Homomorphic Rank Sort Using Surrogate Polynomials

Gizem S. Çetin[1] and Berk Sunar[1]

Worcester Polytechnic Institute
{gscetin,sunar}@wpi.edu

Abstract. In this paper we propose a rank based algorithm for sorting encrypted data using monomials. Greedy Sort is a sorting technique that achieves to minimize the depth of the homomorphic evaluations. It is a costly algorithm due to excessive ciphertext multiplications and its implementation is cumbersome. Another method Direct Sort has a slightly deeper circuit than Greedy Sort, nevertheless it is simpler to implement and scales better with the size of the input array. Our proposed method minimizes both the circuit depth and the number of ciphertext multiplications. In addition to its performance, its simple design makes it more favorable compared to the alternative methods which are hard to parallelize, e.g. not suitable for fast GPU implementations. Furthermore, we improve the performance of homomorphic sorting algorithm by adapting the SIMD operations alongside message slot rotation techniques. This method allow us to pack N integers into a single ciphertext and compute N comparisons at once, thus reducing $\mathcal{O}(N^2)$ comparisons to $\mathcal{O}(N)$.

Keywords: Private computation, encrypted computing, fully homomorphic encryption, homomorphic sorting

1 Introduction

Blind sort is basically arranging a set of encrypted integers in order by using a somewhat, leveled or Fully Homomorphic Encryption (FHE) scheme [1–7, 10, 11, 13–16, 19, 23, 26] without knowledge of neither the plaintext data or the secret key used in encryption. The operations do not require decryption either. The crucial drawback of using FHE schemes in practice is their poor performance in high-level computations due to the high noise growth at the end of deep circuit evaluations. Most recent FHE schemes make use of noise reduction techniques in order to overcome this problem by setting the FHE parameters to evaluate only up to a certain depth. As a result, deep circuits require large parameters and they perform poorly.

In [8], Çetin et. al. analyze different sorting algorithms and compare their performances for ordering encrypted data. Their survey include well-known algorithms such as Bubble Sort, Merge Sort and two sorting networks: Bitonic Sort and Odd-Even Merge Sort [21]. Due to the high depths of known algorithms, the authors propose two new depth-optimized methods: Greedy Sort and Direct Sort. Both of these algorithms require a circuit of depth $\mathcal{O}\left(\log(N)+\log(\ell)\right)$ where N is the number of elements and ℓ is the bit-length of the elements. Other encrypted sorting works in the literature are of Chatterjee et al. [9] and Emmadi et al.'s [12]. In [9], the authors introduce a hybrid technique, i.e. Lazy Sort. This method first uses Bubble Sort to nearly sort the input elements. Then, the list is sorted again by using Insertion Sort. The authors claim that this method has better complexity than the worst case scenario. This is refuted by both [8] and [12]. Emmadi et al. implements and compares Bubble Sort, Insertion Sort, Bitonic Sort and Odd-Even Merge Sort in [12] and their observations are parallel with the analysis of [8]. Recently, Narumanchi et al. compared bitwise and integer-wise encryption within the context of comparison and sorting in [24]. Their analysis shows that it is more efficient to use bitwise encryption in terms of performance. All of the previous works still perform poorly in case of sorting a large data set. In the experiments, the largest N used is around 64.

The main contribution of this work is proposing an alternative way of sorting numbers by computing the Hamming weight of N bits with only N ciphertext multiplications. In comparison to our proposed method with $\mathcal{O}(N)$ multiplications, Direct Sort and Greedy Sort require $\mathcal{O}(N\log N)$ and $\mathcal{O}(2^N)$ multiplications, respectively. Our algorithm implements Direct Sort method with the minimum number of operations. We observe that our proposed method is also a compact implementation of Greedy Sort. Therefore, it both minimizes the circuit depth and the number of homomorphic evaluations. Furthermore, efficient evaluation of the Hamming weight can be used in many other homomorphic applications. In addition to performance improvements, the proposed algorithm is easier to analyze and implement in comparison to previous methods. Even when batching is not applicable, the highly parallelizable nature of the algorithm makes it an efficient candidate for a GPU implementation. Our sorting method is generic and can be implemented with existing software libraries, e.g. HElib [20], SEAL [22].

Our second contribution is adapting Single-Instruction Multiple-Data (SIMD) idea from [25] and permutation technique from [17] to evaluate parallel homomorphic comparisons to sort the elements of a single set. In previous works, batching is used to sort separate number sets simultaneously. This results in not taking advantage of batching when there is only one set to be sorted. We propose placing the set elements into message slots of a single plaintext and using rotation method from [17] when across-slot computation is required. Gentry et.al. used this technique to evaluate an AES circuit homomorphically in [18]. This method requires key switching after every rotation. However we are able to reduce the number of comparisons from N^2 to $N/2$.

2 Background

Following a similar notation to [17,18,25], we define the plaintexts with lowercase letters $a \in \mathbb{A}_p$, batched plaintexts with Greek letters $\alpha \in \mathbb{A}_p$ and the ciphertexts with uppercase letters $A \in \mathbb{A}_q$ where $\mathbb{A}_p = \mathbb{Z}_p/\Phi_m(X)$ and $\mathbb{A}_q = \mathbb{Z}_q/\Phi_m(X)$. We use prime p, q in our scheme and $\Phi_m(X)$ is the m^{th} cyclotomic polynomial. Given two ciphertexts A, B that are encrypted under an FHE scheme, computing $A + B$ and $A \times B$ in $\mathbb{A}_q$ gives us encryptions of $a + b$ and $a \times b$ in $\mathbb{A}_p$.

When a number a has k base-p digits, we use an array index notation to represent its digits $[a_0, a_1, \cdots, a_{k-1}]$ and $a = \sum_{i=0}^{k-1} p^i a_i$. An encryption of a would be a vector of ciphertexts with encryptions of its digits, i.e. $[A_0, A_1, \cdots, A_{k-1}]$. When there is a list of numbers, we use the double-indexed positioning with the first one being the number's position and the second one is for the digit's index. For instance, a plaintext $\alpha_{i,j}$ is the j^{th} digit of the i^{th} number and similarly $A_{i,j}$ is an encryption of the j^{th} digit of the i^{th} number.

2.1 Batching and Rotation of the Message Slots

From [25], we know that with specific parameters we can enable batching, a technique that is used for parallel evaluations in the message slots. For example, when we choose the cyclotomic polynomial with m that divides $p^d - 1$ with smallest such d, we have the factorization,

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

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

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

We define a vector with ℓ messages $\boldsymbol{\alpha} = \langle \alpha_1, \cdots, \alpha_\ell \rangle$ with each α_i belonging to the field $\mathbb{L}_i = \frac{\mathbb{F}_p[X]}{F_i}$. Applying inverse Chinese Remainder Theorem (CRT), we derive a single message in $\mathbb{A}_p$. We write this as:

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

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

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

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

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

for a g that is not a power of 2 in $(\mathbb{Z}/m\mathbb{Z})^*$ with order ℓ performs a permutation. To see why this works and for the underlying Galois field theory we refer readers to [17].

2.2 Problem Definition and Existing Algorithms

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

In the following part of this section, we will briefly describe the previously proposed methods to solve the encrypted sorting problem. Before the algorithm descriptions, we shall define a comparator. In order to sort a set of elements, we need to be able to compare two numbers with respect to their magnitude. In [8], where the input elements are bitwise encrypted, this comparison is converted to a binary circuit as follows[1]:

$$A \lessdot B = \sum_{i=0}^{k-1} (A_i \oplus 1)\, B_i \prod_{j=i+1}^{k-1} (A_j \oplus B_j \oplus 1) \tag{1}$$

where A_i, B_i are the encryptions of i^{th} bit of k bit numbers a and b, respectively. The decryption of $A \lessdot B$ outputs a 1 if $a < b$ and a 0 otherwise.

Greedy Sort In [8], authors propose Greedy Sort as an alternative method to classical sorting algorithms, due to its low circuit depth. However, due to the algorithm's exhaustive nature, the method is not efficient when implemented without optimizations. In this section, we describe the algorithm and later in Section 3, we propose our optimization that minimizes the total number of ciphertext multiplications.

Greedy Sort works by computing every possible permutation of the input set. In order to find the minimum element b_0, one needs to compare each a_i to every other element in the set. If it is smaller than all a_js where $i \neq j$, then we can conclude it is the smallest element and set $b_0 = a_i$. Similarly, in order to find the next smallest element b_1, we need to compare every a_i to every other element and if it is smaller than all but one, then we know that it is the second

[1] If the plaintext modulus p is initialized as 2, an XOR "$\oplus$" operation is performed via addition. Otherwise, $A \oplus B = A + B - 2(A \times B)$ and $A \oplus 1 = 1 - A$.

minimum and set $b_1 = a_i$. We can follow the same idea until the last element of the sorted array b_{n-1} which is the maximum element is found.

This method requires comparison of every pair in the set, thus the initial step is to build a matrix M that holds the comparison outputs. Entries of this matrix, let them be $M_{i,j}$s are evaluated using the binary circuit given in Equation 1 such that $M_{i,j} = A_i \lessdot A_j$. Here, note that $M_{i,j}$s and $M_{j,i}$s naturally complement each other[2] and $M_{i,i}$s are always zero. Therefore $M_{i,j}$s are only evaluated for $i < j$. Given the comparison matrix, the ordered elements are computed as follows:

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

where

$$\theta_{r,i} = \sum_{\substack{k_1=0 \\ k_1 \neq i}}^{N-r-1} M_{k_1,i} \cdots \sum_{\substack{k_r=k_{r-1}+1 \\ k_r \neq i}}^{N-1} M_{k_r,i} \prod_{\substack{j=0 \\ j\neq i \\ j \neq k_1,\cdots,k_r}}^{N-1} M_{i,j} \ . \tag{2}$$

$\theta_{r,i}$ values can be seen as the binary place indicators or a decision flag that indicates whether the input A_i will be mapped to output B_r. In other words, if the input A_i has the rank r, it is an encrypted one. This requires computation of all $\theta_{r,i}$s for $i, r = 0, \cdots, N-1$, thus making the method inefficient. The Greedy Sort algorithm steps can be seen in Appendix C of [8].

Direct Sort In the same work [8], the authors propose another low-depth method for sorting, i.e. Direct Sort. The algorithm implements Rank Sort in the homomorphic setting. The first step is constructing the same comparison matrix M as in Greedy Sort. Then, it computes the ranks by performing a column-wise summation of the entries of M. Since the elements of M are bits, the summation result gives the Hamming weights of the columns of M. It is implemented using a Wallace Tree of depth $\mathcal{O}(\log_{3/2} N)$. The challenge is having the rank values encrypted after this step. This requires an additional homomorphic equality check to place the elements in the output in an ordered manner. This equality check is performed on rank values which are $\log N$ bit numbers, hence requires a homomorphic evaluation of depth $\log\log N$. The method can be found in Algorithm 1 in [8].

3 Our Proposal: Polynomial Rank Sort

In a nutshell, the idea is to use the rank of an input as the degree of a monomial –which we call a *rank monomial*– and place the input element in the coefficient of the monomial.

[2] $M_{j,i} = M_{i,j} \oplus 1$

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

Claim. If we can find the rank monomials $\{x^{r_0}, x^{r_1}, \cdots, x^{r_{N-1}}\}$ of all of the set elements, then we can have an output polynomial with input elements lined up in its coefficients with respect to their rank:

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

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

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

Challenge. Implementing this method on private data has two challenges: finding the ranks of encrypted inputs and placing the encrypted rank values in the exponent. In the following section, we propose a solution to this problem.

3.1 Finding Rank Monomials

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

We shortly describe the method as follows: For an N-element input set, we start by finding the zero-based rankings in each 2-subset, being either 0 or 1. We start by constructing surrogate monomials using these ranks. Then, we merge the subsets by performing polynomial multiplication among the surrogates. At the end, this gives us the rank monomials of each input element.

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

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

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

Now, we include a third element c in the input set. In this case, the first step is to find the ranks in each 2-subset. We look at all two element subsets; $\{a, b\}$, $\{a, c\}$ and $\{b, c\}$ and define the following surrogate monomials:

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

where r_{ij} is the rank of input i in the 2-subset $\{i, j\}$ and ρ_{ij} is the rank monomial of the same element in the same subset. The next step is merging two subsets to find the final rank monomials. Multiplying two surrogates adjusts the degree of the rank monomial depending on the subset-wise rank values. When we want to find the rank monomial for a particular input, we multiply each surrogate that is pertinent to that element:

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

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

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

Then the rank monomials will be:

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

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

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

$$\rho_{ij}(x) = x^{r_{ij}} \tag{3}$$

for all 2-subsets $\{a_i, a_j\}$ that contain a_i, i.e. $\forall j \in [0, N-1]$ and $j \neq i$. Then, computing the product of all the surrogates would carry the overall rank of a_i in the power of:

$$\rho_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{N-1} \rho_{ij}(x) \ . \tag{4}$$

The degree of $\rho_i(x)$ is the rank of a_i in the N-element set. This operation can be viewed as increasing the rank by one whenever an element a_i is larger than another element. In other words, it is counting the number of smaller elements, i.e. the rank of a_i.

Connection to Direct Sort This method is equivalent to summing the column entries of the comparison matrix M, i.e. computing the Hamming weights, as performed in Direct Sort (see Algorithm 1 in [8]). In order to see this connection, we must first confirm that comparison matrix elements m_{ij} and 2-subset ranks r_{ij} complement each other. This is because m_{ij} is the Boolean output of the comparison $a_i < a_j$, which is 1 if and only if a_i is smaller than a_j. However r_{ij} is 0 in this case, since it is the smallest element in the same 2-set. Hence, r_{ij} and m_{ij} complement each other and we can assert that $r_{ij} = m_{ji}$. We expand the product as in Equation 5 and notice that the summation in the exponent is equivalent to the Hamming weight of the columns of M.

$$\rho_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{N-1} x^{r_{ij}} = x^{\sum_{\substack{j=0 \\ j\neq i}}^{N-1} r_{ij}} = x^{\sum_{\substack{j=0 \\ j\neq i}}^{N-1} m_{ji}} \tag{5}$$

Until now, we have showed that we can find the rank monomials given the base case surrogates by applying polynomial multiplication. In the following section we describe how to apply this method to encrypted input sets.

3.2 Finding Rank Monomials in the Encrypted Domain

In the encrypted domain, we have a set of N encrypted numbers $\{A_0, A_1, \cdots, A_{N-1}\}$ by utilizing an FHE scheme. Recall that the first step of the proposed algorithm constructs the surrogates for all 2-subsets as in Equation 3. This requires finding the encrypted rank R_{ij} in set $\{A_i, A_j\}$. Hence we use the comparison circuit from Equation 1 and compute the encrypted rank of a_i as:

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

Using this value, we can set the base case surrogate monomial using the following operation that requires arithmetic suitable for homomorphic evaluation:

$$P_{ij}(x) = 1 - R_{ij} + R_{ij} \cdot x \tag{6}$$

Rechecking the base case example with $a < b$, we can confirm that[3];

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

What happens when two elements are equal to each other? Then, both $a \lessdot b$ and $b \lessdot a$ are expected to output a zero. To fix this problem, we always perform only the first comparison and fix the second comparison to its complement. In other words, computing first one of the ranks $R_{ji} = A_i \lessdot A_j$ and setting the other $R_{ij} = 1 - R_{ji}$ solves the equality problem by making sure that the ranks

[3] Here, we use $[\![\,\cdot\,]\!]$ to represent an encryption of a constant value.

always complement each other. Thereby, we also avoid redundant homomorphic comparisons. Note that, in [8], only the upper half of the comparison matrix M is computed and the lower half entries are set as the complements. This is identical to computing R_{ij} for half of the element pairs.

The next step is to compute the final rank monomials by using Equation 4. This operation which only includes polynomial multiplication, can simply be evaluated using homomorphic evaluations:

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

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

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

where $b_0 \leq b_1 \leq \cdots \leq b_{N-1}$. The overall algorithm is summarized in Algorithm 1 with given encrypted input set and the set size N. In comparison to intricate Greedy Sort Algorithm given in Appendix C [8], the simplicity and elegance of Algorithm 1 makes it more favorable and convenient to implement and less troublesome to analyze. As we shall see in the next section, we also gain further in efficiency.

Algorithm 1 Polynomial Rank Sort

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

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

$$\begin{aligned}\rho_i(x) = \prod_{\substack{j=0\\ j\neq i}}^{N-1} \rho_{ij}(x) &= \prod_{\substack{j=0\\ j\neq i}}^{N-1} (m_{i,j} + m_{j,i}x)\\ &= \prod_{\substack{j=0\\ j\neq i}}^{N-1} m_{i,j} + \left[\sum_{\substack{k=0\\ k\neq i}}^{N-1} m_{k,i} \prod_{\substack{j=0\\ j\neq i\\ j\neq k}}^{N-1} m_{i,j}\right] x\\ &+ \cdots + \left[\prod_{\substack{j=0\\ j\neq i}}^{N-1} m_{j,i}\right] x^{N-1}\end{aligned} \tag{7}$$

3.3 Comparison with the Previous Methods

All three algorithms work by computing the comparison matrix M. Since that is a mutual step for all of the methods, we disregard the cost of constructing M in this analysis. The rest of the operations can be summarized for each method in Table 1.

Table 1. Comparison of the proposed algorithm with the previous methods in terms of number of ciphertext multiplications, multiplicative depth and output size.

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

The Naïve Greedy implements the Greedy Sort in a straightforward manner, hence the multiplications can be counted in the $\theta_{r,i}$ computations. In order to find the binary place indicators for an arbitrary element X_i, we need to compute $\theta_{0,i}, \theta_{1,i}, \cdots, \theta_{N-1,i}$ as in Equation 2 with each one contributing

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

multiplications.

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

Instead, our proposed method requires only a product of $N-1$ ciphertexts, in order to find a single rank.

4 Batching Input Elements

In this section, we describe how to pack input elements in the plaintexts to evaluate parallel comparisons in the message slots. All previously mentioned sorting algorithms require $\frac{N^2-N}{2}$ comparisons. By enabling batching, we can reduce it to $N/2$. Finding the rank monomial of one element requires $N-2$ ciphertext multiplications, thus finding all rank monomials requires $N(N-2)$ multiplications. We also reduce this number to $N-2$ with batching. Finally, when we encrypt the input elements, instead of having Nk encryptions, we only have k encryptions when batching is used, where k is the bit-length of the input elements. As a result, we have both a memory and performance gain with a factor of N. However, there is an additional cost of key switching that we need to perform after each rotation. This operation also increases the noise in the ciphertext, thus we try to limit the number of rotations.

For our application, we use the plaintext modulus $p = 2$. Choosing Φ_m with $m | 2^d - 1$, we utilize ℓ message slots defined over $\mathbb{F}_{2^d}$. Next we choose a $g \in (\mathbb{Z}/m\mathbb{Z})^*$ with order ℓ. We fix the first factor $F_1(X)$ and reorder the rest of the factors so that the permutation given by κ_g is a cyclic shift to the left. There is an isomorphism in between the different slot fields, hence we define a mapping in between $\mathbb{L}_1$ and $\mathbb{L}_i$ for each i:

$$\psi_i : \begin{cases} \mathbb{L}_1 \longmapsto \mathbb{L}_i \\ X \longmapsto X^{g^{(1-i)} \mod \ell} & \mod F_i(X) \end{cases}$$

Using this mapping we transfer the input elements from $\mathbb{L}_1$ to $\mathbb{L}_i$ and pack them as follows

$$\alpha = \mathsf{CRT}^{-1}\left(\langle \psi_1(\alpha_1), \psi_2(\alpha_2), \cdots, \psi_\ell(\alpha_\ell) \rangle\right)$$

and rotate the slots by computing the automorphism $\gamma = \kappa_g(\alpha)$.

$$\mathsf{CRT}(\gamma) = \langle \kappa_g(\psi_2(\alpha_2)), \kappa_g(\psi_3(\alpha_3)), \cdots, \kappa_g(\psi_\ell(\alpha_\ell)), \kappa_g(\psi_1(\alpha_1)) \rangle$$

Using the inverse of the above mappings ψ_i^{-1}, we can retrieve the original message contents after computing $\mathsf{CRT}(\gamma)$:

$$\left\langle \psi_1^{-1}(\kappa_g(\psi_2(\alpha_2))), \cdots, \psi_\ell^{-1}(\kappa_g(\psi_1(\alpha_1))) \right\rangle = \langle \alpha_2, \cdots, \alpha_1 \rangle \ .$$

Note that, performing a cyclic shift of j is the same as rotating the slots j times, in other words it is equivalent to computing κ_{g^j}. We also define two helpful basic operations to clear out a slot content and to select a slot content. Both can be performed with a logical AND operation by defining a bit-mask vector. If we want to clear the data in the i^{th} slot, we create a vector $\boldsymbol{\zeta}$ with $\zeta_i = 0$ and $\zeta_j = 1$ for all $j \neq i$ and use its batched polynomial, i.e. $\zeta = \mathsf{CRT}^{-1}(\boldsymbol{\zeta})$, to perform a logical AND with the input. Similarly, if we want to select the data in the i^{th} slot, we create a vector $\boldsymbol{\sigma}$ with $\sigma_i = 1$ and $\sigma_j = 0$ for all $j \neq i$, batch it $\sigma = \mathsf{CRT}^{-1}(\boldsymbol{\sigma})$ and multiply it with the input.

$$\begin{aligned}
\mathsf{CRT}(\alpha \times \zeta) &= \langle \alpha_1, \cdots, \alpha_{i-1}, 0, \alpha_{i+1}, \cdots, \alpha_{\ell-1} \rangle \\
\mathsf{CRT}(\alpha \times \sigma) &= \langle 0, \cdots, 0, \alpha_i, 0, \cdots, 0 \rangle
\end{aligned}$$

Going back to our sorting algorithm, initially we have a set of k-bit elements $\{a_0, a_1, \cdots, a_{N-1}\}$. We declare a vector for each bit,

$$\begin{aligned}
\boldsymbol{\alpha}_0 &= \langle a_{0,0}, a_{1,0}, \cdots, a_{N-1,0} \rangle \\
\boldsymbol{\alpha}_1 &= \langle a_{0,1}, a_{1,1}, \cdots, a_{N-1,1} \rangle \\
&\vdots \\
\boldsymbol{\alpha}_{k-1} &= \langle a_{0,k-1}, a_{1,k-1}, \cdots, a_{N-1,k-1} \rangle
\end{aligned}$$

where the i^{th} slot is reserved for the i^{th} element. Hence, each plaintext corresponding to a bit is batched, i.e. $\alpha_j = \mathsf{CRT}^{-1}(\boldsymbol{\alpha}_j)$, for $j = 0, \cdots, k-1$. Note that, we do not need to apply the mapping ψ_i before batching, when the slot content is an integer. Hence a vector $\boldsymbol{\alpha}$ can be viewed as holding different elements in each slot as,

$$\boldsymbol{\alpha} = \langle a_0, a_1, \cdots, a_{N-1} \rangle$$

for simplicity. Then, $\alpha = \mathsf{CRT}^{-1}(\boldsymbol{\alpha})$ is the batched plaintext with each element in a separate slot. The first step in Algorithm 1 is computing the comparisons $M_{ij} = A_i \lessdot A_j$ for all $i < j$. To this end, we apply rotation and obtain the following vectors:

$$\begin{array}{lllllll}
\boldsymbol{\gamma}_0 & = < & a_0, & a_1, & \cdots, & a_{N-2}, & a_{N-1} & > \\
\boldsymbol{\gamma}_1 & = < & a_1, & a_2, & \cdots, & a_{N-1}, & a_0 & > \\
\boldsymbol{\gamma}_2 & = < & a_2, & a_3, & \cdots, & a_0, & a_1 & > \\
 & & & & \vdots & & & \\
\boldsymbol{\gamma}_{\lfloor \frac{N}{2} \rfloor} & = < & a_{\lfloor \frac{N}{2} \rfloor}, & a_{\lfloor \frac{N}{2} \rfloor+1}, & \cdots, & a_{\lfloor \frac{N}{2} \rfloor-2}, & a_{\lfloor \frac{N}{2} \rfloor-1} & >
\end{array}$$

by deploying the cyclic shift as $\gamma_i = \kappa_{g^i}(\alpha)$. We clear second half of the last vector when N is even, i.e. clear slot contents of $\boldsymbol{\gamma}_{\frac{N}{2}}$ in the range $\left[\frac{N}{2}, \cdots, N-1\right]$. This is due to the fact that we need exactly $\frac{N(N-1)}{2}$ comparisons. By aligning the elements as above, we can perform parallel comparisons of $M_{ij} = A_i \lessdot A_j$ for $0 < j - i \leq \left\lfloor \frac{N}{2} \right\rfloor$ and $M_{ji} = A_j \lessdot A_i$ where $j - i > \left\lfloor \frac{N}{2} \right\rfloor$ by executing the comparison circuit as;

$$
\begin{aligned}
&\mu_1 = \gamma_0 \lessdot \gamma_1\\
&\boldsymbol{\mu}_1 = \langle a_0 \lessdot a_1, \cdots, a_{N-2} \lessdot a_{N-1}, a_{N-1} \lessdot a_0\rangle\\
&\mu_2 = \gamma_0 \lessdot \gamma_2\\
&\boldsymbol{\mu}_2 = \langle a_0 \lessdot a_2, \cdots, a_{N-2} \lessdot a_0, a_{N-1} \lessdot a_1\rangle\\
&\quad\vdots\\
&\mu_t = \gamma_0 \lessdot \gamma_{\lfloor \frac{N}{2}\rfloor}\\
&\boldsymbol{\mu}_t = \begin{cases} \left\langle a_0 \lessdot a_{\frac{N-1}{2}}, \cdots, a_{N-2} \lessdot a_{\frac{N-1}{2}-2}, a_{N-1} \lessdot a_{\frac{N-1}{2}-1}\right\rangle, \text{if } N \text{ is odd,}\\ \left\langle a_0 \lessdot a_{\frac{N}{2}}, \cdots, a_{\frac{N}{2}-1} \lessdot a_{N-1}, 0, \cdots, 0\right\rangle, \text{if } N \text{ is even.}\end{cases}
\end{aligned}
$$

Their complements are $\bar{\mu}_i = 1 - \mu_i$ for $i = 1, \cdots, t$. By rotating each complement i times to the right, we successfully align all M_{ij} values with only $N/2$ comparisons and N rotations in total. It is important to remark that even though rotation operation by itself is not expensive it is followed by a key switching operation which is both costly and increasing the noise. Therefore the rotation is not completely free. We define ν_is as follows:

$$
\begin{aligned}
&\boldsymbol{\mu}_i = \langle m_{0,i}, m_{0,i+1}, \cdots, m_{N-1,i-1}\rangle\\
&\bar{\mu}_i = 1 - \mu_i\\
&\bar{\boldsymbol{\mu}}_i = \langle m_{i,0}, m_{i+1,0}, \cdots, m_{i-1,N-1}\rangle\\
&\nu_i = \kappa_{g^{-i}}(\mu_i)\\
&\boldsymbol{\nu}_i = \langle m_{0,N-i}, m_{1,N-i+1}, \cdots, m_{i-1,N-1}, m_{i,0}, m_{i+1,0}, \cdots m_{N-1,N-i-1}\rangle\\
&\bar{\nu}_i = 1 - \nu_i, \qquad\qquad \text{for } i = 1, \cdots, t\ .
\end{aligned}
$$

The vectors, $\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_t, \boldsymbol{\nu}_1, \cdots, \boldsymbol{\nu}_t$ hold all comparison values $m_{i,j}$ for $i \neq j$ in the $(i+1)^{\text{th}}$ slot and $\bar{\boldsymbol{\mu}}_1, \cdots, \bar{\boldsymbol{\nu}}_1, \cdots$ hold their complements, i.e. $m_{j,i} = 1 - m_{i,j}$ in the same slot. Next step of the algorithm is computing the product $\prod_{i\neq j}(M_{ij} + M_{ji}x)$. For multiplication with constant x, we batch an x in each slot and multiply with the complement vectors $\bar{\mu}_i$ and $\bar{\nu}_i$. However, we first need to apply the mapping Ψ_i that is defined in the beginning of this section.

$$\boldsymbol{\chi} = \langle \psi_1(x), \psi_2(x), \cdots, \psi_{N-1}(x)\rangle \text{ and } \chi = \mathsf{CRT}^{-1}(\boldsymbol{\chi})$$

Next, we define the surrogates in the parallel slots as

$$
\begin{aligned}
\varphi_i &= \mu_i + \chi\bar{\mu}_i\\
\varphi_{t+i} &= \nu_i + \chi\bar{\nu}_i, \qquad \text{for } i = 1, \cdots, t\ .
\end{aligned}
$$

Finally, the last step is computing the product of the surrogates to find the rank monomials. We write

$$
\begin{aligned}
\varphi &= \prod_{i=1}^{2t} \varphi_i\\
\boldsymbol{\varphi} &= \langle \psi_1(\rho_1), \psi_2(\rho_2), \cdots, \psi_{N-1}(\rho_{N-1})\rangle
\end{aligned}
$$

where ρ_is are the rank monomials of the input elements. The final rotation is performed so that all rank monomials can be summed in the first message slot.

$$\beta = \sigma_i \sum_{i=0}^{N-1} \kappa_{g^i}(\varphi)$$
$$\boldsymbol{\beta} = \langle b, 0, \cdots, 0 \rangle$$

We have the sorted polynomial $b = \sum_{i=0}^{N-1} \rho_i \in \mathbb{L}_1$ in the first message slot.

We give a breakdown of the bandwidth and bottleneck operations in homomorphic evaluation of Polynomial Rank Sort to compare batching with no batching in the Table 2. The cost of a key switching operation depends on the underlying FHE scheme, however it is similar to the relinerization technique and mostly requires $\log q$ multiplications in $\mathbb{A}_q$. A single k-bit comparison circuit requires k ciphertext multiplications. Therefore, the cost of the key switching can exceed the N^2 comparisons, especially with large q. In that case, batching may not be preferable. On the other hand, some new schemes such as GSW [19] and F-NTRU [11] may need to compute $\log q$ multiplications in $\mathbb{A}_q$ for a single ciphertext multiplication. In that case, batching would perform better since the cost of key switching is same as a single ciphertext multiplication.

Table 2. A comparison of the storage and computation with and without batching.

	Inputs	Comparisons	Key Switching	Multiplications	Outputs
No Batching	kN	$\frac{N^2-N}{2}$	–	$N^2 - 2N$	k
Batching	k	$\frac{N}{2}$	$2N$	$N - 2$	k

4.1 Choosing m and d

In most recent FHE schemes, the security parameter heavily depends on the degree of $\mathbb{A}_q$, i.e. $\phi(m)$ and it is usually a large number. For instance, in the homomorphic evaluation of AES [18], $(m, d) = (11441, 48)$ is given as an example for a circuit of depth 10. This choice of m and d utilizes $\ell = 224$ message slots. In that scenario, d is chosen to be a multiple of 8 because of the underlying algebra of AES operations. In our case, the only restrictions are:

1. The output of the algorithm is an $N - 1$ degree polynomial, hence we need $d \geq N$.
2. There must be at least N message slots, thus $\ell \geq N$.
3. The depth of the circuit is $\log(N) + \log(k) + 1$ when sorting k-bit numbers.

Thus the parameters depend on the number of elements N. We also have $m|2^d-1$ and $\ell = \phi(m)/d$ by definition. For example, when $N \leq 28$, we can use $m = 16385$ and $d = 28$ with ring degree $\phi(m) = 12544$ and $\ell = 448$ message slots. This means that, we can sort $\ell/N = 16$ sets in parallel.

5 Conclusion

In conclusion, our proposed method Polynomial Rank Sort performs significantly better than previous algorithms and provides a depth and cost-optimized circuit for homomorphic sorting. It reduces the number of ciphertext multiplications to $\mathcal{O}(N^2)$ for sorting an array of N elements without packing. Furthermore, it is a refined algorithm suitable for parallelization. When batching is enabled, we sort the whole list with only $N/2$ comparisons and following with only N multiplications. Proposed batching method also reduce the data size from kN to N where k is the bit-length of the input elements. However it requires costly key switching operation in exchange. All of our proposed homomorphic algorithms are generic and can be used with the recent FHE schemes. The performance gain however depends on the choice of the scheme and the trade-off between the cost of key switching and ciphertext multiplication.

6 Acknowledgment

This work was supported by the US National Science Foundation CNS Award #1561536.

References

1. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapSVP. IACR Cryptology ePrint Archive 2012, 78 (2012)
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Electronic Colloquium on Computational Complexity (ECCC) 18, 111 (2011)
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ACM (2012)
4. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: Ostrovsky, R. (ed.) FOCS. pp. 97–106. IEEE (2011)
5. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 6841, pp. 505–524. Springer (2011)
6. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Advances in Cryptology–CRYPTO 2011, pp. 505–524. Springer (2011)
7. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing 43(2), 831–871 (2014)

8. Çetin, G.S., Doröz, Y., Sunar, B., Savaş, E.: Low Depth Circuits for Efficient Homomorphic Sorting. In: LatinCrypt (2015)
9. Chatterjee, A., Kaushal, M., Sengupta, I.: Accelarating 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)
10. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 6110, pp. 24–43. Springer (2010)
11. Doröz, Y., Sunar, B.: Flattening ntru for evaluation key free homomorphic encryption. Cryptology ePrint Archive, Report 2016/315 (2016), http://eprint.iacr.org/2016/315
12. Emmadi, N., Gauravaram, P., Narumanchi, H., Syed, H.: Updates on sorting of fully homomorphic encrypted data. In: 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI). pp. 19–24 (Oct 2015)
13. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive 2012, 144 (2012)
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.: Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. IACR Cryptology ePrint Archive 2011, 279 (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. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: CRYPTO. pp. 75–92. Springer (2013)
20. Halevi, S., Shoup, V.: Algorithms in helib. Cryptology ePrint Archive, Report 2014/106 (2014), http://eprint.iacr.org/2014/106
21. Knuth, D.E.: The Art of Computer Programming, Fundamental Algorithms, vol. 1. Addison Wesley Longman Publishing Co., Inc., 3rd edn. (1998), (book)
22. Laine, K., Player, R.: Simple encrypted arithmetic library - seal (v2.0). Tech. rep. (September 2016), https://www.microsoft.com/en-us/research/publication/simple-encrypted-arithmetic-library-seal-v2-0/
23. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
24. Narumanchi, H., Goyal, D., Emmadi, N., Gauravaram, P.: Performance analysis of sorting of fhe data: Integer-wise comparison vs bit-wise comparison
25. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. IACR Cryptology ePrint Archive 2011, 133 (2011)
26. Stehlé, D., Steinfeld, R.: Faster fully homomorphic encryption. Cryptology ePrint Archive 2010/299 (2010)