Efficient Ranking, Order Statistics, and Sorting under CKKS

Federico Mazzone
University of Twente

Maarten Everts
University of Twente
Linksight

Florian Hahn
University of Twente

Andreas Peter
Carl von Ossietzky
Universität Oldenburg

Abstract

Fully Homomorphic Encryption (FHE) enables operations on encrypted data, making it extremely useful for privacy-preserving applications, especially in cloud computing environments. In such contexts, operations like ranking, order statistics, and sorting are fundamental functionalities often required for database queries or as building blocks of larger protocols. However, the high computational overhead and limited native operations of FHE pose significant challenges for an efficient implementation of these tasks. These challenges are exacerbated by the fact that all these functionalities are based on comparing elements, which is a severely expensive operation under encryption.

Previous solutions have typically based their designs on swap-based techniques, where two elements are conditionally swapped based on the results of their comparison. These methods aim to reduce the primary computational bottleneck: the *comparison depth*, which is the number of non-parallelizable homomorphic comparisons in the algorithm. The current state of the art solutions for sorting by Lu et al. (IEEE S&P'21) and Hong et al. (IEEE TIFS 2021), for instance, achieve a comparison depth of $\log^2 N$ and $k\log_k^2 N$, respectively.

In this paper, we address the challenge of reducing the comparison depth by shifting away from the swap-based paradigm. We present solutions for ranking, order statistics, and sorting, that achieve a comparison depth of up to 2 (constant), making our approach highly parallelizable and suitable for hardware acceleration. Leveraging the SIMD capabilities of the CKKS FHE scheme, our approach re-encodes the input vector under encryption to allow for simultaneous comparisons of all elements with each other. The homomorphic re-encoding incurs a minimal computational overhead of $O(\log N)$ rotations. Experimental results show that our approach ranks a 128-element vector in approximately 5.76s, computes its argmin/argmax in 12.83s, and sorts it in 78.64s.

1 Introduction

Fully Homomorphic Encryption (FHE) is a cryptographic primitive that enables performing unbounded operations on encrypted data, without decrypting them first. It is a fundamental building block for designing non-interactive protocols in privacy-preserving applications, and can be used to maintain the confidentiality of data stored in the cloud, while enabling outsourced computations on it. Despite the effort of recent developments to make this technology more efficient and closer to be usable in real-world applications [5–7, 25], computing under FHE is still problematic due to both its serious computational overhead and limited native operations. In particular, it is challenging to realize even fundamental functions efficiently, like *ranking*, computing *order statistics*, and *sorting*, which are frequently required database operations. These functionalities also find applications in diverse fields, for instance in privacy-preserving machine learning, where they can be used to evaluate max-pooling layers and the argmax output layer in neural networks [19, 27], or to perform private inference of decision trees [24, 26].

Several works in the literature have attempted to implement these functionalities efficiently (see Table 1). Many studies have focused on sorting under both the Smart-Vercauteren (SV) scheme [25] and the Cheon-Kim-Kim-Song scheme (CKKS), which is particularly relevant as it enables floating-point arithmetic on vectors of data in a Single Instruction Multiple Data (SIMD) fashion. These approaches typically implement swap-based sorting methods, where at each round two elements are compared and conditionally swapped [8, 9, 16, 18, 22]. Similarly, other works have focused on computing the argmax of a vector of elements encrypted in a CKKS ciphertext, also relying on swap-based techniques [19, 27]. However, comparing two values under encryption is significantly expensive, resulting in the bot-

[1] Precisely, in terms of multiplicative depth, ranking requires up to D_C+4 levels, while the extraction of k-statistics and sorting require up to D_C+D_I+4 and D_C+D_I+6 levels, respectively. Here, D_C and D_I represent the multiplicative depth of the comparison and indicator circuits (see Section 3).

Table 1: Summary of related work.

Paper	Function	Comp. Depth	Comput. Complexity	FHE Scheme	Remarks
Chatterjee et al. [8] (Indocrypt 2013)	Bubble Sort, Insertion Sort	N^2 N^2	$O(N^2)$ $O(N^2)$	SV SV	Tested up to 40 elements, for which it runs in around 359.42 minutes (Bubble Sort) and 362.62 minutes (Insertion Sort).
Chatterjee et al. [9] (IEEE TSC 2017)	Quick Sort	N^2	$O(N^2)$	SV	Tested up to 40 elements, for which it runs in around 779.28 minutes.
Emmadi et al. [16] (ICCCRI 2015)	Bitonic Sort, Odd-Even Merge Sort	$\log^2 N$ $\log^2 N$	$O(N\log^2 N)$ $O(N\log^2 N)$	SV SV	Tested up to 64 elements, for which it runs in around 52.63 minutes (Bitonic Sort) and 42.65 minutes (Odd-Even Merge Sort).
Lu et al. [22] (IEEE S&P 2021)	Bitonic Sort (switching to FHEW)	$\log^2 N$	$O(N\log^2 N)$	CKKS	Tested up to 64 elements, for which it runs in 409.09s in a 4-thread setting (without taking into account the scheme switching cost).
Hong et al. [18] (IEEE TIFS 2021)	k-Way Sorting Networks	$k\log_k^2 N$	$O(Nk\log_k^2 N)$	CKKS	It takes around 87.35 minutes to sort 625 elements.
Jovanovic et al. [19] (ACM CCS 2022)	Argmax	N	$O(N)$	CKKS	It computes the argmax of 128 elements in approximately 92.31 minutes.
Zhang et al. [27] (NDSS 2025)	Argmax	$\log N+1$	$O(\log N)$	CKKS	It computes the argmax of 128 elements in approximately 5.05 minutes.
Our work	Ranking k-Statistics (incl. Argmax) Sorting	**1**[1] **2**[1] **2**[1]	$O(L^2)$ $O(L^2)$ $O(L^2)$ with $L=N/B$ $B\in\{128,256\}$	CKKS CKKS CKKS	Tested up to 16384 elements. It ranks 128 elements in 5.76s, computes their argmin/argmax in 12.83s, and sorts them in 78.64s.

tleneck of these methods being the *comparison depth*. The comparison depth refers to the number of comparisons that must be executed in series, and hence cannot be parallelized. Consequently, **the main problem lies in designing algorithms capable of achieving a low comparison depth**. This task is made particularly challenging by the fact that any swap-based algorithm will have worst case complexity under encryption [16].

In this paper, we design and implement novel algorithms for the aforementioned functionalities which, for the first time, require a constant comparison depth of 2 only. To overcome the limitations of previous solutions, we avoid relying on the strategy of swapping elements under encryption. Our approach heavily exploits the SIMD capabilities of CKKS. The core idea is to use suitable homomorphic rotations and masking to re-encode the encrypted elements in such a way that allows us to compare all elements against each other simultaneously. By aggregating the outcome of this comparison, we compute the rank of each element within the vector. Then, by leveraging appropriate indicator functions, it is possible to use the computed ranks to extract any order statistic and their position, including minimum, maximum, and median (or any percentile). Finally, we show how to parallelize this extraction process to implement a sorting functionality.

By employing only recursive rotation-based operations, we make sure that the re-encoding under encryption requires only $O(\log N)$ rotations, where N is the vector length, thus causing minimal computational overhead. Moreover, the low comparison depth makes our solution highly parallelizable, thus suitable for potential hardware acceleration, such as on GPUs. While we leave this direction to future research, in our present work we show how to further optimize our solution algorithmically in multithreaded environments. Importantly, our approach is agnostic to the specific implementation of the comparison function. Even with a basic implementation, our approach can rank a vector of 128 elements in approximately 5.76s, compute its argmin/argmax in 12.83s, and sort it in 78.64s.

2 Preliminaries

We provide background information and notation regarding CKKS, along with an overview of the homomorphic functionalities upon which our design is constructed.

2.1 CKKS Scheme

CKKS [11] is a fully homomorphic encryption scheme based on the RLWE problem. It works with residual polynomial rings of the form $R_q = \mathbb{Z}_q[x]/(x^n+1)$, where the ring dimension n is a power of two. Messages from $\mathbb{C}^{n/2}$ are encoded into plaintexts, which can embed vectors of up to $n/2$ slots. The scheme operates with floating-point values. The encryption and homomorphic operations (see below) introduce noise on the underlying plaintexts, which is blended with the noise inherent in floating-point arithmetic, making the scheme intrinsically approximate.

CKKS natively supports three homomorphic operations on ciphertexts:

- addition $(X+Y)$ corresponds to the component-wise addition of the underlying plaintext vectors;
- multiplication $(X \cdot Y)$ corresponds to the component-wise multiplication of the underlying plaintext vectors;
- rotation $(X \ll k)$ and $(X \gg k)$ correspond to the left and right rotations of the underlying plaintext vector by a plaintext index k.

The component-wise operations allow processing many inputs concurrently, which makes this encryption scheme suitable for SIMD. Moreover, additions and multiplications can also be performed between a ciphertext and a plaintext. Among all these operations, ciphertext-ciphertext multiplications and rotations are computationally the most expensive.

2.2 Evaluating Non-Polynomial Functions

By combining additions and multiplications it is possible to evaluate any polynomial under encryption. Evaluating non-polynomial functions, such as the comparison operation, is not trivial under CKKS. The usual solution consists of approximating the function by a polynomial. However, a good approximation usually requires a high-degree polynomial, which leads to a deep multiplicative circuit to be evaluated and high computational cost. Thus, this paper focuses on designing algorithms in which the number of non-polynomial evaluations is minimized.

In our design, we use two non-polynomial functions: the comparison function and the indicator function, defined as

$$\mathrm{Cmp}(x,y) = \begin{cases} 1 & \text{if } x > y \\ 0.5 & \text{if } x = y \\ 0 & \text{if } x < y \end{cases}, \tag{1}$$

$$\mathrm{Ind}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}. \tag{2}$$

We approximate both using Chebyshev polynomials, which assure uniform convergence to the original function. The evaluation of the polynomials is then performed using the Paterson-Stockmeyer algorithm adapted to Chebyshev basis [10], which requires only $O(\sqrt{d})$ homomorphic multiplications to evaluate a degree-d polynomial. We denote an approximation of degree d of these functions with $\mathrm{Cmp}(\cdot,\cdot;d)$ and $\mathrm{Ind}_A(\cdot;d)$, respectively. In terms of multiplicative depth, each approximation requires around $\log(d)$ levels.

For assessing our solution against related work, we also implement the comparison function by Cheon et al. [12], which

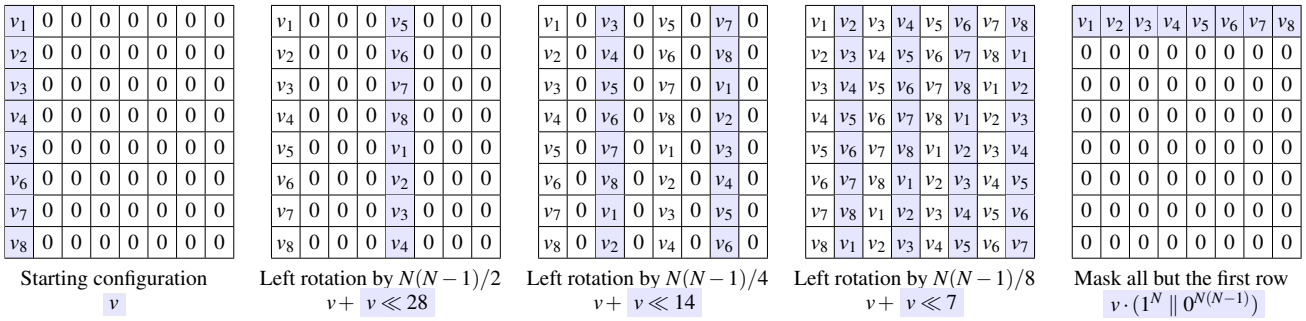


Figure 1: Transposing a vector of length $N = 8$ from column to row (TransC).

is used in the work of Hong et al. [18]. Their implementation is based on the composition of two polynomials

$$f(x) = (35x - 35x^3 + 21x^5 - 5x^7)/2^4$$
$$g(x) = (4589x - 16577x^3 + 25614x^5 - 12860x^7)/2^{10} \quad .$$

In particular, by combining d_f times f and d_g times g, one can get an arbitrarily close approximation of the compare function $\mathrm{Cmp}(x) = (f^{d_f}(g^{d_g}(x-y)) + 1)/2$. While the indicator function for any interval $A = [a,b]$ can be computed as $\mathrm{Ind}_{[a,b]}(x) = \mathrm{Cmp}(x,a)(1 - \mathrm{Cmp}(x,b))$.

Chebyshev usually leads to approximations with low multiplicative depth, while the f,g approach leads to approximations with low evaluation runtime. In general, for a comparison function, the higher the degree of the approximation the better it can correctly compare values that are close to each other. For a discussion on this topic and for different implementations of the comparison function, we refer to [21].

2.3 Matrix Encoding and Operations

Our approach relies on matrices for intermediate computations. To encode a matrix into a ciphertext we adopt the row-by-row approach [20], which consists of concatenating each row into a single vector and then encrypting it. For a square matrix of size N, we have the requirement that $N^2 \leq n/2$, where n is the ring dimension, otherwise multiple ciphertexts are needed to store the entire matrix.

We describe some basic functionalities useful for working with encrypted matrices. These functionalities will be the building blocks of our approach. Given a matrix encoded into a ciphertext X:

- $\mathsf{MaskR}(X,k)$ extracts row k by masking everything else, i.e., setting everything else to zero;
- $\mathsf{SumR}(X)$ sums all the rows together component-wise and stores the result in the first row;
- $\mathsf{ReplR}(X)$ assumes a matrix with only the first row non-zero and replicates it all over by copying its values into the other rows;
- $\mathsf{TransR}(X)$ assumes a square matrix with only the first row non-zero and transposes it (i.e., moving it into the first column).

Similarly for the columns, we have MaskC, SumC, ReplC, TransC. Masking works by multiplying the encrypted matrix by an appropriate plaintext bit mask. For sum and replication there are well-known algorithms in the literature that work recursively, and only require $\log(N)$ rotations, where N is the number of rows/columns of the matrix [17]. For these to work, N must be a power of two, thus the matrix might require padding. We provide their pseudocode in Appendix A. As for transposition, to the best of our knowledge no algorithm that works in $\log(N)$ is available in the literature. Hence, we propose Algorithm 1 and Algorithm 2, which can be of independent interest. Figure 1 shows an example of the TransC algorithm for a generic vector of length $N = 8$.

Algorithm 1 TransR

Input: X encryption of a vector x encoded as a row.
Output: X encryption of the vector x encoded as a column.
1: **for** $i = 1, \ldots, \lceil \log N \rceil$ **do**
2: $\quad X \leftarrow X + (X \gg N(N-1)/2^i)$
3: **end for**
4: $X \leftarrow \mathsf{MaskC}(X, 0)$
5: **return** X

Algorithm 2 TransC

Input: X encryption of a vector x encoded as a column.
Output: X encryption of the vector x encoded as a row.
1: **for** $i = 1, \ldots, \lceil \log N \rceil$ **do**
2: $\quad X \leftarrow X + (X \ll N(N-1)/2^i)$
3: **end for**
4: $X \leftarrow \mathsf{MaskR}(X, 0)$
5: **return** X

3 Our Design for Ranking, Order statistics, and Sorting

The core idea of our design is to manipulate the encrypted vector in such a way that **only a single evaluation of the comparison function is needed to compare all values**. For instance, given a vector $v = (v_1, v_2, v_3)$, we produce

$$v_R = (v_1, v_2, v_3, v_1, v_2, v_3, v_1, v_2, v_3),$$

$$v_C = (v_1, v_1, v_1, v_2, v_2, v_2, v_3, v_3, v_3).$$

The comparison $\mathrm{Cmp}(v_R, v_C)$ contains information about $v_i < v_j$ for all pairs (v_i, v_j). It is easier to visualize this by seeing v_R, v_C as square matrices that have been encoded row-by-row into vectors:

$$v_R = \begin{pmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{pmatrix}, \qquad v_C = \begin{pmatrix} v_1 & v_1 & v_1 \\ v_2 & v_2 & v_2 \\ v_3 & v_3 & v_3 \end{pmatrix}.$$

Turning v into v_R and v_C homomorphically can be done by composing the matrix operations mentioned above, as will be described in detail later. Note that we are implicitly assuming we can fit N^2 elements in a ciphertext, where the vector length N is 3 in the example above. If this assumption does not hold, we require multiple ciphertexts, which we discuss in Section 5.

3.1 Ranking

Ranking associates the elements of a vector to their rank, that is the position they would have if the vector was sorted, starting from 1. In case of ties, we adopt fractional ranking, for which ties receive a rank equal to the average of the ranks they span. For instance, the ranking of $(50, 10, 20, 20, 40)$ is $(5, 1, 2.5, 2.5, 4)$.

Given an input vector v encrypted as V, we think of it as the first row of a null matrix. The encoding v_R is produced by simply applying RepIR, while v_C is produced by first transposing the initial vector to a column with TransR and then replicating it with RepIC. The component-wise comparison of $v_R > v_C$ is ideally a matrix with values in $\{0, 0.5, 1\}$, whose each column j contains information about the position of v_j in the sorted array:

- a number of ones equal to the number of elements smaller than v_j, and
- a number of 0.5 equal to the number of elements equal to v_j.

Thus, summing the elements in the column (and an additional 0.5) gives the fractional rank of v_j. A pseudocode is provided in Algorithm 3, while a schematic with an example can be found in Figure 2.

Algorithm 3 Rank

Input: V encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$, approximation degree $d \in \mathbb{N}$.
Output: R encryption of a vector in $\mathbb{R}^N$ representing the (fractional) ranking of v.

1: $V_R \leftarrow \mathsf{RepIR}(V)$
2: $V_C \leftarrow \mathsf{RepIC}(\mathsf{TransR}(V))$
3: $C \leftarrow \mathrm{Cmp}(V_R, V_C; d)$
4: $R \leftarrow \mathsf{SumR}(C) + (0.5, \ldots, 0.5)$
5: **return** R

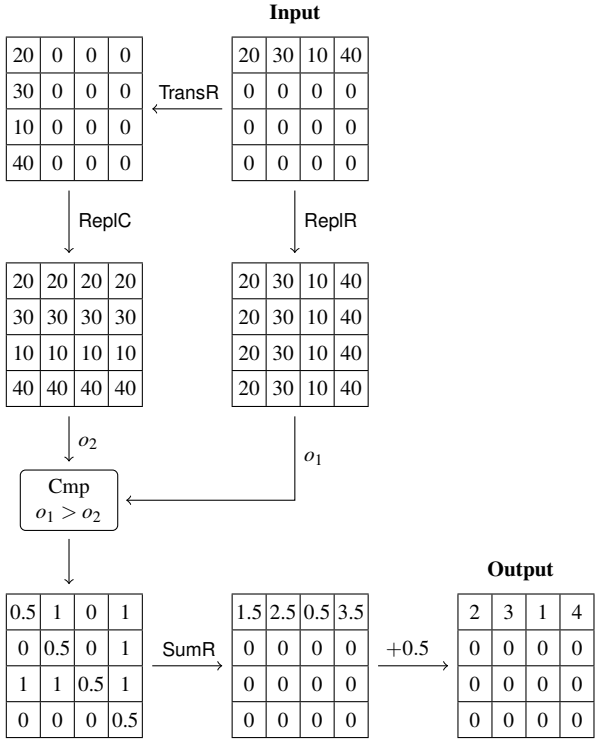


Figure 2: Schematic example of ranking a 4-element vector.

We assume the vector size is a power of 2 to work with recursive operations on matrices. If it is not, we can pad it to the next power of 2 and perform an additional masking after the comparison to remove the excess information.

The cost of Algorithm 3 is $4\lceil \log N \rceil$ rotations, which can be reduced to $3\lceil \log N \rceil$ by parallelizing RepIR and RepIC, and $\sqrt{d}$ ciphertext-ciphertext multiplications, with a multiplicative depth of $\sim \lceil \log d \rceil$.

Correctness Proof Assuming the correctness of the building blocks and the ideal functionality of the algorithm, namely no noise from the scheme and no approximation error for the comparison function, we prove that the output R produced by Algorithm 3 on input the encryption of a vector $v = (v_1, \ldots, v_N)$ is actually the encryption of the fractional

ranking of v. Let $r = (r_1, \ldots, r_N)$ be the decryption of R. Let $j \in \{1, \ldots, N\}$, we prove that r_j is the rank of v_j. Let v_R and v_C be the decryption as matrices of V_R and V_C, respectively. By the correctness of RepIR, $v_{R;i,j} = v_j$ for all $i \in \{1, \ldots, N\}$, where $v_{R;i,j}$ is the element at row i and column j of v_R. Similarly, by the correctness of RepIC and TransR, $v_{C;i,j} = v_i$ for all $i \in \{1, \ldots, N\}$. Let c be the decryption as matrix of C, then $c_{i,j} = \mathrm{Cmp}(v_j, v_i)$, where Cmp is defined in Equation 1. Then $r_j = \sum_{i=1}^{N} c_{i,j} + 0.5$. We split the sum over the following partition of $\{1, \ldots, N\}$:

$$\begin{aligned}
\mathcal{L}_j &:= \{i \in \{1, \ldots, N\} : v_i < v_j\} \\
\mathcal{E}_j &:= \{i \in \{1, \ldots, N\} : v_i = v_j\} \\
\mathcal{G}_j &:= \{i \in \{1, \ldots, N\} : v_i > v_j\}
\end{aligned}$$

and get

$$\begin{aligned}
r_j &= \sum_{\mathcal{L}_j} c_{i,j} + \sum_{\mathcal{E}_j} c_{i,j} + \sum_{\mathcal{G}_j} c_{i,j} + 0.5 \\
&= \sum_{\mathcal{L}_j} 1 + \sum_{\mathcal{E}_j} 0.5 + \sum_{\mathcal{G}_j} 0 + 0.5 \\
&= |\mathcal{L}_j| + 0.5 \cdot |\mathcal{E}_j| + 0.5.
\end{aligned}$$

The elements equal to v_j span a rank from $|\mathcal{L}_j| + 1$ to $|\mathcal{L}_j| + |\mathcal{E}_j|$, thus the fractional rank of v_j is their average, namely

$$\begin{aligned}
\frac{1}{|\mathcal{E}_j|} \sum_{k=1}^{|\mathcal{E}_j|} (|\mathcal{L}_j| + k) &= \frac{1}{|\mathcal{E}_j|} \left(|\mathcal{E}_j| \cdot |\mathcal{L}_j| + 0.5 \cdot |\mathcal{E}_j| \cdot (|\mathcal{E}_j| + 1)\right) \\
&= |\mathcal{L}_j| + 0.5 \cdot (|\mathcal{E}_j| + 1) = r_j. \qquad \square
\end{aligned}$$

3.2 Order Statistics

The k-th order statistic (or k-statistic) of a vector is its k-th smallest value, that is the value of rank k if such rank exists. A value of given rank might not exist if there are ties in the vector. Here, we will show how to work around this issue in the specific case of the first and last order statistics (i.e., min and max). While we will provide a general-purpose solution in Section 4.

We can determine the k-th order statistic of a vector v by first computing its ranking and then applying to it an indicator function "around k", namely for the interval $[k - 0.5, k + 0.5]$. It will output a bit mask whose ones correspond to the positions of the elements with rank k, if they exist (see Algorithm 4). One can then retrieve the actual value of the statistic by computing the inner product $\langle O, V \rangle = \mathsf{SumC}(O \cdot V)$ and dividing it by the L1 norm of the mask $\mathsf{SumC}(O)$. This can be done under encryption by approximating $1/x$ in the range $[0.5, N + 0.5]$, or by using Goldschmidt's division algorithm [13]. The outcome will be zero if no element of rank k exists.

Correctness Proof Assuming the correctness of the building blocks and Algorithm 3, and the ideal functionality of the

Algorithm 4 Order Statistic

Input: V encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$, approximation degrees $d_C, d_I \in \mathbb{N}$, index $k \in \{1, \ldots, N\}$.
Output: O encryption of a Boolean vector in $\{0,1\}^N$ that has value 1 in position i if and only if v_i has rank k.

1: $R \leftarrow \mathsf{Rank}(V; d_C)$
2: $O \leftarrow \mathrm{Ind}_k(R; d_I)$
3: **return** O

algorithm, we prove that Algorithm 4 is correct. The input is the encryption of a vector $v = (v_1, \ldots, v_N)$, together with an index $k \in \mathbb{N}$. Let r, o be the decryption of R, O respectively. For $i \in \{1, \ldots, N\}$,

$$o_i = \mathrm{Ind}_k(r_i) = \begin{cases} 1 & \text{if } \mathsf{Rank}(v_i) = k \\ 0 & \text{if } \mathsf{Rank}(v_i) \neq k \end{cases}. \qquad \square$$

Min and Max We can ensure that we can always compute minimum and maximum (first and last order statistic) by employing a slightly different definition of the comparison function. By using

$$\mathrm{Cmp}_G(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \leq y \end{cases}$$

all the minimal elements are assigned to the first rank, thus the minimum can be retrieved with Ind_1. Similarly, by using

$$\mathrm{Cmp}_{GE}(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

all the maximal elements are assigned to the last rank, thus the maximum can be retrieved with Ind_N.

3.3 Sorting

We sort a vector by extracting all its order statistics simultaneously, parallelizing the strategy presented in Algorithm 4. For now, we assume that all elements in the vector are distinct, ensuring that there is exactly one element for each rank $k \in \{1, \ldots, N\}$, and allowing us to extract any order statistic. We will show how to remove this assumption in Section 4.

To extract all the order statistics in one go, the idea is to compute the ranking of v, replicate it over the rows, and extract a different k-statistic for each row k. Normally, this would require applying N different indicator functions. To avoid this, we shift each row's domain by subtracting the entire row by $(k, \ldots, k)$. Applying an indicator function around zero then produces a one-hot encoding of the position of the k-statistic for each row k. Next, we perform an inner-product with the original vector: we replicate the vector over the rows, multiply it by the previously-computed mask, and sum the result over the columns. This way, the first element of each row k contains

Algorithm 5 Sorting

Input: V encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$ with distinct elements, approximation degrees $d_C, d_I \in \mathbb{N}$.
Output: S encryption of the sorted form of v.

1: $R \leftarrow \mathsf{Rank}(V; d_C)$
2: $R_R \leftarrow \mathsf{RepIR}(R)$
3: $M \leftarrow \mathrm{Ind}_0(R_R - (1^N \parallel \cdots \parallel N^N); d_I)$
4: $V_R \leftarrow \mathsf{RepIR}(V)$
5: $S \leftarrow \mathsf{TransC}(\mathsf{SumC}(M \cdot V_R))$
6: **return** S

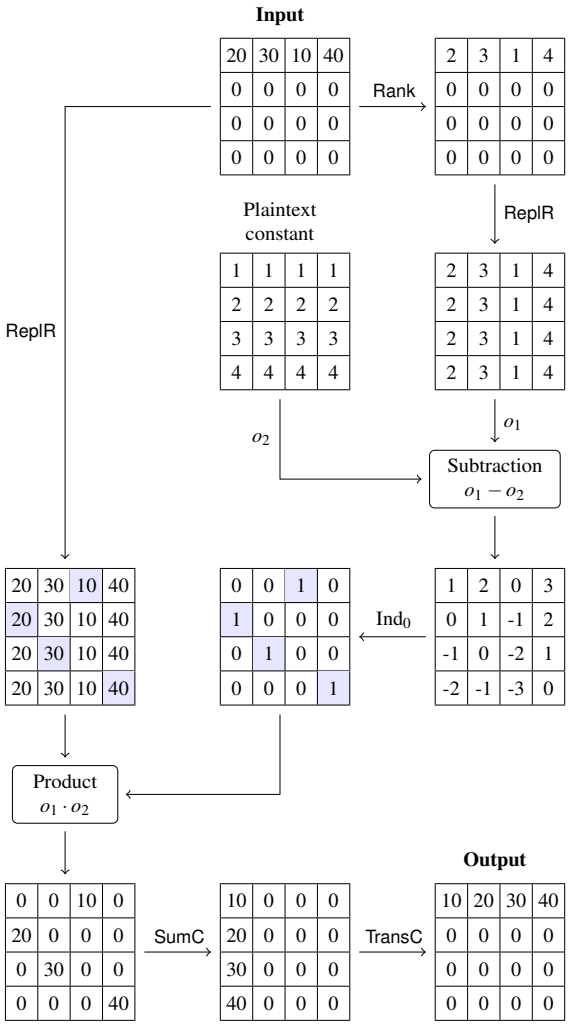


Figure 3: Schematic example of sorting a 4-element vector.

the corresponding k-statistic of v. The pseudocode is provided in Algorithm 5, while a schematic with an example can be found in Figure 3.

Note that the quantity $\mathsf{RepIR}(V)$ is already computed within the ranking algorithm, thus the extra cost is given by only $3\lceil \log N \rceil$ rotations, the evaluation of the indicator function, and the multiplication $M \cdot V_R$. As an additional optimization, we can avoid the final transposition and save $\lceil \log N \rceil$ rotations with a slight tweak in the ranking algorithm. By inverting the order of the operands in the Cmp and replacing the SumR with a SumC, the output of the ranking algorithm will be in column-form instead of row-form. This entails swapping row- and column-operations in the sorting algorithm, and replacing $\mathsf{RepIR}(V)$ with $\mathsf{RepIC}(\mathsf{TransR}(V))$, which is also a quantity computed within the ranking algorithm. The (parallel) cost of the sorting algorithm is then:

- $5\lceil \log N \rceil$ rotations, and
- $\sqrt{d_C} + \sqrt{d_I} + 1$ ciphertext-ciphertext multiplications, with a multiplicative depth of $\sim \lceil \log d_C \rceil + \lceil \log d_I \rceil + 1$.

Correctness Proof Assuming the correctness of the building blocks and Algorithm 3, and the ideal functionality of the algorithm, we prove that the output S produced by Algorithm 5 on input the encryption of a vector with distinct elements $v = (v_1, \ldots, v_N)$ is actually the encryption of the sorted form of v. Let r_R, m, v_R, s be the decryption of R_R, M, V_R, S respectively. As the elements v_i are all distinct, the ranking of v is a permutation of $\{1, \ldots, N\}$. Let $i \in \{1, \ldots, N\}$, we thus have to prove that $\mathsf{Rank}(s_i) = i$. By the correctness of RepIR:

$$m_{i,j} = \mathrm{Ind}_0(r_{R;i,j} - i) = \mathrm{Ind}_0(r_j - i) = \begin{cases} 1 & \text{if } r_j = i \\ 0 & \text{if } r_j \neq i \end{cases}$$

and by the correctness of SumC, TransC, and RepIR, we have

$$s_i = \sum_{j=1}^{N} m_{i,j} \cdot v_{R;i,j} = \sum_{j=1}^{N} m_{i,j} \cdot v_j.$$

Since r is a permutation of $\{1, \ldots, N\}$, there exists one and only one index k such that $\mathsf{Rank}(v_k) = i$. Hence, $s_i = v_k$ and $\mathsf{Rank}(s_i) = i$. □

4 Tie-Correction Offset

If two or more elements are in a *tie*, namely share the same value, they receive the same rank. As noted in Section 3, this causes the ranking function to become non-surjective, which hinders the extraction of certain order statistics and, consequently, prevents a correct sorting of the input vector. For example, the (fractional) ranking of the input vector $v = [10, 20, 20, 40]$ is $r = [1, 2.5, 2.5, 4]$. If we now want to extract the second or third order statistics (which should both correspond to the value 20), we should apply an indicator around rank 2 and 3, respectively, which would miss the actual rank value 2.5. To fix this issue, we build an offset vector that redistributes the fractional ranking of all elements in a tie over the ranks they span. In our example, the offset vector

would be $f = [0, -0.5, 0.5, 0]$, which corrects the fractional ranking to

$$r + f = [1, 2, 3, 4]$$

allowing us to correctly extract all four order statistics. As follows, we explain how to build this tie-correction offset vector under encryption with small computational overhead.

To build this offset, we need to evaluate the equality operator (Eq) among all pairs of elements in the input vector. Similarly to Cmp, the output of this function is a square matrix e of size $N \times N$ such that $e_{i,j} = 1$ if $v_i = v_j$, and 0 otherwise. The equality can be evaluated as an indicator function around 0, which can be done in parallel to the greater-than Cmp in the ranking. However, we note that the information needed to compute the equality matrix e is already contained in the comparison matrix c. We can reuse it to compute the equality:

$$e = 4 \cdot c \cdot (1 - c)$$

mapping both zeros and ones of c to 0, and the values 0.5 to 1. In our implementation we mainly use the latter option, which comes with an overhead of just two multiplications.

Note that each column j contains a number of ones equal to the number of elements that are in a tie with v_j. This includes the trivial equality $v_j = v_j$ on the main diagonal. By masking out the lower triangle of e and summing over the rows, we count the non-trivial identities only once, namely

$$u_j = |\{i \in \{1, \ldots, j\} : v_i = v_j\}| \ .$$

For instance, if the first four elements of v are in a tie, then the corresponding values in u are $1, 2, 3, 4$. We can use u to offset the ranking. But, since we are using fractional ranking, we first need to shift it by half of the tie size, that is the range the elements in the tie span. To do this, we sum directly over the rows of e, without masking it, and get

$$t_j = |\{i \in \{1, \ldots, N\} : v_i = v_j\}| \ .$$

Now, the correction offset for the ranking can be computed as

$$f_j = u_j - 0.5 \cdot t_j - 0.5$$

where the last -0.5 makes the offset start from zero, and it nicely cancels out with the $+0.5$ in the last line of the ranking algorithm. The pseudocode to compute the tie-correction offset is presented in Algorithm 6. This runs at the end of the ranking algorithm (Algorithm 3), and the offset can just be added to the fractional ranking to make it suitable for order statistics extraction and sorting.

As a particular case, we can modify Algorithm 4 to compute the *median* by extracting the $(N+1)/2$ statistic if N is odd, or both the $N/2$ and $(N/2)+1$ statistics if N is even. In the latter case, an additional plaintext multiplication by 0.5 is needed after the inner product. In a similar way, one can compute any percentile of the given vector.

Algorithm 6 Tie-Correction Offset

Input: V encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$.
Output: F encryption of a vector in $\mathbb{R}^N$ representing the tie-correction offset vector of v.

1: $E \leftarrow \mathrm{Eq}(V)$
2: $\mathrm{mask} \leftarrow \delta_{j \geq i}$
3: $U \leftarrow \mathsf{SumR}(E \cdot \mathrm{mask})$
4: $T \leftarrow \mathsf{SumR}(E)$
5: $F \leftarrow U - 0.5 \cdot T - (0.5, \ldots, 0.5)$
6: **return** F

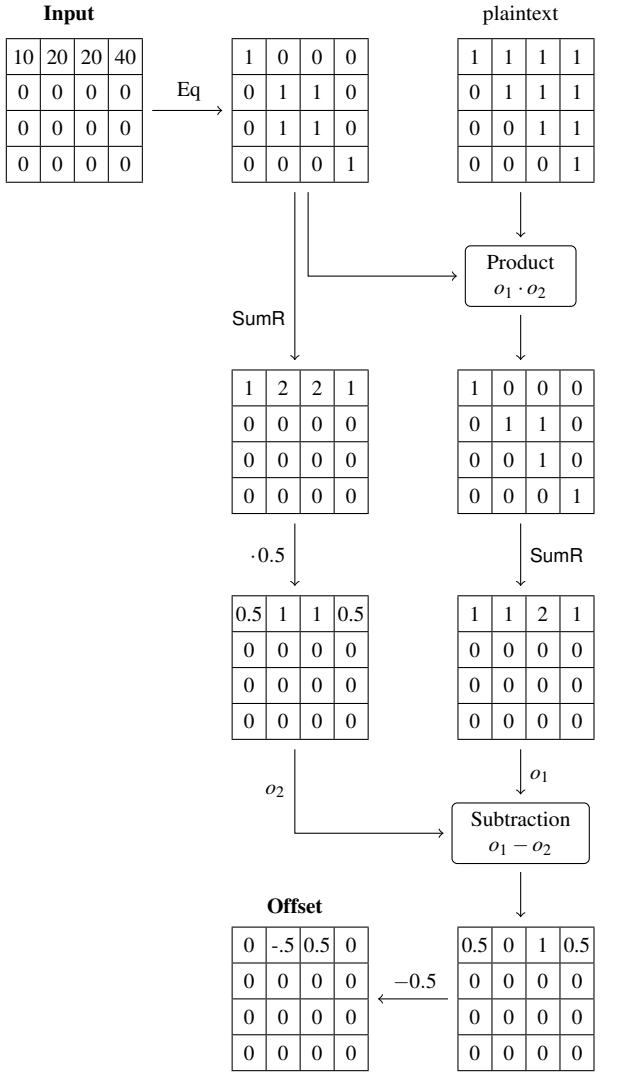


Figure 4: Schematic example of computing the tie-correction offset for a 4-element vector.

Correctness Proof Given a vector $v \in \mathbb{R}^N$, let V be its encryption. Let R and F be the output of Algorithm 3 and Algorithm 6 on input V, respectively. And let r and f be the corresponding decryption. Assuming the correctness of the

building blocks and Algorithm 3, and the ideal functionality of the algorithms, we prove that (1) $k := r + f$ is a permutation of $(1,\dots,N)$, and (2) v_j is the k_j-th order statistic of v, for all $j \in \{1,\dots,N\}$. Let e, u, and t be the decryption of the intermediate computations E, U, and T in Algorithm 6, respectively. Let $j \in \{1,\dots,N\}$, and let us define the following subsets of $\{1,\dots,N\}$:

$$\begin{aligned}\mathcal{L}_j &:= \{i \in \{1,\dots,N\} : v_i < v_j\}\\ \mathcal{E}_j &:= \{i \in \{1,\dots,N\} : v_i = v_j\}\\ \mathcal{U}_j &:= \{i \in \{1,\dots,j\} : v_i = v_j\} \ .\end{aligned}$$

By the correctness of SumR, we have that

$$\begin{aligned}u_j &= \sum_{i=1}^{N} e_{ij}\delta_{j\geq i} = |\mathcal{U}_j|\\ t_j &= \sum_{i=1}^{N} e_{ij} = |\mathcal{E}_j|\end{aligned}$$

and thus

$$\begin{aligned}f_j &= u_j - 0.5\cdot t_j - 0.5\\ &= |\mathcal{U}_j| - 0.5\cdot|\mathcal{E}_j| - 0.5 \ .\end{aligned}$$

On the other hand, by the correctness of Algorithm 3, we know that

$$r_j = |\mathcal{L}_j| + 0.5\cdot(|\mathcal{E}_j|+1) \ .$$

Combining these two identities we get

$$\begin{aligned}k_j &:= r_j + f_j\\ &= |\mathcal{U}_j| - 0.5\cdot|\mathcal{E}_j| - 0.5 + |\mathcal{L}_j| + 0.5\cdot(|\mathcal{E}_j|+1)\\ &= |\mathcal{U}_j| + |\mathcal{L}_j| \ .\end{aligned}$$

Let w be the sorted array, then we note that $w_i = v_j$ for all $i \in \{|\mathcal{L}_j|+1,\dots,|\mathcal{L}_j|+|\mathcal{E}_j|\}$. Since $|\mathcal{U}_j| \geq 1$ (as it contains at least the trivial identity), and $|\mathcal{U}_j| \leq |\mathcal{E}_j|$ (as $\mathcal{U}_j \subseteq \mathcal{E}_j$), we have that $|\mathcal{L}_j|+1 \leq k \leq |\mathcal{L}_j|+|\mathcal{E}_j|$. Hence, $w_{k_j} = v_j$, proving point (2).

Now, we prove that k is a permutation of $(1,\dots,N)$. Let $j \in \{1,\dots,N\}$, then

- $k_j \in \mathbb{N}$: this is trivial, since both $|\mathcal{U}_j|, |\mathcal{L}_j| \in \mathbb{N}$;
- $k_j \geq 1$: since $v_j = v_j$, we have that $k_j \geq |\mathcal{U}_j| \geq 1$;
- $k_j \leq N$: this is true since $\mathcal{U}_j$ and $\mathcal{L}_j$ are non-overlapping subsets of $\{1,\dots,N\}$;
- for all $j' \neq j$, $k_{j'} \neq k_j$: we consider three cases
  1. if $v_j = v_{j'}$, then $\mathcal{L}_j = \mathcal{L}_{j'}$; without loss of generality, we assume $j' > j$, thus $|\mathcal{U}_{j'}| > |\mathcal{U}_j|$, hence $k_{j'} > k_j$;
  2. if $v_j < v_{j'}$, then $\mathcal{L}_j \cup \mathcal{U}_j \subseteq \mathcal{L}_j \cup \mathcal{E}_j \subseteq \mathcal{L}_{j'}$, thus $k_j = |\mathcal{L}_j| + |\mathcal{U}_j| < |\mathcal{L}_{j'}| + 1 \leq k_{j'}$;
  3. if $v_j > v_{j'}$, the proof is symmetric to the previous case.

This proves point (1). □

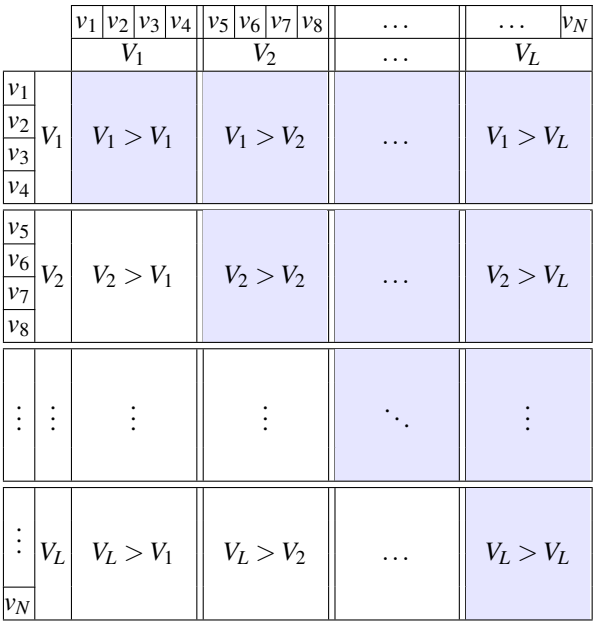


Figure 5: Comparison in multi-ciphertext mode. The vector v is split into blocks of size $B = 4$. The upper triangle contains information about the comparison between all pairs $v_i > v_j$.

5 Multiple Ciphertexts Encoding

When a vector is too long and does not fit in a ciphertext we can split it into multiple blocks. This happens when $N^2 > n/2$, where n is the ring dimension and N is the vector length. In this case, we divide the vector into L blocks $V_1,\dots,V_L$ of size $B = 2^{\lfloor \log\sqrt{n/2} \rfloor}$, which can be done under encryption by suitable rotations and masking. When it comes to comparisons, we also have to consider the comparisons between different blocks, that is, computing

$$C_{i,j} = (V_i > V_j) := \mathrm{Cmp}(\mathsf{ReplR}(V_i), \mathsf{ReplC}(\mathsf{TransR}(V_j)))$$

for all $i, j \in \{1,\dots,L\}$. The results can then be aggregated row-wise and block-wise to compute the ranking

$$R_i = \sum_{j=1}^{L} \mathsf{SumR}(C_{i,j}) + 0.5$$

for $i \in \{1,\dots,L\}$. Note that we can also compute the block-sum first, as $\sum_{j=1}^{L} \mathsf{SumR}(C_{i,j}) = \mathsf{SumR}(\sum_{j=1}^{L} C_{i,j})$, allowing for evaluating SumR only once per block. The split output $R_1,\dots,R_L$ can then be merged back into a single ciphertext if needed and if it fits.

The total number of comparisons is L^2, but they can all be computed in parallel, making this approach suitable for a multi-threaded environment. To reduce the computational burden, we notice that not all the comparisons $C_{i,j}$ are actually needed, as the information in $C_{i,j}$ is already contained in $C_{j,i}$ for all i, j (see Figure 5). In particular, we have that

$$C_{i,j} = (1 - C_{j,i})^{\top}. \tag{3}$$

Algorithm 7 Multi-Ciphertext Ranking

Input: $V_1, \ldots, V_L$ multi-ciphertext encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$, approximation degree $d \in \mathbb{N}$.
Output: $R_1, \ldots, R_L$ multi-ciphertext encryption of a vector in $\mathbb{R}^N$ representing the (fractional) ranking of v.

parallel for $i = 1, \ldots, L$ **do**
  $V_{R;i} \leftarrow \mathsf{RepIR}(V_i)$
  $V_{C;i} \leftarrow \mathsf{RepIC}(\mathsf{TransR}(V_i))$
end for
parallel for $i = 1, \ldots, L$ **do**
  parallel for $j = i, \ldots, L$ **do**
    $C_{i,j} \leftarrow \mathrm{Cmp}(V_{R;i}, V_{C;j}; d)$
  end for
end for
parallel for $i = 1, \ldots, L$ **do**
  $R_i \leftarrow \mathsf{TransC}(\mathsf{SumC}(\sum_{j=1}^{i-1}(1 - C_{j,i}))) + \mathsf{SumR}(\sum_{j=i}^{L} C_{i,j}) + (0.5, \ldots, 0.5)$
end for
return $R_1, \ldots, R_L$

Proof of Equation 3 For the sake of notation, let $A := C_{i,j}$ and $B := C_{j,i}$. Then $A_{m,n} = \mathrm{Cmp}(V_{i;n}, V_{j;m})$, where $V_{x;y}$ denotes the y-th element of block x. On the other hand, $B_{n,m} = \mathrm{Cmp}(V_{j;m}, V_{i;n}) = 1 - \mathrm{Cmp}(V_{i;n}, V_{j;m})$. □

Hence, we compute $C_{i,j}$ only for $j \geq i$ and use Equation 3 for $j < i$. To avoid the transposition of $1 - C_{j,i}$ as a whole matrix, which is expensive, we operate on it column-wise, and only transpose it in the end, after summing it up to a vector:

$$R_i = \mathsf{TransC}\Big(\mathsf{SumC}\Big(\sum_{j=1}^{i-1}(1 - C_{j,i})\Big)\Big) + \mathsf{SumR}\Big(\sum_{j=i}^{L} C_{i,j}\Big).$$

This optimization makes us save $L(L-1)/2$ comparisons. Algorithm 7 describes the full pseudocode for multi-ciphertext ranking. The correctness can be proven similarly as for Algorithm 3, by exploiting Equation 3. The algorithm in case of ranking with tie-correction is similar. We omit its description in the multi-ciphertext pseudocode for the sake of clarity.

We proceed on the same line to adapt sorting to the multi-ciphertext setting. First, a multi-ciphertext ranking is computed. As we have to extract N order statistics, and each one could be in any of the ciphertext blocks, the ranking blocks are replicated both row-wise and block-wise. Then each row of each block is shifted by a constant going from 1 to N, as in Algorithm 5, although this time it spans over multiple instances of the same ranking block. We conclude by applying the indicator function to each instance of each ranking block and summing up the results, both row- and block-wise. A detailed description is presented in Algorithm 8.

Algorithm 8 Multi-Ciphertext Sorting

Input: $V_1, \ldots, V_L$ multi-ciphertext encryption of $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$ with distinct elements, approximation degrees $d_C, d_I \in \mathbb{N}$.
Output: $S_1, \ldots, S_L$ multi-ciphertext encryption of the sorted form of v.

$R_1, \ldots, R_L \leftarrow \mathsf{Rank}(V_1, \ldots, V_L; d_C)$
parallel for $i = 1, \ldots, L$ **do**
  $R_{R;i} \leftarrow \mathsf{RepIR}(R_i)$
end for
parallel for $i = 1, \ldots, L$ **do**
  $V_{R;i} \leftarrow \mathsf{RepIR}(V_i)$
  $S_i \leftarrow 0$
  parallel for $j = 1, \ldots, L$ **do**
    $M_{i,j} \leftarrow \mathrm{Ind}_0(R_{R;j} - ((B(i-1)+1)^N \parallel \cdots \parallel (Bi)^N); d_I)$
    $S_i \leftarrow S_i + M_{i,j} \cdot V_{R;j}$
  end for
  $S_i \leftarrow \mathsf{TransC}(\mathsf{SumC}(S_i))$
end for
return $S_1, \ldots, S_L$

6 Experimental Evaluation

We evaluate the performance of our designs for different vector sizes and compare them with existing work.

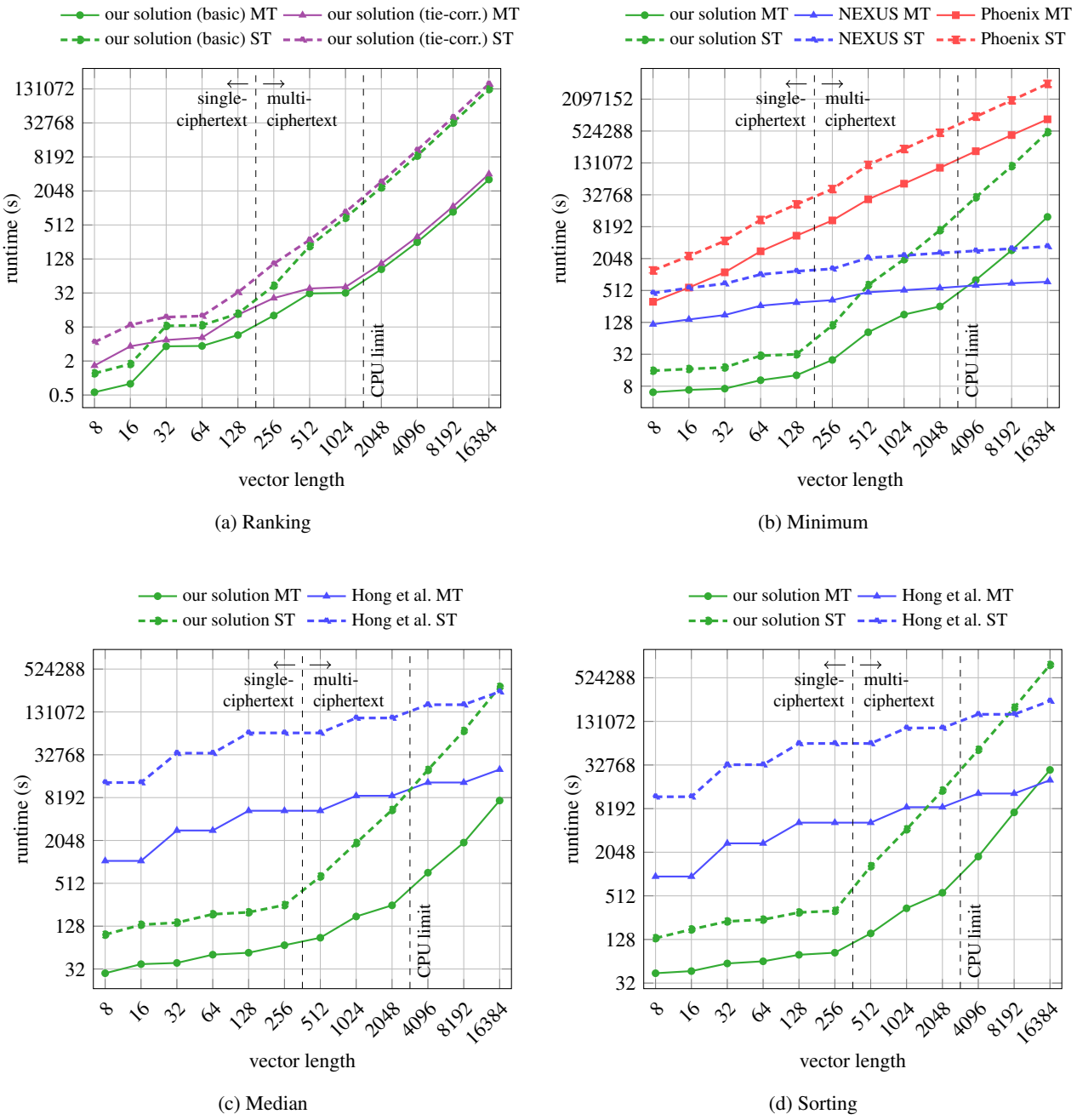
6.1 Experimental Setup

We use the CKKS implementation provided by the OpenFHE library [1],[2] with a scaling factor (decimal precision) ranging from 30 to 59 bits. The ring dimension goes up to 2^{16} for ranking, and 2^{17} for order statistics and sorting, to accommodate the higher multiplicative depth. The parameters are chosen in accordance with the Homomorphic Encryption Standard to assure 128-bit security [2,3]. Our code is available at https://github.com/FedericoMazzone/openfhe-statistics.

We present the runtime of ranking, computing the minimum, median, and sorting elements that are generated uniformly at random in a bounded interval. For ranking and minimum, we use Chebyshev approximation of the comparison function up to degree 2^{11} for $N \leq 256$, while we employ the f, g approximation by Cheon et al. [12] for $N > 256$, as we start benefiting from the lower runtime. The composition degrees used are $d_f = 2$ and $d_g = 3$, which are the same we adopt in the median and sorting experiments for any N.

As the depth of our circuit is upper-bounded by 65, bootstrapping is not needed, and the scheme is used as a leveled homomorphic encryption. All the experiments are performed on a Linux machine with Intel Xeon Platinum 8358 running at 2.60 GHz, with 32 cores (64 threads), and 512 GB RAM.

[2] https://github.com/openfheorg/openfhe-development



(a) Ranking

(b) Minimum

(c) Median

(d) Sorting

Figure 6: Runtime of ranking, minimum, median, and sorting for increasing vector size. Related work's performance is reported as baseline: Phoenix [19] and NEXUS [27] for minimum, and Hong et al. [18] for median and sorting. All solutions are assessed both in single-threaded (ST, dashed lines) and multi-threaded (MT, solid lines) settings. Both axes are in logarithmic scale.

6.2 Empirical Results

In Figure 6, we report the runtime performance of our solution for the different functionalities described in Section 3, both in single-threaded (ST) and multi-threaded (MT) settings, with a maximum of 64 concurrent threads.

Our approach demonstrates particularly fast performance for small-sized input vectors that can be processed within a single ciphertext. For example, computing the minimum of a vector takes only 15.65s for $N = 8$, and up to 31.95s for $N = 128$, in ST. However, as soon as the input vector needs to be split into multiple chunks — of 128 elements for ranking and

minimum, and of 256 elements for median and sorting — our solution exhibits a steep increase in runtime. The runtime for the minimum computation jumps to 112.61s for 2 chunks (256 elements), 648.55s for 4 chunks (512 elements), 1994.30s for 8 chunks (1024 elements), and so on.

This slow-down due to switching from single- to multi-ciphertext mode can be observed in all four functionalities. The primary cause is the increased number of comparisons required in multi-ciphertext mode. This effect is particularly noticeable in the ST setting, where comparisons are evaluated sequentially, making the quadratic growth of our solution's cost evident in the runtime. In the MT setting, the slow-down becomes more pronounced once we exceed the available CPU threads (CPU limit in Figure 6) and can no longer parallelize the comparison evaluations. Specifically, this occurs when the number of blocks increases from 8 to 16, causing the number of comparisons to grow from 36 to 136, which exceeds the 64-thread capacity of our machine. Beyond this point, comparisons begin to execute sequentially, which has a significant impact on the runtime of our solution.

In Figure 6a we can see the computational overhead introduced by the tie-correction offset in the ranking algorithm. In the MT setting, ranking takes 0.56s for $N = 8$ and 12.77s for $N = 256$, while the tie-correction increases these runtimes to 1.66s and 25.97s, respectively. The overhead varies within this range, peaking at 13.2s, while it stabilizes at around 25–27% for $N > 256$. Note that this overhead is not only due to the extra operations required to compute the correction offset, but also to the increased multiplicative depth, which makes all operations computationally more expensive.

In Figure 6c and Figure 6d we report the runtime for the median and sorting algorithms, respectively. We will discuss them in more detail in Section 6.3. For the moment, we only highlight that the median serves as a representative test for order statistic extraction with tie-correction enabled. The cost of computing any other order statistic is identical, as it only changes the interval of the indicator function.

Finally, Table 2 shows that memory consumption grows

Table 2: Memory consumption comparison of different solutions, both in single-thread (ST) and multi-thread (MT).

(a) Ranking

	Our Solution (basic)		Our Solution (tie corr.)	
N	ST	MT	ST	MT
8	191 MB	191 MB	514 MB	508 MB
16	271 MB	259 MB	739 MB	748 MB
32	765 MB	781 MB	952 MB	961 MB
64	881 MB	891 MB	1.06 GB	1.07 GB
128	1.05 GB	1.06 GB	1.42 GB	1.43 GB
256	1.15 GB	1.58 GB	1.54 GB	2.21 GB
512	4.08 GB	6.77 GB	4.67 GB	7.63 GB
1024	4.43 GB	13.3 GB	5.13 GB	15.4 GB
2048	5.09 GB	21.0 GB	6.04 GB	24.4 GB
4096	6.55 GB	22.4 GB	8.01 GB	26.6 GB
8194	9.95 GB	25.1 GB	12.6 GB	30.9 GB
16384	18.8 GB	30.8 GB	25.0 GB	40.2 GB

(b) Minimum

	Our Solution		NEXUS [27]	
N	ST	MT	ST	MT
8	1.34 GB	1.32 GB	9.65 GB	9.68 GB
16	1.61 GB	1.64 GB	11.0 GB	11.2 GB
32	1.91 GB	1.93 GB	13.7 GB	14.6 GB
64	2.60 GB	2.62 GB	15.0 GB	15.6 GB
128	2.93 GB	2.95 GB	15.8 GB	15.9 GB
256	3.47 GB	4.94 GB	18.6 GB	19.8 GB
512	18.9 GB	23.2 GB	34.1 GB	36.3 GB
1024	20.0 GB	39.0 GB	35.3 GB	37.7 GB
2048	22.6 GB	99.6 GB	37.7 GB	39.4 GB
4096	30.4 GB	113 GB	60.2 GB	64.3 GB
8194	34.4 GB	126 GB	60.9 GB	65.0 GB
16384	46.4 GB	165 GB	62.2 GB	66.3 GB

(c) Median

	Our Solution		Hong et al. [18]	
N	ST	MT	ST	MT
8	5.73 GB	5.86 GB	12.9 GB	37.9 GB
16	7.89 GB	8.02 GB	12.9 GB	37.9 GB
32	9.42 GB	9.52 GB	12.9 GB	37.9 GB
64	12.2 GB	12.3 GB	12.9 GB	37.9 GB
128	13.9 GB	14.1 GB	12.9 GB	37.9 GB
256	16.9 GB	17.2 GB	12.9 GB	37.9 GB
512	18.7 GB	23.0 GB	12.9 GB	37.9 GB
1024	21.9 GB	35.2 GB	12.9 GB	37.9 GB
2048	25.6 GB	65.1 GB	12.9 GB	37.9 GB
4096	29.8 GB	111 GB	12.9 GB	37.9 GB
8194	34.6 GB	129 GB	12.9 GB	37.9 GB
16384	40.0 GB	192 GB	12.9 GB	37.9 GB

(d) Sorting

	Our Solution		Hong et al. [18]	
N	ST	MT	ST	MT
8	7.22 GB	7.34 GB	10.9 GB	35.9 GB
16	8.95 GB	9.03 GB	10.9 GB	35.9 GB
32	11.6 GB	11.7 GB	10.9 GB	35.9 GB
64	13.5 GB	13.7 GB	10.9 GB	35.9 GB
128	16.8 GB	17.0 GB	10.9 GB	35.9 GB
256	18.8 GB	19.0 GB	10.9 GB	35.9 GB
512	21.7 GB	28.0 GB	10.9 GB	35.9 GB
1024	23.2 GB	42.2 GB	10.9 GB	35.9 GB
2048	26.9 GB	97.0 GB	10.9 GB	35.9 GB
4096	31.2 GB	116 GB	10.9 GB	35.9 GB
8194	36.0 GB	155 GB	10.9 GB	35.9 GB
16384	41.3 GB	230 GB	10.9 GB	35.9 GB

steadily with the input size, and the switch from single- to multi-ciphertext has a relatively minor impact compared to the jumps caused by an increase in the ring dimension. For example, in the case of ranking (basic) in a single-thread setting, transitioning from single-ciphertext ($N = 128$) to multi-ciphertext ($N = 256$) results in a modest memory increase from 1.05GB to 1.15GB (+10%). In contrast, when the ring dimension increases, the memory consumption grows significantly: from $N = 16$ to $N = 32$ (ring dimension: $2^{14} \rightarrow 2^{15}$), the memory rises from 271MB to 765MB (+182%), and from $N = 256$ to $N = 512$ (ring dimension: $2^{15} \rightarrow 2^{16}$), it increases from 1.15GB to 4.08GB (+255%). This is expected since each ciphertext requires at least twice as many bits to represent after a ring dimension increase.

6.3 Comparison with Previous Work

We compare our approach to state-of-the-art solutions for computing the minimum/maximum, median, and sorting. To ensure a fair comparison, for each experiment we use the same comparison approximation method and degree across all evaluated solutions.

Minimum and Maximum For computing the minimum and maximum, we assess our approach against two existing solutions: Phoenix [19] and NEXUS [27]. Like our solution, both Phoenix and NEXUS operate on elements encrypted within a single ciphertext. Table 3 provides a detailed comparison, focusing on the number of evaluations of the comparison function, homomorphic rotations, and the number of slots required in the ciphertext. While the logarithmic and linear scaling of NEXUS and Phoenix, respectively, enable their solutions to handle larger input vectors more efficiently, our approach performs better for small input sizes. As shown in Figure 6b, our solution is faster for $N \leq 1024$ in single-threaded (ST) settings and $N \leq 2048$ in multi-threaded (MT) settings, given our hardware configuration with 64 threads. For larger input vectors, NEXUS becomes the preferred choice due to its superior scalability. By design, Phoenix consistently lags behind NEXUS in runtime performance.

When the input vector is relatively short, our approach provides a significant speed-up over existing solutions. For example, in the use case of NEXUS [27], where the argmax is applied for secure transformer inference, particularly for computing the output layer in BERT-based and GPT-2 models with $N = 128$ nodes, we observe the following:

- Phoenix [19] requires 128 comparisons, 128 rotations, resulting in a total runtime of 92.31 minutes;
- NEXUS [27] requires 7 comparisons, 7 rotations, with a total runtime of 5.05 minutes;
- our approach requires 2 comparisons, 28 rotations, resulting in a total runtime of 12.83 seconds.

However, it is important to note that NEXUS and Phoenix have different space requirements compared to our approach, as summarized in Table 3. This enables them to utilize extra space for batching computations. For instance, if a ciphertext can encode 16,384 elements, our solution processes only 1 argmax in this example, while NEXUS and Phoenix can process 64 and 128 vectors simultaneously, respectively.

Memory usage follows a similar trend, as shown in Table 2. Our approach consumes less memory than NEXUS for small inputs, but the memory consumption increases significantly with larger inputs. The memory consumption of Phoenix is not explicitly reported as it is equivalent to that of NEXUS.

Median and Sorting For sorting vectors, we compare our approach with the state-of-the-art for CKKS, that is the k-way sorting network approach by Hong et al. [18]. While their method also leverages the SIMD capabilities of the encryption scheme, it results in a comparison depth of $k \log_k^2 N$ when employing a k-way network. Their approach performs a total of $O(N \log_k^2 N)$ comparisons, making it more scalable than our solution for larger input vectors. This scalability advantage is evident in Figure 6d, where we report their runtime for $k = 5$, the optimal parameter choice for their solution. However, our approach outperforms Hong et al. [18] for input sizes up to $N = 4096$ in ST settings, and up to $N = 8192$ in MT settings.

We extended Hong et al.'s scheme to compute the median. Specifically, we sort the input vector using their algorithm, extract the values at indices $N/2$ and $N/2+1$, and then compare these median values with the original vector to determine the median indices. As shown in Figure 6c, the overhead introduced by this extension is minimal. Nonetheless, combined with the slightly lower computational cost of our median algorithm compared to full sorting, it shifts the input size thresholds where our solution is beneficial over theirs to $N = 8192$ in ST and $N = 16384$ in MT. In terms of memory usage, we note that their solution maintains constant memory consumption, whereas our approach scales with the input size.

6.4 Applications and Limitations

The main limitation of our approach resides in its quadratic space complexity, which forces us to split the input vector in many smaller chunks, quickly increasing the number of

Table 3: Summary of different solutions for computing the minimum and maximum functionalities.

	Comparisons	**Rotations**	**Slots**
Phoenix [19]	$O(N)$	$O(N)$	N
NEXUS [27]	$O(\log N)$	$O(\log N)$	$2N$
Our work	$O(L^2)$	$O(\log N)$	N^2

necessary comparison evaluations. This is relevant especially for large inputs or in environments with limited parallelization capabilities. As a result, the scalability of our solution is hindered in such scenarios.

However, our solution is well-suited for applications that do not involve processing large vectors. For example, it may be effective in scenarios involving outsourced data analysis of small datasets, which are often encountered in healthcare studies where hospitals analyze data from hundreds of patients. Additionally, it could perform well on datasets with categorical attributes. Those could be represented in a one-hot encoding, making categorical operations (e.g., mode) straightforward to implement within our approach. In such cases, as categorical attributes typically have a limited number of possible values, we would be dealing with short vectors.

In the context of privacy-preserving machine learning (PPML), our approach can be applied in secure inference tasks. For instance, it can be useful for computing the max-pooling layers of a convolutional neural network (CNN), where the maximum is calculated over a kernel-sized vector, typically 3x3, 5x5, or 7x7. It is also applicable in extracting the argmax from the output layer of most neural networks for classification tasks. In such cases, the vector size corresponds to the number of classes in the classification problem, which is typically in the range of 2 to 100, depending on the application.

Beyond inference, in the context of PPML training, we foresee that our solution could be employed to securely train simple unsupervised models in federated settings. A notable example is k-means clustering, where the main computation involves finding the argmin of distances over k clusters. Here, k often takes values such as 2, 5, or 8, making our approach particularly suitable for such tasks.

7 Related Work

A vast body of literature has focused on either sorting elements or computing their maximum value under encryption. Sorting under FHE has been studied starting from 2010 under the Smart-Vercauteren (SV) scheme [25] using bitwise encoding and comparison based swaps to implement algorithms like Bubble Sort, Insertion Sort [8], and Quick Sort [9], all of which requiring $O(N^2)$ comparisons. Subsequently, Bitonic Sort and Odd-Even Merge Sort were also implemented, reducing the cost to $O(N\log^2 N)$ comparisons, of which N can be potentially run in parallel, making the comparison depth $\log^2 N$ [16]. In 2021, some works started designing sorting for floating-point values under CKKS. Hong et al. [18] use k-way sorting networks to achieve a $k\log_k^2 N$ comparison depth. While Lu et al. (PEGASUS) [22] also implement Bitonic Sort but performing the comparisons using the efficient look-up tables of FHEW [15] after a scheme switching from CKKS.

An entire line of work has focused specifically on improving on the evaluation of the comparison function itself. Chialva et al. [14] use the identity $\tanh(kx) = (e^{kx} - e^{-kx})/(e^{kx} + e^{-kx})$ to approximate the sign function for large $k > 0$, while Boura et al. [4] employ an approximation based on Fourier series. The work by Cheon et al. [13] is the first one to study the max function under CKKS, and it is based on an iterative computation of $u^k/(u^k + v^k)$ for large $k > 0$. The same author proposes a new solution in [12], where a composition of 2 polynomials f, g is used to approximate the sign function, proving an optimal asymptotic complexity. This study was then picked up by Lee et al. [21], who generalized the technique to composition of k polynomials, and found the optimal set of polynomials for any given multiplicative depth.

In Phoenix [19], the authors face the problem of computing the argmax in the output layer of a neural network to perform privacy-preserving inference. There, the elements are stored within a single CKKS ciphertext and they propose a method based on rotations to compute the argmax in N comparisons and N rotations. In NEXUS [27], the authors apply the same strategy recursively, exploiting SIMD slot folding, which results in comparing the elements in a binary tree fashion, thus reducing the cost to $\log N + 1$ rotations and $\log N + 1$ comparisons. They use it for secure transformer inference, in particular for computing the argmax output layer in BERT-based and GPT-2 models.

It is also worth mentioning the work by Lu et al. [23], where the authors propose FHE algorithms that compute a variety of descriptive statistics, including percentile. However, their method is limited to ordinal attributes and requires plaintext encoding dependent on value order, whereas our approach addresses numerical attributes, operating with encrypted vectors without specific plaintext encoding, and thus it can be easily integrated in larger (numerical) circuits. Our paper contributes to these lines of work by introducing a novel approach for implementing comparison-based functionalities that achieve a constant comparison depth of 2. This represents an important reduction with respect to existing solutions, which have higher comparison depths, as also summarized in Table 1.

8 Conclusion

In this paper, we have presented a novel approach for computing ranking, order statistics, and sorting of a vector under CKKS. Our method relies on homomorphic matrix encoding and on the SIMD capabilities of the cryptosystem to compare all elements with each other at once, reducing the comparison depth of these algorithms to 2. This makes our solution highly parallelizable, opening potential future work in the direction of hardware acceleration. We showed that our approach is beneficial over existing solutions when the input vector is within the order of thousand of elements, achieving remarkable speed-ups, and particularly shining in multi-threaded settings. We consider the algorithms we designed practical for a wide range of privacy-preserving scenarios, especially for data outsourcing and secure machine learning, or for serving as fundamental building blocks for larger protocols.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 965315. The results reflect only the authors' view and the European Commission is not responsible for any use that may be made of the information this paper contains. This work was also supported by the Netherlands Organization for Scientific Research under NWO:SHARE project [CS.011].

Ethics Considerations

Our work focuses solely on computational methods under encryption, and no real-world data has been used to test our approach. Consequently, we see no privacy concerns, risks of data misuse, or potential harm to individuals or communities arising from our work. The algorithms and protocols we developed are purely theoretical in nature, designed to enhance computational efficiency and security in encrypted environments. They do not interact with human subjects or physical systems in any way that could cause harm or raise ethical concerns. Our methods can be actually put in place to defend sensitive data in specific applications.

Open Science

In accordance with the principles of Open Science, we have made the complete codebase associated with this paper publicly available under the BSD 2-Clause license. The codebase represents the sole artifact accompanying this work and includes the implementation of all functionalities described in the paper. Specifically, the code consists of a C++ library built on top of the OpenFHE library. The permanent link to the artifact is: https://doi.org/10.5281/zenodo.14673904. The primary components (source files) correspond to the key functionalities discussed in Section 3:

1. ranking,
2. minimum,
3. median,
4. sorting.

Additionally, for each functionality, we provide a benchmarking script that measures runtime performance of our solution on randomly generated vectors under various settings. These scripts were used to produce the runtime results reported in Figure 6, which represents the main experimental assessment of our work. Each script takes as input the vector length and the option to run in either single-threaded or multi-threaded mode. The memory consumption data presented in Table 2 was collected using the Linux top command during these benchmark executions.

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

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

[3] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[4] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[6] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual Cryptology Conference (CRYPTO 2011)*, pages 505–524. Springer, 2011.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.

[8] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In *International Conference on Cryptology in India (Indocrypt 2013)*, pages 262–273. Springer, 2013.

[9] Ayantika Chatterjee and Indranil Sengupta. Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective? *IEEE Transactions on Services Computing (TSC 2017)*, 13(3):545–558, 2017.

[10] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology (EUROCRYPT '19)*, pages 34–54, Cham, 2019. Springer International Publishing.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT '17)*. Springer, 2017.

[12] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *26th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '20)*, pages 221–256, Daejeon, South Korea, December 2020. Springer.

[13] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. Numerical method for comparison on homomorphically encrypted numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '19)*, pages 415–445. Springer, 2019.

[14] Diego Chialva and Ann Dooms. Conditionals in homomorphic encryption and machine learning applications. Cryptology ePrint Archive, Paper 2018/1032, 2018. https://eprint.iacr.org/2018/1032.

[15] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques (EUROCRYPT 2015)*, pages 617–640. Springer, 2015.

[16] Nitesh Emmadi, Praveen Gauravaram, Harika Narumanchi, and Habeeb Syed. Updates on sorting of fully homomorphic encrypted data. In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 19–24. IEEE, 2015.

[17] Shai Halevi and Victor Shoup. Algorithms in helib. In *34th Annual Cryptology Conference (CRYPTO 2014)*, pages 554–571, Santa Barbara, CA, August 2014. Springer.

[18] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. Efficient sorting of homomorphic encrypted data with k-way sorting network. *IEEE Transactions on Information Forensics and Security (TIFS 2021)*, 16:4389–4404, 2021.

[19] Nikola Jovanovic, Marc Fischer, Samuel Steffen, and Martin Vechev. Private and reliable neural network inference. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 1663–1677, 2022.

[20] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):23–31, 2018.

[21] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 19(6):3711–3727, 2021.

[22] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (S&P '21)*, pages 1057–1073. IEEE, 2021.

[23] Wen-jie Lu, Shohei Kawasaki, and Jun Sakuma. Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. In *Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2017.

[24] Wen-jie Lu, Jun-Jie Zhou, and Jun Sakuma. Non-interactive and output expressive private comparison from homomorphic encryption. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (AsiaCCS 2018)*, pages 67–74, 2018.

[25] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography (PKC 2010)*, pages 420–443. Springer, 2010.

[26] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. Non-interactive private decision tree evaluation. In *Data and Applications Security and Privacy XXXIV: 34th Annual IFIP WG 11.3 Conference (DBSec 2020)*, pages 174–194, Regensburg, Germany, June 2020. Springer.

[27] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. In *Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2025.

A Recursive Matrix Operations

We provide the pseudocode for SumR, SumC, ReplR, ReplC for a square matrix with N number of rows/columns. The matrix is assumed to be padded in such a way that N is a power of 2.

Algorithm 9 SumR

Input: X encryption of a square matrix of size N.
Output: X encryption of a row vector.
1: **for** $i = 0, \ldots, \log N - 1$ **do**
2: $\quad X \leftarrow X + (X \ll N \cdot 2^i)$
3: **end for**
4: $X \leftarrow \mathsf{MaskR}(X, 0)$
5: **return** X

Algorithm 10 SumC

Input: X encryption of a square matrix of size N.
Output: X encryption of a column vector.
1: **for** $i = 0, \ldots, \log N - 1$ **do**
2: $\quad X \leftarrow X + (X \ll 2^i)$
3: **end for**
4: $X \leftarrow \mathsf{MaskC}(X, 0)$
5: **return** X

Algorithm 11 RepIR

Input: X encryption of a row vector of size N.
Output: X encryption of a square matrix.
1: **for** $i = 0, \ldots, \log N - 1$ **do**
2: $\quad X \leftarrow X + (X \gg N \cdot 2^i)$
3: **end for**
4: **return** X

Algorithm 12 RepIC

Input: X encryption of a column vector of size N.
Output: X encryption of a square matrix.
1: **for** $i = 0, \ldots, \log N - 1$ **do**
2: $\quad X \leftarrow X + (X \gg 2^i)$
3: **end for**
4: **return** X

B Effect of Chebyshev Approximation Degree on Performance

In this work we employ a relatively basic implementation of comparison and indicator functions. There is an entire body of literature that focuses specifically on this topic, which one may consider for real-life application, see for instance [12, 21]. Nonetheless, we still provide some indication on how different approximation degrees influence the runtime of our algorithms, in exchange of having a more precise result.

We use ranking and minimum computation as case studies for this analysis. Figure 7 shows the impact of the approximation degree of the comparison function, represented in terms of its multiplicative depth, on runtime and approximation error in the ranking task. Higher degrees yield lower error but incur longer runtimes. The reported error is for a vector of 128 elements, indicating how many positions each element is

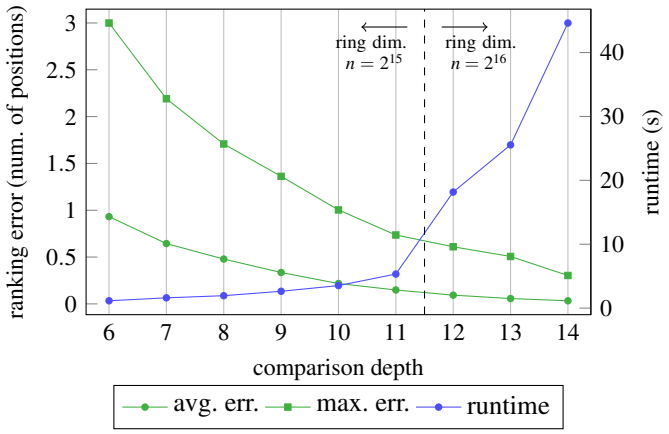
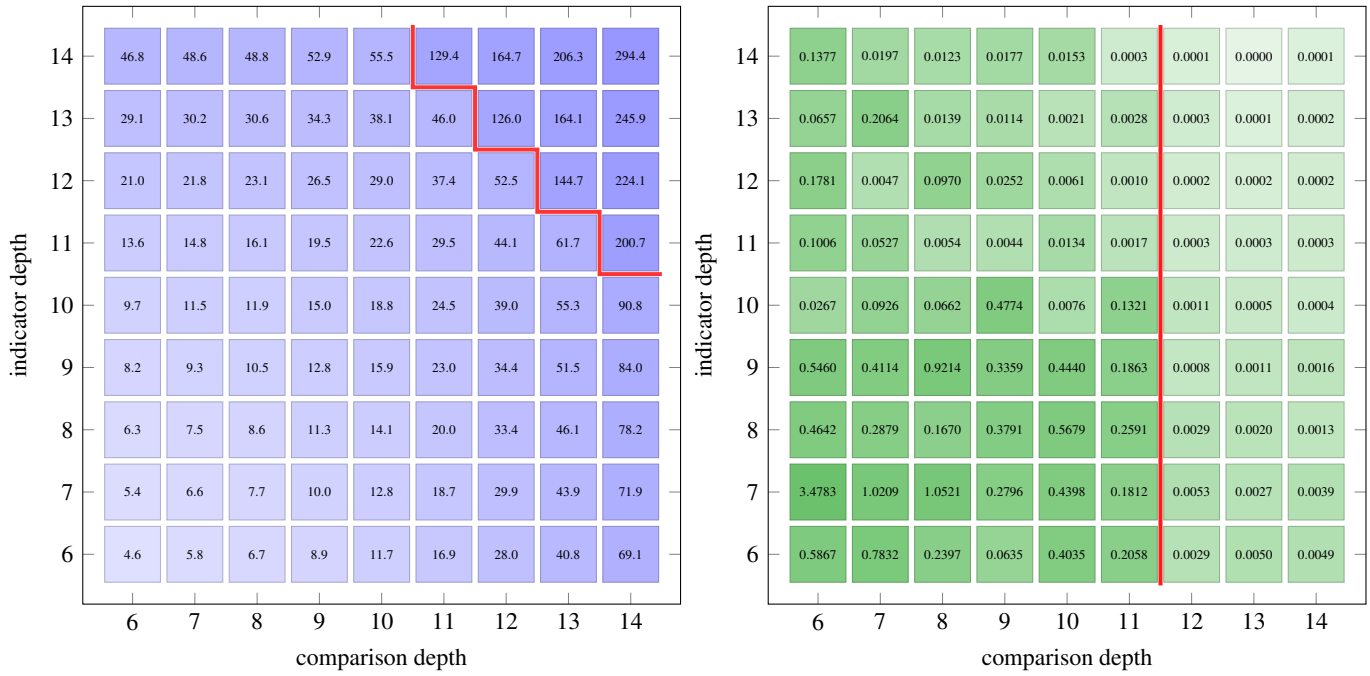


Figure 7: Ranking a vector of 128 elements for different approximation degrees of the comparison function (as multiplicative depth). The ranking error and runtime are reported.

ranked away, on average and in the worst case. The steep increase after depth 11 is due to the fact that the ring dimension must increase to assure 128 bits of security, making the basic homomorphic operations more expensive.

Sweet spots in the trade-off can be noticed at depth 10 and 11, which corresponds to an approximation degree of 2^9 and 2^{10}, respectively. At depth 10, the runtime is around 3.52 seconds, while the elements are ranked no more than 1 position away from their actual rank. Notably, this error is proportional to the separation between elements; closely positioned elements are more susceptible to rank swapping. One could willingly decide to use a lower approximation degree and exploit this effect to achieve a form of differential privacy.

Similar considerations can be applied to the minimum computation. Figure 8 reports the error as the L1 distance between the computed and the expected minimum values. We can see that a sensitive improvement in accuracy occurs when transitioning from comparison depth 11 to 12. Moreover, we notice a sweet spot for the runtime when the sum of the approximation depths equals 24 (the upper-right diagonal), with the (12, 12) combination yielding the lowest error.



(a) runtime in seconds

(b) Error

Figure 8: Computing the minimum of a vector of 32 elements for different approximation degrees of the comparison and indicator functions (as multiplicative depth). The runtime and L1 error are reported.