Lightweight Sorting in Approximate Homomorphic Encryption

Lorenzo Rovida, Alberto Leporati and Simone Basile

University of Milano-Bicocca, DISCo, Milan, Italy
lorenzo.rovida@unimib.it, alberto.leporati@unimib.it,
s.basile1@campus.unimib.it

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 ranking, 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. This allows to build shallow sorting circuits in a very simple way. In this work, we follow up their work and explore different approaches to approximate the nonlinear functions required by the encrypted circuit (where only additions and multiplications can be evaluated), and we propose simpler solutions that allow for faster computations and smaller memory requirements.
In particular, we drastically reduce the upper bound on the depth of the circuits from 65 to 20, making our circuits usable in relatively small rings such as $N = 2^{16}$, even for sorting values while preserving up to three decimal places. As an example, our circuit sorts 128 values with duplicates in roughly 20 seconds on a laptop, using roughly 1 GB of memory, maintaining a precision of 0.01. 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.

Keywords: Fully homomorphic encryption, Sorting algorithms

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 1979 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.

Nowadays, users typically rely on online services to request computations, putting their privacy at risk, since service providers have access to plain user data. By implementing online services using FHE primitives, it is possible for service providers to offer the same service, without having access to the plain user data. The cryptographic security of the most used FHE schemes relies on the hardness of the Ring-Learning with Errors (RLWE) problem [Reg05,LPR13], which has been shown to be as hard as worst-case lattice problems [MR09].

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 nonlinear function. Classical sorting algorithms are heavily based on conditional statements, therefore evaluating them on encrypted values is far from being an easy task. 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.

1.1 Related works

Sorting encrypted values. One of the first attempts to solve the problem of sorting a set of encrypted values was published in 2013 by Chatterjee, Kaushal and Sengupta [CKS13]. A couple of years later, in 2015, Emmadi *et al.* [EGNS15] proposed a list of sorting algorithms based on FHE primitives. Nonetheless, these preliminary works were mostly theoretical and presented methods to sort a small amount of values (no more than 64 values) in a few hours, typically by using Boolean circuits.

A few years later, following the *scheme-switching* approach, Lu *et al.* [LHH+21] proposed an algorithm based on bitonic sorting that exploited the scheme-switching capability to take advantage of look-up-tables for the evaluation of the comparison operation $a < b$ from CKKS to TFHE. Their approach was able to sort 32 values in less than 10 minutes using an Intel Xeon Platinum 8269CY CPU, in single-thread. Nevertheless, the scheme-switching capability is very costly and does not scale much with the number of values, since each value requires a look-up-table evaluation.

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 approximations of the $\operatorname{sgn}(x)$ function. According to their estimations, sorting 512 values required 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, with applications to k-NN classifiers.

Finally, we 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 $\boldsymbol{v}$ (i.e., the index of each value v_i in the sorted vector $\text{sort}(\boldsymbol{v})$), (ii) given the indexing, build the corresponding permutation matrix P, and (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., $n > 256$. 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 the 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 *et al.* [CKK+19] proposed, in 2019, some approximations of the $\text{sgn}(x)$ function by defining compositions of polynomials. Plus, they also defined an approximation of the $\min(a, b)$ function[1] based on the fact that:

$$\min(a, b) = (a + b)/2 - \sqrt{(a - b)^2}/2.$$

In this case, the square root was evaluated using an iterative method [Wil51]. One year later, in 2020, Cheon, Kim and Kim [CKK20] proposed different improvements, especially in terms of computational times, for the approximation of the $\text{sgn}(x)$ function, along with an another approach to $\min(a, b)$ and $\max(a, b)$ functions. In particular, instead of evaluating the square root, they 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. Nevertheless, these approaches required a large multiplicative depth (i.e., the number of multiplications performed on a ciphertext) to ensure a decent value of precision.

In 2022, Lee *et al.* [LLNK22] proposed an approach based on composite Minimax polynomials which showed smaller execution time and a smaller multiplicative depth. We notice that [LLNK22] proposed to evaluate the $\max(a, b)$ and $\min(a, b)$ functions, along with the Rectified-Linear Unit (ReLU) function, as

$$\begin{aligned} \min(a, b) &= \frac{(a + b) - (a - b) \cdot p(a - b)}{2} \quad \text{and} \\ \text{ReLU}(x) = \max(0, x) &= \frac{x + x \cdot p(x)}{2}, \end{aligned} \tag{1}$$

[1] Notice that the max function can be trivially obtained from min as $\max(a, b) = a + b - \min(a, b)$

where $p(x)$ is a polynomial approximation of the $\text{sgn}(x)$ function. We highlight that, with this approach, $\max(a, b)$, $\min(a, b)$ and $\text{ReLU}(x)$ are computed using $p(x)$, which is an approximation of a non continuous step function. Lastly, another line of works propose to perform the evaluation of non-linear functions via the *programmable bootstrapping* approach of the TFHE scheme. Liu, Micciancio and Polyakov [LMP23] proposes a large-precision evaluation of the $\text{sgn}(x)$ function in which the precision scales logarithmically with the ciphertext modulus, making it more usable in practice.

1.2 Our contributions

We split our proposal in two parts:

1. We follow-up [MEHP25] and we 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 20 levels, making the sorting faster and with a smaller memory footprint. In order to keep the circuit efficient, we will not use rings larger than $N = 2^{16}$, keeping $\lambda \geq 128$ 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 in the first ranking procedure, and the $\text{sinc}(x)$ function to perform the equality check required to construct the permutation matrix P and the tie-offset correction. The $\text{sinc}(x)$ function can be accurately approximated without requiring many levels, as it is continuous, 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 be as close as possible to $\mathbb{Z}$. Both will be approximated using Chebyshev polynomials. 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 sort.
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):

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

   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 $d = 9$). We will show that, by performing swaps using Eq. (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., $\mathbf{1}^n = (1, 1, \ldots, 1)$ or scalars $\boldsymbol{i}^n = (i, i, \ldots, i)$. If not specified, the dimension should be clear by the context.

We use the $\leftarrow$ symbol to define an assignment. Given a vector $\boldsymbol{v}$, we define $\boldsymbol{v}_{i:j}$ as the subvector containing the elements from index i to j, namely $\boldsymbol{v}_{i:j} \leftarrow (v_i, v_{i+1}, \ldots, v_j)$.

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. If not specified, we assume $\log(x) \leftarrow \log_2(x)$. 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, we will assume that 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, $[x, y] \otimes [w, z] = [xw, xz, yw, yz]$. This trick allows us to easily create n repetitions of a vector $\boldsymbol{v}$ simply as $\mathbf{1} \otimes \boldsymbol{v}$, or to repeat each element of a vector n times as $\boldsymbol{v} \otimes \mathbf{1}$.

2.1 Some oblivious sorting algorithms

Sorting algorithms in the encrypted realm represents a challenge since most of the common approaches to sort values are *data-dependent*. Indeed, standard algorithms using clear values allow to define worst, average and best cases. In our setting, however, since values are encrypted, standard algorithms fail to be valid alternatives as it is not possible to change the flux of the execution based on the input data. We now present two of the standard approaches used in literature to sort encrypted values: permutation-based and network-based. Note that we will now present the algorithms in plain form, abstracting from any particular encryption scheme.

Permutation-based sorting. The goal of permutation-based sorting is to sort a vector $\boldsymbol{v} \in \mathbb{R}^n$ by finding the permutation matrix $P \in \{0, 1\}^{n \times n}$ such that $P \cdot \boldsymbol{v}$ is correctly sorted. The main objective is thus the construction of the P matrix. In [MEHP25], they propose to find P in two steps:

1. Finding the indexing of the vector $\boldsymbol{v} \in \mathbb{R}^n$. Namely, construct a vector $\boldsymbol{s} \in \mathbb{Z}^n$ such that, if $\boldsymbol{v}$ was sorted, the i-th element of $\boldsymbol{v}$ would be in position s_i, starting from 1. For example, given $\boldsymbol{v} = (4, 7, 1, 8)$ we get $\boldsymbol{s} = (2, 3, 1, 4)$.

This operation can be achieved by comparing each element v_i with all the others. For example, the first element s_1 is given by how many elements in $\boldsymbol{v}$ are smaller than v_1. In order to do that, the $\text{sgn}(x)$ function is used on all the possible combinations $v_i - v_j$, where:

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

Following the previous example, the first element s_1 is evaluated as:

$$s_1 = 0.5 + \text{sgn}(v_1 - v_1) + \text{sgn}(v_1 - v_2) + \text{sgn}(v_1 - v_3) + \text{sgn}(v_1 - v_4).$$

In general, we have that

$$s_i = 0.5 + \sum_{j=1}^{n} \text{sgn}(v_i - v_j).$$

2. The second step is, given the indexing vector $\boldsymbol{s} \in \mathbb{Z}^n$, to construct the corresponding permutation matrix $P \in \{0,1\}^{n \times n}$. Each i-th column $\boldsymbol{p}_i \in \{0,1\}^n$ of P is constructed using an equality check. We define the $\text{equal}(x)$ function as:
$$\text{equal}(x) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ otherwise} \end{cases}.$$
By abuse of notation we allow the function to be applied on vectors by evaluating the corresponding function independently on each entry. Finally, the i-th column $\boldsymbol{p}_i$ of the permutation matrix P can be found as
$$\boldsymbol{p}_i \leftarrow \text{equal}(\boldsymbol{s} - \boldsymbol{i}).$$
with $0 < i \leq n$. Following the example, the first column $\boldsymbol{p}_1$ of P is computed as:
$$\boldsymbol{p}_1 = \text{equal}\,(\boldsymbol{s} - \boldsymbol{1}) = \text{equal}\,((2,3,1,4) - (1,1,1,1)) = (0,0,1,0)$$

Tie-correction offset. One may notice that this method fails to sort a vector which contains repeated values, e.g. $\boldsymbol{v} = (4,7,1,1)$. An effective solution proposed in [MEHP25], is to intervene in the indexing vector, which, using $\boldsymbol{v}$, would be constructed as $\boldsymbol{s} = (3,4,1.5,1.5) \notin \mathbb{Z}^n$. One solution is to apply a *tie-correction offset*. In particular, we require to construct an offset vector that corrects the partial indexing. Following the example, such a vector would be $\boldsymbol{o} = (0,0,-0.5,0.5) \in \mathbb{R}^n$. Therefore, the final indexing would be given by the sum $\boldsymbol{s} + \boldsymbol{o} \in \mathbb{Z}^n$. The offset vector $\boldsymbol{o} \in \mathbb{R}^n$ can be obtained as follows:

1. Count how many elements are repeated, and how many times. Formally, for each element $v_i \in \boldsymbol{v}$, count how many elements v_j are equal to v_i, with

$0 < i, j \leq n$. We build the "repetitions" vector $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ that contains such information, where each $0 < r_i \leq n$, as:

$$r_i \leftarrow \sum_{j=1}^{n} \text{equal}(v_i - v_j).$$

2. Define a vector $\boldsymbol{k}$ which contains information about the partial repetitions. In practice, each k_i is equal to the number of elements in $\boldsymbol{v}_{1:i}$ that are equal to v_i. Single elements will have sum 1, while each repeated value will be assigned a value from 1 to x, where x is the number of repetitions.

$$k_i \leftarrow \sum_{j=1}^{i} \text{equal}(v_i - v_j).$$

3. Compute $\boldsymbol{o} \leftarrow \big(\boldsymbol{k} - (0.5 \cdot \boldsymbol{r}) - \mathbf{0.5}\big) \in \mathbb{R}^n$ to obtain the final tie-correction offset $\boldsymbol{o} \in \mathbb{R}^n$. Following the example, given $\boldsymbol{v} \leftarrow (4, 7, 1, 1) \in \mathbb{R}^n$ and the corresponding indexing $\boldsymbol{s} \leftarrow (3, 4, 1.5, 1.5) \notin \mathbb{Z}^n$:

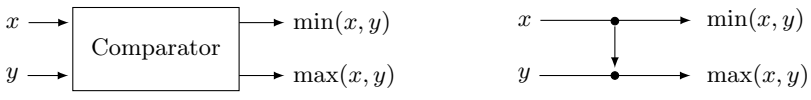
$$\begin{aligned} \boldsymbol{o} \leftarrow \boldsymbol{k} - (0.5 \cdot \boldsymbol{r}) - \mathbf{0.5} &= (1, 1, 1, 2) - (0.5, 0.5, 1, 1) - \mathbf{0.5} \\ &= (0, 0, -0.5, 0.5) \in \mathbb{R}^n \end{aligned}$$

which is precisely the offset we were looking for. Refer to [MEHP25] for a proof of correctness and more details about this procedure.

Network-based sorting. In sorting networks the only operation that depends on the input data is the *compare-and-swap* operation, performed if two elements are not in the correct order. The latter is defined as

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

that can be hardware implemented as shown in Fig. 1a. In the context of sorting networks, this operation is usually represented as illustrated in Fig. 1b. In



(a) In hardware (b) In a sorting network

Fig. 1: Representations of the *compare-and-swap* operation

recent FHE literature [HKC+21], this operation has been implemented using approximations of the $\text{sgn}(x)$ function to replicate the comparison $a > b$.

Bitonic sorting. We propose to build a sorting network using the *bitonic sorter* algorithm [Akl11], which sorts a power of two amount of values. Informally, the bitonic sorting algorithm builds a sorting network that, at the beginning, sorts each 2-consecutive elements, then each 4-consecutive elements, and so on. Let us introduce two definitions that will be useful later.

Definition 1 (k-bitonic sequence). *A k-bitonic sequence is a sequence of n values such that $x_1 \leq x_2 \leq \cdots \leq x_k \geq x_{k+1} \geq \cdots \geq x_n$, with $k \in [1, n]$. For convenience, a non-bitonic sequence is considered a 1-bitonic sequence, and a n-bitonic sequence is considered a sorted sequence.*

Definition 2 (k-bitonic block). *A k-th bitonic block, with $k \in \mathbb{Z}^+$, is a sequence made of $\log(k) + 1$ sorting layers, such that, given as input a k-bitonic sequence, returns as output a $2k$-bitonic sequence.*

Note that, in general, these definitions can be applied to any subsequence. For example, the 2-bitonic sequence $[1, 2, 0, 3, 4, 9, 1, 9]$, given as input to a 2-bitonic block, produces a 4-bitonic sequence $[0, 1, 2, 3, 1, 4, 9, 9]$. A network that sorts n values will be composed of $\log(n)$ bitonic blocks. For instance, Fig. 2 illustrates a network that sorts $n = 8$ input values.

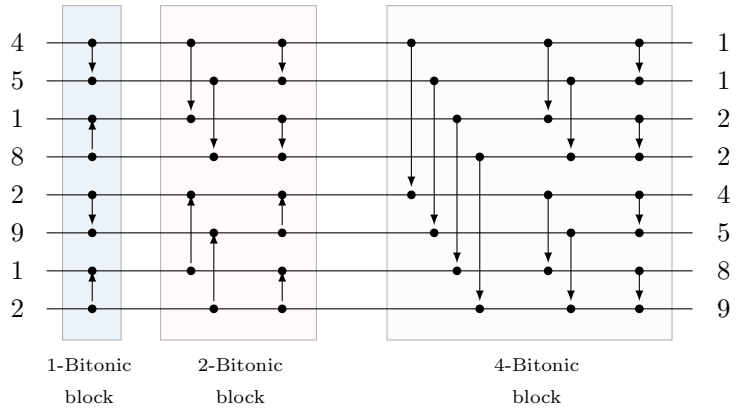


Fig. 2: A bitonic sorting network composed of 6 layers that sorts $n = 8$ input values. The direction of the arrows defines the order of the comparisons

In this example, eight values enter the network from the left, move across the eight horizontal wires, and exit from the right. The network sorts them, placing the largest number at the bottom. Each of the six layers is vertically highlighted, and can be evaluated in a single comparison operation. Declining this observation in the CKKS context, it means that this network, in order to be evaluated, requires six comparison operations.

As opposed to the permutation-based approach, these networks tend to require deep circuits to be evaluated. On the other hand, they do not require to repeat the input n times in the ciphertext.

2.2 Approximate Homomorphic Encryption

The sorting algorithm is implemented using 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 with modulus q. Notice that complex values are rounded to (very large) integers using a scale Δ, in a fixed-point representation. All the operations between ciphertexts are performed *slot-wise* (i.e., between pairs of encrypted vectors elements independently), enabling fast Single Instruction, Multiple Data (SIMD) computations. The available primitive operations are the following:

- $\textsc{Add}/\textsc{Sub}/\textsc{Mul}(c_1, c_2)$: given two ciphertexts c_1 and c_2, performs the slot-wise addition/subtraction/multiplication between each pair of elements. Without loss of generality, c_2 can also be a plaintext.
- $\textsc{Rot}_i(c)$: given a ciphertext 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, which is based on Residual Number System (RNS) for efficient computations. 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 vector v is scaled by a factor Δ, which is a large power of two, typically larger than 2^{40}. 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 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), enabling to evaluate circuits of arbitrary depth. Intuitively, the decryption circuit is evaluated using an encrypted version of the secret key (called *bootstrapping key*).

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

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

Given this set of polynomials, it is also possible to define the so-called *Chebyshev roots* (or *Chebyshev interpolants*) of the first kind, a set of n points defined as:

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

where each coefficient c_k is distributed according to a certain continuous function f to be approximated over $[-1, 1]$. Given the following set:

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

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 roots*. 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 algorithm [PS73] 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 approach to construct more efficient permutation-based sorting algorithms and bitonic sorting networks.

3.1 Permutation-based sorting

As mentioned above, this sorting approach is very efficient and relies on the computation of two nonlinear functions only (three, if accepting repeated elements). Note that our goal is not to prove the correctness of the procedures – whose proofs can be found in [MEHP25] – but to propose concrete approaches to make them computationally lighter and more efficient in terms of memory. Let us define $\boldsymbol{c} \in \mathbb{R}^n$ as the (encrypted) input vector[2], where n is without loss of generality considered a power of two for efficiency reasons.

[2] For the sake of clarity, we will at this point make no difference between clear and encrypted vectors since every operation in the described procedure can be implemented via additions and multiplications, fitting the circuit to FHE schemes.

Indexing based on SIMD. The first function, $\mathrm{sgn}(x)$ is used to count, for every $c_i \in \boldsymbol{c}$, how many elements in the input vector are smaller than c_i. The SIMD capabilities of CKKS allows to compute this very efficiently in a single comparison by repeating the input vector and using $\Theta(n^2)$ slots of the ciphertext. We require the input to be encoded in two different ways: *repeated* encoding (the whole vector $\boldsymbol{v}$ is repeated n times) and *expanded* encoding (each element v_i is repeated n times). Given these encodings, which can be easily obtained as:

$$\boldsymbol{c}_{\text{expanded}} \leftarrow \boldsymbol{c} \otimes \mathbf{1} \in \mathbb{R}^{n^2} \quad \text{and} \quad \boldsymbol{c}_{\text{repeated}} \leftarrow \mathbf{1} \otimes \boldsymbol{c} \in \mathbb{R}^{n^2},$$

we start by performing a subtraction between them[3]. A visual example is given in Figure 3, considering $n = 4$ and the input vector $\boldsymbol{c} = (4, 7, 1, 8)$.

$\boldsymbol{c}_{\text{repeated}}$:	4	7	1	8	4	7	1	8	4	7	1	8	4	7	1	8	−
$\boldsymbol{c}_{\text{expanded}}$:	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	8	

Fig. 3: Computing $\boldsymbol{c}_{\Delta}$ from the two encodings of the input ciphertext $\boldsymbol{c}$ to be sorted

At this point, the idea is to count the occurrence of positive elements in each n-sized subvector. In order to do that, the $\mathrm{sgn}(x)$ function is applied on the whole $\boldsymbol{c}_{\Delta}$ ciphertext. By the means of SIMD, this operation is performed only once. We will later present our approach to evaluate such a function in CKKS. Then, in order to get the sum of the values in positions $k \cdot n$ (which refer to the k-th element in $\boldsymbol{v}$) we rotate and sum elements in positions $k \cdot n$, with $0 \leq k < n$ integer (this procedure is denoted in Figure 4 by the $\sum$ symbol) and we increase the results by 0.5 to correct the bias introduced on elements whose subtraction lead to 0.

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

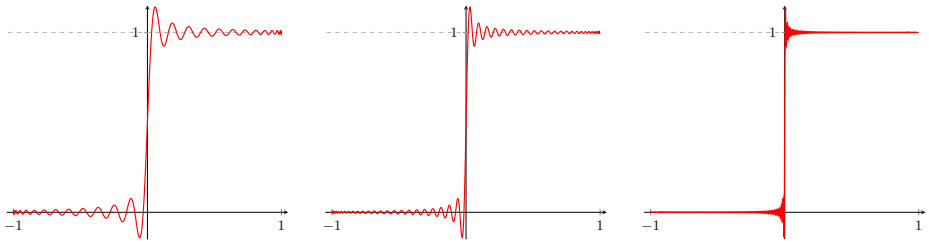
Fig. 4: Given $\boldsymbol{c}_{\Delta}$, the (repeated) indexing of $\boldsymbol{c}$ is given by performing a rotate and sum (indicated with the $\sum$ symbol) over $\mathrm{sgn}(\boldsymbol{c}_{\Delta})$, plus $\mathbf{0.5}$

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

Notice that this procedure returns the repeated indexing vector (following the example, $\mathbf{1} \otimes (2, 3, 1, 4)$) and it is pretty quick, apart from its main bottleneck, i.e., the $\text{sgn}(x)$ function, since it has to be approximated with high degree polynomials. In order to do that, it is possible to follow two directions:

- Using the approach firstly introduced in [CKK20], and followed 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.
- Using Chebyshev polynomials to directly approximate $\text{sgn}(x)$. 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. Approximating the raw $\text{sgn}(x)$ function, though, is not a good choice, as the resulting polynomial will suffer by a strong Gibbs phenomenon, namely large oscillations near the "step" occurring in $x = 0$, as shown in Figure 5.



(a) Approximation with degree $d = 50$

(b) Approximation with degree $d = 100$

(c) Approximation with degree $d = 500$

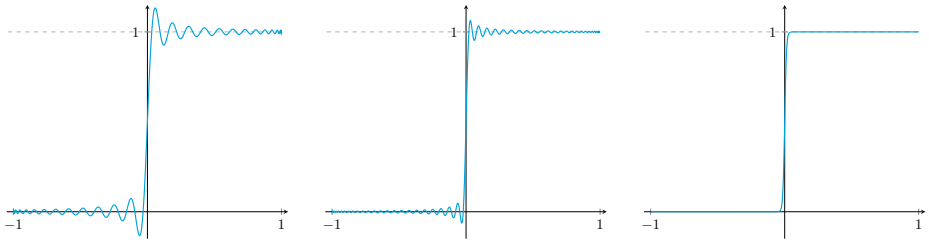
Fig. 5: Chebyshev approximations of the raw $\text{sgn}(x)$ function in $[-1, 1]$, with different degrees

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 function as

$$\text{sigmoid}_k(x) \leftarrow \frac{1}{1 + e^{-kx}}. \tag{3}$$

Notice that this function acts as a "smooth" $\text{sgn}(x)$ function, where the smoothness parameter is given by $k \in \mathbb{Z}^+$. Our strategy is to empirically find

some value k such that the sigmoid well approximates $\mathrm{sgn}(x)$, up to some negligible error, in $[-1,-\delta]\cup[\delta,1]$. As an example, we fix $\delta = 0.1$ and we observe that, with $k = 120$, the error $\ell_\infty(\mathrm{sgn}(x) - \mathrm{sigmoid}_k(x))$ is negligible (i.e., less than 10^{-6}) in the interval $[-1,-\delta]\cup[\delta,1]$. Using this function 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 6.



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

Fig. 6: Chebyshev approximations of the 150-scaled Sigmoid function in $[-1,1]$, with different degrees

Then, the behavior of the polynomial for $x \in [-\delta,\delta]$ is irrelevant, since by assumption there will be no pairs of elements such that their difference lies in that 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. *For any Chebyshev polynomial approximation $p(x)$ of the k-scaled Sigmoid function of degree $d > 0$ in the interval $[-1,1]$, it holds that $p(0) = 0.5$, regardless of the scaling factor $k > 1$.*

Proof. When approximating the scaled sigmoid function using Chebyshev polynomials, the behavior of the approximation depends critically on the parity of the polynomial degree d. Specifically, two distinct cases arise:

– Even degree d: remark that the $d+1$ Chebyshev roots x_i, from x_0 to x_d, are distributed as

$$x_i = \cos\left(\frac{(i+\frac{1}{2})\pi}{d+1}\right) \text{ with } 0 \le i \le d$$

Given the approximation interval $[-1,1]$, we will have $d+1$ roots, and the one "in the middle" is set at x_m, with $m = \frac{d}{2} \in \mathbb{Z}$.

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

Therefore, the polynomial $p(x)$ passing through the Chebyshev roots x_i will necessarily be equal to 0.5 in the origin since, by construction, $p(x_m) = \text{sigmoid}_k(x_m) = \text{sigmoid}_k(0) = 0.5$. Notice that $\text{sigmoid}_k(0) = 0.5$ for any $k > 1$.

– Odd degree d: For now assume without loss of generality that the sigmoid function is odd (just translate it by -0.5). Chebyshev roots are distributed equally between the first $[0, \dots, (d-1)/2] \cap \mathbb{Z}$ and the second $((d-1)/2, \dots, d]) \cap \mathbb{Z}$ set of roots.
Notice that the second set of roots can be written with respect to the first one by reversing signs and order, since for any $0 < i \leq d$:

$$\cos\left(\frac{(d-i+\frac{1}{2})\pi}{d+1}\right) = \cos\left(\pi + \frac{(i+\frac{1}{2})\pi}{d+1}\right) = -\cos\left(\frac{(i+\frac{1}{2})\pi}{d+1}\right)$$

which holds because $\cos(\pi + \theta) = -\cos(\theta)$. This implies that the roots are symmetric with respect to the y-axis since $-x_i = x_{d-i}$. This gives rise to an odd polynomial, i.e., such that $p(-x) = -p(x)$, and as a consequence the polynomial passing through those points will contain even powers only. Any odd-powered term would indeed break such a symmetry, since if x^k is odd, then $p(-x) \neq -p(x)$ since $x^k \neq -x^k$ for odd powers $k > 0$. The odd polynomial will pass through the middle point $p(m)$ between the last point $p(x_{(d-1)/2})$ of the first interval and the first point $p(x_{1+(d-1)/2})$ of the second interval to keep the symmetry with respect to the y-axis, where m is:

$$\begin{aligned} m &= \left(\cos\left(\frac{((d-1)/2+\frac{1}{2})\pi}{d+1}\right) + \cos\left(\frac{(1+(d-1)/2+\frac{1}{2})\pi}{d+1}\right)\right) \cdot \frac{1}{2} = \\ &= \left(\cos\left(\frac{(d/2)\pi}{d+1}\right) + \cos\left(\frac{\pi+(d/2)\pi}{d+1}\right)\right) \cdot \frac{1}{2} \end{aligned}$$

Now, notice that the two angles sum to π:

$$\frac{(d/2)\pi}{d+1} + \frac{\pi+(d/2)\pi}{d+1} = \frac{\pi+d\pi}{d+1} = \frac{\pi(d+1)}{d+1} = \pi$$

as before we use that $\cos(\theta) + \cos(\theta+\pi) = 0$, meaning that the middle point is indeed $m = 0$, therefore $p(m) = \text{sigmoid}_k(m) = 0.5$.

□

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 $\text{sgn}(x)$ in the interval $[-1, -\delta] \cup [0] \cup [\delta, 1]$, with negligible errors.

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. In order to change between the two, at the beginning of the circuit, instead of computing $\boldsymbol{c}_{\text{repeated}} - \boldsymbol{c}_{\text{expanded}}$, compute the opposite $\boldsymbol{c}_{\text{expanded}} - \boldsymbol{c}_{\text{repeated}}$.

Building the corresponding permutation matrix. Let us for a moment ignore the eventual noise contained in the indexing vector $\boldsymbol{s} \in \mathbb{R}^n$ coming from approximate computations. Suppose having $\boldsymbol{v} \leftarrow (4, 7, 1, 8)$ and its corresponding indexing $\boldsymbol{s} \leftarrow (2, 3, 1, 4)$, as computed in the above presented procedure. The procedure to obtain a permutation matrix simply performs an equality check between the indexing vector and n vectors $\boldsymbol{i} \in \{i\}^n$, for $0 < i \leq n$. Again, with SIMD this can be efficiently parallelized. 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:

$$\text{equal}(x) \leftarrow \text{sgn}\left(\frac{x + 0.5}{r}\right) \cdot \left(1 - \text{sgn}\left(\frac{x - 0.5}{r}\right)\right), \tag{4}$$

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 7.

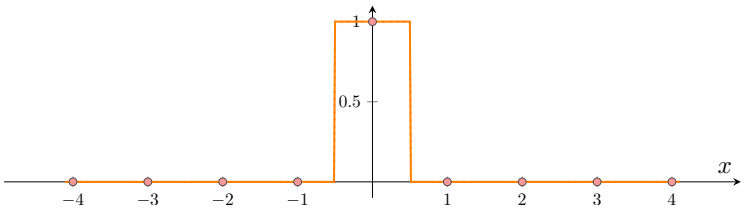


Fig. 7: The approximation for equal(x) used in [MEHP25], for $a = 4$

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 observation.

Lemma 2. *Any function $f : \mathbb{R} \to \mathbb{R}$ such that $f(0) = 1$ and $f(x) = 0$ for all $x \in \mathbb{Z} \setminus \{0\}$ can be used to construct a permutation matrix P from the indexing of a vector $\boldsymbol{v} \in \mathbb{Z}$ such that $P \cdot \boldsymbol{v}$ is sorted.*

Notice that the definition of equal(x) in [MEHP25] respects the requirements of Lemma 2. Nevertheless, 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 respect the barely minimum requirements of Lemma 2: the sinc(x) function, defined as:

$$\text{sinc}(x) = \frac{\sin(x\pi)}{(x\pi)} = \prod_{n=1}^{\infty}\left(1 - \frac{x^2}{n^2}\right).$$

The latter can be efficiently approximated, moreover (i) $\mathrm{sinc}(0) = 1$ and (ii) $\mathrm{sinc}(x) = 0$ for every $x \in \mathbb{Z} \setminus \{0\}$. We conveniently report its plot in Figure 8.

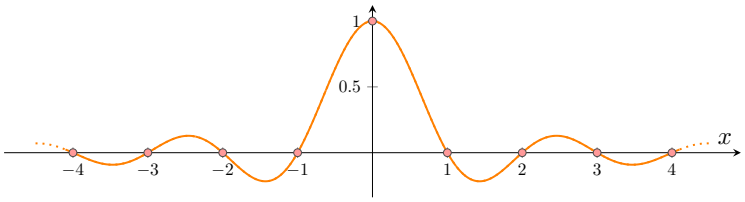


Fig. 8: The $\mathrm{sinc}(x)$ function

Its error tolerance is much smaller than the one in Eq. (4), in particular $\mathrm{sinc}(x)$ approximate equal(x) for all the values $x + \varepsilon \in \mathbb{R}$ for very tiny values of $\varepsilon > 0$; for this reason we require the indexing vector to be as close as possible to $\mathbb{Z}$. We propose to use Chebyshev polynomials to approximate $\mathrm{sinc}(x)$ on the differences between the indexing $\boldsymbol{s}$ and each $\boldsymbol{i}$. This will lead to the permutation matrix P, encoded column-wise, as illustrated in Figure 9.

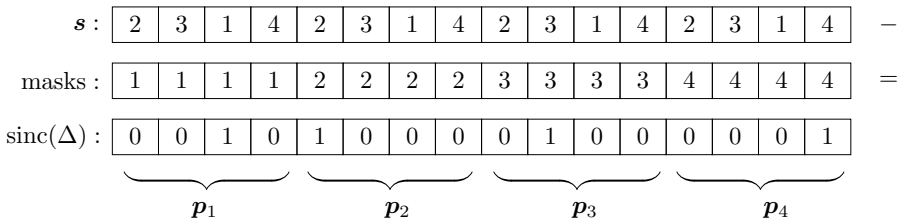


Fig. 9: Generating the columns of P using the $\mathrm{sinc}(x)$ function over the differences between each pair $\boldsymbol{s}$ and $\boldsymbol{i}$.

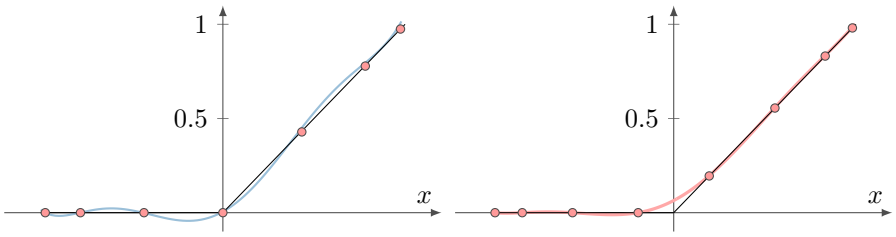
Faster tie-correcting offset. The tie-correction offset in [MEHP25] is computed using some partial results obtained in the indexing phase (using the output of what they call Eq function), while we propose to use $\mathrm{sinc}(x)$ at the beginning of the circuit, on the difference between the expanded and the repeated inputs. The advantages are that: (i) correcting the indexing for repeated elements comes at no cost in terms of circuit depth, since this phase is "parallel" to the indexing one, and additionally (ii) we directly use the input values which contain the least amount possible of noise, and thus are supposed to be close to $\mathbb{Z}$ by assumption. On the other hand, the main drawback is that we compute the equality check based on $\mathrm{sinc}(x)$ from scratch, requiring some extra time – although as experiments will show, this can be computed very quickly and will have a limited impact in the final runtime.

3.2 Network-based sorting

The sorting network relies on comparisons between pairs of values. We propose to evaluate the comparison operation using the following relation:

$$\min(a, b) = a - \mathrm{ReLU}(a - b), \tag{5}$$

where $\mathrm{ReLU}(x)$ is approximated using Chebyshev polynomials. In Figure 10, 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.



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

Fig. 10: Approximations of the ReLU function, using Chebyshev interpolants

By computing the maximum error in $x \in [-1, 1]$ as $||\mathrm{ReLU}(x) - p_d(x)||_\infty$, 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. The definition in Eq. (2) allows to easily approximate $\mathrm{ReLU}(x)$ with a larger precision than previous works. Table 1 presents the depth consumed by evaluating Eq. (2) using Chebyshev polynomials with respect to the state-of-the-art work by Lee *et al.* [LLNK22]. We remark that their approach is to evaluate the ReLU function using Eq. (1).

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 $d = 310$)

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 operation. Therefore, we define the complexity of a sorting network as the number 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 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 11, composed of a single layer, and a sample vector $\boldsymbol{v} \leftarrow (5, 4, 1, 8, 2, 9, 1, 2)$. we evaluate it in four steps.

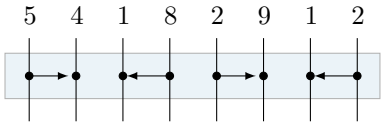


Fig. 11: 1-Bitonic block with 8 input values

1. Obtain the minimum between each pair (refer to pairs of elements connected by the arrows in Figure 2) as shown in Eq. (5):

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

   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. 1a, this operation gives as output all the first wires that contain $\min(a, b)$.

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 2 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 12).

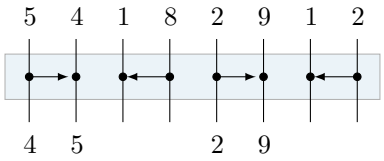


Fig. 12: Values obtained in $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$

In particular, $\boldsymbol{c}_1$ contains the minimum values (4 and 2), whereas $\boldsymbol{c}_2$ 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 13).

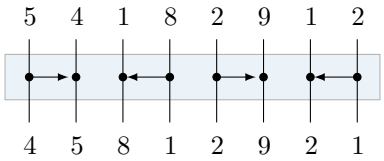


Fig. 13: Complete evaluation of a sorting layer

However, before doing that, unnecessary values, for each 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 1.

Algorithm 1 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 2, which returns a set of four masks to be applied to the different $\boldsymbol{c}_i$.

Algorithm 2 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 $\boldsymbol{m}_j$

This set of masks strongly depends on the current network layer, and they are constructed according to the length of the arrows and their direction. Putting these procedures together, it is possible to build an SIMD-based encrypted bitonic sorting circuit (Algorithm 3). 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-

Algorithm 3 SIMD-based bitonic sorting of $\boldsymbol{c}$

```
n ← length of c
for b ← 1 to ⌈log(n)⌉ do                          ▷ b is the current block
    for ℓ ← 1 to b + 1 do                         ▷ ℓ is the current layer
        α ← 2^(i−j)                               ▷ δ is the length of the arrows
        c ← EvaluateLayer(c, n, α, b, ℓ)
```

based approach. Each experiment uses the OpenFHE implementation of the RNS-CKKS [ABBB+22, KPP22] scheme, a security level of $\lambda > 128$ bits, rings of size $N = 2^{16}$ and sparse ternary secret key distribution. All experiments have been run on a Macbook 2021 M1 Pro laptop with 16 GB of memory and are replicable by running the corresponding script in the experiments folder, contained in our open-source repository[4]. The latter also contains Python notebooks that implement the circuits executed on clear data, useful to better understand their workings. The parameters have been chosen in such a way that sorting of uniform random values, sampled from $[0, 1]$ at maximum distance δ is correct. Formally, 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

$$|c_i - \mathrm{sort}(\boldsymbol{v})_i| < \delta.$$

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

4.1 Permutation-based

The main tunable parameters used in these experiments are: (i) the degree d_1 of the approximation of the $\mathrm{sigmoid}_k(x)$ function, (ii) its scaling factor k, and (iii) the degree of the approximation of the $\mathrm{sinc}(x)$ function d_2.

Concerning the tie correction offset, we use an approximation of the $\mathrm{sinc}(x)$ function of degree of $d_{\mathrm{tie}} < d_1$ so that this phase does not increase the circuit depth, as it can be performed in parallel with the indexing phase[5]. The evaluation of the tie-correction offset does not have a huge impact in the final computation in terms of runtime – just a few seconds – but by being very precise it allows to correct the offset without introducing a significant error to the indexing vector, which, we stress, should be as close as possible to $\mathbb{Z}$. In order to consume less levels, all Chebyshev polynomial approximations are evaluated in $[-1, 1]$, with inputs and outputs scaled accordingly. The CKKS parameter which controls the complexity of decomposition during the key switching, is set to the minimum possible $d_{\mathrm{num}} = 2$ (refer to [KPZ21] for a detailed explaination of key switching). Notice that higher values of d_{num} allow to evaluate deeper circuits without

[4] github.com/lorenzorovida/Lightweight-Sorting-In-Approximate-Homomorphic-Encryption

[5] Taking into account a product by 0.5, so the degree of $\mathrm{sinc}(x)$ should allow for one extra multiplication.

reducing the security level, at the cost of increasing computational complexity (larger d_{num} implies smaller modulus $\log(QP)$, which with the ring size N define the security of the underlying RLWE problem), although we do not require to reduce the size of the modulus, as our circuits are already shallow.

Remark 2. 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] *on single ciphertexts.* In the first experiment, we define the same setup as in [MEHP25] 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 maximum 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 degrees d_1 and d_{tie}, along with 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 approximation

	CKKS parameters			Data parameters			
n	Δ	Circuit depth	$\log(QP)$	$\text{sigmoid}_k(x)$ scale k	$\text{sigmoid}_k(x)$ degree d_1	$\text{sinc}(x)$ degree d_2	$\text{sinc}(x)$ degree d_{tie}
8	2^{45}	17	1233	650	1006	59	495
16	2^{45}	17	1233	650	1006	59	495
32	2^{45}	18	1338	650	1006	119	495
64	2^{45}	19	1383	650	1006	247	495
128	2^{45}	20	1488	650	1006	495	495

As reported in Figure 14, computational times and memory requirements are always lower than in [MEHP25]. 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 levels required by our circuit are at most 20. All the chosen moduli QP are still notably below the 128-bits of classical security threshold – 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

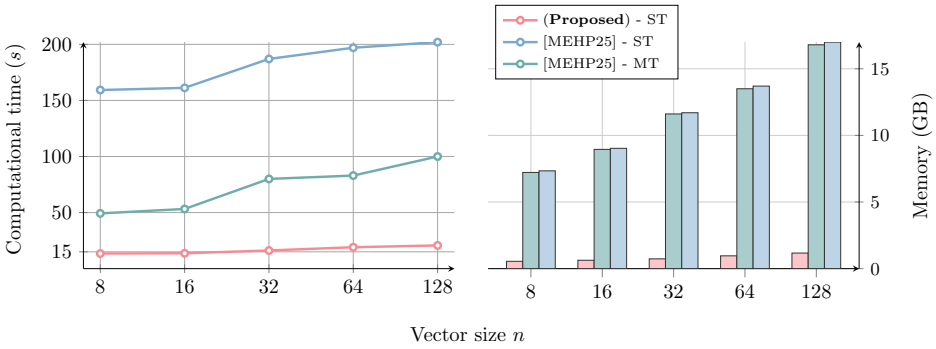


Fig. 14: Comparison on computational time and memory consumption between our proposal and [MEHP25]. ST and MT stand for single and multithread

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 $\mathrm{sgn}(x)$, which is the main bottleneck of the circuit. On the other hand, increasing n results in larger degrees d_2, which are always small with respect to the ones used for $\mathrm{sigmoid}_k(x)$.

Experiment 2, more discretization. We now aim to increase the discretization of the considered $[0, 1]$ interval from $\delta = 0.01$ to $\delta = 0.001$. In order to achieve a correct sorting, we must increase the degrees of approximations of the nonlinear functions, in particular of $\mathrm{sigmoid}_k(x)$. Luckily, from the previous experiment, we still have room to increase the circuit depth maintaining $\lambda > 128$ bits of security without requiring an increment in the ring size. We thus increase the degree d_1 of the scaled sigmoid function to $d_1 = 16000$, with the scaling factor set to $k = 9170$; refer to Table 3 and Figure 15.

Table 3: Parameters used in permutation-based sorting experiments. Degrees d_1, d_2 refer to the Chebyshev polynomials approximation

	CKKS parameters			Data parameters			
n	Δ	Circuit depth	$\log(QP)$	$\mathrm{sigmoid}_k(x)$ scale k	$\mathrm{sigmoid}_k(x)$ degree d_1	$\mathrm{sinc}(x)$ degree d_2	$\mathrm{sinc}(x)$ degree d_{tie}
8	2^{45}	21	1533	9170	16000	59	495
16	2^{45}	21	1533	9170	16000	59	495
32	2^{45}	22	1638	9170	16000	119	495
64	2^{45}	23	1683	9170	16000	247	495
128	2^{45}	24	1728	9170	16000	495	495

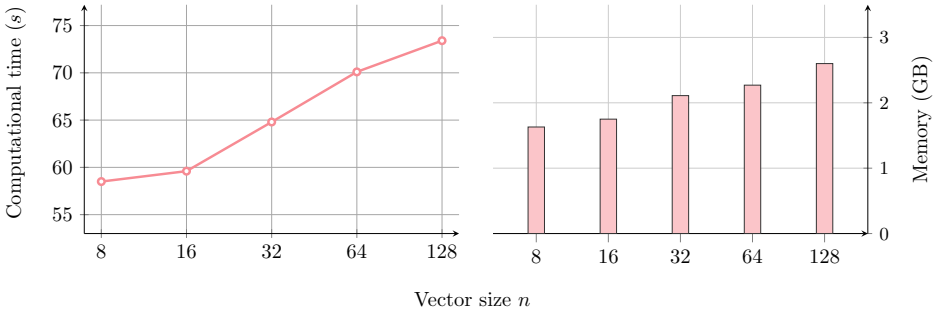


Fig. 15: Time and memory requirements when sorting elements at maximum distance $\delta = 0.001$

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 2 GB, never larger than 3 GB.

4.2 Network-based

We now aim to perform similar experiments as above, but based on the bitonic sorting network. Notice that the keys of the scheme required in this 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 to reduce the errors introduced by the bootstrapping.

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 ReLU(x), which 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	ReLU(x) degree d
from 8 to 8192	2^{55}	25	1765	16	495

We present in Figure 16 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.

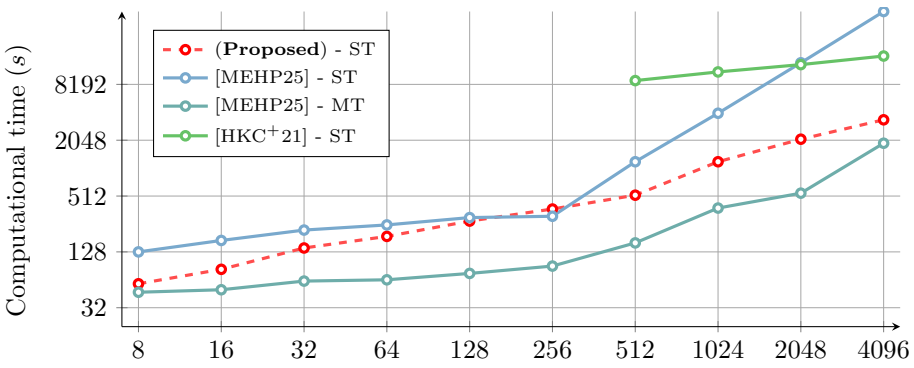


Fig. 16: Comparing our network-based approach with [MEHP25, HKC+21] in terms of runtime

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]. We observe that during the experiments we exceeded the laptop available memory, which may have led to overestimated runtimes, although we are at this point interested to rough estimates and to see how they grow. Although slower, it can be considered as a nice alternative to permutation-based as it can be used on large amounts of elements without assuming to have n^2 slots available.

Although 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 17.

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 fast 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. The ultimate goal would be to be able to run a permutation-based sorting without resorting to $\mathrm{sgn}(x)$. It would be also interesting to explore new functions to use in the

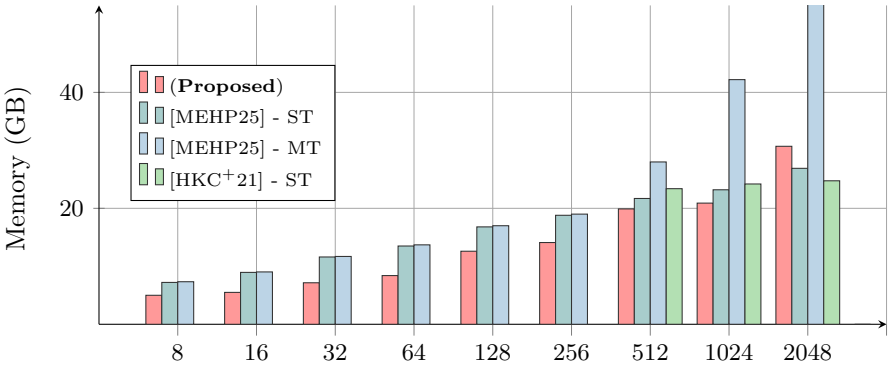


Fig. 17: Comparing our network-based approach with [MEHP25, HKC+21] in terms of memory. The last cut bar is valued 97 GB, cut for readability

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 could be to build a hybrid sorting algorithm where each subvector 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.

Acknowledgements

Lorenzo Rovida would like to thank Federico Mazzone (University of Twente) for his availability, useful discussions and a suggestion about the degrees of the $\mathrm{sinc}(x)$ function.

References

ABBB+22. 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, 2022.

Akl11. Selim G. Akl. *Bitonic Sort*, pages 139–146. Springer US, 2011.

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.

CCS19. Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Advances in Cryptology – EUROCRYPT 2019*, pages 34–54, 2019.

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*, page 3–25, 2025.

CKK+19. Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. Numerical method for comparison on homomorphically encrypted numbers. In *Advances in Cryptology – ASIACRYPT 2019*, pages 415–445, 2019.

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.

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, 2017.

CKS13. Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In *Progress in Cryptology – INDOCRYPT 2013*, pages 262–273, 2013.

EGNS15. 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, 2015.

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.

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.

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.

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), dec 2022.

KPP22. Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. In *Topics in Cryptology – CT-RSA 2022*, pages 120–144, 2022.

KPZ21. Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III*, page 608–639, 2021.

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.

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.

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.

LMP23. Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. In *Advances in Cryptology - ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, pages 130–160, 2023.

LPR13. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), nov 2013.

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)*, aug 2025.

MR09. Daniele Micciancio and Oded Regev. *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

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.

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.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, 2005.

Tre19. Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2019.

Wil51. M.V. Wilkes. *The Preparation of Programs for an Electronic Digital Computer: With Special Reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley mathematics series. Addison-Wesley Press, 1951.