POLITECNICO DI TORINO
Repository ISTITUZIONALE

Lightweight sorting in approximate homomorphic encryption

Original
Lightweight sorting in approximate homomorphic encryption / Rovida, L., Leporati, A., Basile, S.. - In: IACR COMMUNICATIONS IN CRYPTOLOGY. - ISSN 3006-5496. - 3:1(2026), pp. 1-29. [10.62056/a3ksdkmol]

Availability:
This version is available at: 11583/3010677 since: 2026-05-08T10:19:19Z

Publisher:
IACR

Published
DOI:10.62056/a3ksdkmol

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

25 November 2026

IACR Communications in Cryptology
ISSN 3006-5496, Vol. 3, No. 1, 29 pages.

https://doi.org/10.62056/a3ksdkmol

Check for updates

Lightweight sorting in approximate homomorphic encryption

Lorenzo Rovida[a,1], Alberto Leporati[2] and Simone Basile[2]

[1] Dept. of Mathematical Sciences, Polytechnic University of Turin, Turin, Italy
[2] Dept. of Informatics, Systems and Communication, University of Milano-Bicocca, Milan, Italy

Abstract

Sorting encrypted values is an open research problem that plays a crucial role in the broader objective of providing efficient and practical privacy-preserving online services. The current state of the art work by Mazzone, Everts, Hahn and Peter [USENIX Security '25] proposes efficient algorithms for indexing and sorting based on the CKKS scheme, which deviates from the compare-and-swap paradigm, typically used by sorting networks, using a permutation-based approach. In this work, we follow up their work and explore different approaches to approximate the nonlinear functions required by the circuit. We propose simpler and concrete solutions that allow for faster computations, smaller memory requirements and higher precision. For example, our framework allows to sort 128 real elements in roughly 22 seconds, while maintaining a precision of 0.001 and requiring 3 GB of memory. Furthermore, we propose an implementation of a swap-based bitonic network that is not based on approximations of the $\mathrm{sgn}(x)$ function, which scales linearly with the number of values, useful when the number of available slots is small.

1 Introduction

The introduction of the first practical Fully homomorphic encryption (FHE) scheme in 2009 [Gen09], started a new research field that has been only theorized in 1978 by Rivest, Adleman and Dertouzos [RAD78]. The core idea of this cryptographic primitive is to encrypt data in such a way that operations performed in the encrypted space (typically additions and multiplications) are reflected as the same operations evaluated in the plaintext space. When this holds, performing computations on plain or on encrypted data yields the same result. By implementing online services using FHE primitives, it is possible for service providers to offer the same functionalities without accessing users' plain data.

Using FHE, it is possible to perform computations with the same computational power as Turing machines [GKP+13], but over encrypted data; although there is still a huge gap between theory and practice. For example, evaluating a conditional statement represents one of the main challenges in FHE, since it implies the evaluation of a strongly nonlinear function. Classical sorting algorithms are heavily based on conditional statements, therefore evaluating them on encrypted values is far from being an easy task. Moreover, it is not possible to translate arbitrary programs using FHE primitives, since the flow of execution cannot depend on the (encrypted) input data. In particular, selection statements (e.g.,

Emails: lorenzo.rovida@polito.it (Lorenzo Rovida), alberto.leporati@unimib.it (Alberto Leporati), s.basile1@campus.unimib.it (Simone Basile)

[a] A large part of this work has been carried out while the author was a Ph.D. student at the University of Milano-Bicocca.

This work is licensed under a "CC BY 4.0" license.

Received: 2026-02-03 Accepted: 2026-04-10

conditional branches) [BJ66] are not allowed, as they would break the (semantic) security of the system.

Nevertheless, the need for efficient and quick sorting algorithms is high since many protocols, such as encrypted machine learning and private information retrieval, heavily rely on sorting results as a subroutine. The goal of this paper is thus to propose lightweight algorithms to sort encrypted values based on CKKS [CKKS17], which is an approximate FHE scheme.

Motivations. Sorting itself is a procedure that is used in a variety of algorithms as a subroutine. For instance, we can identify some usages in the field of (encrypted) machine learning: think of as an example linear recommender systems [KPAM20; GEMR25], which can be somewhat summarized as

$$\text{recommendation}(\boldsymbol{x}) = \text{sort}(W \cdot \boldsymbol{x}),$$

for some learned weight $W \in \mathbb{R}^{m \times n}$ and input $\boldsymbol{x} \in \mathbb{R}^n$. Another possible usage is that it enables the usage of the ORDER BY keyword in encrypted databases [Zha25; WZD+25]. If some server is storing encrypted data with FHE compatible metadata, one could blindly sort files according to some rule. In general, sorting encrypted data is useful in almost every application that deals with encrypted queries in databases, including private information retrieval [LP23; LMW23; CYW+25].

1.1 Related works

Sorting encrypted values. In 2021, Hong *et al.* [HKC+21] proposed a so-called k-way sorting network, based entirely on the CKKS scheme, in which comparisons are performed using polynomial approximations of the $\text{sgn}(x)$ function. According to their estimations, sorting 512 values requires roughly 150 minutes using an AMD Ryzen 9 3950X CPU, and more than 23 GB of RAM memory. We will refer to this approach as *network-based sorting*. This top-k approach was then improved and extended by Cong, Geelen, Kang and Park [CGKP25] under the TFHE scheme [CGGI20], with applications to k-NN classifiers.

We will consider the state of the art work by Mazzone, Everts, Hahn and Peter [MEHP25], where efficient algorithms for sorting based on CKKS are presented. In particular, they propose procedures that are not based on the swap operations implemented in the previous work based on sorting networks. Their sorting algorithm can be divided into three steps: (i) find the indexing of the input vector $\boldsymbol{v}$, namely the index of each value v_i in the sorted vector $\text{sort}(\boldsymbol{v})$, then, (ii) given the indexing, build the corresponding permutation matrix P, and finally (iii) sort $\boldsymbol{v}$ by evaluating a matrix-vector multiplication. Although very practical for small sets of numbers, this approach requires storing $\Theta(n^2)$ slots, where n is the number of values to be sorted, indeed the authors propose to split the input in *chunks* of data when the number of values is too large, i.e., when $n > 256$. One of the main advantages of this approach is that the circuit depth, in terms of multiplications, is very small compared to sorting networks. We will refer to their approach as *permutation-based sorting*.

Approximate comparison operations. A part of our work includes an efficient sorting network which heavily relies on comparing encrypted values. In literature, comparisons are typically performed using (polynomial approximations of) the $\text{sgn}(x)$ function on the difference between two values. Cheon, Kim and Kim [CKK20] observed that $(\sqrt{x})^2 = x \cdot \text{sgn}(x)$, enabling to use approximations of the $\text{sgn}(x)$ function to compute the minimum and maximum between two elements as $\min(a, b) = (a + b)/2 - \sqrt{(a - b)^2}/2$. Nevertheless, these approaches required a large multiplicative depth to ensure a decent amount of precision. Lee *et al.* [LLNK22] proposed an approach based on composite Minimax

polynomials which showed smaller execution time and a smaller multiplicative depth. We remark that [LLNK22] proposed to evaluate the $\max(a, b)$ and $\min(a, b)$ functions, along with the Rectified-Linear Unit (ReLU) function, as

$$\min(a, b) \approx \frac{(a + b) - (a - b) \cdot p(a - b)}{2} \quad \text{and} \quad \text{ReLU}(x) \approx \frac{x + x \cdot p(x)}{2}, \tag{1}$$

where $p(x)$ is some polynomial approximation of the $\text{sgn}(x)$ function.

Comparison with [KÜYL25]. In a concurrent work, Kim, Ünay, Yilmazer-Metin and Lee [KÜYL25] proposed solutions to sorting which can be considered close to our approach, especially considering the usage of the $\text{sinc}(x)$ function ($\text{sinc}(x) := \sin(\pi x)/\pi x$). We believe that some of [KÜYL25] ideas could be applied to our permutation-based sorting framework, especially when sorting large amount of values. Nevertheless, we do not consider our works as colliding with each other, as their experiments are based on the $N = 2^{17}$ ring and polynomial compositions are used to approximate the $\text{sgn}(x)$ function. On the other hand, our proposal is mainly focused on lightweight computations, and we believe that our framework can be improved with some of [KÜYL25] techniques. As it will be shown, our circuits can be instantiated using the smaller $N = 2^{16}$ ring and Chebyshev polynomials, combined with some "cleaning" polynomials. Additionally we show that our circuit supports the sorting of values at distance at most $\delta = 0.001$, while [MEHP25; KÜYL25] run their experiments with inputs at distance at most $\delta = 0.01$, excluding repeated elements.

1.2 Our contributions

We split our contribution in two parts:

1. We follow-up [MEHP25] and explore different and more efficient approaches to approximate the required nonlinear functions in their permutation-based sorting algorithm. We will show how to reduce the depth of the circuit, reducing the upper bound from 65 to 29 levels in the same setup (i.e., sorting with precision of 0.01), making the sorting faster and with a smaller memory footprint. This makes it easier to embed our solution "blindly" as a building block for more complex CKKS-based architectures. In order to keep the circuit efficient, we will not use rings larger than $N = 2^{16}$, maintaining $\lambda \geq 128$ bits of (classical) security.

   In particular, we propose to simplify their framework by removing non essential approximations of step functions. We will use the k-scaled sigmoid function — followed by some cleaning polynomial à la [DMPS23] — in the first ranking procedure and its derivative in the tie-offset correction, and the $\text{sinc}(x) := \sin(\pi x)/(\pi x)$ function to perform the equality check required to construct the permutation matrix P. The $\text{sinc}(x)$ function can be accurately approximated without requiring many multiplicative levels, as it is continuous and smooth, yet it allows to obtain the same performance as the more complex counterparts based on $\text{sgn}(x)$ — under some assumptions, i.e., the input of $\text{sinc}(x)$ must be as close as possible to $\mathbb{Z}$. We will also show some "cleaning" properties of the $\text{sinc}(x)$ function. The number of required slots for the ciphertexts to contain the input is quadratic $\Theta(n^2)$, where n is the number of values to be sorted.

2. We construct an efficient sorting network using bitonic sorting that, differently from [HKC+21], is based on a definition of the $\min(a, b)$ function that does not depend on the $\text{sgn}(a - b)$ function. This allows us to reduce the depth of the circuit and to obtain more accurate computations. We aim to approximate the $\min(a, b)$

and the $\max(a, b)$ functions using an approximation of the $\max(0, x)$ function (i.e., the ReLU function):

$$\min(a, b) = a - \max(0, a - b) = a - \mathrm{ReLU}(a - b). \tag{2}$$

The advantage of this approach is that $\max(0, a - b)$ is easier to approximate, even with Chebyshev polynomials, with respect to the polynomial $p(x)$ used in [LLKN22; LLNK22], since it is continuous and does not contain steps. We show that, with this approach, it is possible to obtain large level of precision bits α with a small multiplicative depth (e.g., $\alpha > 10$ with depth $d = 9$). By performing swaps using (2), computational time is orders of magnitude smaller than previous works [HKC+21; HWW+22; LHH+21], and better scales with the number of inputs. The number of required slots for the ciphertexts to contain the input is linear $\Theta(n)$ and can become a valid alternative in applications where the number of available slots is not as large as $\Theta(n^2)$ – namely the slots required by the more efficient permutation-based approach.

2 Preliminaries

In what follows, if not stated otherwise, lowercase letters are used to indicate numbers (e.g., a), bold-italic lowercase letters to indicate vectors (e.g., $\boldsymbol{v}$), and uppercase letters are used to indicate matrices (e.g., A). We extend the vectors notation for single numbers, e.g., $\mathbf{1}^n = (1, 1, \ldots, 1)$. If not specified, the dimension should be clear by the context. Vectors are considered as column vectors, the i-th column of a matrix A is referred to as $\boldsymbol{a}_i$ and the i-th element of a vector $\boldsymbol{v}$ as v_i. Given a vector $\boldsymbol{v}$, we define $\boldsymbol{v}_{i:j}$ as the subvector containing the elements from index i to j, namely $\boldsymbol{v}_{i:j} = (v_i, v_{i+1}, \ldots, v_j)$. If not specified, we assume $\log(x) = \log_2(x)$. We denote the flattened outer product of two vectors using the symbol $\boldsymbol{a} \otimes \boldsymbol{b}$: in order to obtain a vector out of the resulting matrix, the result of a outer product is flattened row-wise as a vector, therefore we denote $\otimes : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{n^2}$. For instance, $[x, y] \otimes [w, z] = [xw, xz, yw, yz]$.

2.1 The CKKS scheme

The sorting algorithm is implemented on top of the CKKS scheme, which is an *approximate* homomorphic encryption scheme. It was first introduced in 2017 by Cheon, Kim, Kim and Song [CKKS17]. Informally, the cryptosystem encodes and encrypts vectors of complex values as follows:

$$\boldsymbol{v} \in \mathbb{C}^{N/2} \xrightarrow{\text{Encoding}} \boldsymbol{p} \in \mathbb{Z}[X]/(X^N + 1) \xrightarrow{\text{Encryption}} \boldsymbol{c} \in \left(\mathbb{Z}_Q[X]/(X^N + 1)\right)^2,$$

where N is a power of two, and a ciphertext is a RLWE sample [LPR13] with modulus Q. Notice that the encoded values are injected with some noise and rounded to (very large) integers using a scale Δ, in a fixed-point representation. Let $\mathsf{DFT} : \mathbb{R}[X]/(X^N+1) \to \mathbb{C}^{n/2}$ be a discrete Fourier transform (DFT) defined as $\boldsymbol{m}(X) \mapsto (\boldsymbol{m}(\zeta^{5^i}))_{0 \le i < N/2}$ where ζ is a complex primitive root of unity. Since 5 is a generator of a cyclic group of order $N/2$, it allows to perform rotations via the Galois automorphism $\zeta \to \zeta^5$. Let also $\mathsf{iDFT} : \mathbb{C}^{N/2} \to \mathbb{R}[X]/(X^N + 1)$ be its inverse. We give a brief description of the basic operations:

- Setup(λ): for a given security parameter λ and level $L \in \mathbb{Z}^{>0}$, output the ring dimension N modulus Q and distributions χ_{key}, χ_{err} and χ_{enc}.
- Keygen: sample $\boldsymbol{a}(X) \sim \mathcal{R}_Q$, a secret $\boldsymbol{s} \sim \chi_{\text{key}}$ and an error $\boldsymbol{e} \sim \chi_{\text{err}}$. Set $\mathsf{sk} := (1, \boldsymbol{s})$ and $\mathsf{pk} := (-\boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}, \boldsymbol{a})$. The public key is a structured random q-ary basis for a lattice, containing N short trapdoor vectors [LPR13].

- Encode($\boldsymbol{z}$): given $\boldsymbol{z} \in \mathbb{C}^n$, return $\boldsymbol{p} := \lfloor \Delta \cdot \mathsf{iDFT}(\boldsymbol{z}) \rceil \in \mathcal{R}$.
- Decode($\boldsymbol{p}(X)$): given $\boldsymbol{p}(X) \in \mathcal{R}$, return $\boldsymbol{z} := \mathsf{DFT}(\boldsymbol{p}(X))/\Delta \in \mathbb{C}^n$.
- $\text{Encrypt}_{\mathsf{pk}}(\boldsymbol{p})$: given $\boldsymbol{p} \in \mathcal{R}$, sample $\boldsymbol{u} \sim \chi_{\text{enc}}$ and $\boldsymbol{e}_1, \boldsymbol{e}_2 \sim \chi_{\text{err}}$, set $\mathsf{ct} := [u \cdot \mathsf{pk} + (\boldsymbol{p} + \boldsymbol{e}_1, \boldsymbol{e}_2)]_Q \in \mathcal{R}_Q^2$ and return $\{\mathsf{ct}, Q\}$.
- $\text{Decrypt}_{\mathsf{sk}}(\mathsf{ct})$: given $\mathsf{ct} := (\boldsymbol{c}_1, \boldsymbol{c}_2) \in \mathcal{R}_Q^2$, return $\boldsymbol{p} := \boldsymbol{c}_1 + \boldsymbol{c}_2 \cdot \boldsymbol{s} \in \mathcal{R}$.

Since data is encoded via iDFT, operations in the encoded space are reflected as *slot-wise* operations in the original cleartext space. This enables fast Single Instruction, Multiple Data (SIMD) computations. In our implementation we also take advantage of Residue Number System (RNS) and Number Theoretic Transform (NTT) to accelerate computations on encrypted polynomials. The available primitive operations are the following:

- Add/Sub/Mul($\boldsymbol{c}_1, \boldsymbol{c}_2$): given two ciphertexts $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$, performs the slot-wise addition/subtraction/multiplication between each pair of elements. Without loss of generality, $\boldsymbol{c}_2$ can also be a plaintext.
- $\text{Rot}_i(\boldsymbol{c})$: given a ciphertext $\boldsymbol{c}$, rotate its elements by i positions. Positive values of i indicate a rotation to the left, negative values to the right.

We refer the reader to [KPP22] for the technical details about the used CKKS scheme. Since only basic arithmetic operations are available, the evaluations of nonlinear functions are, as previously stated, performed using polynomial approximations.

Bootstrapping. During the encoding process, each element of the cleartext vector $\boldsymbol{v}$ is scaled by a factor Δ, which is a large power of two, typically larger than 2^{30}. When two ciphertexts are multiplied, the resulting scale will be the squared scale $\Delta \cdot \Delta$. In order to avoid an "explosion" of the scale, a *rescaling* operation is performed after each multiplication, so that the resulting ciphertext c is scaled by a factor $1/\Delta$. This causes a reduction of the modulus of the ciphertext from Q to Q/Δ. The modulus Q of a fresh ciphertext c is constructed using a so-called *moduli chain*: $Q = q_1 \cdot q_2 \cdot \cdots \cdot q_l$; after each rescaling operation, an element $q_i \approx \Delta$ of the chain is lost. When Q reaches its minimum modulus q_1 (i.e., $L - 1$ multiplications have been performed), the ciphertext must be *bootstrapped*. This operation, which is the most computationally expensive in FHE, refreshes its level (i.e., the number of multiplications performed on a ciphertext) by increasing the modulus from q_1 back to Q, enabling to evaluate circuits of arbitrary depth. Intuitively, the main challenge is about removing the extra $k \cdot q_1$ term that appears as part of the message when raising the modulus, with the magnitude of $k > 0$ depending on the norm of secret key [CHK+18].

Chebyshev polynomials. The *Chebyshev polynomials* [Tre19] refer to a family of orthogonal polynomials defined by the following recurrence relation:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{2n}(x) = 2T_n(x)^2 - 1, \quad T_{2n+1}(x) = 2T_n(x) \cdot T_{n+1}(x) - x.$$

Given this set of polynomials, it is also possible to define the so-called *Chebyshev series* representation of a polynomial:

$$p_n(x) = \sum_{k=0}^{n-1} c_k T_k(x),$$

where $p_n(x)$ is a polynomial of degree $n - 1$, $T_k(x)$ is the k-th Chebyshev polynomial of the first kind, and each coefficient c_k is set according to a certain function f to be

approximated, w.l.o.g., over $[-1, 1]$. Given the following set, which we will call *Chebyshev interpolants*:

$$x_i = \cos\left(\frac{(i+\frac{1}{2})\pi}{n}\right) \text{ with } 0 \leq i < n,$$

the coefficients c_k are determined such that $p_n(x_i) = f(x_i)$. Putting everything together, it is possible to define polynomial approximations of any continuous function f by performing a polynomial regression over the Chebyshev interpolants. In particular, an approximation of degree d is performed over the $d+1$ roots. Our circuit uses an algorithm to efficiently evaluate this set of polynomials using a modified version of the Paterson-Stockmeyer [PS73] algorithm introduced by Chen, Chillotti and Song [CCS19] that minimizes the errors and the number of multiplications.

3 Methodology

We will now describe our concrete CKKS-based approaches to construct more efficient permutation-based sorting algorithms and bitonic sorting networks.

3.1 Permutation-based sorting

This approach relies on the computation of two nonlinear functions only. Note that most of the presented procedures are optimized from the original approach introduced in [MEHP25]. Let us define $\boldsymbol{c} \in \mathbb{R}^n$ as the (encrypted) input vector[3], where n is without loss of generality considered a power of two for efficiency reasons.

3.1.1 Indexing based on SIMD.

The $\mathrm{sgn}(x)$ function is used to count, for every $c_i \in \boldsymbol{c}$, how many elements in the input vector are smaller than c_i. We define it as:

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0.5 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}. \tag{3}$$

By an abuse of notation, we may write $\mathrm{sgn}(\boldsymbol{c})$ to denote the vector obtained by applying $\mathrm{sgn}(x)$ to each component $c_i \in \boldsymbol{c}$. As we will later show, the SIMD functionality of CKKS enables counting, for each c_i, the number of c_j satisfying $c_i < c_j$, in a single comparison. This is achieved by replicating the input vector and operate on $\Theta(n^2)$ slots of the ciphertext.

We require the input to be encoded in two different ways: *repeated* encoding (the whole vector $\boldsymbol{c}$ is repeated n times) and *expanded* encoding (each element c_i is repeated n times). We also give a definition of *indexing* (or *ranking*) that will simplify later proofs.

Definition 1 (Repeated encoding)**.** Given a vector $\boldsymbol{v} \in \mathbb{R}^n$, its *repeated encoding* is defined by repeating the $\boldsymbol{v}$ vector n times as:

$$\boldsymbol{v}_{\mathrm{rep}} := \mathbf{1}^n \otimes \boldsymbol{v} = (v_1, \ldots, v_n, v_1, \ldots, v_n, \ldots, v_1, \ldots, v_n) \in \mathbb{R}^{n^2}. \tag{4}$$

Definition 2 (Expanded encoding)**.** Given a vector $\boldsymbol{v} \in \mathbb{R}^n$, its *expanded encoding* is defined by repeating each i-th element of $\boldsymbol{v}$ precisely n times:

$$\boldsymbol{v}_{\mathrm{exp}} := \boldsymbol{v} \otimes \mathbf{1}^n = (v_1, \ldots, v_1, v_2, \ldots, v_2, \ldots, v_n, \ldots, v_n) \in \mathbb{R}^{n^2}. \tag{5}$$

[3]For the sake of clarity, we will for now make no difference between clear and encrypted vectors since every operation in the described procedure can be implemented via additions, multiplications and rotations.

Definition 3 (Indexing)**.** Let $\boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{R}^n$. For each $\boldsymbol{v}_i$, define

$$l_i := \big|\{j : c_j < c_i\}\big| \quad \text{and} \quad e_i := \big|\{j : c_j = c_i\}\big|$$

to be the number of elements less than and equal to v_i, respectively. The *indexing* of v_i is

$$\text{index}(v_i) = l_i + \frac{e_i + 1}{2}.$$

If no equal elements are in $\boldsymbol{v}$, then the indexing of v_i is simply the number of values smaller than v_i in $\boldsymbol{v}$. Otherwise, it is as above plus a fraction that represents the number of repetitions.

Given some ciphertext $\boldsymbol{c}$ and the corresponding $\boldsymbol{c}_{\text{rep}}$, $\boldsymbol{c}_{\text{exp}}$, we start by performing a subtraction between them[4]. Let us for a moment ignore vectors with possibly repeated elements. A visual example is given in Figure 1, considering $n = 4$ and the input vector $\boldsymbol{c} := (4, 7, 1, 8) \in \mathbb{R}^4$.

$\boldsymbol{c}_{\text{rep}}$:	4	7	1	8	4	7	1	8	4	7	1	8	4	7	1	8	−
$\boldsymbol{c}_{\text{exp}}$:	4	4	4	4	7	7	7	7	1	1	1	1	8	8	8	8	=
$\boldsymbol{c}_{\Delta}$:	0	3	−3	4	−3	0	−6	1	3	6	0	7	−4	−1	−7	0	

Figure 1: Computing $\boldsymbol{c}_{\Delta}$ from the two encodings of the input ciphertext $\boldsymbol{c}$ to be sorted

At this point, the indexing – ignoring repetitions – for the i-th element is given by the count of positive elements in $\boldsymbol{c}_{\Delta}$ in positions $k \cdot n + i$, for $0 \leq k < n$. In order to do that, we (i) first apply the $\text{sgn}(x)$ function to each element of $\boldsymbol{c}_{\Delta}$ and (ii) rotate and sum elements at distance n. Finally, we increase each slot by 0.5 to correct the bias introduced on elements whose subtraction lead to 0. We will later present our concrete approach to perform such procedure in CKKS. This procedure is illustrated in Figure 2. Notice that

$\text{sgn}(\boldsymbol{c}_{\Delta})$:	.5	1	0	1	0	.5	0	1	1	1	.5	1	0	0	0	.5
$0.5 + \sum$:	2	3	1	4	2	3	1	4	2	3	1	4	2	3	1	4

Figure 2: Given $\boldsymbol{c}_{\Delta}$, the indexing of $\boldsymbol{c}$ is obtained by performing a rotate and sum procedure (indicated with the $\sum$ symbol) over $\text{sgn}(\boldsymbol{c}_{\Delta})$, plus $\mathbf{0.5}_{n^2}$

this procedure returns the *repeated* indexing vector – following the example, $\mathbf{1}^n \otimes (2, 3, 1, 4)$ – and it is simple, apart from its main bottleneck, i.e., the $\text{sgn}(x)$ function, since in our setting it has to be approximated with high degree polynomials. We will later provide in Algorithm 1 a more extensive definition of this procedure using CKKS primitives.

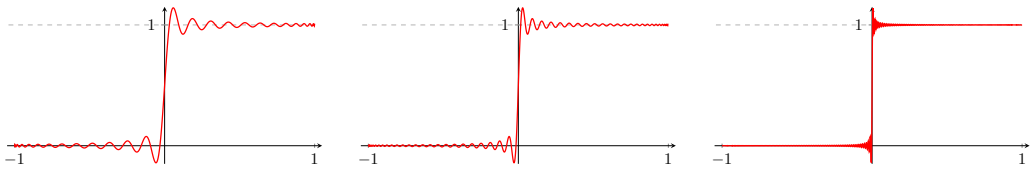
As previously spoiled, the obstacle that prevents this algorithm to be implemented under CKKS is the $\text{sgn}(x)$ function that has to be approximated through some polynomial(s). In order to do that, it is possible to follow two directions:

- Using the approach firstly introduced in [CKK20], and used in [MEHP25], where the authors propose to approximate $\text{sgn}(x)$ using compositions of known polynomials $f(\cdot)$ and $g(\cdot)$. This path leads to smaller runtime and to more accurate computations, although it has the drawback of requiring a large multiplicative depth.

[4]We assume both encodings to be immediately available. If not, one can compute them, starting from $\boldsymbol{c}$, at the cost of 1 multiplication and $\log(n)$ additions and rotations.

- Using Chebyshev polynomials to directly approximate $\mathrm{sgn}(x)$. These have the advantage of giving decent approximations using less levels, although they require more computational time to obtain a large amount of precision.

Since our goal is to present low-depth implementations (i.e., we avoid using large CKKS rings such as $N = 2^{17}$), we stick to Chebyshev polynomials, although we follow a hybrid approach that will improve their precision by composing them with "cleaning polynomials" [DMPS23] afterwards. Approximating the raw $\mathrm{sgn}(x)$ function, though, is not a good choice, as the resulting polynomial will suffer by a strong Gibbs phenomenon [PHE+25], namely large oscillations near the "step" occurring in $x = 0$, as shown in Figure 3.



(a) Approximation with degree $d = 50$ (b) Approximation with degree $d = 100$ (c) Approximation with degree $d = 500$

Figure 3: Chebyshev approximations of the raw $\mathrm{sgn}(x)$ function in $[-1, 1]$, with different degrees

We thus propose to define a priori some value $\delta > 0$ that defines the minimum distance between any pair of elements in the vector to be sorted – ignoring repeated values. In the experiments presented in [MEHP25], this is implicitly set to $\delta = 0.01$, although they conveniently report sets of parameters to achieve any desired level of precision in their open-source repository. We define the k-scaled sigmoid σ_k function as

$$\sigma_k(x) := \frac{1}{1 + e^{-kx}}. \tag{6}$$

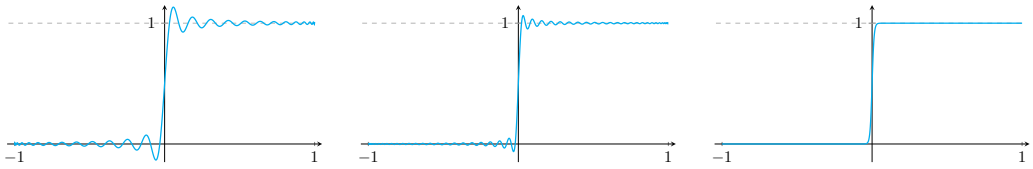
and the cleaning polynomial $c(x)$ as:

$$c(x) := 3x^2 - 2x^3 \tag{7}$$

Notice that the scaled sigmoid function acts as a "smooth" $\mathrm{sgn}(x)$ function, where the smoothness parameter is given by $k \in \mathbb{Z}_{>0}$. Our strategy is to empirically find some value k such that the sigmoid well approximates $\mathrm{sgn}(x)$, up to some small error – with respect to δ – in $[-1, -\delta] \cup [\delta, 1]$.

As an example, let us fix $\delta = 0.1$ and observe that, with $k = 150$, the error $\|\mathrm{sgn}(x) - \sigma_k(x)\|_\infty$ in $[-1, -\delta] \cup [\delta, 1]$ is smaller[5] than 10^{-7}. Using this function, eventually followed by $c(x)$ as, e.g., $(c \circ \sigma_k)(x)$, allows for much smoother approximations, since near $x = 0$ we can control the "steepness" of the jump. We report some illustrations of such approximations in Figure 4.

[5]Note that with "small" we always mean with respect to the input. If its precision is controlled by, e.g., $\delta = 0.01$, then we consider an error of 10^{-7} as small.



(a) Approximation with degree $d = 50$ (b) Approximation with degree $d = 100$ (c) Approximation with degree $d = 500$

Figure 4: Chebyshev approximations of the 150-scaled Sigmoid function in $[-1, 1]$, with different degrees

The behavior of the polynomial for $x \in [-\delta, \delta]$ is irrelevant, since by assumption there will be no pair of elements such that its difference lies in such interval. Nonetheless, we still require a precise approximation for $0 \in [-\delta, \delta]$, which needs to be mapped to 0.5, although for that we have the following result.

Lemma 1 (Chebyshev interpolant of σ_k at $x = 0$)**.** *Let $p_d(x)$ be the Chebyshev interpolant of degree d of the scaled sigmoid function $\sigma_k(x)$ on the interval $[-1, 1]$. Then $p_d(0) = 0.5$ for any $k > 1$ and any $d \geq 1$.*

Proof. Refer to Section B.1. □

Given this lemma, we can safely use Chebyshev approximations of the scaled sigmoid function, by carefully choosing a parameter $k > 1$ that allows for the scaled sigmoid to behave as $\operatorname{sgn}(x)$ in the interval $[-1, -\delta] \cup [0] \cup [\delta, 1]$, with small errors. Also note that $c(0.5) = 0.5$, so applying the cleaning polynomial after the Chebyshev approximation does not change the behavior in $x = 0.5$.

We present the whole procedure that computes the indexing in Algorithm 1 along with a proof of correctness.

Algorithm 1 Generating the indexing vector with SIMD computations

1: **Input:** Two encodings $\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}} \in \mathbb{R}^{n^2}$ of the input vector $\boldsymbol{c} \in \mathbb{R}^n$ with (possibly repeated) elements at distance at most $\delta > 0$ and a polynomial approximation $p(x)$ of the function $\operatorname{sgn}(x)$ with desired accuracy in $[-1, -\delta] \cup [\delta, 1]$.
2: **Output:** Indexing vector $\boldsymbol{idx} \in \mathbb{R}^{n^2}$ of $\boldsymbol{c}$ in repeated encoding, as in Definition 3.
3: **procedure** INDEXING($\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}}$)
4: $\quad \boldsymbol{c}_\Delta := \text{Sub}(\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}})$
5: $\quad \boldsymbol{s} := p(\boldsymbol{c}_\Delta)$
6: $\quad$ **for** $i := 0$ to $\log(n)$ **do**
7: $\quad\quad \boldsymbol{s} := \text{Add}(\boldsymbol{s}, \text{Rot}_{n \cdot 2^i}(\boldsymbol{s}))$ ▷ Rotate and sum
8: $\quad \boldsymbol{idx} := \text{Add}(\boldsymbol{s}, \boldsymbol{0.5}_{n^2})$
9: $\quad$ **return** $\boldsymbol{idx}$

Lemma 2 (Correctness of Algorithm 1)**.** *Algorithm 1 returns the correct indexing, given that the approximation error of $p(x)$ in $[-1, -\delta] \cup [0] \cup [\delta, 1]$ is small compared to $\delta > 0$.*

Proof. Refer to Section B.2. □

Notice that, since $\mathbb{Z} \subset \mathbb{R}$, we will assume that the indexing lives in $\mathbb{R}$, so that subsequent algorithms based on approximations will not require a change of domain. Nevertheless, we will show how to pick parameters and inputs in such a way that the behavior in $\mathbb{R}$ matches the one in $\mathbb{Z}$ (with some arbitrary precision). In other words, we will make sure to keep the CKKS noise out of the $1/\delta$ most significant digits.

Remark 1 (Ascending and descending sorting). Notice that this framework is flexible enough to easily switch between ascending and descending indexing, which then will be reflected in the actual sorting, at no cost. At the beginning of the circuit, instead of computing $\boldsymbol{c}_{\text{rep}} - \boldsymbol{c}_{\text{exp}}$, simply compute the opposite $\boldsymbol{c}_{\text{exp}} - \boldsymbol{c}_{\text{rep}}$.

3.1.2 Correcting repeated elements

The tie-correction offset in [MEHP25] is used to correct the indexing for repeated elements in $\boldsymbol{c}$ and it is computed using some partial results obtained in the indexing phase (using the output of what they call the Eq function). In particular, the goal is to find equal elements in the input vector. We do something similar, but we make the derivative of $\sigma_k(x)$ explicit (in [MEHP25] it is somewhat implicit) and study the propagation of the error during its evaluation. We thus start from the already computed approximation of $\sigma_k(x)$, and use its derivative to perform the equality check. Notice that the derivative of $\sigma_k(x)$ is

$$\sigma_k(x)' := \sigma_k(x) \cdot (1 - \sigma_k(x)), \tag{8}$$

which has exactly the shape we require (i.e., when the difference between two elements is zero, output 1, otherwise output 0), a sample illustration is given in Figure 5. Moreover,

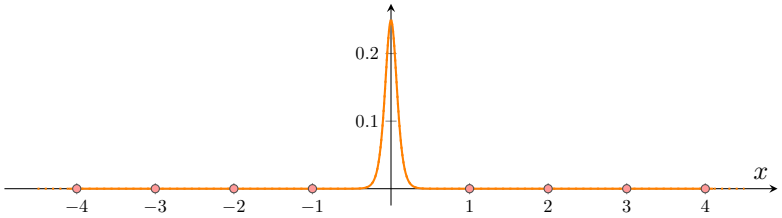


Figure 5: The derivative of $\sigma_k(x)$, for $k = 20$

by recycling the polynomial approximation of $\sigma_k(x)$ used during the indexing phase, this only costs a single multiplication[6]. Additionally, not much precision is lost, namely if in $[-1, -\delta] \cup [0] \cup [\delta, 1]$ the approximation error for $\sigma_k(x)$ is smaller than some $\varepsilon > 0$, its derivative will roughly have the same error, refer to Lemma 3. We only lose some precision due to the scaling of $\sigma_k(x)'$ as it is not equal to 1 in the origin. The loss is less than one order of magnitude, but it should be taken into account when choosing the degrees of the approximations.

Lemma 3 (Stability of the sigmoid-derivative approximation)**.** *Given $0 < \delta < 1$, fix some domain $D = [-1, -\delta] \cup [0] \cup [\delta, 1]$ and let $p(x)$ be some polynomial that satisfies*

$$\|p(x) - \sigma_k(x)\|_\infty \leq \varepsilon$$

for some small $\varepsilon > 0$ and for all $x \in D$. Then

$$\| p(x)(1 - p(x)) - \sigma_k(x)(1 - \sigma_k(x)) \|_\infty \leq \varepsilon(1 + \varepsilon),$$

Proof. Refer to Section B.3 □

By applying this function to the difference between expanded and repeated encodings, we obtain 1 (after scaling) when values are equal and 0 otherwise. Given this information, we can compute the tie-correcting offset. We give a detailed pseudocode showing a possible procedure in the appendix (Algorithm 6). Notice that, since this algorithm is the same as [MEHP25, Algorithm 6], with the difference of the used Eq function, we avoid proof of correctness, that can be found in [MEHP25, Section 4].

[6] During the same step we also scale the approximation of $\sigma_k(x)'$ so that in $x = 0$ it equals 1

3.1.3 Building the corresponding permutation matrix

Let us for a moment ignore the eventual noise contained in the (repeated) indexing vector $\boldsymbol{idx}_{\text{rep}} \in \mathbb{R}^{n^2}$ given by approximate computations.

The procedure to obtain a permutation matrix simply performs an equality check between the indexing vector and n vectors $\boldsymbol{i}^n$, for $0 < i \leq n$. For instance, if the indexing of some element is 3, then the equality check will return 1 only when comparing the value to the number 3, and in other cases it will return 0. That allows to generate the third column of the permutation matrix. More details follow.

In [MEHP25], this equality check is performed by using the equal(x) function — which they call $\mathsf{Ind}_0(x)$ — that is eventually evaluated using approximations of the sgn(x) function. In particular they observed that equal(x) can be implemented as:

$$\text{equal}(x) := \text{sgn}\left(\frac{x+0.5}{r}\right) \cdot \left(1 - \text{sgn}\left(\frac{x-0.5}{r}\right)\right), \tag{9}$$

where the parameter $r > 0$ controls the range of values, and the 0.5 shift gives tolerance to accept values not necessarily very close to 0. We report a plot of this function in Figure 6.

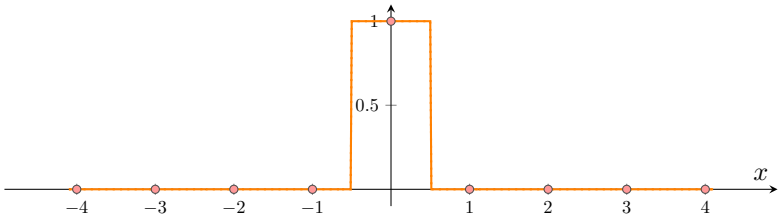


Figure 6: The approximation for equal(x) used in [MEHP25], for $a = 4$

However, we observe that such solution is strongly nonlinear and may be excessively complicated, especially considering that it requires the computation of two sgn(x) functions. Instead, we propose to relax this phase by considering the following.

Lemma 4 (Permutation)**.** *There exists an algorithm that, given any function $f : \mathbb{R} \to \mathbb{R}$ such that $f(0) = 1$ and $f(x) = 0$ for all $x \in \mathbb{Z} \setminus \{0\}$, and the indexing $\boldsymbol{idx}$ of some vector $\boldsymbol{v}$, returns a permutation matrix P such that $P \cdot \boldsymbol{v}$ is sorted.*

Proof. We prove this lemma by showing a possible algorithm. Set $\boldsymbol{x} := \boldsymbol{idx}_{\text{rep}} \in \mathbb{R}^{n^2}$ and assume that its entries are contained in $\mathbb{Z}$ and that the indexing $\boldsymbol{x}$ is correct. For each pair (i, j) with $i, j \in \{1, 2, \ldots, n\}$, compute $r_{i,j} := \boldsymbol{x}_{i,j} - i$. Then, define the permutation P, where each $p_{i,j} := f(r_{i,j})$. By definition of f each entry of P is binary. In particular, $P_{i,j} = 1$ if and only if the j-th element of $\boldsymbol{v}$ must be moved in position i. □

We provide a pseudocode based on a polynomial approximation of $f(x)$ using CKKS primitives in Algorithm 2. Notice that the definition of equal(x) in [MEHP25] respects the requirements of Lemma 4. However, their approach is more generic as the approximation holds for all the values $x + \varepsilon \in \mathbb{R}$, with $x \in \mathbb{Z}$ and $\varepsilon \in [-0.5, 0.5)$ being an "error" term, which introduces some tolerance but at the same time might be excessively generic for some applications. Therefore, we propose to use a simple function that respects the barely minimum requirements of Lemma 4:

$$\text{sinc}(x) := \begin{cases} \sin(x\pi)/(x\pi) & \text{if } x \neq 0 \\ 1 & \text{otherwise} \end{cases} = \prod_{n=1}^{\infty} \left(1 - \frac{x^2}{n^2}\right). \tag{10}$$

Algorithm 2 Generating a permutation matrix with SIMD computations

1: **Input:** A vector $\boldsymbol{idx}_{\text{rep}} \in \mathbb{R}^{n^2}$ containing the repeated indexing of some vector $\boldsymbol{c} \in \mathbb{R}^n$ and a polynomial approximation $p(x)$ of a function $f(x)$ satisfying the constraints in Lemma 4.
2: **Output:** A permutation matrix $P \in \mathbb{R}^{n \times n}$, encoded column-wise, such that $P \cdot \boldsymbol{c} = \text{sort}(\boldsymbol{c})$.
3: **procedure** GeneratePermutation($\boldsymbol{idx}_{\text{rep}}, p(x)$)
4: $\quad \boldsymbol{v} := (1, 2, \ldots, n) \otimes \mathbf{1}_n \in \mathbb{R}^{n^2}$
5: $\quad \boldsymbol{r} := \text{Sub}(\boldsymbol{idx}_{\text{rep}}, \boldsymbol{v})$
6: $\quad$ **return** $p(\boldsymbol{r})$

The latter can be efficiently approximated via Chebyshev polynomials, and it satisfies Lemma 4 as (i) $\text{sinc}(0) = 1$ and (ii) $\text{sinc}(x) = 0$ for every $x \in \mathbb{Z} \setminus \{0\}$. We report its plot in Figure 7.

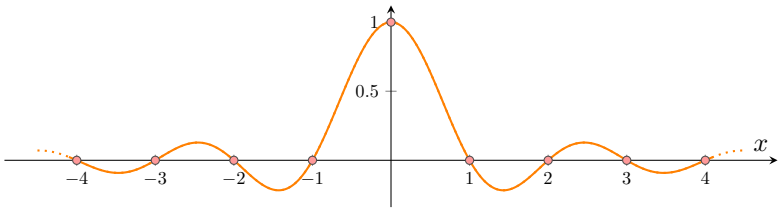


Figure 7: The $\text{sinc}(x)$ function

Its error tolerance is much smaller than the one in Eq. (9), in particular $\text{sinc}(x)$ approximates $\text{equal}(x)$ for all the values $x + \varepsilon \in \mathbb{R}$ for very small values of $\varepsilon > 0$; for this reason we require the indexing vector to be as close as possible to $\mathbb{Z}$. Nonetheless, $\text{sinc}(x)$ is robust enough to handle errors, as it has "cleaning" properties for certain inputs.

Lemma 5 (Cleaning properties of $\text{sinc}(x)$)**.** *For every $x \in \mathbb{Z} \setminus \{\pm 1\}$ and every small $\varepsilon \in \mathbb{R}$ such that $|\varepsilon| < 1$, we have that*

$$\begin{cases} \big|\text{sinc}(x + \varepsilon)\big| \leq |\varepsilon| & \text{if } x \neq 0, \\ \big|\text{sinc}(\varepsilon)\big| \leq |1 + \varepsilon| & \text{if } x = 0. \end{cases}$$

That is, the output error is reduced except in the cases of $x = \pm 1$. In particular, we have that:

$$\text{sinc}(x + \varepsilon) \approx \varepsilon / x.$$

Proof. Refer to Section B.4 □

It can be shown that in the cases of $x = \pm 1$ the error is not reduced, but it is slightly increased, but overall by combining all the different evaluations one obtains a reduction in the total error. We therefore propose to use Chebyshev polynomials to approximate $\text{sinc}(x)$, with degree depending on the number of elements to be sorted, more details will be given in Section 4. This will lead to the permutation matrix P, encoded column-wise as shown in Figure 8.

Remark 2. One might think about using some polynomial that respects requirements of Lemma 4, by simply fitting a polynomial $p(x)$ such that $p(0) = 1$ and $p(x) = 1$ for all

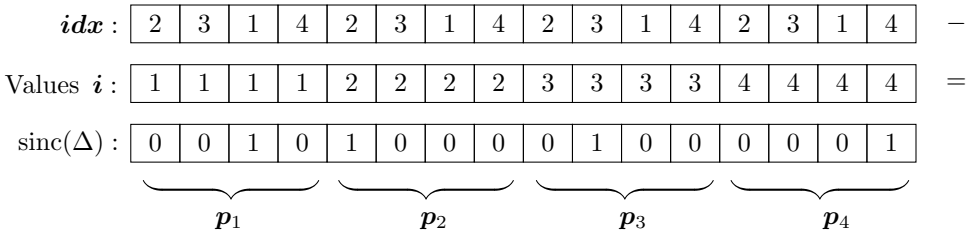


Figure 8: Generating the columns of P using the $\text{sinc}(x)$ function over the differences between each pair $\boldsymbol{s}$ and $\boldsymbol{i}$.

$x \in \{-n, -n+1, \ldots, -1, 1, \ldots, n-1, n\}$. However, the representation of such polynomial in the power basis and in Chebyshev basis are strongly ill-conditioned and hard to be evaluated under CKKS without incurring in plaintext overflows.

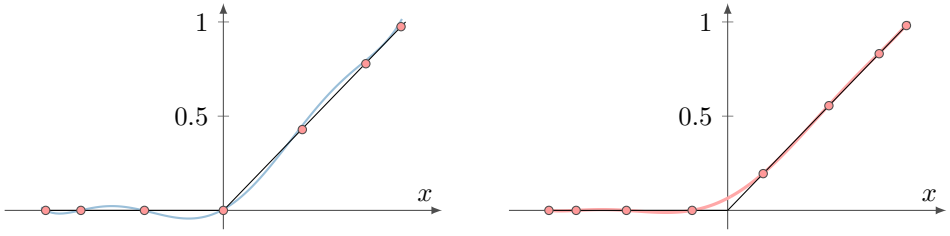
The final sorting is simply computed as a matrix-vector multiplication, which can be done at the cost of one multiplication and $\log(n)$ additions and rotations.

3.2 Network-based sorting

The sorting network relies on comparisons between pairs of values; we refer to Appendix A for an introduction to sorting networks. We propose to evaluate the comparison operation using the following relation:

$$\min(a, b) = a - \text{ReLU}(a - b), \tag{11}$$

where $\text{ReLU}(x)$ is approximated using Chebyshev polynomials. In Figure 9, we present the two different behaviors assumed by the polynomials: when the degree d is even, and when is odd. In general, increasing the degree of the polynomial does not always imply a smaller error (in terms of infinity norm).



(a) Even degree $d = 6$ (b) Odd degree $d = 7$

Figure 9: Approximations of the ReLU function, using Chebyshev interpolants

By computing the maximum error in $x \in [-1, 1]$ as $\|\text{ReLU}(x) - p_d(x)\|_\infty$, where $p_d(x)$ is the Chebyshev approximation with degree d, a much more stable approximation is obtained when d is odd, although the value $x = 0$ will have the largest error. Table 1 presents the depth consumed by evaluating Eq. (2) using Chebyshev polynomials with respect to [LLNK22] – remark that their approach is to evaluate the ReLU function using Eq. (1).

3.2.1 Evaluating a sorting layer via SIMD computations.

As it will be now shown, each layer of the sorting network can be computed in a single SIMD comparison. Therefore, we define the complexity of a sorting network as the number

Table 1: Approximations for the ReLU function. We compare to [LLNK22, Table IV]

Precision bits (α)	Consumed depth	
	[LLNK22]	(**Proposed**)
6	6	5 (using $d = 20$)
7	7	6 (using $d = 38$)
8	8	7 (using $d = 76$)
9	9	8 (using $d = 156$)
10	11	9 (using $d = 310$)

of its layers. In particular, given n inputs (with n being necessarily a power of two), the number of layers of the corresponding sorting network is equal to $(\log(n)(\log(n)+1))/2$. The challenge is to build a circuit that evaluates a sorting network by performing the least possible amount of comparisons and by minimizing the number of consumed levels (i.e., multiplicative depth). We informally introduce an algorithm to evaluate a sorting network layer (namely the *compare-and-swap* operation) using as example a sorting network for eight values. Without loss of generality, the algorithm can be generalized to evaluate any sorting network layer. Given the 1-Bitonic block in Figure 10, composed of a single layer, and a sample vector $\boldsymbol{c} := (5, 4, 1, 8, 2, 9, 1, 2)$. we evaluate it in four steps.

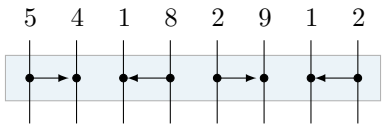


Figure 10: 1-Bitonic block with 8 input values

1. Obtain the minimum between each pair (refer to pairs of elements connected by the arrows in Figure 18) as shown in Eq.(11):

$$\boldsymbol{c}_1 = \underbrace{\boldsymbol{c}}_{a} - \underbrace{p\big(\boldsymbol{c} - \mathrm{Rot}_i(\boldsymbol{c})\big)}_{\mathrm{ReLU}(a-b)}$$

   where $p(x)$ is some polynomial approximation of the $\mathrm{ReLU}(x)$ function (in our experiments we use Chebyshev polynomials, but any approximation can be plugged in), $\boldsymbol{c}$ is the encrypted input vector to be sorted and i is the distance between compared elements; in this example, this is equal to 1. Referring to Fig. 17a, this operation gives as output all wires containing $\min(a, b)$.

2. Obtain the maximum between each pair: recalling that $\max(a, b) = a + b - \min(a, b)$, it is possible to obtain these values as:

$$\boldsymbol{c}_2 = \Big(\underbrace{\boldsymbol{c}}_{a} + \underbrace{\mathrm{Rot}_{-i}(\boldsymbol{c})}_{b} \Big) - \underbrace{\mathrm{Rot}_{-i}(\boldsymbol{c}_1)}_{\min(a,b)}$$

   This gives us the comparisons between the values that in Figure 18 are compared by an arrow pointing down. Considering the previous example, we computed the values in the bottom part of following the 1-bitonic block (Figure 11).

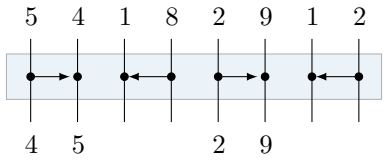


Figure 11: Values obtained in $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$

In particular, $\boldsymbol{c}_1$ contains the minimum values (4 and 2), while $\boldsymbol{c}_2$ the maximum ones (5 and 9).

3. Is is not necessary to evaluate the minimum function again, since the minimum values of the remaining wires (whose arrows are pointing left) have already been computed and are available in $\boldsymbol{c}_1$; they just need to be rotated:

$$\boldsymbol{c}_3 = \mathrm{Rot}_{-i}(\boldsymbol{c}_1)$$

4. As before, the maximum values of the remaining wires are computed using the previously computed minimum:

$$\boldsymbol{c}_4 = (\boldsymbol{c} + \mathrm{Rot}_i(\boldsymbol{c})) - \boldsymbol{c}_1$$

At this point, all the required elements have been computed, and can be obtained by summing up the four ciphertexts $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ and $\boldsymbol{c}_4$ (Figure 12).

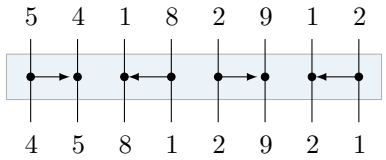


Figure 12: Complete evaluation of a sorting layer

However, before doing that, unnecessary values, for each partial ciphertext $\boldsymbol{c}_i$, must be cancelled (i.e., multiplied by zero), otherwise incorrect values will ruin the results. Think, as an example, to the second slot of $\boldsymbol{c}_1$, which will contain partial computations to be removed before the sum. The EvaluateLayer procedure is formally presented in Algorithm 3.

Algorithm 3 Evaluation of a sorting network layer

procedure EvaluateLayer($\boldsymbol{c}, n, \alpha, b, \ell$)
  $\boldsymbol{c}_1 \leftarrow \min(\boldsymbol{c}, \mathrm{Rot}_\alpha(\boldsymbol{c}))$ ▷ Approximate comparison function
  $\boldsymbol{c}_2 \leftarrow \big(\boldsymbol{c} + \mathrm{Rot}_{-\alpha}(\boldsymbol{c})\big) - \boldsymbol{c}_1$
  $\boldsymbol{c}_3 \leftarrow \mathrm{Rot}_{-\alpha}(\boldsymbol{c}_1)$
  $\boldsymbol{c}_4 \leftarrow \big(\boldsymbol{c} + \mathrm{Rot}_\alpha(\boldsymbol{c})\big) - \boldsymbol{c}_1$
  $\boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3, \boldsymbol{m}_4 \leftarrow$ GenMasks(n, b, ℓ)
  return $\mathrm{Add}(\mathrm{Mul}(\boldsymbol{c}_1, \boldsymbol{m}_1), \mathrm{Mul}(\boldsymbol{c}_2, \boldsymbol{m}_2), \mathrm{Mul}(\boldsymbol{c}_3, \boldsymbol{m}_3), \mathrm{Mul}(\boldsymbol{c}_4, \boldsymbol{m}_4))$

This procedure takes as input a ciphertext $\boldsymbol{c}$, the value of α (namely the length of the arrows), the current bitonic block b and the layer $0 < \ell \leq b$ of the current block. It computes the four ciphertexts $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ and $\boldsymbol{c}_4$, implementing the procedure described above. It relies on the GenMasks procedure, described in Algorithm 4, which returns a set of four masks to be applied to the different $\boldsymbol{c}_i$. This set of masks strongly depends on the current network layer, and they are constructed according to the length of the arrows and

Algorithm 4 Generation of the masks for the comparisons c_i

$\boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3, \boldsymbol{m}_4 \leftarrow$ empty vectors
procedure GenMasks(n, b, ℓ)
  while size($\boldsymbol{m}_1$) $< n$ **do**
    for 2^ℓ times **do**
      for $2^{b-\ell}$ times **do** PushMask(1)
      for $2^{b-\ell}$ times **do** PushMask(2)
    if size($\boldsymbol{m}_1$) $+ 2^b >= n$ **then break**
    for 2^ℓ times **do**
      for $2^{b-\ell}$ times **do** PushMask(3)
      for $2^{b-\ell}$ times **do** PushMask(4)
  return $\boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3, \boldsymbol{m}_4$
procedure PushMask(idx)
  for $j \leftarrow 1$ to 4 **do**
    if j = idx **then** push 1 to $\boldsymbol{m}_j$ **else** push 0 to $\boldsymbol{m}_j$

Algorithm 5 SIMD-based bitonic sorting of $\boldsymbol{c}$

$n \leftarrow$ length of $\boldsymbol{c}$
for $b \leftarrow 1$ **to** $\lceil \log(n) \rceil$ **do** ▷ b is the current block
  for $\ell \leftarrow 1$ **to** $b + 1$ **do** ▷ ℓ is the current layer
    $\alpha \leftarrow 2^{(i-j)}$ ▷ δ is the length of the arrows
    $\boldsymbol{c} \leftarrow$ EvaluateLayer($\boldsymbol{c}, n, \alpha, b, \ell$)

their direction. Putting these procedures together, it is possible to build an SIMD-based encrypted bitonic sorting circuit (Algorithm 5). In our setup, each EvaluateLayer operation is followed by a CKKS bootstrapping operation on $\boldsymbol{c}$ which refreshes the level of the ciphertext and enables the evaluation of the subsequent layer.

4 Experiments

As above, we split this section in two parts: in the first we present experiments based on the permutation-based approach, in the second one on the network-based approach. Each experiment uses the OpenFHE implementation of the CKKS [ABB+22; KPP22] scheme, a security level of $\lambda \geq 128$ bits, rings of size $N = 2^{16}$ and sparse-key encapsulation method [BTH22]. All experiments have been run on a Macbook Pro with M4 Max and 36 GB of memory and are replicable by running the corresponding script in the experiments folder, contained in our open-source repository[7]. The latter also contains Python notebooks that implement the circuits executed on clear data, useful to better understand their workings.

We evaluate our sorting as follows: given the clear vector $\boldsymbol{v} \in \mathbb{R}^n$ and the decrypted vector (after the evaluation of the circuit) $\boldsymbol{c} \in \mathbb{R}^n$, we say that c_i is correct if its absolute difference with v_i is smaller than δ, i.e., if

$$|c_i - \text{sort}(\boldsymbol{v})_i| \leq \delta.$$

The sorting is said to be correct if this holds for all c_i, with $0 < i \leq n$.

[7] https://github.com/lorenzorovida/Lightweight-Sorting-In-Approximate-Homomorphic-Encryption

4.1 Permutation-based

The main tunable parameters of the cryptosystem used in these experiments are: (i) the degree d_1 of the polynomial approximation for $\sigma_k(x)$ function, (ii) its scaling factor k, (iii) the number of cleaning operations after $\sigma_k(x)$, (iv) the degree of the approximation of the $\text{sinc}(x)$ function d_2 and lastly (v) the number of cleaning operations after $\text{sinc}(x)$.

All experiments are run with tie correction active. To stress-test our method under various conditions, we run three types of experiments for each parameters set using different inputs: (i) elements randomly sampled from $[0, 1]$ with minimum distance $\delta > 0$, (ii) sequentially spaced elements $[0, \delta, 2\delta, \dots]$ and (iii) repeated equal elements $[\delta, \delta, \dots]$.

Remark 3. The polynomial approximations in the actual implementation are always performed without loss of generality over the $[-1, 1]$ interval, as this allows to save one multiplication. The indexing vector will thus not live in $\mathbb{Z}^n$ but in $\delta\mathbb{Z}^n$. This is simply an implementation choice that does not change the output, but makes the computation faster.

Experiment 1, comparison with [MEHP25; KÜYL25] on single ciphertexts. In the first experiment, we define the same setup as in [MEHP25; KÜYL25] in single-ciphertext mode, namely number of values n ranging from 8 up to 128 — our small ring does not allow to encrypt up to 256 values — and minimum distance between pair of elements $\delta = 0.01$, with tie-correction active. The degree d_2 of the $\text{sinc}(x)$ approximation depends on the number of values n (remark that this function is evaluated on the indexing vector), while the degree d_1 and the scaling factor k depend on δ. For every run, values have been sampled randomly from a discretization of $[0, 1]$, with step size equal to δ, with possible repeated values, whose indexing is corrected by the tie-correction offset. We present in Table 2 the parameters used in the various tests.

Table 2: Parameters used in the first permutation-based sorting. Degrees d_1, d_2 refer to the Chebyshev polynomials approximations. The cleaning polynomials $c(x)$ are evaluated after the function in the previous column

	CKKS parameters			Polynomials parameters				
n	Δ	Depth	$\log(QP)$	$\sigma_k(x)$ scale k	$\sigma_k(x)$ degree d_1	Number of $c(x)$	$\text{sinc}(x)$ degree d_2	Number of $c(x)$
8	2^{38}	22	1238	650	495	2	59	0
16	2^{38}	23	1276	360	495	2	119	0
32	2^{38}	26	1390	360	495	2	247	1
64	2^{38}	27	1488	360	495	2	495	1
128	2^{38}	29	1564	360	495	2	495	2

As reported in Figure 13, computational times and memory requirements are always lower than in [MEHP25; KÜYL25]. The main factor that reduces both time and memory is the choice of the ring $N = 2^{16}$, which can be safely used since the moduli required by our circuits are always notably below the threshold for 128-bits of classical security — roughly 2^{1772} — established by the Lattice Estimator [APS15]. Curiously, our runtimes and memory requirements does not grow much with respect to the number of elements n. This is a natural consequence of the fact that the complexity of the circuit mainly depends on the factor δ, which in turn defines the degree d_1 of the polynomial approximations of $\sigma_k(x)$, which is the main bottleneck of the circuit. On the other hand, increasing n results in larger degrees d_2, which are often small with respect to the ones used for $\sigma_k(x)$.

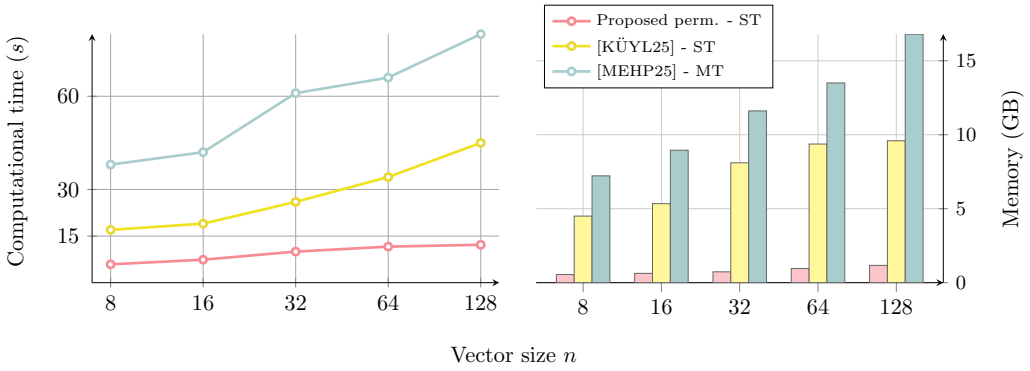


Figure 13: Comparison on computational time and memory consumption between our proposal and [MEHP25; KÜYL25]. ST and MT stand for single and multithread. Our experiments have been run on a M4 Max CPU, while we reported times from both other experiments from [KÜYL25, Table 1] that used an Intel Xeon Gold 6226R. Memory requirements have been obtained using the public source code of [KÜYL25]

Experiment 2, more discretization and stress tests. We now aim to increase the granularity of the considered $[0, 1]$ interval from $\delta = 0.01$ to $\delta = 0.001$. Moreover, we want to stress our circuits with worst case inputs, namely all equal elements and all elements at distance δ. The first stresses the tie correction offset, the second the polynomial approximation of $\sigma_k(x)$, that close to δ has its largest errors. To keep the modulus small, we set $d_{\text{num}} = 3$ (see [KPP22, Section 2] for an explanation of that parameter).

In order to achieve a correct sorting, we must increase the degrees of approximations of the nonlinear functions and the number of cleanings. Luckily, from the previous experiment, we still have room to increase the circuit depth maintaining $\lambda \geq 128$ bits of security without requiring an increment in the ring size. Refer to Table 3.

Table 3: Parameters used in permutation-based sorting experiments. Degrees d_1, d_2 refer to the Chebyshev polynomials approximation. The modulus marked with * is smaller as for that setup d_{num} has been set to 4

	CKKS parameters			Polynomials parameters				
n	Δ	Depth	$\log(QP)$	$\sigma_k(x)$ scale k	$\sigma_k(x)$ degree d_1	Number of $c(x)$	$\text{sinc}(x)$ degree d_2	Number of $c(x)$
8	2^{38}	26	1390	2400	2031	3	59	0
16	2^{38}	27	1488	2400	2031	3	119	0
32	2^{38}	30	1662	2400	2031	3	247	1
64	2^{38}	31	1700	2400	2031	3	495	1
128	2^{38}	33	1656*	2400	2031	3	495	2

All the experiments were successful on all the run stress tests, we report in Figure 14 the runtimes and the (peak) memory requirements, compared to the previous case. Results confirm that the runtime is somehow stable regardless of the vector size n; it only fluctuates by a few seconds at most. Also memory requirements are always around 3 GB, never larger than 4 GB.

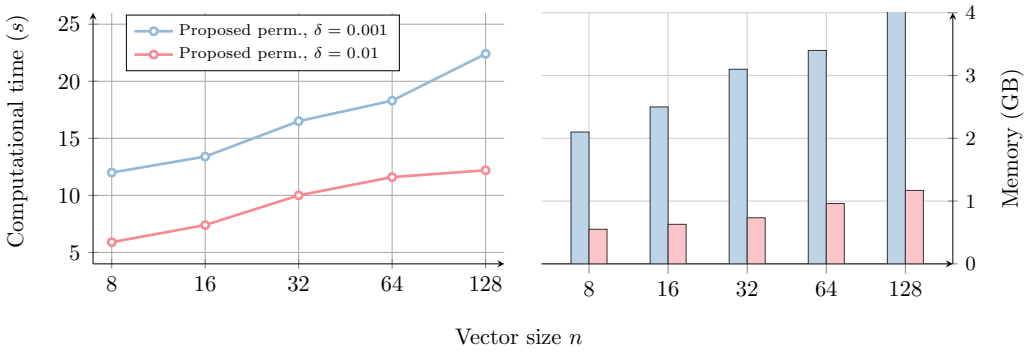


Figure 14: Time and memory requirements when sorting elements at minimum distance $\delta = 0.001$, compared to $\delta = 0.01$. The experiments have been tested on different inputs: random, uniformly equal and sequential

4.2 Network-based

We now aim to perform similar experiments as above, but based on the bitonic sorting network. Notice that the key material of the scheme required in such experiments will be drastically larger — and in turn, larger memory requirements — as a consequence of the bootstrapping operation, run after each layer evaluation. This also requires larger values of Δ that allow to reduce the new errors introduced by the bootstrapping procedure

Experiment 1, sorting at distance $\delta = 0.01$. Differently from before, we can now sort up to 8192 elements without relying on multiple ciphertexts, since the number of required slots by the sorting network is simply $\Theta(n)$. The set of parameters is fixed for any value of n, since the only nonlinear function is $\mathrm{ReLU}(x)$, whose precision depends on δ. We present in Table 4 the parameters used in this phase.

Table 4: Parameters used in network-based sorting experiments.

	CKKS parameters				Data parameters
n	Δ	Circuit depth	$\log(QP)$	Bootstrapping depth	$\mathrm{ReLU}(x)$ degree d
from 8 to 8192	2^{51}	27	1730	16	495

We present in Figure 15 the results comparing to the permutation-based sorting approach and to [MEHP25]. We also include, for the sake of completeness, a comparison with the CKKS-based sorting network in [HKC+21], based on approximations of the $\mathrm{sgn}(x)$ function.

Our network-based approach does not represent a significant improvement with respect to the permutation-based sorting in [MEHP25], although it is orders of magnitude faster than the network in [HKC+21]. Despite not being significantly faster, it can be considered as a decent alternative to permutation-based as it can be used on large amounts of elements without assuming to have n^2 slots available.

Even though the network-based approach requires larger amounts of memory with respect to the premutation-based one (mainly because of the bootstrapping operation), the $N = 2^{16}$ ring allows to stay competitive with [MEHP25], as illustrated in Figure 16.

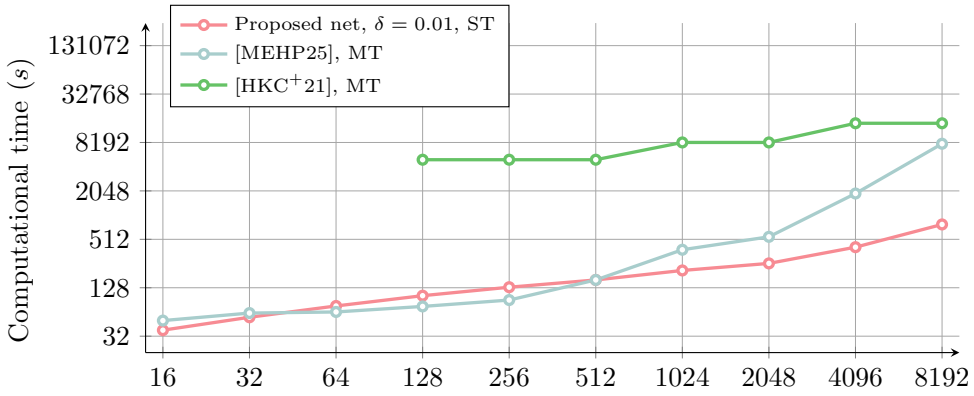


Figure 15: Comparing our network-based approach with [MEHP25; HKC+21] in terms of runtime. Our runtimes are obtained using a M4 Max with 36GB of RAM, the others are reported from the corresponding papers

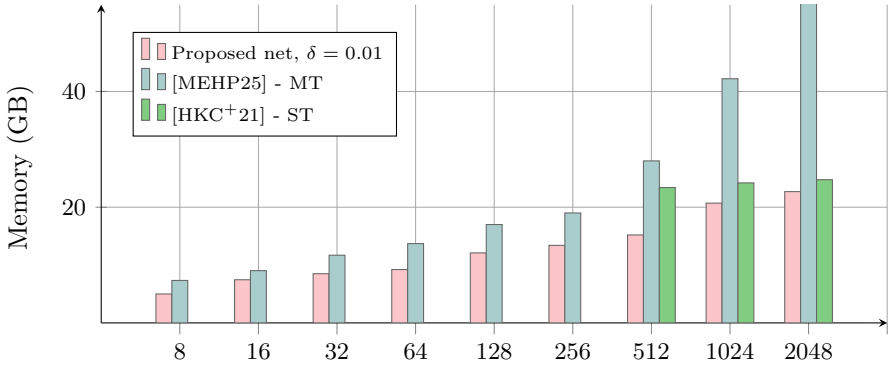


Figure 16: Comparing our network-based approach with [MEHP25; HKC+21] in terms of memory. The last cut bar is valued 97 GB, cut for readability

5 Conclusions and open questions

We proposed two approaches to sort encrypted values under the CKKS scheme. In particular, the permutation-based approach is very quick and do not have large memory requirements, but it has the drawback of requiring $\Theta(n^2)$ slots in order to efficiently evaluate the SIMD operations. On the other hand, the network-based approach is slower and requires more memory, but has the advantage of requiring only $\Theta(n)$ slots, making it more suitable in certain applications. It would be interesting to explore different ways to tackle the procedures in the permutation-based sorting – using $\mathrm{sinc}(x)$ is just one of them. It would be also interesting to explore new functions to use in the sorting network, different from $\mathrm{ReLU}(x)$, which might lead to shorter circuits and lighter computations.

An interesting open point is about dealing with large amount of values in the permutation-based approach. In [MEHP25], they propose to split the input vector in multiple ciphertexts, while one idea for instance could be to build a hybrid sorting algorithm where each sub-vector is partially sorted using permutations, and the whole vector is reconstructed by evaluating only the last block of a bitonic network, which takes as input sequences that are partially sorted. Another idea for future work is about using ideas from [YNW+25] to put together ReLU and bootstrapping, leading to a faster sorting network that can hopefully be competitive with respect to the permutation-based sorting.

References

[ABB+22] Ahmad Al Badawi et al. Openfhe: open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, 2022. DOI: 10.1145/3560827.3563379.

[Akl11] Selim G. Akl. *Bitonic sort*. In *Encyclopedia of Parallel Computing*. Springer US, 2011, pages 139–146. DOI: 10.1007/978-0-387-09766-4_124.

[APS15] 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. DOI: 10.1515/jmc-2015-0016.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, Atlantic City, New Jersey, 1968. DOI: 10.1145/1468075.1468121.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966. DOI: 10.1145/355592.365646.

[BTH22] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022*, pages 521–541, 2022. DOI: 10.1007/978-3-031-09234-3_26.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2019*, pages 34–54, 2019. DOI: https://doi.org/10.1007/978-3-030-17656-3_2.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. DOI: 10.1007/s00145-019-09319-x.

[CGKP25] Kelong Cong, Robin Geelen, Jiayi Kang, and Jeongeun Park. Revisiting Oblivious Top-k Selection with Applications to Secure k-NN Classification. In *Selected Areas in Cryptography – SAC 2024: 31st International Conference, Montreal, QC, Canada, August 28–30, 2024, Revised Selected Papers, Part I*, pages 3–25, 2025. DOI: 10.1007/978-3-031-82852-2_1.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2018*, pages 360–384, 2018. DOI: 10.1007/978-3-319-78381-9_14.

[CKK20] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *Advances in Cryptology – ASIACRYPT 2020*, pages 221–256, 2020. DOI: 10.1007/978-3-030-64834-3_8.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham. Springer International Publishing, 2017. ISBN: 978-3-319-70694-8. DOI: 10.1007/978-3-319-70694-8_15.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN: 0070131511.

[CYW+25] Zehao Chen, Honghui You, Qian Wei, Hang Lu, Lei Ju, and Zhaoyan Shen. Smartpir: a private information retrieval system using computational storage devices. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, pages 1749–1762, 2025. DOI: 10.1145/3725843.3756060.

[DMPS23] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. Bleach: cleaning errors in discrete computations over ckks. *J. Cryptol.*, 37(1), November 2023. DOI: 10.1007/s00145-023-09483-1.

[GEMR25] Karthik Garimella, Austin Ebel, Gabrielle De Micheli, and Brandon Reagen. He-lrm: encrypted deep learning recommendation models using fully homomorphic encryption, 2025. arXiv: 2506.18150 [cs.CR].

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, 2009. DOI: 10.1145/1536414.1536440.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In *Advances in Cryptology – CRYPTO 2013*, pages 536–553, 2013. DOI: 10.1007/978-3-642-40084-1_30.

[HKC+21] 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*, 16:4389–4404, 2021. DOI: https://doi.org/10.1109/tifs.2021.3106167.

[HWW+22] Hai Huang, Yongjian Wang, Luyao Wang, Huasheng Ge, and Qiang Gu. Secure word-level sorting based on fully homomorphic encryption. *J. Inf. Secur. Appl.*, 71(C), December 2022. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2022.103372.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN: 0201896850.

[KPAM20] Shristi Shakya Khanal, P. W. C. Prasad, Abeer Alsadoon, and Angelika Maag. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies*, 25(4):2635–2664, July 2020. DOI: 10.1007/s10639-019-10063-9.

[KPP22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. In Steven D. Galbraith, editor, *Topics in Cryptology – CT-RSA 2022*, pages 120–144, 2022. DOI: 10.1007/978-3-030-95312-6_6.

[KÜYL25] Seunghu Kim, Eymen Ünay, Ayse Yilmazer-Metin, and Hyung Tae Lee. Optimized rank sort for encrypted real numbers. Cryptology ePrint Archive, Paper 2025/1170, 2025.

[LHH+21] 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 (SP)*, pages 1057–1073. IEEE, 2021. DOI: 10.1109/sp40001.2021.00043.

[LLKN22] Eunsang Lee, Joon-Woo Lee, Young-Sik Kim, and Jong-Seon No. Optimization of homomorphic comparison algorithm on rns-ckks scheme. *IEEE Access*, 10:26163–26176, 2022. DOI: 10.1109/access.2022.3155882.

[LLNK22] 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*, 19(6):3711–3727, 2022. DOI: 10.1109/tdsc.2021.3105111.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 595–608, 2023. DOI: 10.1145/3564246.3585175.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. Near-optimal private information retrieval with preprocessing. In *Theory of Cryptography Conference*, pages 406–435, 2023. DOI: 10.1007/978-3-031-48618-0_14.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), November 2013. DOI: 10.1145/2535925.

[MEHP25] Federico Mazzone, Maarten Everts, Florian Hahn, and Andreas Peter. Efficient ranking, order statistics, and sorting under ckks. In *34th USENIX Security Symposium (USENIX Security '25)*, August 2025. DOI: 10.1007/978-981-95-3182-0_18.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2(02n03):205–211, 1992. DOI: 10.1142/s0129626492000337.

[PHE+25] Thomas Prantl, Lukas Horn, Simon Engel, André Bauer, and Samuel Kounev. Quo vadis ckks: comparison of the realization of basic mathematical functions for the homomorphic cryptosystem ckks using de bello and polynomial approximations. *Journal of Information Security and Applications*, 93:104110, 2025. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2025.104110.

[PS73] Michael S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. DOI: https://doi.org/10.1137/0202007.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Tre19] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2019. DOI: 10.1137/1.9781611975949.

[WZD+25] Xuan Wang, Minxuan Zhou, Gabrielle De Micheli, Yujin Nam, Sumukh Pinge, Augusto Vega, and Tajana Rosing. Pathe: a privacy-preserving database pattern search platform with homomorphic encryption. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2025. DOI: 10.1109/iccad66269.2025.11240997.

[YNW+25] Zhaomin Yang, Chao Niu, Benqiang Wei, Zhicong Huang, Cheng Hong, and Tao Wei. RBOOT: accelerating homomorphic neural network inference by fusing ReLU within bootstrapping. Cryptology ePrint Archive, Paper 2025/1534, 2025.

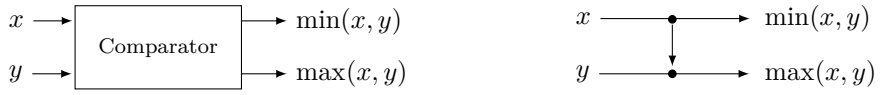
[Zha25] Dongfang Zhao. Hermes: high-performance homomorphically encrypted vector databases, 2025. arXiv: 2506.03308 [cs.CR].

A Sorting networks

Generally, most of the sorting algorithms are *data-dependent*. This means that their behavior depends on the input values. As a consequence, it is possible to define *best* and *worst* case complexities. Nevertheless, this approach can not be pursued in encrypted circuits: if one would be able to compare two ciphertexts and change the instruction flow according to the content of the ciphertexts, the IND-CPA security notion would be broken. A possible solution to this problem is represented by *oblivious algorithms*. This family of algorithms always performs the same sequence of instructions, independent of the input. *Sorting networks* are an example of oblivious sorting algorithms; the only operation that depends on the input data is the *swap* operation, performed if two elements are not in the correct order. Refer to Knuth [Knu98] for a detailed introduction. In Cormen *et al.* [CLRS01], they suggest that a *compare-and-swap* operation defined as:

$$\textbf{if } x > y \textbf{ then } \text{swap}(x, y)$$

can be hardware implemented as shown in Fig. 17a. In the context of Sorting networks, this operation is usually represented as illustrated in Fig. 17b.



(a) In hardware (b) In a Sorting network

Figure 17: Representations of the *compare-and-swap* operation

In literature [HKC+21], this operation has been implemented using approximations of the *sgn* function to replicate the comparison $a > b$; we instead propose to use Eq. (2). It is possible to find many approaches to Sorting networks, such as the *Bitonic sorter* [Akl11], the *Batcher odd-even Mergesort [Bat68]*, and *Pairwise Sorting Network* [Par92], each of them constructs networks of depth $\mathcal{O}(\log^2(n))$. These algorithms are well-suited to the context of CKKS since they are highly parallelizable. CKKS is based on the SIMD computational paradigm, making it possible to evaluate one layer of the network at each step.

Bitonic sorting

In this paper, we propose to build sorting networks using the *Bitonic sorter* algorithm [Akl11]. In particular, given n inputs (with n being a power of two), the number of layers (that are defined below) of the corresponding sorting network will be equal to:

$$\frac{\log(n)(\log(n)+1)}{2} \tag{12}$$

Informally, the bitonic sorting algorithm constructs a sorting network that first sorts every pair of consecutive elements, then every block of four elements, and continues doubling the block size until the entire sequence is sorted. Let us introduce some definitions that will be useful later.

Definition 4 (Bitonic sequence)**.** A sequence $S = (x_0, x_1, \ldots, x_{n-1})$ is called *bitonic* if there exists an index $i \in \{0, \ldots, n-1\}$ such that

$$x_0 \leq x_1 \leq \cdots \leq x_i \geq x_{i+1} \geq \cdots \geq x_{n-1}.$$

In practice, a sequence is bitonic if it is the concatenation of a monotone non-decreasing sequence and a monotone non-increasing sequence.

Definition 5 (k-bitonic sequence)**.** Let $k \in [0, \dots, n/2]$. A sequence $S = (x_0, x_1, \dots, x_{n-1})$ is called a *k-bitonic sequence* if it can be partitioned into k contiguous subsequences of equal length

$$S = B_1 \| B_2 \| \cdots \| B_k,$$

such that each subsequence B_j is bitonic. For $k = 0$, a k-bitonic sequence is defined to be a monotone non-decreasing sequence. The maximal value of $k = n/2$ corresponds to a completely unsorted sequence.

Definition 6 (Sorting layer)**.** A *sorting layer* is a set of compare-and-swap operations (comparators) that act on pairwise disjoint indices of a sequence and can therefore be executed in parallel.

Definition 7 (k-bitonic block)**.** Let k be a power of two. A *k-bitonic block* is a sequence of sorting layers that takes as input a k-bitonic sequence and produces as output a $\lfloor k/2 \rfloor$-bitonic sequence of the same length.

A network that sorts n values will be composed of $\log(n)$ Bitonic blocks, as each bitonic block transforms a k-bitonic sequence into a $(k/2)$-bitonic sequence. For instance, Fig. 18 illustrates a network that sorts $n = 8$ input values.

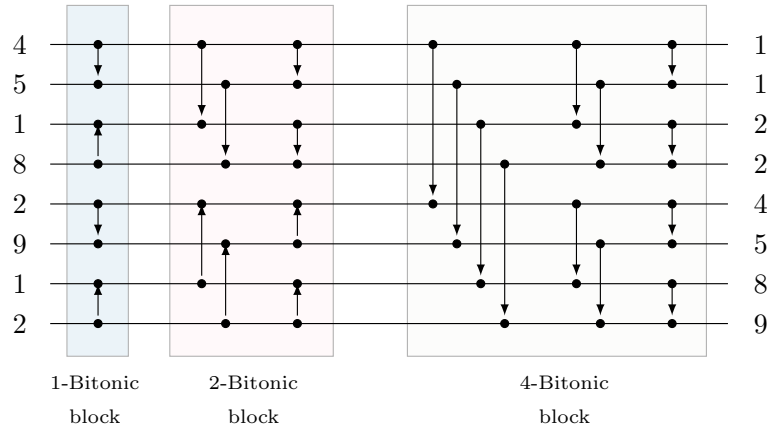


Figure 18: A Sorting network composed of 6 layers that sorts $n = 8$ input values constructed using Bitonic Sorter. Notice that the direction of the arrows defines the order in which the elements are compared

Example 1 (Evolution of bitonicity under bitonic blocks)**.** Consider the sequence of length $n = 8$ in Figure 18:

$$S = (4, 5, 1, 8, 2, 9, 1, 2).$$

The sequence is non-bitonic, so a $n/2$-bitonic sequence. As a consequence, the first block will first produce a $n/4$-bitonic sequence. The 1-bitonic block operates on adjacent pairs of elements, producing two bitonic subsequences of length 4:

$$\underset{\nearrow}{(4,5)} \mid \underset{\searrow}{(8,1)} \parallel \underset{\nearrow}{(2,9)} \mid \underset{\searrow}{(2,1)}.$$

Hence, the resulting sequence is a 2-bitonic sequence. Next, the 2-bitonic block operates on pairs of adjacent bitonic subsequences of length 2, producing one bitonic sequence of length 8:

$$\underset{\nearrow}{(1,4,5,8)} \mid \underset{\searrow}{(9,2,2,1)}.$$

Thus, the sequence becomes a 1-bitonic sequence.

Eventually, the 4-bitonic block operates on the entire sequence and produces the sorted sequence.

$$\overset{\nearrow}{(1,1,2,2,4,5,8,9)}.$$

Therefore, the resulting sequence is a 0-bitonic sequence. Overall, the application of bitonic blocks reduces the degree of bitonicity while preserving the sequence length, according to the transformation

$$k=4 \xrightarrow{\text{1-bitonic block}} k=2 \xrightarrow{\text{2-bitonic block}} k=1 \xrightarrow{\text{4-bitonic block}} k=0.$$

B Proofs of lemmas

B.1 Lemma 1

Proof. Let the Chebyshev nodes of the first kind be

$$x_i = \cos\left(\frac{(i+\frac{1}{2})\pi}{d+1}\right), \qquad i=0,\dots,d.$$

Two distinct cases arise depending on the parity of the degree. If $d=2m$, then the central node is

$$x_m = \cos\left(\frac{(m+\frac{1}{2})\pi}{d+1}\right) = \cos\left(\frac{(\frac{d}{2}+\frac{1}{2})\pi}{d+1}\right) = \cos\left(\frac{\pi}{2}\right) = 0.$$

Since p_d interpolates σ_k at the Chebyshev nodes, we have

$$p_d(0) = p_d(x_m) = \sigma_k(x_m) = \sigma_k(0) = 0.5.$$

This proves the first case. On the other hand, if the degree is odd, i.e., $d=2m+1$, the Chebyshev nodes are symmetric with respect to the origin:

$$x_{d-i} = -x_i, \qquad i=0,\dots,d.$$

Therefore, for each node x_i,

$$p_d(x_i) = \sigma_k(x_i), \qquad p_d(-x_i) = \sigma_k(-x_i) = 1-\sigma_k(x_i).$$

Now, let us define the polynomial

$$q_d(x) = 1 - p_d(-x).$$

Then for every Chebyshev node x_i,

$$q_d(x_i) = 1 - p_d(-x_i) = 1 - \sigma_k(-x_i) = \sigma_k(x_i),$$

so q_d interpolates the same data as p_d at the $d+1$ Chebyshev nodes. Since a polynomial of degree d is uniquely determined by its values at $d+1$ distinct points, we conclude that $q_d(x) = p_d(x)$. In particular $p_d(0) = q_d(0) = 1 - p_d(0)$, so it must hold that

$$p_d(0) = 0.5.$$

Thus, in both cases, $p_d(0) = 0.5$. □

B.2 Lemma 2

Proof. Lines 4-5 compute the following:

$$s_{ij} = p(c_i - c_j) \approx \begin{cases} 1, & c_i > c_j + \delta, \\ 0, & |c_i - c_j| \leq \delta, \\ -1, & c_i < c_j - \delta. \end{cases}$$

For the sake of the proof we assume s_{ij} to be error-free. After the rotate and sum procedure (Line 7), each block for index i contains

$$s_i = \sum_{j=1}^{n} s_{ij} = g_i - l_i, \tag{13}$$

where

$$g_i = |\{j : c_i > c_j + \delta\}| \quad \text{and} \quad l_i = |\{j : c_i < c_j - \delta\}|,$$

where g and l stand for greater and less, respectively. Let

$$e_i = |\{j : |c_i - c_j| \leq \delta\}|$$

be the number of ties (including $j = i$). Notice that in the case of no repeated elements we have $e_i = 1$ for all $i < n$. Observe that, since the three sets (greater, less, equal) are disjoint and cover all indices, we have

$$g_i + l_i + e_i = n.$$

Solving for g_i, and replacing l_i using (13), yields:

$$g_i = \frac{n - e_i + s_i}{2}.$$

The rank of c_i with ties (as defined in Definition 3) is

$$\text{rank}(c_i) = n - g_i - \frac{e_i - 1}{2} = \frac{n + 1 - s_i}{2}. \tag{14}$$

Finally, the algorithm outputs

$$\boldsymbol{idx} = \mathbf{s} + \frac{1}{2},$$

so that, by combining (13) and (14):

$$\boldsymbol{idx}_i = \frac{n + 1 - s_i}{2} = \text{rank}(c_i).$$

Hence, the returned vector correctly assigns the indexing to all the values of the vector. Since each s_{ij} is approximate, so it will be $\boldsymbol{idx}_i$. □

B.3 Lemma 3

Proof. For each $x \in D$ we compute the difference between the exact and the approximate derivative:

$$p(x)(1 - p(x)) - \sigma_k(x)(1 - \sigma_k(x)) = (p(x) - \sigma_k(x))(1 - (p(x) + \sigma_k(x))).$$

Taking absolute values gives

$$|p(x)(1 - p(x)) - \sigma_k(x)(1 - \sigma_k(x))| \leq |p(x) - \sigma_k(x)|\,|1 - (p(x) + \sigma_k(x))|.$$

Since $\|p(x) - \sigma_k(x)\|_\infty \leq \varepsilon$ in D by assumption, we have $|p(x) - \sigma_k(x)| \leq \varepsilon$ for all $x \in D$. Also, $0 \leq \sigma_k(x) \leq 1$ and $|p(x) - \sigma_k(x)| \leq \varepsilon$ imply $|1 - (p(x) + \sigma_k(x))| \leq 1 + \varepsilon$. Thus for all $x \in D$,

$$|p(x)(1 - p(x)) - \sigma_k(x)(1 - \sigma_k(x))| \leq \varepsilon(1 + \varepsilon).$$

Therefore, the ℓ_∞ used to compute the approximation error in D is such that:

$$\|p(x)(1 - p(x =)) - \sigma_k(x)(1 - \sigma_k(x))\|_\infty \leq \varepsilon(1 + \varepsilon).$$

□

B.4 Lemma 5

Proof. For the first case, given

$$|\mathrm{sinc}(x + \varepsilon)| = \left|\frac{\sin((x + \varepsilon)\pi)}{\pi(x + \varepsilon)}\right| = \left|\frac{\sin(x\pi + \varepsilon\pi)}{\pi(x + \varepsilon)}\right|,$$

and since x is integer, we have that $|\sin(\pi x + \pi\varepsilon)| = |\sin(\pi\varepsilon)|$, also observe that $|\sin(\pi\varepsilon)| \leq |\pi\varepsilon|$, so we can bound $|\mathrm{sinc}(x + \varepsilon)|$ as:

$$|\mathrm{sinc}(x + \varepsilon)| \leq \left|\frac{\pi\varepsilon}{\pi(x + \varepsilon)}\right| = \left|\frac{\varepsilon}{(x + \varepsilon)}\right|.$$

We can thus loosely reduce the original statement to:

$$\frac{|\varepsilon|}{|(x + \varepsilon)|} \leq |\varepsilon|$$

that is true as long as $|x| > 1$. In particular, the distance grows linearly in x and in particular we have that

$$\mathrm{sinc}(x + \varepsilon) \approx \varepsilon/x,$$

with approximation given by the fact that we used $|\sin(\pi\varepsilon)| \leq |\pi\varepsilon|$. This proves the first case. For the case of $x = 0$, simply observe that

$$|\mathrm{sinc}(\varepsilon)| = \frac{|\sin(\pi\varepsilon)|}{|\pi\varepsilon|}$$

which is always true since $|\sin(\pi\varepsilon)| \leq |\pi\varepsilon|$. □

C Pseudocodes

C.1 Tie-correction offset

We present in Algorithm 6 the procedure used to compute the tie-correcting offset. Notice that line 4 is already computed in Algorithm 1, so there is no need to recompute them in the actual implementation.

Algorithm 6 Computing the tie-correcting offset

Input: Two encodings $\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}} \in \mathbb{R}^{n^2}$ of the input vector $\boldsymbol{c} \in \mathbb{R}^n$ with (possibly repeated) elements at distance at most $\delta > 0$ and a polynomial approximation $p(x)$ of the function $\sigma_k(x)$ with desired accuracy in $[-1, -\delta] \cup [\delta, 1]$.
Output: The tie-correcting vector $\boldsymbol{t} \in \mathbb{R}^{n^2}$

procedure TieCorrection($\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}}, p(x)$)
    $\boldsymbol{d} := p(\text{Sub}(\boldsymbol{c}_{\text{rep}}, \boldsymbol{c}_{\text{exp}}))$
    $\boldsymbol{e} := \text{Mul}(\boldsymbol{d}, \text{Sub}(\alpha, \text{Mul}(\alpha, \boldsymbol{d})))$ ▷ α is a scaling factor s.t. $\alpha \cdot \sigma_k(0)' = 1$
    $\boldsymbol{s}_1 := \boldsymbol{e}$
    for $i := 0$ to $\log(n)$ **do**
        $\boldsymbol{s}_1 := \text{Add}(\boldsymbol{s}_1, \text{Rot}_{n \cdot 2^i}(\boldsymbol{s}_1))$
    $\boldsymbol{s}_2 := \boldsymbol{e} \cdot T$
    for $i := 0$ to $\log(n)$ **do**
        $\boldsymbol{s}_2 := \text{Add}(\boldsymbol{s}_2, \text{Rot}_{n \cdot 2^i}(\boldsymbol{s}_2))$
    return $\text{Sub}(\text{Mult}(0.5, \boldsymbol{s}_1), \text{Add}(\boldsymbol{s}_2, 0.5))$