementation of the proposed sorting algorithm using this SWHE scheme. Our results, confirm our theoretical analysis, i.e. that the performance of the proposed sorting algorithm scales favorably as $N$ increases.

## 2 Background

We start by giving a brief summary of the multi-key LTV-FHE scheme and provide a brief explanation on the primitive functions that are proposed by López-Alt, Tromer and Vaikuntanathan. Later, we give details of the DHS FHE library, that is used in the implementation, based on a specialized LTV-FHE version.

### 2.1 The LTV-SWHE Scheme

In 2012 López-Alt, Tromer and Vaikuntanathan proposed a leveled multi-key FHE scheme (ATV) [25]. The scheme based on a variant of NTRU encryption scheme proposed by Stehlé and Steinfeld [30]. The introduced

scheme uses a new operation called relinearization and existing techniques such as modulus switching for noise control. We use the same construction as in [11] which is a single key version of ATV with reduced key size technique. The operations are performed in $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ where $n$ is the polynomial degree and $q$ is the prime modulus. The scheme also defines an error distribution $\chi$, 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 $\chi$ distribution. The scheme consists of four primitive functions, namely **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. A public and secret key pair is computed for each level: $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ and $f^{(i)} = 2u^{(i)} + 1$, where $\{g^{(i)}, u^{(i)}\} \in \chi$. Later we create evaluation keys for each level: $\zeta_\tau^{(i)}(x) = h^{(i)}s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau(f^{(i-1)})^2$, where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i \rfloor]$.

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

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

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

### 2.2 The DHS SWHE Library

We use a customized version of the LTV-SWHE scheme that is previously proposed in [11] by Doröz, Hu and Sunar (DHS). The code is written in C++ using NTL package that is compiled with GMP library. The library contains some special customizations that improve the efficiency in running time and memory requirements. The customizations of the DHS implementation are as follows:

- We select a special $m^{th}$ cyclotomic polynomial $\Psi_m(x)$ as our polynomial modulus. The degree of the polynomial $n$ is equal to the euler totient function of $m$, i.e. $\varphi(m)$. In each level the arithmetic is performed over $\mathrm{R}_{q_i} = \mathbb{Z}_{q_i}[x]/\langle \Psi_m(x)\rangle$, where modulus $q_i$ is equal to $p^{k-i}$. The value $p$ is a prime number that cuts $(\log_p)$-bits of noise and the value $k$ is equal to the depth plus 1.
- Due to the special structure of the moduli $p^{k-i}$, the evaluation keys in one level can also be promoted to the next level via modular reduction. For any level we can evaluate the evaluation key as $\zeta_\tau^{(i)}(x) = \zeta_\tau^{(0)}(x) \pmod{q_i}$. This technique reduces the memory requirement significantly and makes it possible to evaluate higher depth circuits.
- The specially selected cyclotomic polynomial $\Psi_m(x)$ is used to batch multiple message bits into the same polynomial for parallel evaluations as proposed by Smart and Vercauteren [17, 29] (see also [18]). The polynomial $\Psi_m(x)$ is factorized over $\mathbb{F}_2$ into equal degree polynomials $F_i(x)$ which define the message slots in which message bits are embedded using the Chinese Remainder Theorem. We can batch $\ell = n/t$ number of messages, where $t$ is the smallest integer that satisfies $m|(2^t - 1)$.
- The DHS library can perform 5 main operations; KeyGen, Encryption, Decryption, Modulus Switch and Relinearization. The most time consuming operation is Relinearization, which is generally the bottleneck.
  Therefore, the most critical operation for circuit evaluation is Relinearization. The other operations have negligible effect on the run time.

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

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

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

## 3 Basic Circuits

As stated earlier, given level $i$ in a homomorphic circuit, we will have $c_1^{(i)} = \mathsf{Encrypt}(b_1)$ and $c_2^{(i)} = \mathsf{Encrypt}(b_2)$ where $b_1$ and $b_2$ are encrypted by the owner of data. We are allowed to use two fundamental operations on encrypted inputs; bit multiplication ($AND$, "$\cdot$") and bit addition ($XOR$, "$\oplus$"). Hence, we can evaluate $c^{(i)} = c_1^{(i)} \oplus c_2^{(i)}$ and $\tilde{c}^{(i)} = c_1^{(i)} \cdot c_2^{(i)}$. Finally, the holder of the secret key can compute and retrieve $\mathsf{Decrypt}(c^{(i)}) = b_1 \oplus b_2$. $\mathsf{Decrypt}(\tilde{c}^{(i)}) = b_1 \cdot b_2$. Using these two, we can define the following circuits.

**Equality Circuit $\mathcal{C}_{\text{EQ}}$:** It compares two encrypted $\ell$-bit integers $E(X) = X^{(i)} = (x_{\ell-1})^{(i)} \dots (x_1)^{(i)} (x_0)^{(i)}$ and $E(Y) = Y^{(i)} = (y_{\ell-1})^{(i)}, \dots, (y_1)^{(i)}, (y_0)^{(i)}$, and outputs $\tilde{z}^{(j)}$. Here $(x_k)^{(i)}$ and $(y_k)^{(i)}$ represent the $k$-th bits of $X^{(i)}$ and $Y^{(i)}$, respectively. Also $D(\tilde{z}^{(j)})$ returns 1 if $X$ equals $Y$ and 0 otherwise. We can formalize the comparison circuit as follows; $\tilde{z}^{(j)} = (E(X) = E(Y)) = \prod_{k \in [\ell]} (E(x_k) = E(y_k)) = \prod_{k \in [\ell]} ((x_k)^{(i)} \oplus (y_k)^{(i)} \oplus 1)$. The product chain of $\ell$ bits may be evaluated using a binary tree of AND gates which creates a circuit with $\lceil \log(\ell) \rceil$ multiplicative depth. Therefore, $d(\mathcal{C}_{\text{EQ}}) \approx \mathcal{O}(\log(\ell))$.

**Less Than Circuit $\mathcal{C}_{\text{LT}}$:** In a similar manner, this circuit compares two $\ell$-bit integers $E(X)$ and $E(Y)$, and outputs $\tilde{z}^{(j)}$ where $D(\tilde{z}^{(j)}) = 1$ if $X$ is smaller than $Y$ and $D(\tilde{z}^{(j)}) = 0$ otherwise. We can formalize the comparison circuit as follows; $\tilde{z}^{(j)} = (E(X) < E(Y)) = \sum_{k \in [\ell]} \left[ (E(x_k) < E(y_k)) \prod_{k<t<\ell} (E(x_t) = E(y_t)) \right]$ where $(E(x_k) < E(y_k)) = (y_k)^{(i)} \cdot ((y_k)^{(i)} \oplus 1)$ and $(E(x_j) = E(y_j)) = (y_t)^{(i)} \oplus (x_t)^{(i)} \oplus 1$. The expansion of the formula gives a sum of products expression where the product with the maximum number of bits occurs when $i = 0$, in which case the product chain contains $\ell + 1$ bits, where 2 bits are contributed by the $(E(x_0) < E(y_0))$ term and the rest are from the $(E(x_t) = E(y_t))$ terms. For the product of $\ell + 1$ elements, we may use again a binary tree in which case we achieve the minimum depth of $\lceil \log(\ell + 1) \rceil$. Therefore $d(\mathcal{C}_{\text{LT}}) \approx \mathcal{O}(\log(\ell + 1))$.

**Compare and Swap Block $\mathcal{C}_{\text{CS}}$:** Since our main goal is the construction of a sorting circuit, we will extensively use the comparators followed by a swap operation. The $\mathcal{C}_{\text{CS}}$ block basically compares two $\ell$-bit integers $X$ and $Y$ using previously defined $\mathcal{C}_{\text{LT}}$ circuit and swaps them if $X \not< Y$. Overall circuit can be defined as; $\tilde{X}^{(j+1)} = [\tilde{z}^{(j)} \cdot E(X)] \oplus [(\tilde{z}^{(j)} \oplus 1) \cdot E(Y)]$ and $\tilde{Y}^{(j+1)} = [\tilde{z}^{(j)} \oplus 1) \cdot E(X)] \oplus [\tilde{z}^{(j)} \cdot E(Y)]$, where $\tilde{z}^{(j)} = (E(X) < E(Y))$. Therefore, $d(\mathcal{C}_{\text{CS}}) = d(\mathcal{C}_{\text{LT}}) + 1$.

## 4 Sorting Algorithms

Sorting is one of the most natural and crucial tasks in computing. Numerous sorting algorithms have been proposed in the literature [21]. These algorithms have been heavily investigated and characterized according

to their time and space requirements, as well as to the degree of their suitability for parallelization. As far as homomorphic evaluation is concerned we have another requirement. Since most of the FHE and SWHE schemes are designed to evaluate circuits, and do not scale well when the multiplicative depth of the circuit is high, we need to add another metric; namely multiplicative circuit depth, before we can build a homomorphic sorting scheme. For this we need to first convert the serial sorting algorithm, into a circuit by unrolling loops and eliminating conditional assignments by *arithmetization*. In this paper, the term "circuit depth" is used in lieu of multiplicative depth of the circuit and it should not be confused with "comparison depth", i.e. depth of the circuit measured in terms of comparative levels, which is used in the analysis of classical sorting algorithms in the literature.

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

### 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. [8] 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 swapped to move the smaller element to the left (in case of a horizontal array). The average and worst case performance for an array of $N$ elements are the same: $\mathcal{O}(N^2)$. During homomorphic evaluation since we have no way of knowing when the array is sorted for a possible early termination, we need to make $N-1$ passes over the array always achieving the worst case complexity. Since another element in the rightmost portion is sorted the passes decrease by one in number of elements compared and swapped after each pass. Thus, overall we will have $[(N-1)+(N-2)+\ldots+1] = (N^2-N)/2$ $\mathcal{C}_{\text{CS}}$ blocks and the depth of the Bubble Sort circuit will be $[(N^2-N)/2]d(\mathcal{C}_{\text{CS}})$. Considering $\ell$-bit wide array elements, we have $d(\mathcal{C}_{\text{BUBS}}) = \mathcal{O}(N^2\log(\ell))$. We can gain some economy by not waiting to start the next pass until a pass is finished. We can *overlap* the passes which creates a network version of Bubble Sort, known as Odd Even Sort, detailed in the next section. [3]

### 4.2 Odd-Even Sort

A trellis shaped circuit arrangement of Bubble sort is known as Odd-Even Sort. The circuit admits $N$ inputs and computes the $N$ sorted output values after $N$ passes. In the first pass, considering a zero-indexed array, every even indexed element is compared and swapped with its right neighbor. In the second pass, every odd indexed element is compared and swapped with its right neighbor. Considering these two steps as a round, the identical operations are applied in each round. The total number of comparisons is $N-1$ in each round, there are $N$ passes which means $N/2$ rounds and so overall, there are $N(N-1)/2$ comparators. And the depth of the circuit is $Nd(\mathcal{C}_{\text{CS}})$. Therefore $d(\mathcal{C}_{\text{OES}}) = \mathcal{O}(N\log(\ell))$.

[3] Note that in their implementation Chatarjee et al. [8] perform the comparison using a carry propagate adder based subtraction circuit result in a circuit depth $(N^2-N)(\ell+1)/2$. While the computational complexity of the scheme is low, the $\mathcal{O}(N^2)$ circuit depth is prohibitive.

### 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 an element smaller 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 and the worst case complexity of the algorithm is $\mathcal{O}(N^2)$ while the best case is only $\mathcal{O}(N)$. When considered as a circuit for homomorphic evaluation we need to run the algorithm with the worst case complexity, 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 $[1+2+\ldots+N-1]d(\mathcal{C}_{\mathrm{CS}}) = (N^2-N)/(2)d(\mathcal{C}_{\mathrm{CS}})$. Therefore $d(\mathcal{C}_{\mathrm{INS}}) = \mathcal{O}(N^2\log(\ell))$. This circuit can be used in a more efficient way by overlapping some comparisons, similar to $\mathcal{C}_{\mathrm{BUBS}}$. Consequently we can see that, Insertion Sort and Bubble Sort reveal the same construction, when they are considered as sorting networks.

In [8] Chatarjee et al. rely on the fact that after the *imperfect* application of Bubble Sort the array is *nearly* sorted. Thus Insertion Sort performs nearly in linear time. But even if the array is *nearly* sorted, the algorithm should run as in the worst case, since we do not have any knowledge of the misplaced elements.

### 4.4 Merge Sort

Merge Sort is an asymptotically faster algorithm and allows early termination in normal execution, which reduces its complexity. The algorithm is recursively applied by splitting arrays into smaller ones. In the innermost recursion, arrays of two elements are sorted, where only one comparison is needed in one sub-array. In the merging step, which combines two individually sorted arrays into a single sorted array, at most three comparisons are applied in each partition. This eventually requires $\mathcal{O}(N\log(N))$ comparisons in the worst case. But in our case, the merging step requires many more comparisons, due to algorithm's input dependent nature and our lack of input knowledge. For instance, in the classic Merge Sort, to merge two sub-arrays each of size two, as in Figure 1, we follow one of the paths until all the elements are placed in the sorted sub-array of size four. Let our output array be $Z$ in a merge step. Then, if $\mathcal{C}_{\mathrm{LT}}(E(X_0), E(Y_0))$ output is 1; we can conclude that $Z_0 = X_0$, otherwise $Z_0 = Y_0$. But in homomorphic sorting, we cannot follow any specific path as the output of each $\mathcal{C}_{\mathrm{LT}}(E(X_i), E(Y_j))$ block is also encrypted. Hence, we need to consider every single possible outcome of all comparison operations, i.e. every single path, which eventually necessitates comparing every possible pair.

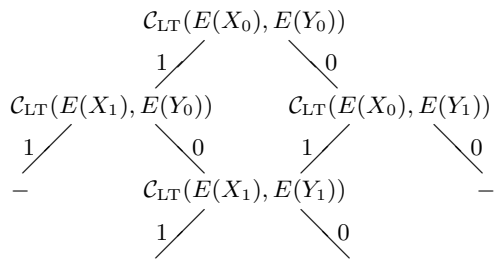


Fig. 1: Merging two individually sorted arrays: $\langle E(X_0), E(X_1)\rangle$ and $\langle E(Y_0), E(Y_1)\rangle$

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

two new sorting circuits, with the same number of comparators $\mathcal{O}(N^2)$ and the total comparison depth of $\mathcal{O}(1)$, in Section 5.1.

Indeed, we can reduce the number of comparisons in $\mathcal{C}_{\text{MS}}$ using $\mathcal{C}_{\text{CS}}$ blocks. In the first step, the run of $\mathcal{C}_{\text{CS}}(X_0^{(i)}, Y_0^{(i)})$ will yield $X_0^{(j)}$ and $Y_0^{(j)}$ as the smaller and larger elements, respectively. Then, we can safely use $Y_0^{(j)}$ in the following step as one of the inputs to $\mathcal{C}_{\text{CS}}$ blocks while the second input can be either $X_1$ or $Y_1$. But, it is impossible to know which one of them should be used since we do not know the previous comparison result. Therefore, we should apply an additional $\mathcal{C}_{\text{CS}}(X_1^{(i)}, Y_1^{(i)})$. This yields $X_1^{(j)}$ as the smaller element. So, as a final step, evaluating $\mathcal{C}_{\text{CS}}(X_1^{(j)}, Y_0^{(j)})$ will be sufficient to complete the merging of the two arrays. This algorithm yields to Odd-Even Merge Sort whose details are given in the next section.

### 4.5 Odd-Even Merge Sort

Odd-Even Merge Sort is a sorting network devised by Batcher [1]. It has a recursive structure similar to Merge Sort. The algorithm considers two already sorted half-lists at each merge step. In a merge step, the merging process is recursively applied to even and odd indexed elements separately while arranging them into two halves. This process continues until there is only one element in each half list in which case a $\mathcal{C}_{\text{CS}}$ is applied in order to merge them into an array. Once the even indexed half and the odd indexed half are both internally merged, $\mathcal{C}_{\text{CS}}$ are applied to inner adjacent elements only. The merging process is illustrated in Figure 2.

Assume we have two lists with $k$ elements as input to the merge step. In case $k = 1$, we only need one level of $\mathcal{C}_{\text{CS}}$, thus the depth of the circuit $t_1 = 1$. In case $k = 2$, the first step recursively applies the merge step with $k = 1$ twice in parallel. Then, the second step applies inner adjacent comparisons which increment the depth by one, i.e., $t_2 = t_1 + 1 = 2$. For any $k$, we can conclude that $t_k = t_{k/2} + 1 = \log(k) + 1$. Hence, when we sort $N$ elements, the overall depth can be computed as $\sum_{i=0}^{\log(N)-1} t_k = \sum_{i=0}^{\log(N)-1}(\log(k) + 1)$ where $k = 2^i$ which yields $\sum_{i=0}^{\log(N)-1}(i-1) = [\log^2(N) + \log(N)]/2$. Therefore, $d(\mathcal{C}_{\text{MS}}) = \mathcal{O}(\log^2(N))d(\mathcal{C}_{\text{CS}})$. Similarly, let total number of comparison and swaps in a merge step in a single block be $c_k$. Then $c_1 = 1$ and $c_2 = 2 \cdot c_1 + 1$, since in a single merge block with $k = 2$, there are two merge blocks with $k = 1$ and only one inner adjacent element pair. In the general case, we have $c_k = 2 \cdot c_{k/2} + k - 1$ for arbitrary $k$. Consequently, for $k = 2^i$, we have $c_i = 2 \cdot c_{i-1} + \log(i) - 1$, which is equal to $c_i = (i+1)2^i - \sum_{j=0}^{i-1} 2^j = i \cdot 2^i + 1$. Since there are $[N/(2k)] = N/2^{i+1}$ parallel blocks in each merge step, the total number of $\mathcal{C}_{\text{CS}}$ will be $\sum_{i=0}^{\log(N)-1}[N2^{-(i+1)}c_i] = N\sum_{i=0}^{\log(N)-1}[2^{-(i+1)} + i/2]$, which results in an asymptotic complexity of $\mathcal{O}(N\log^2(N))$.

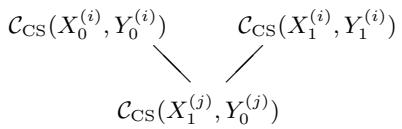


Fig. 2: Odd-Even Merging two individually sorted encrypted arrays: $\langle X_0^{(i)}, X_1^{(i)}\rangle$ and $\langle Y_0^{(i)}, Y_1^{(i)}\rangle$

### 4.6 Bitonic Sort

Bitonic Sorter is another sorting network created by Batcher [1]. It has similar complexity to Odd-Even Merge Sort, but with slightly different number of comparisons. The algorithm again consists of recursive sort and merge operations. The base case occurs when there are only two elements in the input array in which case only one $\mathcal{C}_{\text{CS}}$ is applied. In order to merge two sorted arrays, first of all their elements are compared and swapped so that all elements of the first subsequence are smaller than the second one. Then the subsequences

are individually sorted. The depth is computed as $d(\mathcal{C}_{OEM-SORT}) = (\log^2(N)+\log(N))/2d(\mathcal{C}_{\text{CS}})$. The depth is $\mathcal{O}(\log^2(N)\log(\ell))$ and a total of $\mathcal{O}(N\log^2(N))$ comparison complexity.

## 5 Proposed Sorting Algorithms

Given the inadequacies of existing sorting algorithms in permitting shallow circuit evaluation, we develop two new sorting algorithms, Direct Sort and Greedy Sort, optimized for this purpose. Both algorithms take an input vector and compute the sorted vector by evaluating the sorting circuits $\mathcal{C}_{\text{DS}}$ and $\mathcal{C}_{\text{GS}}$. The circuit evaluation makes it easy to apply the SWHE algorithm for homomorphic evaluation. The first circuit $\mathcal{C}_{\text{DS}}$ makes use of the equality check $\mathcal{C}_{\text{EQ}}$ and comparison circuits $\mathcal{C}_{\text{LT}}$ defined in Section 1 as building blocks whereas the second $\mathcal{C}_{\text{GS}}$ uses only the comparison circuit $\mathcal{C}_{\text{LT}}$.

**Sorting Circuits $\mathcal{C}_{\text{DS}}$, $\mathcal{C}_{\text{GS}}$**
**Encrypted Input vector:** $E(X) = \langle X_0^{(\alpha)}, X_1^{(\alpha)}, \ldots, X_{N-1}^{(\alpha)}\rangle$
**Encrypted Output vector:** $Y^{(\beta)} = \langle Y_0^{(\beta)}, Y_1^{(\beta)}, \ldots, Y_{N-1}^{(\beta)}\rangle$ The first step, which is mutually used by both of the circuits, constructs a comparison matrix $M$:

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

Each $m_{i,j}^{(\gamma)}$ is computed as follows[4]:

$$m_{ij}^{(\gamma)} = \mathcal{C}_{\text{LT}}((X_i)^{(\alpha)}, (X_j)^{(\alpha)}) = \begin{cases} m_{ij} = 1 \text{ if } \ X_i < X_j \\ m_{ij} = 0 \text{ else} \end{cases}$$

where $i, j < N$ and $i < j$. The diagonal elements are self comparisons, i.e. $X_i < X_i$, therefore $m_{i,i} = 0,\ \forall i \in [0, N-1]$. The remaining entries in the lower triangular part of $M$, whose indices satisfy $i > j$, are computed as $m_{ji}^{(\gamma)} = m_{ij}^{(\gamma)} \oplus 1$. Note that the lower triangular part holds the comparison $m_{ji}^{(\gamma)} = (X_i \geq X_j)$. The adopted approach is straightforward as we simply compare every element with every other element in the input vector. But in terms of depth, it has a significant advantage, as performing all comparisons in the beginning reduces the depth by $d(\mathcal{C}_{\text{LT}})$ in each comparison level. In the construction of $M$ we perform $N(N-1)/2$ parallel $\mathcal{C}_{\text{LT}}$ 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 constructing $M$ at the outset we simply avoid further $\mathcal{C}_{\text{LT}}$ computations during the execution of the later steps and the multiplicative depth will thus be minimized.

### 5.1 Direct Sort

The next step for $\mathcal{C}_{\text{DS}}$ is computing the index vector, $\sigma$, which indicates the positions of the vector elements in the sorted output vector, and is computed using the comparison matrix $M$ as

$$\sigma^{(\delta)} = \left( \sum_{i=0}^{N-1} m_{i,0}^{(\gamma)} \sum_{i=0}^{N-1} m_{i,1}^{(\gamma)} \cdots \sum_{i=0}^{N-1} m_{i,N-1}^{(\gamma)} \right).$$

Note that in $M$, the sum of all elements in a column gives the number of elements, which the element with the index of the column number is greater than. For instance, the sum of all elements in column $j$ is the

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

number of elements, which the element $X_j$ is larger than, as we add 1 to the sum for each such element. Therefore, the sum is also 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 $k+1^{st}$ largest element and its index is $k$ in a zero-based output vector. Now, since all data is in an encrypted form, we have no knowledge about the elements of the $\sigma$; therefore we cannot use it directly for homomorphic sorting. Here, we simply compare each element of the index vector $\sigma^{(\gamma)}$ (i.e., $\sigma_i^{(\gamma)}$) with each possible index value (which is in the interval $[0, N-1]$); the equality places the corresponding input element in the current position of the output vector. For this, we make use of $\mathcal{C}_{\text{EQ}}$ circuit as follows

$$Y_j^{(\beta)} = \sum_{i \in [N]} (\sigma_i^{(\delta)} = j) X_i^{(\alpha)} \quad \text{for } j \in [N].$$

The overall method for open version $\mathcal{C}_{\text{DS}}$ is described in Algorithm 1.

**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
```

*Example 1.* For an input vector $X = \langle 1, 3, 4, 3 \rangle$, the comparison matrix $M$ and the index vector $\sigma$ will be obtained as

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

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

Here we need to remember that, all inputs ,i.e each element of $X$ and correspondingly all entries of $M$ and $\sigma$ are encrypted in the. Using $\mathcal{C}_{\text{DS}}$ we compute the sorted output sequence elements as follows.

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

$$
\begin{aligned}
Y_1 &= \sum_{i\in[N]} (\sigma_i = 1)X_i \\
&= (\sigma_0 = 1)X_0 + (\sigma_1 = 1)X_1 + (\sigma_2 = 1)X_2 + (\sigma_3 = 1)X_3 \\
&= (0 = 1)X_0 + (2 = 1)X_1 + (3 = 1)X_2 + (1 = 1)X_3 \\
&= (0)X_0 + (0)X_1 + (0)X_2 + (1)X_3 \\
&= X_3.
\end{aligned}
$$

$$
\begin{aligned}
Y_2 &= \sum_{i\in[N]} (\sigma_i = 2)X_i \\
&= (\sigma_0 = 2)X_0 + (\sigma_1 = 2)X_1 + (\sigma_2 = 2)X_2 + (\sigma_3 = 2)X_3 \\
&= (0 = 2)X_0 + (2 = 2)X_1 + (3 = 2)X_2 + (1 = 2)X_3 \\
&= (0)X_0 + (1)X_1 + (0)X_2 + (0)X_3 \\
&= X_1.
\end{aligned}
$$

$$
\begin{aligned}
Y_3 &= \sum_{i\in[N]} (\sigma_i = 3)X_i \\
&= (\sigma_0 = 3)X_0 + (\sigma_1 = 3)X_1 + (\sigma_2 = 3)X_2 + (\sigma_3 = 3)X_3 \\
&= (0 = 3)X_0 + (2 = 3)X_1 + (3 = 3)X_2 + (1 = 3)X_3 \\
&= (0)X_0 + (0)X_1 + (1)X_2 + (0)X_3 \\
&= X_2.
\end{aligned}
$$

Consequently, the output vector will be $Y = \langle X_0, X_3, X_1, X_2\rangle = \langle 1, 3, 3, 4\rangle$.

From the discussions in Section 1, we already know that $d(\mathcal{C}_{\text{LT}}) = \log(\ell + 1)$ and $d(\mathcal{C}_{\text{EQ}}) = \log(\ell)$. In the computations of the entries of $\sigma$ we add $N$ bits to form a $\log(N)$-bit sum. In this step full and half adders are used in a Wallace Tree structure, hence the depth of the circuit for the $N$-bit summation can be given approximately as $d(\sigma) = \mathcal{O}(\log_{3/2}(N))$. Taking into account the parallel $\mathcal{C}_{\text{LT}}$ and $\mathcal{C}_{\text{EQ}}$ comparisons and single multiplication in the final summation the total depth becomes $d(\mathcal{C}_{\text{LT}}) + d(\sigma) + d(\mathcal{C}_{\text{EQ}}) + 1$. Therefore, we can obtain the following expression for the overall depth of the circuit that implements the proposed algorithm: $d(\mathcal{C}_{\text{DS}}) = \mathcal{O}(\log(N) + \log(\ell))$.

### 5.2 Greedy Sort

In this scheme, we compute every possible permutation of indices for the sorted array. For instance, to determine the smallest element $Y_0$ in the sorted array we need to check if a candidate element $X_i$ is smaller than all the other elements in $X$, to be set as the smallest element of the sorted array. We can express the conditions yielding the $Y_0$ assignment explicitly as in Algorithm 2. Similarly, for $Y_1$ if an element is

**Algorithm 2** Finding the minimum element

1: **if** $(X_0 < X_1) \wedge (X_0 < X_2) \wedge \ldots \wedge (X_0 < X_{N-1})$ **then**
2: $\quad Y_0 = X_0$
3: **else if** $\neg(X_0 < X_1) \wedge (X_1 < X_2) \wedge \ldots \wedge (X_1 < X_{N-1})$ **then**
4: $\quad Y_0 = X_1$
5: **else if** $\ldots$ **then**
6: $\quad \vdots$
7: **end if**

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 in Algorithm 3. Using the comparison matrix $M^{(\gamma)}$, defined in Section 5.1, we can

**Algorithm 3** Finding the second minimum element

1: **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**
2: $\quad Y_1 = X_0$
3: **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**
4: $\quad Y_1 = X_1$
5: **else if** $\ldots$ **then**
6: $\quad \vdots$
7: **end if**

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

$$Y_0^{(\beta)} = \left(m_{0,1}^{(\gamma)} \ldots m_{0,N-1}^{(\gamma)}\right) X_0^{(\alpha)} \oplus \left(m_{1,0}^{(\gamma)} \ldots m_{1,N-1}^{(\gamma)}\right) X_1^{(\alpha)} \ldots \oplus \left(m_{N-1,0}^{(\gamma)} \ldots m_{N-1,N-2}^{(\gamma)}\right) X_{N-1}^{(\alpha)}.$$

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 $t = 0$, $\theta_{0,i} = \prod_{\substack{j=0 \\ j \neq i}}^{N-1} m_{ij}$ and the overall equation simply becomes $Y_0 = \theta_{0,0} X_0 \oplus \ldots \oplus \theta_{0,N-1} X_{N-1}$ .

In condition evaluations we can also convert the OR gates (i.e., logical disjunction $\vee$ in Algorithm 3) to XOR gates. To see why this works, first note that $a + b = a \oplus b \oplus (a \cdot b)$ where $a$ and $b$ are bits. If $a \cdot b = 0$ then $a + b = a \oplus b$. We can make 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 1** *In the expression of $\theta_{t,i}$ of the element $X_i$, for any two distinct conjunctions $\rho$ and $\rho'$ 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 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 $\oplus$ and the circuit for $Y_1$ becomes $Y_1^{(\beta)} = \theta_{1,0}^{(\gamma)} X_0^{(\alpha)} \oplus \ldots \oplus \theta_{1,N-1}^{(\gamma)} X_{N-1}^{(\alpha)}$ where $\theta_{1,i}^{(\gamma)} = \sum_{\substack{k_1=0 \\ k_1 \neq i}}^{N-1} m_{k_1 i}^{(\gamma)} \prod_{\substack{j=0 \\ j \neq i, k_1}}^{N-1} m_{ij}^{(\gamma)}$ . More generally, for other $t$ values, following a

similar logical expression, we will have $\binom{N-1}{t}$ possibilities, and $\theta_{t,i}^{(\gamma)}$ will be computed as

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

and the output values of $\mathcal{C}_{\mathrm{GS}}$, $Y_t$ for $t \in [N]$ as $Y_t^{(\beta)} = \sum_{i=0}^{N-1} X_i^{(\alpha)} \theta_{t,i}^{(\gamma)}$ . Each output of the circuit $\mathcal{C}_{\mathrm{GS}}$ computes a summation of the input values $X_0, \ldots, X_{N-1}$ where values are weighted with $\theta_{t,i}$. Note that $\theta_{t,i}$ evaluates a logic expression that determines whether $X_i$ ends up in position $t$, i.e. $Y_t = X_i$, after sorting. The overall method for unencrypted $\mathcal{C}_{\mathrm{GS}}$ is described in Algorithm 4.

**Algorithm 4** Greedy Sorting Algorithm

**function** SORT($X, Y, N$)
  $iter \leftarrow \lceil \log(N-1) \rceil$
  $row \leftarrow 1$, $col \leftarrow (N-1)$
  **for** $i \leftarrow 0$ to $N-1$ **do** ▷ Construct $M$ table
    $M[i][i] \leftarrow 0$
    **for** $j \leftarrow i+1$ to $N-1$ **do**
      $M[i][j] \leftarrow (X[i] < X[j])$
      $M[j][i] \leftarrow (M[j][i] \oplus 1)$
    **end for**
  **end for**
  **for** $i \leftarrow 0$ to $N-1$ **do**
    **for** $j_1 \leftarrow [(i+1) \mod N]$ to $[(i-2) \mod N]$ **do**
      $j_2 \leftarrow (j_1+1) \mod N$, $j \leftarrow (j_1 - i) \mod N$
      $T[i][0][j] \leftarrow (M[i][j_1] \cdot M[i][j_2])$
      $T[i][1][j] \leftarrow (M[i][j_1] \oplus M[i][j_2])$
      $T[i][2][j] \leftarrow T[i][0][j] + T[i][1][j] + 1$
      $j_1 \leftarrow (j_1+1) \mod N$
    **end for**
    **if** $N$ is even **then**
      $j \leftarrow (N/2-1)$
      $T[i][0][j] \leftarrow M[i][j_1]$
      $T[i][1][j] \leftarrow M[j_1][i]$
      $T[i][2][j] \leftarrow 0$
    **end if**
  **end for**
  $Row \leftarrow 3$, $Col \leftarrow \lceil (N-1)/2 \rceil$
  **for** $k \leftarrow 1$ to $iter-1$ **do**
    $Row' \leftarrow 2^{k+1}+1$
    **for** $i \leftarrow 0$ to $N-1$ **do**
      **for** $j \leftarrow 0$ to $Col-1$ **do**
        **for** $r_1 \leftarrow 0$ to $Row-1$ **do**
          **for** $r_2 \leftarrow 0$ to $Row-1$ **do**
            $T[i][r_1+r_2][j/2] \leftarrow T[i][r_1+r_2][j/2]$
            $+\mathrm{AND}(C[i][r_1][j], C[i][r_2][j])$;
          **end for**
        **end for**
        $j \leftarrow j+1$
      **end for**

```
            if Col is odd then
                for r ← 0 to Row − 1 do
                    T[i][r][j/2] ← T[i][r][j]
                end for
                for r ← Row to Row′ − 1 do
                    T[i][r][j/2] ← 0
                end for
            end if
        end for
        Row ← Row′
        Col ← Col/2
    end for
    for t ← 0 to N − 1 do                                  ▷ Compute θ_{i,t}X_i products
        for i ← 0 to N − 1 do
            TX[t][i] ← AND (T[i][t][0], X[i])
        end for
    end for
    for t ← 0 to N − 1 do                                  ▷ Sum θ_{i,t}X_i products
        for i ← 0 to N − 1 do
            Y[t] ← Y[t] + TX[t][i]
        end for
    end for
end function
```

*Example 2.* For the same input vector $X = \langle 1, 3, 4, 3\rangle$, the comparison matrix $M$ will be obtained as

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

The circuit $y = \mathcal{C}_{\text{GS}}(X)$ is instantiated for $N = 4$ as

$$Y_0 = X_0(m_{01}m_{02}m_{03}) \oplus X_1(m_{10}m_{12}m_{13}) \oplus X_2(m_{20}m_{21}m_{23}) \oplus X_3(m_{30}m_{31}m_{32})$$
$$Y_1 = X_0[m_{10}(m_{02}m_{03}) \oplus m_{20}(m_{01}m_{03}) \oplus m_{30}(m_{01}m_{02})] \oplus X_1[m_{01}(m_{12}m_{13}) \oplus m_{21}(m_{10}m_{13}) \oplus m_{31}(m_{10}m_{12})]$$
$$\oplus X_2[m_{02}(m_{21}m_{23}) \oplus m_{12}(m_{20}m_{23}) \oplus m_{32}(m_{20}m_{21})] \oplus X_3[m_{03}(m_{31}m_{32}) \oplus m_{13}(m_{30}m_{32}) \oplus m_{23}(m_{30}m_{31})]$$
$$Y_2 = X_0[m_{10}(m_{20}(m_{03}) \oplus m_{30}(m_{02})) \oplus m_{20}(m_{30}m_{01})] \oplus X_1[m_{01}(m_{21}(m_{13}) \oplus m_{31}(m_{12})) \oplus m_{21}(m_{31}m_{10})]$$
$$\oplus X_2[m_{02}(m_{12}(m_{23}) \oplus m_{32}(m_{21})) \oplus m_{12}(m_{32}m_{20})] \oplus X_3[m_{03}(m_{13}(m_{32}) \oplus m_{23}(m_{31})) \oplus m_{13}(m_{23}m_{30})]$$
$$Y_3 = X_0(m_{10}(m_{20}m_{30})) \oplus X_1(m_{01}(m_{21}m_{31})) \oplus X_2(m_{02}(m_{12}m_{32})) \oplus X_3(m_{03}(m_{13}m_{23}))$$

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

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

Note that in each group $\theta_{t,i}$ selects only one source $i$ value for each output position $t$. Finally, compute the output vector $Y_t = \sum_{i=0}^{N-1} X_i\theta_{t,i}$ for $t \in [N]$.

$$\begin{aligned}
Y_0 &= \sum_{i\in[N]} \theta_{0,i}X_i \\
&= \theta_{0,0}X_0 + \theta_{0,1}X_1 + \theta_{0,2}X_2 + \theta_{0,3}X_3 \\
&= (1)X_0 + (0)X_1 + (0)X_2 + (0)X_3 \\
&= X_0. \\
Y_1 &= \sum_{i\in[N]} X_i \\
&= \theta_{1,0}X_0 + \theta_{1,1}X_1 + \theta_{1,2}X_2 + \theta_{1,3}X_3 \\
&= (0)X_0 + (0)X_1 + (0)X_2 + (1)X_3 \\
&= X_3. \\
Y_2 &= \sum_{i\in[N]} \theta_{2,i}X_i \\
&= \theta_{2,0}X_0 + \theta_{2,1}X_1 + \theta_{2,2}X_2 + \theta_{2,3}X_3 \\
&= (0)X_0 + (1)X_1 + (0)X_2 + (0)X_3 \\
&= X_1. \\
Y_3 &= \sum_{i\in[N]} \theta_{3,i}X_i \\
&= \theta_{3,0}X_0 + \theta_{3,1}X_1 + \theta_{3,2}X_2 + \theta_{3,3}X_3 \\
&= (0)X_0 + (0)X_1 + (1)X_2 + (0)X_3 \\
&= X_2.
\end{aligned}$$

Consequently, the output vector will be $Y = \langle X_0, X_3, X_1, X_2\rangle = \langle 1, 3, 3, 4\rangle$.

The overall depth is $d(\mathcal{C}_{\mathrm{GS}}) = d(\mathcal{C}_{\mathrm{LT}}) + d(\theta_{t,i}X_i)$., where $d(\mathcal{C}_{\mathrm{LT}}) = \log(\ell + 1)$ as given in Section 3. During the $\theta_{t,i}$ computations we employ a circuit arranged in a binary tree of depth $d(\theta_{t,i}) = \lceil log(N-1)\rceil$ and $d(\theta_{t,i}X_i) = d(\theta_{t,i}) + 1$. Consequently, the overall circuit depth is found as $d(\mathcal{C}_{\mathrm{GS}}) = \lceil\log(\ell+1)\rceil + \lceil\log(N-1)\rceil + 1 = \mathcal{O}(\log(N) + \log(\ell))$.

## 6 Implementation Results

We implemented the proposed depth optimized sorting method described in Algorithm 1 using the SWHE scheme of [11] and evaluated $\mathcal{C}_{\mathrm{DS}}$ for a number of array lengths. Here, we briefly summarize the parameter selection process and present the simulation results.

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

Table 1: The multiplicative depth of different sorting circuits given size $N$ and $\ell$

**Parameter Selection.** According to [11] the NTRU based SWHE Scheme requires Hermite factor $\delta < 1.0066$ to achieve a security level of 80-bit. We set the per level cutting rate $\log p$ depending both on the circuit itself and its total depth, similarly we choose a polynomial degree $n$ according to security threshold and maximum coefficient modulus size. We implemented $\mathcal{C}_{\text{OES}}$, $\mathcal{C}_{\text{OEMS}}$, $\mathcal{C}_{\text{DS}}$ and $\mathcal{C}_{\text{GS}}$ circuits, simulated them for both $\ell = 8$-bit and $\ell = 32$-bit integer inputs and selected array size $N$ as powers of two[5]. In Table 2 (see Appendix), we enumerate the parameters which we used in our experiments for various circuit depths. The largest Hermite factor among our parameter choices is $\delta = 1.0060$, ensuring a security level of 99-bits, which is the lowest security level for all cases.

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

Table 2: Cutting size $\log p$, maximum coefficient size $\log q_0$, Polynomial degree $n$, message batching slot size $S$ and Hermite Factor $\delta$ for different depths $d$

**Performance Results.** We implemented homomorphic Odd Even Sort, Batcher's Odd Even Merge Sort and both of the proposed algorithms in C++ using DHS-SWHE Library [11]. All simulations were performed on an Intel Xeon @ 2.9 GHz server running Ubuntu Linux 13.10. We compiled our code using Shoup's NTL library version 6.0 and with GMP version 5.1.3. The sorting times for 8 and 32 bit integers are given in Table 3. For $N = 64$ our algorithm runs in about 14.15 hours whereas the amortized running time, where we use batching with slot size 630, is about 1.35 minutes per sort. For $N = 4$ the sorting takes as low as 0.20 seconds per sort. In comparison, the homomorphic Lazy Sort implementation of [8] takes about 976 and 1400 seconds for array sizes of 10 and 40, respectively. For array sizes $N = 16$ and $N = 64$ our implementation takes 4.28 and 50 seconds, respectively.

## 7 Conclusion

We proposed two depth optimized sorting algorithms for efficient homomorphic evaluation. Circuit depth is intimately related to the parameter sizes in leveled homomorphic encryption implementations and therefore directly affect the overall performance of the homomorphic circuit evaluation. Existing sorting algorithms are not optimized for homomorphic evaluation. To close this gap we presented the depth analysis for several classical sorting algorithms: Bubble sort, Insertion Sort, Odd Even Sort, Odd Even Merge Sort, Merge Sort, and Bitonic Sort. Inspired by the performance of Merge Sort we introduced two new depth-optimized sorting algorithms which achieve a circuit depth of $\mathcal{O}(\log(N) + \log(\ell))$.

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

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

Table 3: Amortized execution time of circuits for different array sizes $N$ and input bit sizes $\ell$

To study the real-life performance of our sorting algorithms, we instantiated an NTRU based SWHE scheme in the DHS SWHE library and presented simulation results for selected array lengths. For this we determined the ideal parameter choices, e.g. modulus cutting levels to cope with noise growth and Hermite work factor estimates to ensure reasonable security margins. The implementation performs favorably achieving significant speedup over the proposal in [8] for similar array lengths.

## References


1. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 307–314. AFIPS '68 (Spring), ACM, New York, NY, USA (1968), http://doi.acm.org/10.1145/1468075.1468121
2. Bos, J.W., Lauter, K., Naehrig, M.: Private predictive analysis on encrypted medical data. Tech. Rep. MSR-TR-2013-81 (September 2013), http://research.microsoft.com/apps/pubs/default.aspx?id=200652
3. Bos, J., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) Cryptography and Coding, Lecture Notes in Computer Science, vol. 8308, pp. 45–64. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-45239-0_4
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapSVP. IACR Cryptology ePrint Archive 2012, 78 (2012)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Electronic Colloquium on Computational Complexity (ECCC) 18, 111 (2011)
6. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: FOCS. pp. 97–106 (2011)
7. Brenner M., Perl H., S.M.: libscarab software library., https://hcrypt.com/
8. Chatterjee, A., Kaushal, M., Sengupta, I.: Accelerating sorting of fully homomorphic encrypted data. In: Paul, G., Vaudenay, S. (eds.) Progress in Cryptology ? INDOCRYPT 2013, Lecture Notes in Computer Science, vol. 8250, pp. 262–273. Springer International Publishing (2013), http://dx.doi.org/10.1007/978-3-319-03515-4_17
9. Cheon, Jung Hee, M.K., Lauter., K.: Secure dna-sequence analysis on encrypted dna nucleotides., http://media.eurekalert.org/aaasnewsroom/MCM/\discretionary{-}{}{}FIL_000000001439/EncryptedSW.pdf
10. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: EUROCRYPT. pp. 24–43 (2010)
11. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic aes evaluation using ntru (2014), https://eprint.iacr.org/2014/039.pdf, iACR ePrint Archive
12. Doröz, Y., Sunar, B., Hammouri, G.: Bandwidth efficient pir from ntru. In: Bhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 8438, pp. 195–207. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-662-44774-1_16
13. Fischlin, M.: A cost-effective pay-per-multiplication comparison method for millionaires (2001)
14. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009)
15. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. pp. 169–178 (2009)
16. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: EUROCRYPT. pp. 129–148 (2011)
17. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. IACR Cryptology ePrint Archive Report 2011/566 (2011), http://eprint.iacr.org/
18. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. IACR Cryptology ePrint Archive 2012 (2012)

19. Goldwasser, S., Micali, S.: Probabilistic encryption & how to play mental poker keeping secret all partial information. In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. pp. 365–377. STOC '82, ACM, New York, NY, USA (1982), http://doi.acm.org/10.1145/800070.802212
20. Graepel, T., Lauter, K., Naehrig, M.: Ml confidential: Machine learning on encrypted data. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) Information Security and Cryptology ? ICISC 2012, Lecture Notes in Computer Science, vol. 7839, pp. 1–21. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-37682-5_1
21. Knuth, D.E.: The Art of Computer Programming, Fundamental Algorithms, vol. 1. Addison Wesley Longman Publishing Co., Inc., 3rd edn. (1998), (book)
22. Lagendijk, R., Erkin, Z., Barni, M.: Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. Signal Processing Magazine, IEEE 30(1), 82–105 (Jan 2013)
23. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical. Cloud Computing Security Workshop pp. 113–124 (2011)
24. Lauter, K., Lopez-Alt, A., Naehrig, M.: Private computation on encrypted genomic data. Tech. Rep. MSR-TR-2014-93 (June 2014), http://research.microsoft.com/apps/pubs/default.aspx?id=219979
25. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
26. López-Alt, A., Naehrig., M.: Large integer plaintexts in ring-based fully homomorphic encryption. in preparation (2014)
27. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation pp. 169–180 (1978)
28. Sander, T., Young, A., Yung, M.: Non-interactive cryptocomputing for nc1. In: Foundations of Computer Science, 1999. 40th Annual Symposium on. pp. 554–566 (1999)
29. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. IACR Cryptology ePrint Archive 2011, 133 (2011)
30. Stehlé, D., Steinfeld, R.: Making ntru as secure as worst-case problems over ideal lattices. Advances in Cryptology – EUROCRYPT '11 pp. 27–4 (2011)
31. Vaidya, J., Clifton, C.: Privacy-preserving k-means clustering over vertically partitioned data. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 206–215. KDD '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/956750.956776
32. Yao, A.C.: Protocols for secure computations. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science. pp. 160–164. SFCS '82, IEEE Computer Society, Washington, DC, USA (1982), http://dx.doi.org/10.1109/SFCS.1982.88
33. Yildizli, C.B., Pedersen, T., Saygin, Y., Savas, E., Levi, A.: Distributed privacy preserving clustering via homomorphic secret sharing and its application to vertically partitioned spatio-temporal data. Int. J. Data Warehous. Min. 7(1), 46–66 (Jan 2011), http://dx.doi.org/10.4018/jdwm.2011010103