Oblivious Sorting under Fully Homomorphic Encryption: A Comprehensive Survey and Performance Analysis

Omar Ahmed, Rostin Shokri, and Nektarios Georgios Tsoutsos

University of Delaware
{oaaa, rostinsh, tsoutsos}@udel.edu

Abstract. Outsourcing computations to cloud providers raises significant data privacy concerns, making Privacy-Preserving Computation via Fully Homomorphic Encryption (FHE) increasingly vital. However, adapting data sorting routines to the FHE domain introduces severe performance bottlenecks. This survey systematizes the state-of-the-art in FHE-based sorting algorithms. A novel complexity metric, FHE-Effort, is introduced to accurately evaluate homomorphic circuit efficiency. Eighteen algorithms are benchmarked across three major FHE schemes using a unified codebase. The analysis concludes that TFHE is currently the most efficient scheme for sorting applications, and sorting networks like Odd-Even Merge and Bitonic Sort offer the optimal algorithmic architectures.

Keywords: Fully Homomorphic Encryption · Privacy-Preserving Computation · Oblivious Sorting · Asymptotic Analysis · Sorting Networks · Secure Cloud Computing.

1 Introduction

Data sorting is a fundamental operation that has driven the evolution of the computer science discipline. Sorting algorithms serve as the fundamental building blocks for organizing massive volumes of data generated in the digital age. Since the dawn of technology, sorting has been an active area; researchers are continually developing efficient sorting algorithms that can sort billions of data elements to accommodate the massive data growth [7, 22, 23, 30, 33, 39, 49].

Beyond mere data organization, sorting is an indispensable technique to optimize numerous downstream computational processes. For example, [55, 57] utilize sorting to optimize the performance of the K-Nearest Neighbor (KNN) classifier. Furthermore, sorting is found in information retrieval systems to enhance the data retrieval process. For instance, [38] uses the count sort algorithm to partially sort retrieved documents. In addition, [21] relies on sorting to allow efficient duplicate removal, grouping, and aggregating database queries. Sorting algorithms are also critical tools that are used to enhance financial and big-data systems [43, 47], wireless networking [6], and graph algorithms [27, 54].

With the heavy reliance on sorting, especially when performed as part of outsourced computations on cloud systems, a gap manifests between the requirement for data privacy and performing sorting. In particular, with the increasing privacy concerns and threats to cloud attacks [5, 20, 37, 44], and the need to outsource heavy computations to remote cloud providers, users prefer to rely on privacy-preserving computation techniques to ensure the confidentiality of their private data. Privacy-preserving computation (PPC) refers to the utilization of cryptographic techniques to allow processing sensitive data without the need to expose them.

One of the critical techniques to enable PPC is Fully Homomorphic Encryption (FHE). FHE [28] is a cryptographic technique that enables users to perform computation directly on encrypted data without ever decrypting them. The processor of FHE data (e.g., a cloud provider) generates encrypted results which can only be recovered by the data owner. Typically, when a user outsources a computation, they use an FHE scheme to encrypt the data and send them to a cloud provider. The cloud will process the data while they are encrypted and generate encrypted outputs. The outputs are sent to the user, who maintains the private key to recover the results. Data in clear format are called plaintext, whereas encrypted data are called ciphertext. Data owners hold a private key that can decrypt ciphertexts, while cloud providers maintain a set of public keys (generated by the data owner) that allows performing computations on ciphertexts.

This powerful capability of FHE has attracted the attention of the research community and industry to develop efficient solutions to enable PPC. However, there are still some challenges that limit the adoption of FHE. One of the most important challenges is the performance of FHE computations. Homomorphic computations are orders of magnitude slower than the corresponding plaintext computations. This is due to the complexity of the structures of FHE ciphertexts, which are often represented as very large polynomials. Translating basic arithmetic operations, such as addition and multiplication, involves implementing more complicated arithmetic on the polynomial level.

Another challenge related to FHE computations is **noise management**. Essentially, FHE depends on random noise added to the plaintext data to protect them. The problem is that this noise increases as we do more homomorphic computations. Eventually, the noise accumulates and destroys the underlying data, causing incorrect results. Therefore, some existing FHE schemes are equipped with a procedure called **bootstrapping**, which reduces the noise within a ciphertext to allow more computations. Unfortunately, bootstrapping is one of the performance bottlenecks of FHE computations. Without bootstrapping, an FHE scheme would be used to perform a limited number of operations.

Given the significant importance of sorting algorithms and their ubiquitous applications, it has become imperative to implement sorting algorithms in the FHE domain. However, this requirement is challenging because the mechanics of sorting algorithms are not compatible with homomorphic operations. Basically, sorting algorithms inherently rely on comparing and swapping values of a dataset; this implies that a processor must have access to plaintext values to tell if a variable is greater/less than another variable, and based on such decision, an algorithm decides to swap the values or not. Intuitively, since FHE hides the true values of the variables (i.e. encrypted), a processor becomes unable to tell if some variable is greater/less than another variable. Therefore, sorting algorithms must be tweaked carefully to operate *obliviously*. In other words, comparing two ciphertexts generates an encrypted Boolean value, and the selection of a ciphertext must be performed *homomorphically* based on the encrypted Boolean value.

In order to compare and swap ciphertexts, we have to homomorphically evaluate a comparison operator (e.g., $>$), which outputs an encrypted 1 or 0. Based on this encrypted condition value, we can select the ciphertext that is larger/less than the other. Hence, an FHE scheme must be able to perform a comparison operations, return an encrypted condition bit, and use this bit to select the larger/smaller ciphertext. Implementing this compare-select functionality depends on the FHE schemes, and in some schemes it is infeasible. If a scheme can compare two ciphertexts and returns and encrypted bit s, then a multiplexer gate (MUX) can be used to return one of the ciphertexts. A MUX is a circuit that takes three inputs: the encrypted selector bit s, the first ciphertext a that is returned if s is true, and the second ciphertext b that is returned if s is false, which then evaluates $result \leftarrow (s \times a) + (1 - s) \times b$. Since compare-select is a frequent operation in sorting algorithms, throughout the article, we will use the term "comparison" to refer to evaluating a comparison operation or a comparison operation with a selection.

In this survey, we present a systematic review and analysis of the state-of-the-art works that realize sorting algorithms in the FHE domain. Our goal is to benchmark both classical and FHE-specific sorting algorithms across the three most common FHE schemes on various data sizes. To support this, we introduce, for the first time, a new complexity analysis notation to express the order of growth of FHE sorting algorithms. Our notation, dubbed FHE-Effort, provides a unified framework to demonstrate how an oblivious sorting algorithm scales with increasing input size. Our contributions are summarized as follows:

- We systematize the literature on FHE-based sorting algorithms and empirically evaluate them across various FHE schemes and data sizes.
- We establish a comprehensive empirical baseline by adapting and evaluating classical sorting algorithms that have not previously been analyzed in the FHE literature.
- We introduce the FHE-Effort notation, a novel analytical framework that standardizes the complexity modeling of FHE-based circuits and clearly demonstrates their scalability.

Related Work. In 2015, Emmadi et al. [25] showed the implementation results of basic sorting algorithms, including Bubble, Insertion, Bitonic and Odd-Even Merge (OEM) Sort. All of these algorithms were implemented in the SV [51] homomorphic encryption scheme. The algorithms were tested on datasets of up to 64

elements, demonstrating that OEM Sort is the most efficient algorithm. In 2024, Kim et al. [34] proposed a survey on sorting techniques using FHE. The authors focused on providing a canonical arrangement of early sorting circuit and reviewing the key ideas of each work. Their article did not provide an empirical analysis of any algorithm.

The rest of the manuscript is organized as follows: Section 2 presents a technical background about the main topics discussed in this survey, Section 3 introduces the fundamental constructs that are used to construct our study, Section 4 presents and analyzes the sorting algorithms in the FHE domain, Section 5 presents the empirical evaluation of the algorithms, and finally Section 6 discusses the results and key observations of this study.

2 Background

2.1 Fully Homomorphic Encryption Schemes

The most common FHE schemes are BGV [12], BFV [26], CKKS [16], FHEW [24], and TFHE [17]. Each of these schemes are based on the Learning with Errors (LWE) or Ring-LWE (RLWE) problems, which are hard lattice problems that are post-quantum secure [45]. However, they differ on the mathematical operations supported, the way they manage noise growth, the types of data they operate on, and other special features to execute the computations. OpenFHE [9] and TFHE-rs [56] are the most common open-source FHE libraries found in many FHE works and are maintained regularly by the FHE community. OpenFHE provides efficient implementations of the BGV, BFV, CKKS, and FHEW schemes. TFHE-rs is the official implementation of the TFHE scheme.

BGV and BFV are similar FHE schemes; they perform modular arithmetic over encrypted integers. However, BGV and BFV differ in how they manage noise growth. In particular, BGV manages noise growth through **modulus switching**, which is an operation that switches the ciphertext modulus to another smaller modulus. However, BFV relies on encoding the message in the higher bits (MSBs) of the ciphertext space and keeps the noise in the lower bits (LSBs) of the ciphertext. Unlike integer-based FHE schemes, CKKS is designed to process encrypted real or complex numbers. CKKS is based on approximate arithmetic, where the values lose precision as we perform more computations. The FHEW and TFHE schemes are two similar schemes in the sense that they are both Boolean schemes, i.e, they process encrypted bits. Therefore, the basic operations supported by these schemes are Boolean gates (e.g., AND gate).

Each of these five scheme exhibit different pros and cons. BGV, BFV, and CKKS perform arithmetic operations significantly faster than FHEW and TFHE. In addition BGV, BFV, and CKKS feature the Single-Instruction Multiple-Data (SIMD) capability, which means that a single ciphertext object can encrypt multiple plaintext messages and a single mathematical operation on the ciphertext is reflected on all the encoded messages. This technique is sometimes called packing or batching, and it significantly improves performance by reducing the number of expensive FHE operations. However, TFHE and FHEW excel at evaluating non-linear and comparison operations efficiently, which is a feature that is not supported in other schemes.

Additionally, FHEW and TFHE feature fast bootstrapping procedures that are automatically invoked after each Boolean gate operation. This has the advantage that users can execute circuits of arbitrary depth without handling noise growth manually. However, CKKS and BGV/BFV are originally leveled FHE schemes; they can execute circuits of limited depth depending on the scheme parameters. The current implementation of BGV/BFV in OpenFHE does not support bootstrapping to allow unlimited circuit depth. However, CKKS has received much attention from the OpenFHE community; it has a dedicated bootstrapping procedure to allow for deep circuits. In addition, its OpenFHE implementation supports evaluating Look-Up Tables (LUT) that allow executing non-linear and comparison functions.

Table 1 summarizes the capabilities of each FHE scheme. Both BGV/BFV process encrypted integers and support SIMD operations, but they cannot process real-numbers, perform Boolean operations, or evaluate logical operators. CKKS is a standard for SIMD real-number arithmetic, and its implementation features an efficient bootstrapping procedure. However, CKKS is not suitable for exact integer arithmetic because

encrypted values lose precision during FHE operations. The current implementation of CKKS in OpenFHE allows evaluating Boolean gates and logical operations using a technique called scheme switching (transforming a CKKS ciphertext to FHEW ciphertext to evaluate Boolean gates) or look-up tables. However, these features are not efficient enough yet and are limited in terms of their usability and scalability. The FHEW scheme encrypts individual bits and thus can perform Boolean operations efficiently. The current FHEW implementation in OpenFHE offers the functions to evaluate Boolean gates only, but users can design circuits to perform integer arithmetic and logical operations. Finally, the TFHE scheme as implemented in TFHE-rs library offers optimized functions to perform integer and Boolean arithmetic.

Based on the previous discussion of FHE schemes, we design the sorting routines on CKKS, FHEW, and TFHE-rs only. BGV and BFV schemes do not support a practical evaluation of comparison functions, which are necessary for sorting algorithms. Moreover, sorting circuits are often deep and are not practically feasible to execute in the BGV/BFV schemes. Yet, BGV/BFV can be customized to support a bespoke sorting circuit only on a legacy library called HELib [29] as presented in [32], which is evaluated on Table 5.

Table 1: Common FHE schemes and their capabilities according to their implementation in OpenFHE or TFHE-rs. ● means full support of a feature, ◐ means partial support of a feature, ○ means does not inherently support a feature.

Scheme	Boolean Arithmetic	Integer Arithmetic	Real-number Arithmetic	Bootstrapping	SIMD	Logical Operators
BFV/BGV	○	●	○	○	●	○
CKKS	◐	○	●	●	●	◐
FHEW	●	◐	○	●	○	◐
TFHE	●	●	○	●	○	●

2.2 Asymptotics Analysis and Order of Growth

A typical sorting algorithm works by finding the minimum value among a list of N values and placing it in the correct position in the list. The way to find the minimum value and to iterate over the entire list varies from an algorithm to another. Some algorithms rely on only comparisons and swapping to sort the input, while others involve other mathematical operations, such as the Count Sort algorithm. Regardless of the operations involved in an algorithm, the gold standard approach to estimate how efficient a sorting algorithm is asymptotic analysis.

Asymptotic analysis is the mathematical framework that is used to define the limits of an algorithm's performance asymptotically as the input size N approaches infinity. Essentially, it provides a standardized notation to group algorithms with the same order of growth. Asymptotic analysis involves three standard notations: O (Big-O) to describe an upper bound (worst-case) running time, Ω (Big-Omega) to describe a best-case running time, and Θ (Big-Theta) to describe an average-case running time. We refer the reader to [19] for a detailed discussion about algorithm analysis.

Asymptotic analysis does not consider the subtle details of a particular algorithm and mainly focus on how an algorithm scales with the input size. For example, if we assume that some sorting algorithm takes one full iteration on a list of N values to sort it, then we say that it is a $O(N)$ algorithm, since its running time is linear. Here, the notation does not mention anything about the operations involved in each iteration; whether the algorithm performs comparisons only or comparisons with other operations, the algorithm is still $O(N)$. In fact, this concept is consistent with the purpose of asymptotic analysis to describe how an algorithm scales with the input size.

However, we argue that asymptotic analysis becomes less meaningful in the FHE-sorting domain for two reasons. First, the type of operations involved in a sorting algorithm exhibit significant performance differences that can be utilized to discriminate between various sorting algorithms. For example, two FHE

sorting algorithms are $O(N^2)$, but one involves comparisons only, while the second involves comparisons and multiplications. In this case, the $O(N^2)$ does not tell which routine is more efficient. The second reason is that asymptotic notations omit constant and low-order factors that describe the order of growth of a routine. However, it turns out that these constant factors are critically important for analyzing an FHE-sorting routine.

Therefore, in this article, we propose a new notation to estimate the order of growth of FHE circuits. Our notation, dubbed *FHE-Effort*, is a dialect of asymptotic analysis notation that describes the order of growth of oblivious algorithms in terms of the basic FHE operations, namely addition, multiplication, comparison, bit-wise operations (Boolean gates) and shifting and rotation. **Our notation does not replace or omit the fundamental asymptotic notation, but instead it specially complements the analysis for oblivious sorting algorithms to characterize the complexity of an algorithm**. The FHE-Effort notation is explained in Section 3.2.

2.3 Sorting in the FHE Domain

Implementing sorting algorithms in the FHE domain is not a plug-and-play process, as it imposes several challenges that impede adapting the unique merits of each sorting circuit. A sorting circuit can either be a *Primitive Iterative Circuit* (PIC) or a *Variable Iterative Circuit* (VIC). We define PIC circuits to refer to algorithms that involve a known number of iterations, whereas VIC defines the algorithms that involve an unknown number of iterations. PICs include algorithms that are often based on loops where the number of iterations is fixed or based on the input size. Meanwhile, VICs include algorithms that keep iterating over the dataset until the data becomes sorted. VIC circuits depend on the current state of the dataset and do not terminate until some condition indicates that the dataset is sorted. For example, Quicksort is a VIC circuit, it works by selecting a pivot element and moves elements to the left or right based on their values relative to that pivot. In this case, the entire Quicksort approach relies mainly on the movement of the pivot with other values. If we cannot see the results of the comparisons to terminate the algorithm, then it will never terminate.

Furthermore, the depth of a sorting circuit is a critical bottleneck in FHE. Each multiplication or comparison operation increases the noise of a ciphertext. If the noise is too large, bootstrapping has to be invoked, otherwise the data will be corrupted. Bootstrapping itself introduces long latencies that increases the running time of a circuit.

3 Building Blocks

All FHE schemes presented in Table 1 support addition, subtraction, and multiplications. Other operations, such as shifting, comparison and selections, and bit-wise operations, differ from scheme to another. In this section, we review how these operations are implemented in each scheme.

TFHE is the most flexible scheme as its official library implementation (TFHE-rs) provides native functions to perform various encrypted computations. This flexibility stems from the fact this scheme encrypts bits, and thus can execute bit-wise and shifting operations efficiently. In addition, TFHE-rs API enables executing comparisons through two approaches. The first is through evaluating a comparator circuit (e.g., $>$ and $<=$) that outputs encrypted $True$ or $False$. Then, we evaluate the MUX circuit to select between two values. The second approach is using the built-in `max` and `min` functions that return the maximum or minimum of two values. Our experiments show that the first approach is faster than the second when we need to get the maximum/minimum of two values. This is because we evaluate the condition once and then use the same condition twice to select the maximum and minimum value using MUX circuit. Meanwhile, the second approach evaluates the condition and performs MUX selection every time we call `max` or `min`.

FHEW is similar to TFHE, where it performs bootstrapped Boolean gates efficiently. Unlike TFHE-rs, OpenFHE implements FHEW with the basic Boolean gates only and does not provide built-in functions to perform other arithmetic operations. To utilize FHEW, we implemented the Instruction Set Architecture (ISA) of the PulpFHE processor [1]. This open-source ISA provides a functionally complete set of operations

(e.g, multiplication and comparison) implemented for the TFHE scheme. We ported the circuits logic to FHEW. Moreover, PulpFHE ISA is originally implemented to be multi-threaded, but our implementation is single-threaded to ensure a fair comparison among other schemes.

CKKS natively supports SIMD addition, subtraction, and multiplication over vectors of encrypted real numbers, so we realize the arithmetic operations (σ and π) directly with the library primitives: `EvalAdd`/ `EvalSub` for addition and subtraction and `EvalMult` for multiplication. Because our comparison context uses manual rescaling (the FIXEDMANUAL scaling technique), every ciphertext–ciphertext or ciphertext–plaintext product is followed by an explicit rescale that returns the ciphertext to a unit scaling factor before it is reused. Shifting and rotation (λ) are performed with `EvalRotate` under precomputed rotation keys; as in the other schemes the shift amount is a public (plaintext) constant, so a rotation is a single key-switch. Because CKKS packs all n elements into the slots of one ciphertext, a single rotation aligns every element with its comparison partner at once, which is what makes the batched compare–swap networks practical.

Unlike TFHE and FHEW, CKKS provides no native comparison, selection, or Boolean gates. Unlike the earlier CKKS sorting works [35, 40, 46], our FBT implementation does not approximate the comparison with a high-degree Chebyshev polynomial of the sign function; instead we evaluate comparisons *exactly* through CKKS functional bootstrapping (FBT) [4]. To compare two packed operands a and b, we compute their difference $d = a - b$ with a single subtraction and then extract its exact sign. The sign is recovered by peeling the value one digit (here one bit) at a time: each digit is isolated by a CKKS functional bootstrap that evaluates the Boolean bit-extraction look-up table through a complex-exponential / trigonometric Hermite interpolation, the reconstructed digit is subtracted off, and the modulus is halved before the next iteration. Once the lower digits have been peeled away, the remaining most-significant digit is the sign of the difference, which is 1 exactly when $a < b$ and 0 otherwise. Because this result comes from a table evaluation rather than a polynomial approximation, the comparison (α) is exact over the full P-bit signed domain, at the cost of one functional bootstrap per digit (the dominant cost of every CKKS sorting routine we implement).

Selection (the MUX in α) and the min/max used by compare–swap are then linear. Given the comparison bit $s = [\, a > b \,]$ for an ascending pair, we form the transport amount $\delta = s \cdot (a - b)$ with one multiplication and update the pair additively. Consequently, the left slot becomes $a - \delta$ (the minimum) and the right slot becomes $b + \delta$ (the maximum), placing both results in-place from a single comparison and mirroring the compare-select optimization described for the other schemes. Our transport compare–swap goes one step further and fuses the comparison with the noise refresh: peeling d already reconstructs a fresh, full-precision value alongside its sign, so a single functional bootstrap performs the comparison and re-exactifies the moved data with no separate refresh.

Finally, CKKS has no native bit-wise operators (β). When an algorithm needs a bit-level view of a value (for example the digit keys of Radix Sort), we obtain the individual bits from the same functional-bootstrap digit-peeling used for comparison, and realize the Boolean gates arithmetically on the resulting 0/1 slots (AND as ab, XOR as $a + b - 2ab$, OR as $a + b - ab$).

Table 2 lists the latency (in milliseconds) of each FHE operation in the CKKS, TFHE, and FHEW schemes. The Symbol column identifies the category of the operation as explained later in Section 3.2.

3.1 Blind Array Operations

Some oblivious sorting algorithms (e.g., Selection Sort) require accessing ciphertexts of an array using an encrypted index. This may seem counter-intuitive, because since an index of an element is encrypted, then we cannot fetch the element at this index. In fact, this can be implemented in the FHE domain, however, the solution is quite expensive. Blind Array Access (`BAA`) is a method that was introduced in [8] that utilized Look-Up Tables (LUT) in TFHE to blindly rotate a ciphertext array by an encrypted amount (index) and extract the first element. This specific approach is fast, however, it is specifically designed to work on TFHE-rs library and is not directly extendable to other schemes.

To utilize `BAA` across various algorithms and schemes, we redefined its implementation to adapt a brute-force search approach. In a nutshell, our `BAA` implementation takes an encrypted array and an encrypted index as inputs, iterates over every element in the array, encrypts the array counter and compares it with the index. The comparison yields an encrypted true (1) or false (0), which is then used by the `MUX` function to

return the target element. Algorithm 1 lists the `BAA` procedure. We also extended BAA to implement Blind Array Update (`BAU` Algorithm 2), which updates a ciphertext at an encrypted index, Blind Array Increment (`BAI` Algorithm 3), which increments a ciphertext at an encrypted index by 1, and Blind Array Swap (`BAS` Algorithm 4), which swaps two ciphertexts at encrypted index ia and ib.

Algorithm 1: Blind Array Access

```
1 Procedure BAA(A, index):
2     result ← Enc(0)
3     for i ← 0 to length(A) − 1 do
4         is_match ← index = Enc(i)
5         result ← is_match ? A_i : result
6     return result
```

Algorithm 2: Blind Array Update

```
1 Procedure BAU(A, index, val):
2     n ← length(A)
3     for j ← 0 to n − 1 do
4         is_target ← index = Enc(j)
5         A_j ← is_target ? val : A_j
```

Algorithm 3: Blind Array Increment

```
1 Procedure BAI(A, index):
2     result ← empty A
3     for j ← 0 to length(A) − 1 do
4         is_target ← index = Enc(j)
5         append(result, (A_j + is_target))
6     return result
```

Algorithm 4: Blind Array Swap

```
1 Procedure BAS(A, ia, ib):
2     va ← BAA(A, ia) ; vb ← BAA(A, ib)
3     for i ← 0 to length(A) − 1 do
4         is_a ← ia = Enc(i) ; is_b ← ib = Enc(i)
5         val ← is_a ? vb : A_i
6         A_i ← is_b ? va : val
```

3.2 The FHE-Effort Notation

The FHE-Effort $\mathcal{F}$ notation describes how the complexity of FHE algorithms grows in terms of the basic FHE operations as the input size increases. The $\mathcal{F}$ notation is a dialect of the asymptotic analysis notations to expound a tight bound on the order of growth of an algorithm. We define $\mathcal{F}$ as in Formula 1, where n is the input size, α means n comparison operations, σ means n addition/subtraction operations, π means n multiplication operations, β means n bit-wise (Boolean) operations, λ means n shifting/rotation operations. The terms c_1, c_2, c_3, c_4 and c_5 represent non-negative algorithmic coefficients that scale based on the specific circuit complexity.

$$\mathcal{F}(n) = c_1\alpha + c_2\sigma + c_3\pi + c_4\beta + c_5\lambda \;\equiv\; \mathcal{F}(c_1\alpha + c_2\sigma + c_3\pi + c_4\beta + c_5\lambda) \tag{1}$$

To understand the $\mathcal{F}$ notation, we provide the example in Algorithm 5. This example iterates over a list of n values and performs a sequence of addition, multiplication, comparison, and bit-wise operations. The algorithmic complexity of Algo.5 is $O(n)$, since the function visits each element of the list exactly once. To expound this algorithm in the FHE-Effort notation, we analyze the types of operations and their number. In each iteration, the algorithm performs an addition and a multiplication (lines 4 and 5), 2 comparisons (line 6), and a bit-wise AND operation (line 7), and no shifting operations. Since the total number of iterations is n, we say that the FHE-Effort of this algorithm is $\mathcal{F}(n) = (2\alpha + 1\sigma + 1\pi + 1\beta + 0\lambda)$, or equivalently $\mathcal{F}(2\alpha + \sigma + \pi + \beta)$.

FHE-Effort can be generalized to represent the set of algorithms that exhibit a similar order of growth by omitting constants or keeping the dominant factors only. For example, we can let $\mathcal{F}(\alpha^2)$ to denote the set of algorithms that perform n^2 comparisons, regardless whether there are other operations or not. However, we recommend the fine-grained notation to give the full picture about an algorithm complexity, which in turns helps discriminating between various algorithms. Definition 3.1 provides a formal definition of the FHE-Effort notation and lists the rules to be followed to analyze an FHE algorithm.

Algorithm 5: List Processing with Arithmetic and Boolean Operations

Input: A list **values** containing N numerical elements
Output: The total sum, total product, and a count of specific elements

```
1 Procedure ProcessList(values):
2     total_sum ← 0 ; total_product ← 1
3     foreach element ∈ values do
4         total_sum ← total_sum + element
5         total_product ← total_product × element
6         cond_1 ← (element <= 100) ; cond_2 ← (element > 0)
7         in_bounds ← cond_1 ∧ cond_2
8     return total_sum, total_product, matching_count
```

DEFINITION 3.1 — FHE-EFFORT

Let $\alpha, \sigma, \pi, \beta, \lambda$ denote n comparison, addition/subtraction, multiplication, bit-wise, and shifting operations, respectively. The FHE-Effort $\mathcal{F}$ notation expresses a tight bound on the complexity of an FHE function in terms of $\alpha, \sigma, \pi, \beta$, and λ. It can be written as

$$\mathcal{F}(c_1\alpha + c_2\sigma + c_3\pi + c_4\beta + c_5\lambda) \mid c_{[1:5]} \text{ are non-negative algorithmic coefficients}$$

FHE-Effort analysis follows the following rules:

1. *Count only plaintext-ciphertext and ciphertext-ciphertext operations.*
2. *Do not count the following: data movement, variables initialization, plaintext-plaintext operations, homomorphic negation (*`not`*), and array indexing.*
3. *A single comparison operation (i.e.,* `x > y`*), a single selection operation (i.e.,* `res = cond? A:B`*, or a comparison followed by a selection operation (e.g.,* `cond = x > y; res = cond? A:B`*) is counted as one.*
4. *A single comparison operation followed by L selection operations are counted as L. For example,* `cond = x > y; r1 = cond? A:B; r2 = cond? R:Q` *means 2 comparisons (we compute* `cond` *once and performs selection twice using same* `cond`*).*

Note on Bootstrapping: To maintain a scheme-agnostic abstraction, FHE-Effort evaluates core logical operations independently of bootstrapping overhead. Noise management mechanics are highly scheme-dependent, ranging from implicit per-gate evaluations in TFHE to explicit, depth-based invocations in CKKS. By abstracting these variables, FHE-Effort establishes a standardized baseline for comparing algorithmic efficiency.

Based on the FHE-Effort definition, we analyze blind array operations (Algorithms 1, 2, 3, and 4) as follows:

- `BAA` performs n iterations. In each iteration it performs a comparison (an equality check at line 4 and a selection at line 5). Therefore, its FHE-Effort is $\mathcal{F}(\alpha)$. The same applies to `BAU` which also costs $\mathcal{F}(\alpha)$.
- `BAI` performs n iterations. In each iteration it performs an equality check (line 4) and an addition (line 5). Therefore, its cost is $\mathcal{F}(\alpha + \sigma)$.
- `BAS` performs two `BAA` operations (line 2) and n iterations. In each iteration it performs two comparisons (two equality checks at line 4 and two selections at lines 5 and 6). Since each `BAA` costs $\mathcal{F}(\alpha)$ and the for loop costs $\mathcal{F}(2\alpha)$. Hence, the total cost of the procedure is $\mathcal{F}(4\alpha)$.

Table 2: Average latency of basic operations in each FHE scheme based on 10 runs on two 8-bit ciphertext. For the λ operations, the shift amount was in plaintext. All the results are in milliseconds, unless reported otherwise. CKKS latencies are measured at ring dimension (N=65,536) and reported per operation per element, amortized across all (32,768) packed SIMD slots.

Operation	**Symbol**	**CKKS**	**TFHE**	**FHEW**
Addition	σ	0.0002	58	1709
Subtraction		0.0003	56	1703
Multiplication	π	0.004	122	16736
Bit-wise AND	β	0.003	19	377
Bit-wise OR		0.004	19	371
Bit-wise XOR		0.004	19	370
Left Shift/Rotate	λ	0.002	4 (μ secs)	4.5 (μ secs)
Right Shift/Rotate		0.003	19	1.5 (μ secs)
Greater Than	α	2	55	2981
Less Than		2	55	2958
Equals To		4	37	2959
Selection (MUX)		0.004	49	1115

4 Oblivious Sorting

In this section we review sorting algorithms that were specifically proposed for FHE applications in addition to foundational sorting algorithms that are commonly found in the field of computer science. Also, we provide the FHE-Effort analysis of each algorithm and show how it is different from the classical asymptotic analysis notations. Table 3 lists the sorting algorithms analyzed in this article along with the corresponding reference in which an algorithm is implemented using FHE. As shown in Table 3, we have 18 algorithms implemented in different FHE schemes. Also, only 5 of the works provide public artifacts for their source code. This imposes challenges that impede replicating the algorithms and reproducing the results across various schemes. Note that some works implement their algorithms in a legacy or non-standard FHE implementation. In particular, the Insertion and Bubble Sort algorithms in [14, 15] are implemented in the SV scheme [51] using Scarab library.[1] This scheme is not supported any more and may lack the current requirements for an efficient and secure FHE system. The Direct and Greedy Sort algorithms (in [13, 18]) are implemented on a custom schemes which are not publicly available.

We implemented all the algorithms in a unified codebase to ensure a fair analysis of the algorithms and to ensure reproducible results and replication across various schemes and libraries. We note that our implementation may be slightly different from the actual algorithm in cases where the algorithm relies on specific assumptions or capabilities of the underlying FHE scheme or library. In the following discussion, we list the algorithms in an idiomatic syntax matching very closely our implementations.

4.1 Bubble and Insertion Sort

Bubble and Insertion sort are classic comparison-based sorting algorithms, but they operate using different approaches. Basically, Bubble Sort works by repeatedly comparing adjacent elements and swapping them if necessary. Meanwhile, Insertion Sort builds a sorted sub-list of one element at a time and inserts it into the correct position among the already sorted elements. The algorithmic complexity of both algorithms is $O(n^2)$. The oblivious version of Bubble and Insertion are listed in Algorithms 6 and 7, respectively. The oblivious version of Bubble Sort was presented in the works [53], which is implemented in the TFHE scheme using the C++ API and in [15], which is implemented in the SV scheme. Oblivious Insertion Sort was also implemented in SV in [14].

[1] https://github.com/hcrypt-project/libScarab

Table 3: The sorting algorithms covered in this survey, including their corresponding references and links to open-source artifacts. Cells with a dash (—) indicate that the algorithm has not been found in the FHE literature or does not have a publicly accessible artifact (prior to our implementation). Notably, while previous works targeted specific FHE schemes, our unified codebase adapts and benchmarks all listed algorithms across CKKS, TFHE, and FHEW.

Algorithm	**Reference**	**Target Scheme**	**Existing Open-source Artifacts**
Bubble Sort	[53]	TFHE	—
	[15]	SV	—
Insertion Sort	[14]	SV	—
Count Sort	[8]	TFHE	github.com/sofianeazogagh/knn
Addition Sort	[53]	TFHE	—
Direct Sort	[32]	BGV/BFV	github.com/iliailia/comparison-circuit-over-fq
	[13]	NTRU-based Scheme	—
	[41]	BGV	—
	[18]	Variant of BGV	—
Matrix-Rank Sort	[40]	CKKS	github.com/FedericoMazzone/openfhe-statistics
Greedy Sort	[13]	NTRU-based Scheme	—
	[18]	Variant of BGV	—
Bitonic Sort	[46]	CKKS	github.com/lorenzorovida/Lightweight-Sorting-In -Approximate-Homomorphic-Encryption
Rank Sort	[35]	CKKS	github.com/oksuman/sorting-fhe
Radix Sort	[42]	TFHE	—
Odd-Even Merge Sort	[41]	BGV	—
K-Way Sort	[31]	CKKS	—
Selection Sort	—	—	—
Cocktail Shaker Sort	—	—	—
Shell Sort	—	—	—
Gnome Sort	—	—	—
Heap Sort	—	—	—
Merge Sort	—	—	—

To perform the FHE-Effort analysis, we notice that the algorithms have two nested loops and perform two comparisons, namely `max` and `min` in Bubble Sort and compare-select ($>$ and MUX) in Insertion Sort. Hence, a simple analysis tells us the number of comparisons is α^2. However, to provide a more precise analysis, we notice that the second loop of Bubble Sort iterates from $j = 0$ to $n - i - 1$, Hence, the second loop performs fewer iterations as the counter of the first loop increases. Similarly, the second loop of Insertion Sort starts from $j = i$ instead of 0, and also performs fewer iterations as the counter of the first loop increases. By tracing the loops of both algorithms, we find that the second loops decrease their iteration by 1 after each iteration of the first loop. Hence, the FHE-Effort of both algorithms is $\mathcal{F}(\alpha^2 - \alpha)$.

Notice that the pseudocode of both algorithms is similar, however there is a slight difference that is intentionally made to reflect the implementation details. Specifically, the comparison in Bubble Sort uses the `max` and `min` functions. Meanwhile, the comparison of Insertion Sort relies on the $>$ operator (line 5),

then uses the same comparison bit to get the minimum and maximum value using MUX gate (lines 6 and 7). Recall that `max/min` internally behaves like executing a comparison operator and uses the comparison bit to perform selection using MUX. This introduces redundancy because the comparison bit is computed twice even when comparing the same elements. Therefore, we argue that the compare-select approach is more efficient than using `max/min` functions, and to support our claim, we keep the Bubble Sort using `max/min` to show that it will introduce a higher latency than Insertion Sort, although they belong to the same algorithmic and FHE-Effort complexity.

Algorithm 6: Oblivious Bubble Sort

1 **Procedure** `BubbleSort(`D`)`**:**
2 $n \leftarrow \text{length}(\mathbf{D})$
3 **for** $i \leftarrow 0$ **to** $n-1$ **do**
4 **for** $j \leftarrow 0$ **to** $n-i-1$ **do**
5 $max \leftarrow \max(\mathbf{D}_j, \mathbf{D}_{j+1})$
6 $min \leftarrow \min(\mathbf{D}_j, \mathbf{D}_{j+1})$
7 $\mathbf{D}_j \leftarrow min$
8 $\mathbf{D}_{j+1} \leftarrow max$

Algorithm 7: Oblivious Insertion Sort

1 **Procedure** `InsertionSort(`D`)`**:**
2 $n \leftarrow \text{length}(\mathbf{D})$
3 **for** $i \leftarrow 1$ **to** $n-1$ **do**
4 **for** $j \leftarrow i$ **to** 1 *by* -1 **do**
5 $swap \leftarrow \mathbf{D}_{j-1} > \mathbf{D}_j$
6 $new_left \leftarrow swap\ ?\ \mathbf{D}_j : \mathbf{D}_{j-1}$
7 $new_right \leftarrow swap\ ?\ \mathbf{D}_{j-1} : \mathbf{D}_j$
8 $\mathbf{D}_{j-1} \leftarrow new_left$
9 $\mathbf{D}_j \leftarrow new_right$

4.2 Selection Sort

Selection Sort works by repeatedly finding the minimum element from the unsorted sub-list and swapping it with the first element of the unsorted list. Its algorithmic complexity is $O(n^2)$. The plaintext algorithm can be found in [3], while the oblivious version is listed in Algorithm 8. The algorithm works by assuming i is the index of the minimum value (line 4), then performs `BAA` to get the value at that minimum index and compares with the current minimum value and swaps them using MUX (lines 6-7). After the second loop, the algorithm fetches the minimum value by calling `BAA` (line 10), and finally it updates the value at index `min_index` with the value `tmp` at index i (line 11). The FHE-Effort of Selection Sort is $\mathcal{F}((\alpha^3 + 4\alpha^2 - \alpha)/2)$, which is broken down as shown in Proof 4.1.

Algorithm 8: Oblivious Selection Sort

1 **Procedure** `SelectionSort(`D`)`**:**
2 $n \leftarrow \text{length}(\mathbf{D})$
3 **for** $i \leftarrow 0$ **to** $n-1$ **do**
4 $min_idx \leftarrow \text{Enc}(i)$
5 **for** $j \leftarrow i+1$ **to** $n-1$ **do**
6 $min_val \leftarrow \text{BAA}(\mathbf{D}, min_idx)$
7 $is_gt \leftarrow min_val < \mathbf{D}_j$
8 $min_idx \leftarrow is_gt\ ?\ min_idx : \text{Enc}(j)$
9 $tmp \leftarrow \mathbf{D}_i$
10 $\mathbf{D}_i \leftarrow \text{BAA}(\mathbf{D}, min_idx)$
11 $\text{BAU}(\mathbf{D}, min_idx, tmp)$

Algorithm 9: Oblivious Addition Sort

1 **Procedure** `AdditionSort(`D`)`**:**
2 $n \leftarrow \text{length}(\mathbf{D})$
3 **ranks** $\leftarrow [\text{Enc}(0), \ldots, \text{Enc}(0)]$ of size n
4 **for** $i \leftarrow 0$ **to** $n-1$ **do**
5 $m \leftarrow \text{Enc}(0)$
6 **for** $j \leftarrow 0$ **to** $n-1$ **do**
7 $m \leftarrow m + (\mathbf{D}_i \geq \mathbf{D}_j)$
8 $ranks_i \leftarrow m - \text{Enc}(1)$
9 **return ranks**

PROOF 4.1 — Selection Sort FHE-Effort

1. The outer loop runs n times, and the inner loop runs from $j = i + 1$ to n. Hence, lines 6-8 are executed $Iters = \sum_{i=0}^{n-1}(n - 1 - i) = \frac{n(n-1)}{2}$ times.
2. A single call to `BAA` or `BAU` costs n comparisons, i.e., α.
3. Line 6 calls `BAA` $Iters$ times, line 10 calls `BAA` n times, and line 11 calls `BAU` n times.
4. Lines 7-8 are considered as a one comparison operation, which is invoked $Iters$ times.

$$\therefore \text{The number of inner loop comparisons is } Iters \times (n + 1) = \frac{n(n^2 - 1)}{2} \equiv \frac{\alpha^3 - \alpha}{2}$$

and

$$\therefore \text{The outer loop comparisons are } n \times 2n \equiv 2\alpha^2$$

Hence, the FHE-Effort is

$$\frac{\alpha^3 - \alpha}{2} + 2\alpha^2 \equiv \frac{\alpha^3 + 4\alpha^2 - \alpha}{2}$$

□

4.3 Addition Sort

Addition Sort was proposed as an FHE-specific algorithm [53], which was implemented in TFHE using the C++ API. The pseudocode is listed in Algorithm 9. It works by computing the rank position of the i-th element in the list. It does not sort the array directly, instead it outputs a rank list, which is decrypted by users and is used to arrange the elements in order. When implementing the algorithm as described in the original paper, we found that it does not always output the correct ranks. Upon analyzing the algorithm, we made two key modifications to resolve this. First, the inner loop of the original algorithm iterates from $j = i$ to $n - 1$, we changed the starting index to $j = 0$. Second, the output ranks were consistently offset by $+1$ from the expected results, therefore we added a -1 operation at line 8. Although these modifications increase the average latency of the algorithm, they ensure that the output is always correct.

To analyze the FHE-Effort of Addition Sort, we notice that it has two nested loops, each performs n iterations. The second loop performs an addition and a comparison (line 7); meanwhile the outer loop performs subtraction only. Therefore, it is clear that for a dataset of size n, this algorithm performs n^2 additions and comparisons and n subtractions. Hence, the FHE-Effort is $\mathcal{F}(\alpha^2 + \sigma^2 + \sigma)$.

4.4 Cocktail Shaker Sort

Cocktail Shaker Sort [36], also known as Bidirectional Bubble Sort, is a variation of the classic Bubble Sort algorithm that performs two passes over the dataset. Specifically, it performs a forward pass (moving from left to right) to "bubble" the largest element to the end of the list, and a backward pass to "bubble" the smallest element to the beginning of the list. Like Bubble Sort, it is an $O(n^2)$ algorithm. Cocktail Shaker is not found in any FHE work before, but we include it in our study to ensure a comprehensive coverage of sorting algorithms in the FHE domain and to stress the fact that similar sorting algorithms can exhibit notable performance difference when applied in the FHE domain.

Algorithm 10 lists the oblivious Cocktail Shaker Sort. If n is even, the FHE-Effort is $\mathcal{F}(\alpha^2)$. If n is odd, the FHE-Effort is $\mathcal{F}(\alpha^2 - 1)$. For sake of simplicity, we fix the FHE-Effort to be $\mathcal{F}(\alpha^2)$. This is proved in Proof 4.2.

PROOF 4.2 — Cocktail-Shaker Sort FHE-Effort

1. The outer loop iterates from $pass = 0$ to $\lfloor n/2 \rfloor$, the first inner loop iterates from $i = pass$ to $n - 1 - pass$, and the second inner loop iterates from $i = n - 1 - pass$ to $pass$.

2. $\therefore$ The first inner loop performs $(n-1-pass)-pass = n-1-2pass$ and the second inner loop performs $(n-pass)-(pass+1) = n-1-2pass$.
3. $\therefore$ The total number of iterations is

$$\sum_{pass=0}^{\lfloor \frac{n}{2} \rfloor - 1} 2(n-1-2\cdot pass) = \left\lfloor \frac{n^2}{2} \right\rfloor$$

4. $\because$ The algorithm only performs comparisons, and each of the inner loops performs 2 comparisons (1 compare operator) and two selections. Then, the cost of comparisons is 2α.
5. $\therefore$ The FHE-Effort is $\left\lfloor \frac{2\alpha^2}{2} \right\rfloor \equiv \begin{cases} \mathcal{F}(\alpha^2) & \text{if } n \pmod 2 = 0 \\ \mathcal{F}(\alpha^2-1) & \text{Otherwise} \end{cases}$
6. We simplify the effort to be $\mathcal{F}(\alpha^2)$

□

4.5 Shell Sort

Shell Sort [36] is an optimized comparison-based sorting algorithm that compares elements that are far apart using "gap sequence" steps rather than just comparing adjacent elements. The main feature of the plaintext Shell Sort is that it can terminate early based on a condition in the very inner loop. However, since early termination is impossible in FHE, the algorithm has to traverse the entire dataset unconditionally. Like Cocktail Shaker Sort, the algorithm is not found in FHE works, but it is included for the sake of comprehensiveness. The complexity of Shell Sort depends on the gap sequence; for a basic gap sequence of $n/2$, it becomes $O(n^2)$. The oblivious Shell Sort is shown in Algorithm 11. Assuming the gap sequence is $n/2$ and $n = 2^k \mid k \in \mathbb{Z}$, its FHE-Effort becomes $\mathcal{F}(2\alpha^2 - 2\alpha - \alpha\log_2(\alpha))$. This is obtained by analyzing the behavior of the algorithm bottom-up as in Proof 4.3.

PROOF 4.3 — Shell Sort FHE-Effort

1. The number of comparisons is 2: 1 comparison operator (line 8) and 2 selections (lines 9 and 10)
2. Assuming a fixed `gap` for the inner `while` loop. The number of iterations is $\left\lfloor \frac{j}{gap} \right\rfloor$ (j reduces by `gap`).
3. The middle `for` loop runs for n iterations. Hence, the total number of iterations from line 5 to 13 is given by $I(n, gap) = \sum_{i=gap}^{n-1} \left\lfloor \frac{i}{gap} \right\rfloor$.
4. Since the outer `while` loop runs from $gap = \lfloor \frac{n}{2} \rfloor$ to 0 stepping by $gap/2$, it gives $\log_2(n)$ iterations.
5. $\therefore$ The total number of iterations is given by $\sum_{k=1}^{\lfloor \log_2(n) \rfloor} I(n, gap) \mid gap = \left\lfloor (\frac{n}{2^k}) \right\rfloor$ for $k \in [1 : \log_2(n)]$.
6. Under the assumption that n is powers of 2, the FHE-Effort simplifies to $\mathcal{F}(2\alpha^2 - 2\alpha - \alpha\log_2(\alpha))$.

□

4.6 Gnome Sort

Gnome Sort [48] is an optimized version of the Insertion Sort that does not use nested loops. It works by iterating over an array, comparing adjacent elements, and swapping them. It then cascades values into their correct position by stepping backward. Its time complexity is $O(n^2)$. The plaintext algorithm is quite simple to implement and fairly efficient. However, it exhibits a significant complexity when implemented in FHE. This is because the original algorithms relies on a single indexing variable `pos` that is used to access the elements

Algorithm 10: Oblivious Cocktail Shaker Sort

```
1  Procedure CocktailShakerSort(D):
2      n ← length(D)
3      for pass ← 0 to ⌊n/2⌋ − 1 do
4          for i ← pass to n − 2 − pass do
5              cond ← D_i > D_{i+1}
6              max ← cond ? D_i : D_{i+1}
7              min ← cond ? D_{i+1} : D_i
8              D_i ← min
9              D_{i+1} ← max
10         for i ← n − 1 − pass to pass + 1 by −1
               do
11             cond ← D_{i−1} > D_i
12             max ← cond ? D_{i−1} : D_i
13             min ← cond ? D_i : D_{i−1}
14             D_{i−1} ← min
15             D_i ← max
```

Algorithm 11: Oblivious Shell Sort

```
1  Procedure ShellSort(D):
2      n ← length(D)
3      gap ← ⌊n/2⌋
4      while gap > 0 do
5          for i ← gap to n − 1 do
6              j ← i
7              while j ≥ gap do
8                  cond ← D_{j−gap} > D_j
9                  j_val ← cond ? D_{j−gap} : D_j
10                 j_gap ← cond ? D_j : D_{j−gap}
11                 D_j ← j_val
12                 D_{j−gap} ← j_gap
13                 j ← j − gap
14         gap ← ⌊gap/2⌋
```

of the array and putting them into their correct position. This index `pos` will be encrypted to ensure a correct oblivious version of the algorithm as listed in Algorithm 12. Its FHE-Effort is $\mathcal{F}(6\alpha^3 + 5\alpha^2 + 3\sigma^2 + 3\beta^2)$, which is obtained as in Proof 4.4.

PROOF 4.4 — Gnome Sort FHE-Effort

1. The algorithm has 1 loop that performs $max_steps = n^2$ iterations.
2. Lines 7 and 9 resemble 1 comparison operation, lines 8 and 11 resemble another comparison operation, line 14 is a third comparison operation, and lines 19 and 20 are 2 selection operations, which are counted as 2 comparisons. Hence, so far we have 5 comparisons, which is equivalent to $5\alpha^2$.
3. Lines 12 and 13 are two blind array access functions, each cost α. The blind array swap function at line 21 costs 4α. Since lines 12, 13, and 21 are inside a n^2 loop, their total cost is $2\alpha^3 + 4\alpha^3 = 6\alpha^3$.
4. Line 10 is a subtraction operation and line 22 is an addition and a subtraction. They count as $3\sigma^2$ as they are inside the loop. Similarly, lines 16-18 are $3\beta^2$ operations. Notice that FHE-Effort does not include $\neg$ operation (line 15) because it is a cheap operation.
5. Hence, the total FHE-Effort becomes $\mathcal{F}(6\alpha^3 + 5\alpha^2 + 3\sigma^2 + 3\beta^2)$.

□

4.7 Rank Sort

Rank-based Sorting is a sorting technique proposed in [35] that is tailored for sorting real-numbers encrypted in the CKKS scheme. The algorithm consists of two phases: The first phase is rank construction, in which the algorithm constructs a rank vector `ranks`, such that `ranks[i]` denotes the number of elements smaller than the i-th element. The second phase is rotation index checking, in which the input dataset is rearranged (rotated) according to the rank vector `ranks`. This algorithm is a bit similar to the Addition Sort in 9, but this time the rank is applied obliviously at the server side. The original algorithm is designed specifically to exploit the SIMD properties of the CKKS scheme; hence, it is not an agnostic solution that can be seamlessly

Algorithm 12: Oblivious Gnome Sort

```
1  Procedure GnomeSort(D):
2      n ← length(D)
3      max_steps ← n × n
4      pos ← Enc(0)
5      n_enc ← Enc(n)
6      for step ← 0 to max_steps − 1 do
7          is_done ← pos ≥ n_enc
8          is_zero ← pos = Enc(0)
9          pos_s ← is_done ? Enc(0) : pos
10         pos_1 ← pos_s − Enc(1)
11         pos_prev ← is_zero ? Enc(0) : pos_1
12         val_current ← BAA(D, pos_s)
13         val_prev ← BAA(D, pos_prev)
14         is_ge ← val_current ≥ val_prev
15         not_done ← ¬is_done
16         ge_0 ← is_zero ∨ is_ge
17         mv_fwd ← ge_0 ∧ not_done
18         mv_bkwd ← ¬ge_0 ∧ not_done
19         swap_a ← mv_bkwd ? pos_s : Enc(0)
20         swap_b ← mv_bkwd ? pos_prev : Enc(0)
21         BAS(D, swap_a, swap_b)
22         pos ← pos + mv_fwd − mv_bkwd
```

Algorithm 13: Oblivious Rank Sort

```
1  Procedure RankSort(D):
2      n ← length(D)
3      ranks ← [Enc(0), ..., Enc(0)] of size n
4      for i ← 0 to n − 1 do
5          accum ← Enc(0)
6          for j ← 0 to n − 1 do
7              if i = j then
8                  continue
9              is_gt ← D_i > D_j
10             if i > j then
11                 is_eq ← D_i = D_j
12                 accum ← accum + is_gt + is_eq
13             else
14                 accum ← accum + is_gt
15         ranks_i ← accum
16     result ← [Enc(0), ..., Enc(0)] of size n
17     for k ← 0 to n − 1 do
18         slot_val ← Enc(0)
19         for i ← 0 to n − 1 do
20             is_match ← ranks_i = k
21             slot_val ← slot_val + (is_match × D_i)
22         result_k ← slot_val
23     D ← result
```

ported to any FHE scheme. To ensure compatibility and portability across various schemes, we redesigned the algorithm as shown in Algorithm 13 while maintaining its core logic and behavior.

The authors analyzed the circuit depth as follows: for a dataset of size n, the algorithm requires $4\sqrt{n} + 2\log_2(n)$ rotations, and three multiplications. The multiplicative depth is $\lceil \log_2(2n_g + 1) \rceil \cdot d_g + \lceil \log_2(2n_f + 1) \rceil \cdot d_f + 1$ for performing comparisons, where n_g, n_f, d_g, and d_f are precision parameters, plus an additional $\log_2(n_s)$ depth for a helper function to rearrange the array according to the ranks, where n_s is the degree of the approximation polynomial. Based on pseudocode in Algorithm 13, it is easy to tell that is complexity is $O(n^2)$. However, the FHE-Effort is more complex: $\mathcal{F}(2\alpha^2 - \alpha + \frac{\alpha^2-\alpha}{2} + \sigma^2 + \frac{3(\sigma^2-\sigma)}{2} + \pi^2)$, which is proved in Proof 4.5.

PROOF 4.5 — Rank Sort FHE-Effort

1. The loops at lines 4, 6, 17, and 19 perform n iterations. Hence, the block from line 7 to 14 is evaluated n^2 times and the block from line 20 to 21 is evaluated n^2 times.
2. $\because$ The `if` condition at line 7 will skip the rest of the loop body whenever $i = j$, which is n iterations. Therefore, the total number of iterations becomes $n^2 - n$.
3. $\therefore$ The comparison at line 9 is evaluated $n^2 - n$ times, which costs $\alpha^2 - \alpha$. Similarly, the comparison at line 11 is evaluated only when $i > g$. Hence, it is evaluated $\sum_{i=0}^{n-1} i = \frac{n^2-n}{2}$ times, which costs $\frac{\alpha^2-\alpha}{2}$.
4. At lines 10 to 14, we either execute 2 addition operations (lines 12) or 1 addition operation (line 14). This can be expressed as executing 3 additions $\frac{n^2-n}{2}$ times, which is equivalent to $\frac{3(\sigma^2-\sigma)}{2}$.

5. The comparison at line 20 is clearly executed n^2, which costs as α^2. Similarly, the line 21 performs n^2 additions and multiplications, which cost as σ^2 and π^2, respectively.
6. By summing up the terms in previous steps, we get $\mathcal{F}\left(2\alpha^2 - \alpha + \frac{\alpha^2-\alpha}{2} + \sigma^2 + \frac{3(\sigma^2-\sigma)}{2} + \pi^2\right)$.

□

4.8 Heap Sort

The plaintext Heap Sort is an efficient sorting algorithm that sorts a dataset by organizing it into a binary heap data structure. The algorithm works by rearranging the elements to satisfy the "min heap property" (or max heap property), which requires every parent node to be less than or equal to its child node. The minimum values always ends up at the root. To get a sorted dataset, the algorithm repeatedly extracts the root (which is always the minimum or maximum value) and reorganizes the heap structure again. The time complexity of the plaintext algorithm is $O(n \log_2(n))$.

Although Heap Sort has not been found in the FHE literature, we include it in our study as a demonstration on how one of the most efficient sorting algorithm can become one of the worst oblivious sorting algorithms. Algorithm 14 lists the pseudocode of the oblivious Heap Sort algorithm. The FHE-Effort is $\mathcal{F}(g \cdot (\alpha+\sigma+\pi+\beta))$, where $g = \left(\lfloor \frac{n}{2} \rfloor \lceil \log_2(n) \rceil + \sum_{i=1}^{n-1} (\lceil \log_2(i) \rceil)\right)$ is the number of iterations, $\alpha = 7n+5$, $\sigma = 2,\ \pi = 1$, and $\beta = 2$ (Proof 4.6).

PROOF 4.6 — Heap Sort FHE-Effort

1. `HeapSort` calls `Heapify` at lines 4 and 7. At line 4, `Heapify` is invoked $\lfloor n/2 \rfloor$ times. At line 7, `Heapify` is invoked n times. Notice that `Heapify` itself performs $\lceil \log_2(size) \rceil$ iterations. Hence, `Heapify` at line 4 performs $\lceil \log_2(n) \rceil$ iterations and at line 7 it performs performs $\sum_{i=1}^{n-1} \lceil \log_2(i) \rceil$ iterations. Therefore, the total number of iterations is $g = \left(\lfloor \frac{n}{2} \rfloor \lceil \log_2(n) \rceil + \sum_{i=1}^{n-1} \lceil \log_2(i) \rceil\right)$.
2. The total number of comparisons made by `Heapify` is $7n + 5$, computed as follows: 3 `BAA` invocations (lines 15–17) and 1 `BAS` invocation (line 25) together require $7n$ comparisons; lines 18 and 19 require 2 comparisons; lines 20–22 account for 2 comparisons (1 comparison operator and 2 selections); and lines 23–24 account for 1 comparison (1 comparison operator and 1 selection).
3. `Heapify` performs 2 addition operations (lines 13 and 14), 1 multiplication (line 13), and 2 bit-wise operations (lines 20 and 23). Therefore, we have $\sigma = 2g$, $\pi = g$, and $\beta = 2g$.
4. By grouping the terms in previous steps, we get the FHE-Effort $\mathcal{F}(g \cdot (\alpha + \sigma + \pi + \beta))$, assuming $\alpha = 7n + 5$, $\sigma = 2,\ \pi = 1$, and $\beta = 2$.

□

4.9 Odd-Even Merge Sort

Odd-Even Merge (OEM) Sort [10], also known as Batcher's odd-even mergesort, is a highly efficient sorting network introduced by Ken Batcher in 1968. Its efficiency stems from idea of instead of comparing items sequentially, it recursively splits the input, sorts each half independently, and uses the "odd-even merge" approach to weave the sorted sub-lists together into a single list. Its time complexity is $O(n(\log_2^2(n)))$. In addition, it is an FHE-friendly algorithm and, as demonstrated later in the experiments, it is the most efficient oblivious sorting algorithm. One limitation of OEM is that it requires the size of input to be powers of 2, i.e., $n = 2^k \mid k \in \mathbb{Z}$.

An oblivious OEM Sort has been introduced in [41] and is implemented in the BGV scheme using HELib library. Unlike other scheme-specific sorting algorithms, OEM Sort is considered a portable algorithm that can implemented in any FHE scheme that support comparisons. Algorithm 15 lists the pseudocode of the

Algorithm 14: Oblivious Heap Sort

```
1  Procedure HeapSort(D):
2      n ← length(D)
3      for i ← ⌊n/2⌋ − 1 to 0 by −1 do
4          Heapify(D, n, i)
5      for i ← n − 1 to 1 by −1 do
6          swap(D_0, D_i)
7          Heapify(D, i, 0)
8  Procedure Heapify(D, size, root):
9      max_depth ← ⌈log_2(size)⌉
10     current_idx ← Enc(root)
11     enc_size ← Enc(size)
12     for step ← 0 to max_depth − 1 do
13         l_idx ← (current_idx × 2) + 1
14         r_idx ← l_idx + 1
15         root_val ← BAA(D, current_idx)
16         l_val ← BAA(D, l_idx)
17         r_val ← BAA(D, r_idx)
18         l_valid ← l_idx < enc_size
19         r_valid ← r_idx < enc_size
20         l_root ← (l_val > root_val) ∧ l_valid
21         max_val ← l_root ? l_val : root_val
22         max_idx ← l_root ? l_idx : current_idx
23         r_max ← (r_val > max_val) ∧ r_valid
24         max_idx ← r_max ? r_idx : max_idx
25         BAS(D, current_idx, max_idx)
26         current_idx ← max_idx
```

Algorithm 15: Oblivious Odd-Even Merge Sort

```
1  Procedure OddEvenMergeSort(D):
2      OddEvenMergeSortRec(D, 0, length(D))
3  Procedure OddEvenMergeSortRec(D, lo, n):
4      if n > 1 then
5          m ← ⌊n/2⌋
6          OddEvenMergeSortRec(D, lo, m)
7          OddEvenMergeSortRec(D, lo + m, m)
8          OddEvenMerge(D, lo, n, 1)
9  Procedure OddEvenMerge(D, lo, n, r):
10     m ← r × 2
11     if m < n then
12         OddEvenMerge(D, lo, n, m)
13         OddEvenMerge(D, lo + r, n, m)
14         i ← lo + r
15         while i + r < lo + n do
16             cond ← D_i > D_{i+r}
17             min ← cond ? D_{i+r} : D_i
18             max ← cond ? D_i : D_{i+r}
19             D_i ← min
20             D_{i+r} ← max
21             i ← i + m
22     else
23         cond ← D_lo > D_{lo+r}
24         min ← cond ? D_{lo+r} : D_lo
25         max ← cond ? D_lo : D_{lo+r}
26         D_lo ← min
27         D_{lo+r} ← max
```

oblivious OEM Sort. Its FHE-Effort is $\mathcal{F}\left(\frac{n}{2}\log_2(\alpha) \times (\log_2(\alpha) - \frac{\alpha}{n}) + 2(\alpha - \frac{\alpha}{n})\right)$, which is derived as in Proof 4.7.

PROOF 4.7 — OEM Sort FHE-Effort

1. `OddEvenMergeSort` recursively splits the input into two halves and invokes the `OddEvenMerge` procedure to sort and merge sub-lists. After the recursive calls, it executes the `while` loop that steps by $m = 2r$. Hence, the number of iterations is $n/2 - 1$.
2. $\therefore$ We can express this recurrence relation as $M(n) = 2M(n/2) + (n/2 - 1)$. When reaching the base case of $M(2) = 1$, the `else` block (lines 23–27) executes once. Therefore, this recurrence resolves to $M(n) = \frac{n}{2}\log_2(n) - \frac{n}{2} + 1$.
3. Since `OddEvenMergeSort` recursively calls itself twice and executes `OddEvenMerge`, then its recurrence relation is given by $S(n) = 2S\left(\frac{n}{2}\right) + M(n)$.
4. By expanding the above relation and substituting the recurrence of $M(n)$, we get the total number of iterations in a closed form as $\frac{n\log_2^2(n) - n\log_2(n)}{4} + n - 1$.
5. $\because$ Each of the `while` and `else` blocks perform 2 comparisons, then the cost of comparisons is 2α.
6. $\therefore$ The FHE-Effort is obtained by substituting 2α in step 5: $\mathcal{F}\left(\frac{\alpha}{2}\log_2(\alpha) \times (\log_2(\alpha) - \frac{\alpha}{n}) + 2(\alpha - \frac{\alpha}{n})\right)$.

□

4.10 Merge Sort

Merge Sort [36] is stable sorting algorithm that relies on the divide-and-conquer approach. It recursively breaks the input list into smaller sub-lists, then sorts and merges them together to output the sorted list. The complexity of Merge Sort is $O(n \log_2(n))$, which is less than the OEM Sort. However, this does not imply that the oblivious Merge Sort is more efficient than OEM Sort. In fact, oblivious Merge Sort, shown in Algorithm 16, costs $\mathcal{F}\left(2\alpha^2 + \alpha \log_2(\alpha) - \frac{2\alpha}{n} + 2\sigma \log_2(\sigma)\right)$, which is significantly more expensive than oblivious OEM Sort. Specifically, due to the heavy reliance on Blind Array Access (BAA) during the merge phase, the oblivious adaptation of Merge Sort degrades to a quadratic complexity, rendering it highly inefficient compared to its plaintext counterpart. This FHE-Effort is obtained as in Proof 4.8.

PROOF 4.8 — Merge Sort FHE-Effort

1. The divide pattern in Merge Sort gives $\log_2(n)$ merge levels. The work at level i is expressed as $2^i \cdot \frac{n}{2^i} = n$.
2. `Merge` executes 2 additions (lines 21 and 22), which is a cost of 2σ. Therefore, at level i, the cost of additions is $2^i \cdot \frac{2\sigma}{2^i} = 2\sigma$. Hence, the total cost of additions in the entire algorithm is $2\sigma \log_2(\sigma)$.
3. To determine the cost of the comparisons, we notice that there is 1 comparison (lines 18 and 19), which is a total cost of $\alpha \log_2(\alpha)$. Lines 16 and 17 are 2 `BAA` invocations on half of the dataset. Since we have $\log_2(n)$ levels and at each level i the dataset is reduced to $n/2^i$, the cost of `BAA`s expressed as $\sum_{i=0}^{\log_2(n)-1} \left(\frac{n^2}{2^i}\right)$. By solving the summation, the total cost of `BAA` functions becomes $2\alpha^2 - 2\alpha$.
4. The total cost of comparisons of the entire algorithm becomes the cost of `BAA`s plus the cost of the single comparison at lines 18 and 19, which is $2\alpha^2 - 2\alpha/n + \alpha \log_2(\alpha)$.
5. The FHE-Effort becomes $\mathcal{F}\left(2\alpha^2 + \alpha \log_2(\alpha) - \frac{2\alpha}{n} + 2\sigma \log_2(\sigma)\right)$.

□

4.11 Count Sort

Count Sort [19], is a non-comparison-based integer sorting algorithm that operates in linear time. As its name suggests, it works by counting the occurrences of each unique element in the dataset and then uses the counts to directly map the elements into their correct positions. Its time complexity is $O(n+r)$, where r is the range of potential values. The authors of [8] introduced an efficient implementation of the oblivious version of Count Sort. Although the algorithm can be ported to other FHE schemes, the original implementation is not portable because its efficiency stems from custom-built `BAA` and addition functions implemented in the TFHE scheme.

In particular, the authors of [8] implemented the algorithm utilizing the `BAA` and `BAAdd` (Blind Array Add) functions. These functions are implemented using LUTs, which exhibit a notably fast performance. A limitation of this work is that it is limited to 8-bit ciphertexts only. Our implementation is shown in Algorithm 17, which reflects the core logic of the algorithm but different from [8]; this to ensure compatibility and portability across various schemes and to support arbitrary bit size. The FHE-Effort of Algorithm 17 is $\mathcal{F}(\alpha(2r - \frac{\alpha}{n}) + \sigma(2r - \frac{\sigma}{n}) + \frac{r\sigma}{n})$, which is obtained as in Proof 4.9.

PROOF 4.9 — Count Sort FHE-Effort

1. The `BAI` function (line 5) costs r comparisons and additions (based on the size of `histogram`). Since it is invoked n times, then it costs $r(\alpha + \sigma)$.
2. The `for` loop (line 8) executes addition (line 10) r times. Therefore, it costs $\frac{r\sigma}{n}$.
3. The `for` loop at line 12 makes n iterations and the `for` loop at line 15 makes $r - 1$ iterations. Hence, the comparison at line 17 is evaluated $n(r-1)$ times, which is a cost of $\alpha(r - \frac{\alpha}{n})$. Similarly, the addition at line 18 is a cost of $\sigma(r - \frac{\sigma}{n})$.

4. By summing up all the previous terms, we get the FHE-Effort$\mathcal{F}(\alpha(2r - \frac{\alpha}{n}) + \sigma(2r - \frac{\sigma}{n}) + \frac{r\sigma}{n})$.

□

Algorithm 16: Oblivious Merge Sort

Procedure `MergeSort(`$\boldsymbol{D}$`)`**:**
- $n \leftarrow$ length(**D**)
- **if** $n \leq 1$ **then**
  - **return**
- $mid \leftarrow \lfloor n/2 \rfloor$
- `MergeSort(`$\boldsymbol{D}_{0...mid-1}$`)`
- `MergeSort(`$\boldsymbol{D}_{mid...n-1}$`)`
- **merged** $\leftarrow$ [Enc(0), ..., Enc(0)] of size n
- `Merge(`$\boldsymbol{D}_{0...mid-1}$, $\boldsymbol{D}_{mid...n-1}$, ***merged***`)`
- **D** $\leftarrow$ **merged**

Procedure `Merge(`***left***, ***right***, ***merged***`)`**:**
- $total_len \leftarrow$ length(**left**) + length(**right**)
- $left_idx \leftarrow$ Enc(0)
- $right_idx \leftarrow$ Enc(0)
- **for** $i \leftarrow 0$ **to** $total_len - 1$ **do**
  - $lval \leftarrow$ BAA(**left**, $left_idx$)
  - $rval \leftarrow$ BAA(**right**, $right_idx$)
  - $cond \leftarrow lval \leq rval$
  - $min \leftarrow cond\ ?\ lval : rval$
  - $\textbf{merged}_i \leftarrow min$
  - $left_idx \leftarrow left_idx + cond$
  - $right_idx \leftarrow right_idx + \neg cond$

Algorithm 17: Oblivious Count Sort

Procedure `CountSort(`$\boldsymbol{D}$, r`)`**:**
- $n \leftarrow$ length(**D**)
- **histogram** $\leftarrow$ [Enc(0), ..., Enc(0)] of size R
- **for** $x \in \boldsymbol{D}$ **do**
  - **histogram** $\leftarrow$ BAI(**histogram**, x)
- **prefix_sums** $\leftarrow$ array of size R
- $running_sum \leftarrow$ Enc(0)
- **for** $count \in$ ***histogram*** **do**
  - append(**prefix_sums**, $running_sum$)
  - $running_sum \leftarrow running_sum + count$
- **sorted_output** $\leftarrow$ array of size n
- **for** $k \leftarrow 0$ **to** $n - 1$ **do**
  - $val_k \leftarrow$ Enc(0)
  - $k_enc \leftarrow$ Enc(k)
  - **for** $v \leftarrow 1$ **to** $r - 1$ **do**
    - $start_v \leftarrow \textbf{prefix_sums}_v$
    - $past_bounds \leftarrow start_v \leq k_enc$
    - $val_k \leftarrow val_k + past_bounds$
  - append(**sorted_output**, val_k)
- **D** $\leftarrow$ **sorted_output**

4.12 Radix Sort

Radix Sort [19] is a non-comparative integer sorting algorithm that works by processing elements digit by digit. It is considered a sorting paradigm instead of being a sorting algorithm in and of itself; at its core it utilizes a stable sorting routine to sort the digits. A typical plaintext Radix Sort implementation extracts base-10 digits, groups them into buckets (0 to 9) and uses Count Sort to sort the data. The time complexity of Radix Sort varies depending on its parameters: given n b-bit numbers and positive integer radix bits $r \leq b$, and a stable sorting algorithm that takes $\Theta(n + k)$ for inputs in range k, Radix Sort can sort the dataset in $\Theta((b/r)(n + 2^r))$ [19].

Our implementation of oblivious Radix Sort is shown in Algorithm 18. Notice that this algorithm does not rely on Count Sort as its core sorting logic, instead we utilize a stable Rank Sort paradigm. This is because oblivious Count Sort introduces a massive overhead, and the stable rank-based sorting paradigm is a more lightweight procedure to enhance the performance of the entire oblivious Radix Sort. For ciphertext elements of bit size b and radix bits r, the FHE-Effort of oblivious Radix Sort is $\mathcal{F}\left(\frac{b}{r}\left(\lambda+\beta+2\alpha^2+\frac{\alpha^2-\alpha}{2}+2\sigma^2+\frac{\sigma^2-\sigma}{2}\right)\right)$, which is derived as in Proof 4.10.

PROOF 4.10 — Radix Sort FHE-Effort

1. The first inner `for` loop (line 6) performs n iterations, and each iteration executes a shift and a bit-wise operation. Hence, the cost of the `for` loop is $\lambda + \beta$.
2. The `for` loop at line 10 makes n iterations and the loop at line 12 makes $\sum_{i=0}^{n-1}(i) = \frac{n^2-n}{2}$ iterations. Each iteration executes a comparison and an addition. Hence, the cost of this loop is $\frac{\alpha^2-\alpha}{2} + \frac{\sigma^2-\sigma}{2}$.
3. The second innermost `for` loop (line 15) makes n iteration, each performs a comparison and an addition. Hence, the cost of this loop is $n \cdot n$ additions and comparisons, or equivalently $\alpha^2 + \sigma^2$.
4. Lines 20 and 21 are nested loops, performing n^2 iterations. Each iteration executes a comparisons (lines 23 and 24) and an addition (line 25). Hence, the cost of this block is $\alpha^2 + \sigma^2$.
5. By summing up the above terms and considering that the outer loop (line 4) performs b/r iterations, we get the FHE-Effort $\mathcal{F}\left(\frac{b}{r}\left(\lambda + \beta + 2\alpha^2 + \frac{\alpha^2-\alpha}{2} + 2\sigma^2 + \frac{\sigma^2-\sigma}{2}\right)\right)$

□

4.13 Greedy Sort

Greedy Sort was proposed in [13, 18] as an oblivious sorting circuit of depth $O(\log(n) + \log(l))$, where l is the bit width of the inputs. The algorithm relies heavily on bit-wise operations. The official implementation is based on a variant of the LTV HE scheme [52]. The pseudocode of Greedy Sort is shown in Algorithm 19. The algorithm works by constructing a comparison array *compars* to compare every element in the dataset with every other element. Then, it utilizes the `GreedyMatch` procedure to evaluate every possible combination of indices for the sorted array.

Despite the optimized circuit depth, the greedy behavior of the algorithm incurs an expensive FHE-Effort of $\mathcal{F}\left(2\alpha^2 - \alpha + \frac{\alpha^2-\alpha}{2} + \sigma^2 + \beta^2 \times 2^{\beta-1} + \beta^2 - \beta\right)$, which is derived as in Proof 4.11.

PROOF 4.11 — Greedy Sort FHE-Effort

1. For the nested loop block at lines 4 to 13, the operations inside it are evaluated n^2 times. Given the condition at line 7 skips n iterations, the precise cost of lines 8 and 13 is $\alpha^2 - \alpha + \beta^2 - \beta$. Similarly, the comparison at line 10 is only evaluated when $i > j$. Hence, its cost is $\sum_{i=0}^{n-1} i = \frac{n^2-n}{2}$.
2. Based on the previous step, the total cost of the nested loop block (lines 4 to 13) is $\alpha^2 - \alpha + \frac{\alpha^2-\alpha}{2} + \beta^2 - \beta$.
3. The `for` loop at line 14 runs n times inside another loop (line 4). Hence, the cost of line 16 is $\alpha^2 + \sigma^2$.
4. The `for` loop in `GreedyMatch` makes $C_k^m = \frac{m!}{k!(m-k)!}$ iterations, where $m = length(\mathbf{B}) = n - 1$. Hence, the `for` loop performs $\frac{(n-1)!}{k!(n-1-k)!}$ iterations.
5. The `for` loop at line 22 runs $n - 1$ times, executing $n - 1$ bit-wise operations. After the loop, it executes 1 bit-wise operation (line 27). Hence, the number of bit-wise operations from line 22 to 27 is n.
6. $\therefore$ The number of operations for a rank k is $n \cdot \frac{(n-1)!}{k!(n-1-k)!}$.
7. $\therefore$ The total number of operations across all possible ranks $k = 0 : n - 1$ is $\sum_{k=0}^{n-1} n \cdot \frac{(n-1)!}{k!(n-1-k)!} = n \times 2^{n-1}$.
8. Given that we have the outer loop runs n iterations, the final cost `GreedyMatch` across the entire algorithm is $n^2 \times 2^{n-1}$, or equivalently $\beta^2 \times 2^{\beta-1}$ (since it performs bit-wise operations only).
9. By summing up the terms from previous steps, we get $\mathcal{F}\left(2\alpha^2 - \alpha + \frac{\alpha^2-\alpha}{2} + \sigma^2 + \beta^2 \times 2^{\beta-1} + \beta^2 - \beta\right)$.

□

Algorithm 18: Oblivious Radix Sort

```
1  Procedure RadixSort(D, r, b):
2      n ← length(D)
3      mask ← Enc(1 ≪ (r − 1))
4      for shift ← 0 to b − 1 step r do
5          digits ← [Enc(0), ..., Enc(0)] of size n
6          for i ← 0 to n − 1 do
7              shifted ← D_i ≫ shift
8              digits_i ← shifted & mask
9          destinations ← array of size n
10         for i ← 0 to n − 1 do
11             d_idx ← Enc(0)
12             for j ← 0 to i − 1 do
13                 is_eq ← digits_j = digits_i
14                 d_idx ← d_idx + is_eq
15             for j ← 0 to n − 1 do
16                 is_lt ← digits_j < digits_i
17                 d_idx ← d_idx + is_lt
18             destinations_i ← d_idx
19         result ← [Enc(0), ..., Enc(0)] of size n
20         for i ← 0 to n − 1 do
21             for dest ← 0 to n − 1 do
22                 c_idx ← Enc(dest)
23                 target ← destinations_i = c_idx
24                 new_val ← target ? D_i : Enc(0)
25                 result_dest ←
                     result_dest + new_val
26         D ← result
```

Algorithm 19: Oblivious Greedy Sort

```
1  Procedure GreedySort(D):
2      n ← length(D)
3      out ← [Enc(0), ..., Enc(0)] of size n
4      for i ← 0 to n − 1 do
5          compars ← array of size n − 1
6          for j ← 0 to n − 1 do
7              if i ≠ j then
8                  is_gt ← D_i > D_j
9                  if i > j then
10                     tie ← D_i = D_j
11                 else
12                     tie ← Enc(false)
13                 append(compars, (is_gt ∨ tie))
14         for k ← 0 to n − 1 do
15             rank ← GreedyMatch(compars, k)
16             out_k ← out_k + (rank ? D_i : zero)
17     D ← out
18 Procedure GreedyMatch(B, k):
19     m ← length(B); k_true ← Enc(false)
20     for t ∈ (m choose k) do
21         match ← Enc(true)
22         for idx ← 0 to m − 1 do
23             if idx ∈ t then
24                 match ← match ∧ B_idx
25             else
26                 match ← match ∧ (¬B_idx)
27         k_true ← k_true ∨ match
28     return k_true
```

4.14 Direct Sort

Direct Sort is an oblivious sorting algorithm proposed in the same article of Greedy Sort [13] and also reproduced by [18, 32, 41]. Like Greedy Sort, Direct Sort features a low depth of $O(\log(n) + \log(l))$, where l is the bit size. However, Direct Sort is significantly more efficient than Greedy Sort because it eliminates the greedy behavior and minimizes the total number of homomorphic operations. The pseudocode of Direct Sort is shown in Algorithm 20. It works by creating an $n \times n$ comparison matrix `cmpr_mat`, where each entry is the evaluation of the $<$ operator between $\mathbf{D}_i$ and $\mathbf{D}_j$. Then, it computes the index vector $\boldsymbol{S}$ using the transposition of `compr_mat`, which indicates the positions of the vector elements in the sorted list. The FHE-Effort of Direct Sort is $\mathcal{F}(\alpha^2 + \frac{\alpha^2-\alpha}{2} + 2\sigma^2)$, obtained as in Proof 4.12.

PROOF 4.12 — Direct Sort FHE-Effort

1. The `for` loop block (lines 4–7) runs for $\frac{n^2-n}{2}$ times. Hence, the cost of comparison at line 5 is $\frac{\alpha^2-\alpha}{2}$.
2. The nested `for` loop block at lines 11 to 15 runs for n^2 times, and in each iteration it executes a comparison and an addition. Hence, the total cost of this block is $\alpha^2 + \sigma^2$.

3. The `HammingWeightRows` procedure executes n `sum` function on every row on the matrix *rows*, and every row has n elements. Hence, the total cost of this procedure is σ^2.
4. By summing up the terms in the previous steps, we get the FHE-Effort $\mathcal{F}(\alpha^2 + \frac{\alpha^2-\alpha}{2} + 2\sigma^2)$.

□

4.15 Bitonic Sort

Bitonic Sort [2, 10] is a sorting network similar to OEM sort and is commonly used in high-performance computing systems. The core logic of the algorithm relies on the bitonic sequence, which is a sequence of numbers that first monotonically increases and then decreases. More specifically, Bitonic Sort creates a bitonic sequence by recursively dividing the input list into two halves, sorting the first half in ascending order and the second half in descending order. Then it merges the halves to form the bitonic sequence. Finally, the elements are merged into a sorted list. Like OEM Sort, Bitonic Sort requires the input size to be powers of 2. Its time complexity is $O(n \log_2^2(n))$. The authors of [46] implemented Bitonic Sort in CKKS to sort encrypted real-numbers. The algorithm is shown in Algorithm 21, which costs $\mathcal{F}\left(\frac{\alpha}{2}\log_2(\alpha) \times (\log_2(\alpha)+1)\right)$. Its FHE-Effort is obtained as in Proof 4.13.

Algorithm 20: Oblivious Direct Sort

```
1  Procedure DirectSort(D):
2      n ← length(D)
3      cmpr_mat ← n × n matrix
4      for i ← 0 to n − 1 do
5          for j ← i + 1 to n − 1 do
6              cmpr_mat_{i,j} ← (D_i < D_j)
7              cmpr_mat_{j,i} ← ¬cmpr_mat_{i,j}
8      cmpr_mat ← Transpose(cmpr_mat)
9      S ← HammingWeightRows(cmpr_mat)
10     out ← [Enc(0), ..., Enc(0)] of size n
11     for i ← 0 to n − 1 do
12         enc_i ← Enc(i)
13         for j ← 0 to n − 1 do
14             val ← (S_j = enc_i) ? D_j : Enc(0)
15             out_i ← out_i + val
16     D ← out
17 Procedure HammingWeightRows(rows):
18     n_rows ← length(rows)
19     weights ← array of size n_rows
20     for i ← 0 to n_rows − 1 do
21         weights_i ← sum(rows_i)
22     return weights
```

Algorithm 21: Oblivious Bitonic Sort

```
1  Procedure BitonicSort(D):
2      n ← length(D)
3      k ← 2
4      while k ≤ n do
5          j ← k ≫ 1
6          while j > 0 do
7              for i ← 0 to n − 1 do
8                  l ← i ⊕ j
9                  if l > i then
10                     cond ← D_l < D_i
11                     min ← cond ? D_l : D_i
12                     max ← cond ? D_i : D_l
13                     if (i & k) = 0 then
14                         a ← i
15                         b ← l
16                     else
17                         a ← l
18                         b ← i
19                     D_a ← min
20                     D_b ← max
21             j ← j ≫ 1
22         k ← k ≪ 1
```

PROOF 4.13 — Bitonic Sort FHE-Effort

1. The outer `while` loop runs for $\log_2(n)$ iterations; its counter $k = 2$ doubles every iteration (line 22).

2. The inner `while` loop (line 6) has its counter j starting at $k/2$ and it halves after each iteration. More specifically, if $k = 2^i$, then $j = 2^{i-1}$. The loop runs until $j = 1$, thus it runs for $i = \log_2(n)$ iterations.
3. We sum the iterations as i goes from 1 to $\log_2(n)$ to get $\sum_{i=1}^{\log_2(n)} i = \frac{\log_2^2(n)+\log_2(n)}{2}$.
4. The `for` loop (line 7) runs exactly n times, but the homomorphic comparisons (lines 10 to 12) are only executed when $l > i$. Therefore, the comparisons are evaluated $n/2$ times.
5. Given the number of iterations of previous steps, the comparisons block runs $\frac{n}{2} \times \frac{\log_2^2(n)+\log_2(n)}{2}$ times.
6. We have 2 comparisons (1 compare operator and 2 selections), then the total cost is $2 \times \frac{n}{2} \times \frac{\log_2^2(n)+\log_2(n)}{2}$, or equivalently $\mathcal{F}\left(\frac{\alpha}{2}\log_2(\alpha) \times (\log_2(\alpha)+1)\right)$.

□

4.16 Matrix-Rank Sort

The authors of [40] introduced a CKKS-specific solution to perform ranking and sorting on encrypted real-numbers. The efficiency of this algorithm stems from leveraging SIMD capabilities of CKKS and the design of the circuit achieves a constant comparison depth. The complexity of the CKKS-based implementation is $O((n/b)^2)$, where $b \in \{128, 256\}$ is the block size when dividing the input into blocks. Like earlier scheme-specific algorithms, this algorithm is not portable and compatible with FHE schemes other than CKKS. Consequently, it loses its efficiency when applied in other schemes. We redefined the algorithm to be scheme-agonistic as shown in Algorithm 22 and 23, showing only the procedures that involve homomorphic operations, while other procedures (e.g, `RepRow`) can be accessed from [40].

The algorithm works by computing the ranking of the input $\mathbf{D}$ using `Rank` procedure, replicating the ranks over the rows, and extracting different statistics for each row. If the input has duplicated elements, they will get the same rank, which consequently causes incorrect output. Therefore, the authors introduced the `TieCorrect` procedure to resolve equal ranks. The algorithm exhibits FHE-Effort of $\mathcal{F}(3\alpha^2+4\sigma^2+2\sigma+\pi^2)$, which is obtained as in Proof 4.14.

PROOF 4.14 — Matrix-Rank Sort FHE-Effort

1. At line 11, there is a subtraction, a comparison, and a multiplication. Since, these are within a $n \times n$ loop, the total cost is $\alpha^2 + \sigma^2 + \pi^2$.
2. Both `SumRow` and `SumCol` procedures exhibit the same performance cost; they perform additions within an $n \times n$ loop. Therefore, each costs σ^2.
3. The `TieCorrect` procedure invokes `SumRow` at line 28. Therefore, its cost is σ^2.
4. The `Rank` procedure has an $n \times n$ loop (lines 19 and 20), in which it performs 2 comparisons. Hence, the cost of this block is $2\alpha^2$. At line 23, there is an invocation of `SumRow` and `TieCorrect`, together they cost $2\sigma^2$. Finally, it performs 2 additions at line 26 within an n-iteration loop, which costs 2σ.
5. `MatrixSort` invokes `SumCol` at line 12, which costs σ^2.
6. By summing up the previous terms, we get the FHE-Effort $\mathcal{F}(3\alpha^2 + 4\sigma^2 + 2\sigma + \pi^2)$.

□

4.17 K-Way Sort

K-Way Sort [50] can be viewed as a generalized sorting network of Merge, OEM, and Bitonic Sort. Basically, it is a sequence merge paradigm that takes k sorted lists and merge them into a single sorted list. The authors of [31] introduced an oblivious K-Way Sort implementation in the CKKS scheme. The implementation leverages

Algorithm 22: Matrix-Rank Sort (Part 1)

Procedure `MatrixSort(`D`)`:
- $n \leftarrow \text{length}(\mathbf{D})$
- $\mathbf{v} \leftarrow Vec2Sq(\mathbf{D})$; $\mathbf{r} \leftarrow Rank(\mathbf{D})$
- $\mathbf{rr} \leftarrow RepRow(r)$; $\mathbf{vr} \leftarrow RepRow(v)$
- $\mathbf{consts}, \mathbf{tmp} \leftarrow n \times n$ matrix
- **for** $i \leftarrow 0$ **to** $n-1$ **do**
  - **for** $j \leftarrow 0$ **to** $n-1$ **do**
    - $\mathbf{consts}_{i,j} \leftarrow i+1$
- **for** $i \leftarrow 0$ **to** $n-1$ **do**
  - **for** $j \leftarrow 0$ **to** $n-1$ **do**
    - $\mathbf{tmp}_{i,j} \leftarrow ((\mathbf{rr}_{i,j} - \mathbf{consts}_{i,j}) = Enc(0)) \times \mathbf{vr}_{i,j}$
- $\mathbf{res} \leftarrow TransposeCol(SumCol(\mathbf{tmp}))$
- $\mathbf{D}_i \leftarrow \mathbf{res}_{0,i}$

Procedure `Rank(`D`)`:
- $n \leftarrow \text{length}(\mathbf{D})$; $\mathbf{vec2d} \leftarrow Vec2Sq(\mathbf{D})$
- $\mathbf{vr} \leftarrow RepRow(\mathbf{vec2d})$
- $\mathbf{vc} \leftarrow RepCol(TransposeRow(\mathbf{vec2d})$
- $\mathbf{g}, \mathbf{e} \leftarrow n \times n$ matrix
- **for** $i \leftarrow 0$ **to** $n-1$ **do**
  - **for** $j \leftarrow 0$ **to** $n-1$ **do**
    - $\mathbf{g}_{i,j} \leftarrow \mathbf{vr}_{i,j} > \mathbf{vc}_{i,j}$
    - $\mathbf{e}_{i,j} \leftarrow \mathbf{vr}_{i,j} = \mathbf{vc}_{i,j}$
- $\mathbf{r_initial} \leftarrow SumRow(\mathbf{g})$; $\mathbf{t} \leftarrow TieCorrect(\mathbf{e})$
- $\mathbf{ranks} \leftarrow n \times n$ matrix
- **for** $j \leftarrow 0$ **to** $n-1$ **do**
  - $\mathbf{ranks}_{0,j} \leftarrow \mathbf{r_initial}_{0,j} + \mathbf{t}_{0,j} + 1$
- **return ranks**

Algorithm 23: Matrix-Rank Sort (Part 2)

Procedure `SumRow(`$data$`)`:
- $n \leftarrow \text{length}(\mathbf{data})$
- $\mathbf{res} \leftarrow n \times n$ matrix
- **for** $col \leftarrow 0$ **to** $n-1$ **do**
  - $sum \leftarrow \text{Enc}(0)$
  - **for** $row \leftarrow 0$ **to** $n-1$ **do**
    - $sum \leftarrow sum + \mathbf{data}_{row,col}$
  - $\mathbf{res}_{0,col} \leftarrow sum$
- **return res**

Procedure `SumCol(`$data$`)`:
- $n \leftarrow \text{length}(\mathbf{data})$
- $\mathbf{res} \leftarrow n \times n$ matrix
- **for** $row \leftarrow 0$ **to** $n-1$ **do**
  - $sum \leftarrow \text{Enc}(0)$
  - **for** $col \leftarrow 0$ **to** $n-1$ **do**
    - $sum \leftarrow sum + \mathbf{data}_{row,col}$
  - $\mathbf{res}_{row,0} \leftarrow sum$
- **return res**

Procedure `TieCorrect(`e`)`:
- $n \leftarrow \text{length}(\mathbf{e})$
- $\mathbf{masked} \leftarrow n \times n$ matrix
- **for** $j \leftarrow 0$ **to** $n-1$ **do**
  - **for** $i \leftarrow 0$ **to** $n-1$ **do**
    - **if** $i < j$ **then**
      - $\mathbf{masked}_{i,j} \leftarrow \mathbf{e}_{i,j}$
    - **else**
      - $\mathbf{masked}_{i,j} \leftarrow \text{Enc}(0)$
- **return** $SumRow(\mathbf{masked})$

the SIMD capabilities of CKKS to speed up the algorithm. The authors reported that the algorithm's complexity is $O(k \log_k^2 n)$, where k is the number of sub-lists split from the input. We outline this algorithm in Algorithm 24 and 25, utilizing $k = \lfloor n/2 \rfloor$. The FHE-Effort is $\mathcal{F}((\alpha^2 - \alpha \log_2(\alpha) - \alpha/n) + 2\sigma \log_2(\sigma))$, which is obtained as shown in Proof 4.15.

The previous analysis of the FHE-Effort assumed that $k = \lfloor n/2 \rfloor$. The choice of k affects the cost of comparisons only, while additions remain the same. We can generalize it to an arbitrary $k = qn$ for $0 < q \leq 1$. In this case the cost of `MthMax` and `MthMin` in step 3 can be written as $C_{merge} = (qn)^2 + (n - qn - q)^2 = q^2n^2 + ((1-q)(n-1))^2$. Injecting C_{merge} into the recurrence relation yields:

$$T(n) = 2\left(\frac{n}{2}\right) + C_{merge} \tag{2}$$

$$= 2(q^2 + (1-q)^2)n^2 - 2(1-q)n\log_2(n) - (2q^2 - 2q + 1)2n + n - 1 \tag{3}$$

$$\equiv 2(q^2 + (1-q)^2)\alpha^2 - 2(1-q)\alpha\log_2(\alpha) - (2q^2 - 2q + 1)2\alpha + \alpha - \alpha/n \tag{4}$$

PROOF 4.15 — K-Way Sort FHE-Effort

1. Starting with the `MthMax` procedure, it performs 1 comparison (line 5) when reaching the base case and another comparisons (line 13 with 16) when invoking a recursive step. Since, the function divides $k = m+1$ into left and right path, the recursive call invokes `MthMax(j-1)` and `MthMax(i-1)`. This gives the recurrence relation $C(m) = C(i-1) + C(j+1) + 1 \equiv 2m - 1$. The same applies to `MthMin`.
2. `MthMax` is invoked within a loop at line 18 in `KSorterMerge`, where its depth depends on the loop counter. The total cost becomes $\sum_{i=0}^{\frac{n}{2}-1}(2i+1) = \left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$. The same applies to `MthMin`; its total cost with the loop at line 20 is $\sum_{m=0}^{\frac{n}{2}-2}(2m+1) = \left(\frac{n}{2}-1\right)^2$. Notice that for the loop at line 20 in `KSorterMerge`, although the loop counter j is incrementing, the m parameter of `MthMin` is actually decreasing.
3. From the previous step, the cost of `MthMax` and `MthMin` in `KSorterMerge` is $\frac{n^2}{4} + \left(\frac{n}{2}-1\right)^2 = \frac{n^2}{2} - n + 1$.
4. The top-level `KSorter` splits the input into two halves and recursively sorts and merges them. This yields the recurrence relation $C(n) = 2\left(\frac{n}{2}\right) + \frac{n^2}{2} - n + 1$. By solving the recurrence, we get $n^2 - n\log_2(n) - 1$.
5. The previous step estimated the total cost for the `MthMax` and `MthMin` procedures, which both perform comparisons. Hence, the the total cost of comparisons is $\alpha^2 - \alpha\log_2(\alpha) - \alpha/n$. Notice the -1 is expressed as α/n, since we are assuming α means n comparisons.
6. `KSorterMerge` performs a compound addition at line 22, which is equivalent to performing $length(B) + length(C) = (s-1) + (t-1) + 1 = s + t - 1 = n - 1$. Line 30 is another addition in an n iteration loop, which is a total of n addition. Finally, line 31 is a single addition. Therefore, the total number of additions in a single `KSorterMerge` invocation is $2n$, or equivalently, 2σ.
7. By incorporation the recurrence relation introduced by `KSorter`, we get the cost of additions as $C(n) = 2C\left(\frac{n}{2}\right) + 2n = 2n\log_2(n) \equiv 2\alpha\log_2(\alpha)$.
8. From steps 5 and 7, the FHE-Effort becomes $\mathcal{F}((\alpha^2 - \alpha\log_2(\alpha) - \alpha/n) + 2\sigma\log_2(\sigma))$.

□

4.18 Discussion on FHE-Effort

Our novel FHE-Effort analysis serves as a general-purpose holistic complexity analysis model to estimate the efficiency of an FHE algorithm independently from a particular scheme. Similar to the order of growth notation, FHE-Effort has the advantage of estimating how efficient an algorithm is compared to another without the need to implement and run the code. This is a particularly significant advantage because, unlike plaintext algorithms, FHE algorithms are quite hard to implement and running them can take hours to judge their performance. In this case, the complexity analysis notations (e.g., big-O) is not sufficient to give a concrete performance view. Figure 1 summarizes the FHE-Effort of the previous sorting algorithms and visualizes the order of growth of each FHE operation according to their FHE-Effort.

There are two ways to interpret FHE-Effort. The first way is to determine how many times a particular operation $\rho \in \{\alpha, \beta, \lambda, \sigma, \pi\}$ is executed. By substituting n for a target ρ in the FHE-Effort formula and zeroing out other terms, this yields the exact number of times the operation is executed. For example, to determine how many comparisons Greedy Sort executes on a dataset of 8 elements, we substitute 8 for every $\rho = \alpha$ and eliminate other terms. Hence, Greedy Sort executes $2 \cdot 8^2 - 8 + \frac{8^2-8}{2} = 148$ comparisons, which can be verified by a manual tracing of the algorithm.

The second way to interpret FHE-Effort is by estimating the running time of the actual circuit. This can be obtained by first calculating the exact count of each operation for a given input size n, and then multiplying that count by the operation's average running time. More formally, let C_ρ denote the total count

Algorithm 24: Oblivious K-Way Sort (Part 1)

Procedure `KWaySort(`D`)`:
  $\mathbf{D} \leftarrow reverse(KSorter(\mathbf{D}))$
Procedure `KSorter(`D`)`:
  $n \leftarrow \text{length}(\mathbf{D})$
  if $n \leq 1$ **then return D**
  $s \leftarrow \lfloor n/2 \rfloor$
  $\mathbf{B} \leftarrow KSorter(\mathbf{D}_{0\ldots s-1})$
  $\mathbf{C} \leftarrow KSorter(\mathbf{D}_{s\ldots n-1}))$
  return $KSorterMerge(\mathbf{B}, \mathbf{C})$
Procedure `KSorterMerge(`B, C`)`:
  $s \leftarrow \text{length}(\mathbf{B})$; $t \leftarrow \text{length}(\mathbf{C})$
  $n \leftarrow s + t$
  if $n = 0$ **then return** []
  if $s = 0$ **then return C**
  if $t = 0$ **then return B**
  $\mathbf{Z} \leftarrow$ array of size n
  $k \leftarrow \lfloor n/2 \rfloor$
  for $i \leftarrow 0$ **to** $k - 1$ **do**
    $\mathbf{Z}_i \leftarrow MthMax(i, \mathbf{B}, \mathbf{C})$
  for $j \leftarrow k + 1$ **to** $n - 1$ **do**
    $\mathbf{Z}_j \leftarrow MthMin((n - 1 - j), \mathbf{B}, \mathbf{C})$
  $total_sum \leftarrow sum(\mathbf{B}) + sum(\mathbf{C})$
  if $k = 0$ **then**
    $Z_sum \leftarrow \mathbf{Z}_1$
  else
    $Z_sum \leftarrow \mathbf{Z}_0$
  for $m \leftarrow 0$ **to** $n - 1$ **do**
    $x \leftarrow k \vee (k = 0 \wedge m = 1) \vee (m = 0 \wedge k \neq 0)$
    if $m = x$ **then continue**
    $Z_sum \leftarrow Z_sum + \mathbf{Z}_m$
  $\mathbf{Z}_k \leftarrow total_sum - Z_sum$
  return Z

Algorithm 25: Oblivious K-Way Sort (Part 2)

Procedure `MthMax(`m, B, C`)`:
  if B *is empty* **then return** $\mathbf{C}_m$
  if C *is empty* **then return** $\mathbf{B}_m$
  if $m = 0$ **then**
    return $\mathbf{B}_0 > \mathbf{C}_0$? $\mathbf{B}_0 : \mathbf{C}_0$
  $k \leftarrow m + 1$
  $i \leftarrow \min(\lfloor k/2 \rfloor, \text{len}(\mathbf{B}))$
  $j \leftarrow k - i$
  if $j > len(C)$ **then**
    $j \leftarrow \text{len}(\mathbf{C})$
    $i \leftarrow k - j$
  $cmp_i \leftarrow i - 1$; $cmp_j \leftarrow j - 1$
  $cond \leftarrow \mathbf{B}_{cmp_i} > \mathbf{C}_{cmp_j}$
  $l \leftarrow MthMax(j - 1, \mathbf{B}_{i\ldots\text{end}}, \mathbf{C}_{0\ldots j-1})$
  $r \leftarrow MthMax(i - 1, \mathbf{B}_{0\ldots i-1}, \mathbf{C}_{j\ldots\text{end}})$
  return $cond$? $l : r$
Procedure `MthMin(`m, B, C`)`:
  if B *is empty* **then return** $\mathbf{C}_{\text{len}(\mathbf{C})-1-m}$
  if C *is empty* **then return** $\mathbf{B}_{\text{len}(\mathbf{B})-1-m}$
  if $m = 0$ **then**
    $cond \leftarrow \mathbf{B}_{\text{len}(\mathbf{B})-1} < \mathbf{C}_{\text{len}(\mathbf{C})-1}$
    return $cond$? $\mathbf{B}_{\text{len}(\mathbf{B})-1} : \mathbf{C}_{\text{len}(\mathbf{C})-1}$
  $k \leftarrow m + 1$; $j \leftarrow k - i$
  $i \leftarrow \min(\lfloor k/2 \rfloor, \text{len}(\mathbf{B}))$
  if $j > len(C)$ **then**
    $j \leftarrow \text{len}(\mathbf{C})$
    $i \leftarrow k - j$
  $cmp_i \leftarrow \text{len}(\mathbf{B}) - i$; $cmp_j \leftarrow \text{len}(\mathbf{C}) - j$
  $cond \leftarrow \mathbf{B}_{cmp_i} < \mathbf{C}_{cmp_j}$
  $l \leftarrow MthMin(j-1, \mathbf{B}_{0\ldots cmp_i-1}, \mathbf{C}_{cmp_j\ldots\text{end}})$
  $r \leftarrow$
    $MthMin(i - 1, \mathbf{B}_{0\ldots cmp_i-1}, \mathbf{C}_{0\ldots cmp_j-1})$
  return $cond$? $l : r$

of a specific operation ρ derived from the FHE-Effort formula. Given the latency of a particular operation T_ρ, the estimated running time T_{est} is computed as follows:

$$T_{est} = \sum_{\rho \in \{\alpha, \beta, \lambda, \sigma, \pi\}} C_\rho \cdot T_\rho \tag{5}$$

For example, to estimate how long Matrix-Rank Sort will take on a dataset of 8 elements in TFHE-rs, we first assume the cost of each operation ρ is based on Table 2 as follows (we select the larger latency value in each category): $T_\alpha = 55$, $T_\sigma = 58$, and $T_\pi = 122$ milliseconds. Using the FHE-Effort formula $\mathcal{F}(3\alpha^2 + 4\sigma^2 + 2\sigma + \pi^2)$, we compute the operation counts for $n = 8$ and multiply by their respective latencies:

$$T_{est} = (3 \cdot 8^2 \cdot 55) + ((4 \cdot 8^2 + 2 \cdot 8) \cdot 58) + (8^2 \cdot 122) = 34,144 \text{ ms}. \tag{6}$$

This calculation estimates the running time of Matrix-Rank Sort on 8 elements in TFHE-rs as 34.1 seconds. Rather than a strict upper bound, FHE-Effort provides a deterministic baseline estimation of the

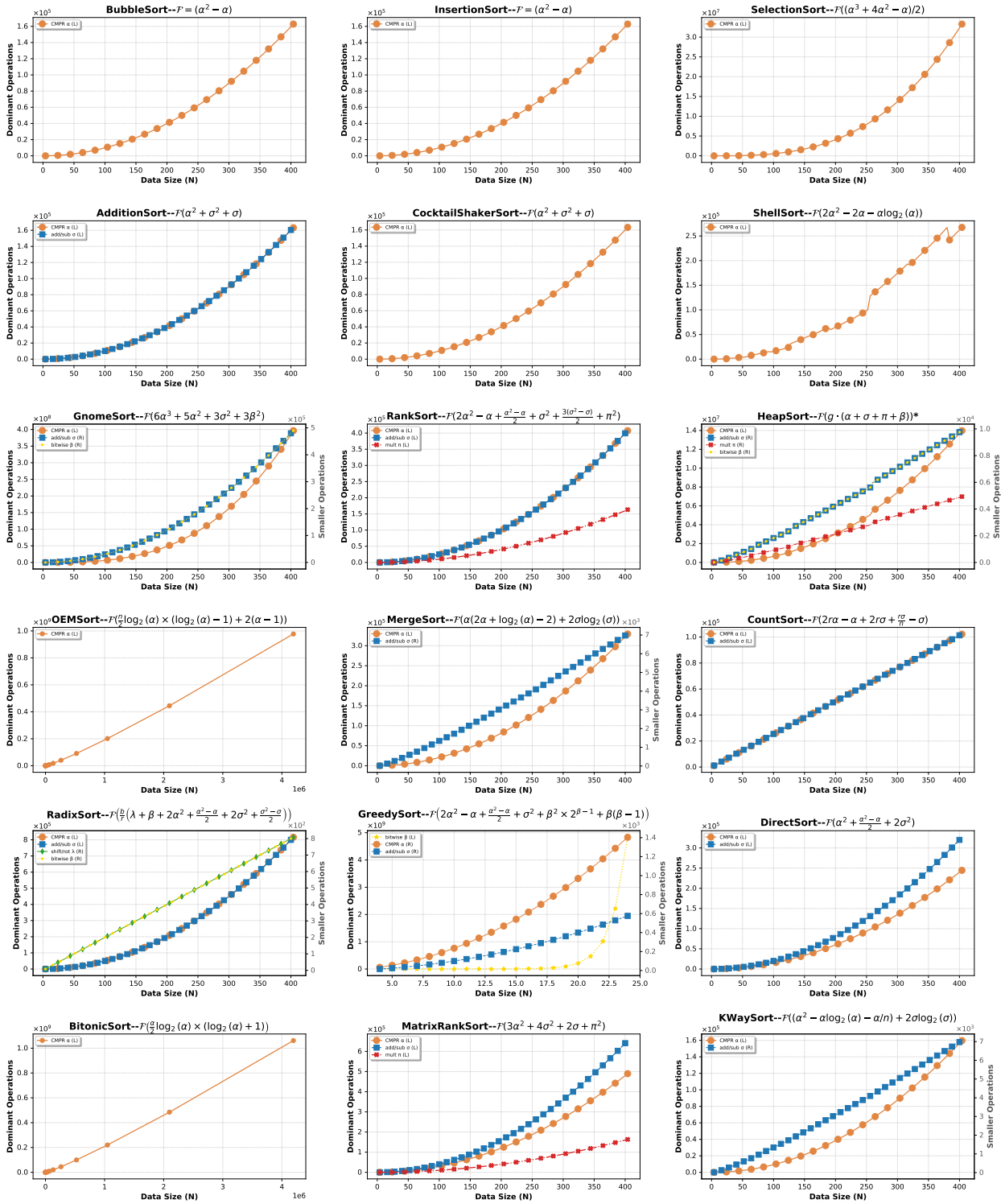


Fig. 1: The order of growth of oblivious sorting algorithms as described by their FHE-Effort analysis. The major operations are plotted on the left side of the plot, while minor operations are plotted on the right side. For Heap Sort, $g = (\lfloor \frac{n}{2} \rfloor \lceil \log_2(n) \rceil + \sum_{i=1}^{n-1} (\lceil \log_2(i) \rceil)$, $\alpha = 7n + 5$, $\sigma = 2$, $\pi = 1$, and $\beta = 2$.

core cryptographic workload. The actual empirical runtime may be lower because underlying FHE libraries often optimize independent homomorphic evaluations, whereas FHE-Effort aggregates maximum latencies sequentially. Conversely, empirical run times can exceed this baseline if the execution incurs heavy unmodeled costs, such as scheme-specific bootstrapping, complex plaintext operations, or frequent disk I/O.

5 Experimental Evaluation

This section presents the execution times of the earlier oblivious sorting algorithms. All experiments were conducted on a Linux machine equipped with an Intel i9-12900 CPU (24 threads) and 100 GB of RAM. Although our implementations were single-threaded, multi-threading was enabled for the underlying FHE library to reflect the realistic performance expected from a default library installation. To balance the evaluation scope with practical deployment considerations, we enforced a strict timeout threshold of 2 hours per benchmark run. In the context of secure data processing, computations exceeding this duration transition from being merely latent to practically unfeasible for real-world deployments. Thus, a 2-hour limit serves as a realistic ceiling for evaluating the viability of these FHE sorting algorithms.

We chose not to parallelize the sorting circuits themselves for one key reason: Sorting circuits exhibit fundamentally different parallelization potentials. While algorithms like OEM and Bitonic Sort are naturally embarrassingly parallel, others would require radical algorithmic changes to support parallel execution. Consequently, evaluating parallelized circuits would introduce confounding variables—such as disparate thread-scheduling overheads—preventing a fair, apples-to-apples comparison of the core sorting algorithms' intrinsic efficiency.

Table 4 shows the running time in seconds (rounded to the nearest integer) for each algorithm evaluated on CKKS, TFHE, and FHEW on 8-bit ciphertext at a uniform 128-bit security level. As we can see in Table 4 OEM and Bitonic Sort are the most efficient sorting algorithms due to their low circuit complexity. Meanwhile, Gnome, Count, and Greedy Sort are the most inefficient oblivious sorting algorithms due to their circuit complexity as demonstrated by their FHE-Effort.

To present a thorough analysis of the oblivious sorting literature, we compare the performance of original source code of the works that provide their open source artifacts. Table 5 shows the execution time of the open-source codebases (rounded to the nearest integer). The Count Sort circuit [8] is limited to only 8-bit data. The Direct Sort of [32] does not specify a particular bit size, but BGV defines its limits using a plaintext modulus. We have tested Direct Sort on $13, 29, 37, 67,$ and 131 plaintext moduli for data sizes of $8, 16, 32, 64,$ and 128 elements, respectively. For the CKKS artifacts (i.e., Matrix-Rank Sort [40], Rank Sort [35], and both the Bitonic and Matrix-Rank sorts of [46]) we ran each on 8-bit real inputs at the 128-bit security level, scaling the ring dimension with the data size to preserve security (from 2^{16} up to 2^{18}). Rank Sort and Rovida's Matrix-Rank Sort keep a fixed ring dimension across all sizes, whereas Mazzone's Matrix-Rank Sort and the Bitonic Sort raise theirs for larger inputs. Additionally, all open-source baselines leverage Functional Bootstrapping (FBT). To ensure a fair and stable comparison, the CKKS scaling factor was maintained at 50 bits (2^{50}), and the initial multiplicative depth was tailored to accommodate the specific circuit requirements of each baseline without triggering premature precision loss.

6 Discussion

Which FHE Scheme Should I Choose?. Based on the results in Table 4, it is clear that TFHE is the most efficient FHE scheme for oblivious sorting routines. However, TFHE has two limitations: first, it does not natively support real-number arithmetic, and second, it does not support batching. While a recent work [11] realized an extended version of TFHE to support real-number arithmetic, this has not been officially integrated into the main TFHE-rs library. Real-number arithmetic in TFHE is expected to be faster than CKKS due to the native capability of TFHE to perform comparisons, though it will inherently be slower than the integer arithmetic of vanilla TFHE. We believe that if TFHE inherently supported batching like CKKS, it would offer significant performance benefits to the sorting circuit.

Table 4: The running time (in seconds) for all sorting algorithms on sets of 8, 16, and 32 elements. The bit size of each element is fixed to 8 bits. A dash (—) means the corresponding experiment did not finish within 2 hours.

Algorithm	**CKKS**			**TFHE**			**FHEW**		
Data Size	**8**	**16**	**32**	**8**	**16**	**32**	**8**	**16**	**32**
Bubble Sort	630	2825	—	6	25	125	240	1032	4270
Insertion Sort	633	2825	—	4	23	91	153	656	2711
Selection Sort	1120	4816	—	30	255	1887	1630	—	—
Addition Sort	—	—	—	7	36	148	281	1152	4648
Cocktail-Shaker Sort	640	2789	—	5	24	96	240	1026	4265
Shell Sort	1007	4784	—	7	39	173	241	1132	4989
Gnome Sort	—	—	—	277	2586	—	—	—	—
Rank Sort	11	13	20	16	81	333	1755	7168	—
Heap Sort	2752	—	—	114	871	4316	7163	—	—
Odd-Even Merge Sort	433	1520	4875	3	12	34	104	345	1045
Merge Sort	1953	5928	—	13	63	241	670	2567	—
Count Sort	474	870	1657	287	522	1100	—	—	—
Radix Sort	471	556	830	35	182	726	1646	6599	—
Greedy Sort	45	79	221	151	—	—	7117	—	—
Direct Sort	69	123	319	13	54	217	633	2511	—
Bitonic Sort	58	104	157	3	15	44	131	439	1304
Matrix-Rank Sort	26	28	72	24	119	483	2180	—	—
K-Way Sort	151	288	548	6	25	106	217	991	4175

Although our empirical benchmarks focus on 8-bit integers to establish a baseline, scaling to larger bit-widths introduces inherent architectural trade-offs. In leveled schemes like CKKS, extending deep, non-linear sorting circuits to higher precisions rapidly exacerbates noise accumulation, inherently demanding frequent and highly expensive bootstrapping operations to prevent precision loss. In contrast, TFHE's native, continuous noise management via Boolean gates bypasses this specific depth penalty, granting it a distinct structural advantage for deep sorting routines.

Which Oblivious Sorting Algorithm should I Choose?. Based on the FHE-Effort analysis presented in Figure 1, it is obvious that OEM and Bitonic algorithms are the most efficient oblivious sorting routines. This is due to their relatively shallow circuit depth that stems from how they split and merge the data. In addition, OEM and Bitonic involve comparisons only; there are no expensive homomorphic operations (e.g., multiplication) that may cause the noise in a ciphertext to explode, especially for leveled schemes such as CKKS.

Furthermore, while substituting brute-force Blind Array Access (BAA) with scheme-specific Look-Up Tables (such as TFHE's Programmable Bootstrapping) provides constant-factor optimizations for indexing-dependent algorithms like Heap Sort, Selection Sort, or Gnome Sort, these algorithms inherently possess a heavier computational footprint. As the FHE-Effort analysis demonstrates, the strict requirement for mixed mathematical operations (additions and multiplications, or an excessive volume of brute-force comparisons) to obliviously manage array indices guarantees an algorithmic bottleneck compared to the comparison-only elegance of sorting networks.

Parallelization Opportunities and Resource Scheduling. Modern FHE libraries already heavily parallelize underlying computations at the ciphertext level. However, during the running of a sorting circuit, we noticed that the FHE libraries do not utilize all the available system cores to accelerate the computations. In particular, we analyzed the usage of system resources of FHEW in OpenFHE (v1.4.0) and TFHE in TFHE-rs (v1.5.2) and noticed that only a small number of cores were fully utilized, while the remaining system cores were idle or under-utilized. Also, we noted that the utilization of the system cores increases with the

Table 5: The run time of the open-source sorting algorithms based on the original implementation as appeared in the corresponding work, evaluated across varying data sizes (N=8 to 128). All the times are reported in seconds, unless stated otherwise.

Algorithm	**Target Scheme**	**8**	**16**	**32**	**64**	**128**
Count Sort [8]	TFHE	588 (ms)	1	7	40	353
Direct Sort [32]	BGV/BFV	188 (ms)	596 (ms)	4	26	124
Matrix-Rank Sort [40]	CKKS	26	28	72	77	95
Bitonic Sort [46]	CKKS	58	104	157	504	700
Rank Sort [35]	CKKS	11	13	20	28	47
Rovida Matrix-Rank Sort [46]	CKKS	10	11	14	46	53

bit size of the data. CKKS, by contrast, already saturates the available cores, since OpenFHE parallelizes RNS CKKS across its limbs, so core utilization scales with the number of limbs (moduli) in the ciphertext.

This under-utilization presents a structural opportunity: the high-level FHE sorting circuits can be parallelized without risking severe CPU oversubscription. In fact, in our experiments we ran 2-3 sorting processes concurrently to efficiently utilize the system resources without causing any notable performance degradation. However, this advantage may not be accessible in CKKS, as it already saturates the CPU. Therefore, attempting to parallelize the high-level sorting logic on top of CKKS may trigger severe thread contention, ultimately degrading performance rather than improving it.

Open Problems and Future Research Directions. Oblivious sorting techniques are still far from being practical and suitable for real-world deployments. Thereupon, there are several open problems that can drive future research direction in this field. First, various oblivious sorting circuits exhibit numerous scheduling opportunities that can be exploited to leverage accelerated performance and optimal utilization of system resources. The most promising approach is by leveraging the divide-and-conquer paradigm to divide the problem and solve smaller problems independently and then merging them. This approach is evident in algorithms such as OEM and Bitonic Sort. This challenge is not trivial because the parallelization of the top-level circuit has to be meticulously coordinated with the underlying FHE libraries to avoid thread contention problems.

Another promising direction involves focusing on having more ciphertext-plaintext (CP) operations rather than ciphertext-ciphertext (CC) operations. Although our FHE-Effort notation treats them equally, a sorting routine that relies on CP operations is guaranteed to be faster than an algorithm that relies on CC operations if they exhibit the same FHE-Effort. Furthermore, we recommend that future research should focus on minimizing the multiplicative depth and the cost of bootstrapping operations, especially in leveled schemes such as CKKS. For very large-scale sorting tasks, such as sorting banking data with trillions of entries, an oblivious sorting routine may never succeed in finishing its task due to the massive noise accumulation in ciphertexts. Even when utilizing large parameters to initialize the CKKS scheme, the final output values will lose their precision. This is definitely intolerable in sensitive domains, such as banking or health domains.

Finally, we emphasize the need for a *portable* sorting routine that is not tightly coupled to a specific FHE implementation. In particular, we argue that a sorting algorithm that gets its efficiency from a custom FHE implementation or relies on bespoke constructs becomes less usable. This is because these bespoke designs limit the algorithm's interoperability and render it as a specific-purpose rather than a general-purpose algorithm.

7 Conclusion

Sorting is a critically important practice in our digital landscape. Its applications range from database management and AI applications to the banking and health domains. With the increasing needs to securely outsource computations on (potentially curious) cloud providers, it becomes urgent that these systems rely on

Privacy-preserving computation methods to perform data processing securely. Therefore, to the best of our knowledge, this work is the first systematic review of the sorting routines applied in the Fully Homomorphic Encryption (FHE) domain. On top of surveying the current literature, we analyzed other fundamental sorting algorithms that have not been found in the FHE literature.

Furthermore, we introduced a novel complexity analysis notation, FHE-Effort, that is used to expound on the complexity details of any FHE circuit, providing more details that a classic Big-O notation cannot provide. We thoroughly analyzed 18 sorting algorithms and implemented them in three FHE schemes, CKKS, TFHE, and FHEW. Our results demonstrated that TFHE is the most efficient FHE scheme for sorting applications, and that Odd-Even Merge Sort (OEM) and Bitonic Sort are the most optimal sorting algorithms.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Omar Ahmed and Nektarios Georgios Tsoutsos. Pulpfhe: Complex instruction set extensions for fhe processors. *Cryptology ePrint Archive*, 2024.
2. Selim G Akl. *Parallel sorting algorithms*, volume 12. Academic press, 2014.
3. Mohammed Alaa Ala'Anzy, Zhanar Mazhit, Alaa Fadhil Ala'Anzy, Abdulmohsen Algarni, Ramis Akhmedov, and Abdiyev Bauyrzhan. Comparative analysis of sorting algorithms: A review. In *2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, pages 88–100. IEEE, 2024.
4. Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using ckks. In *Annual International Cryptology Conference*, pages 304–337. Springer, 2025.
5. Bader Alouffi, Muhammad Hasnain, Abdullah Alharbi, Wael Alosaimi, Hashem Alyami, and Muhammad Ayaz. A systematic literature review on cloud computing security: threats and mitigation strategies. *Ieee Access*, 9:57792–57807, 2021.
6. Abdulmohsen Alsaui, Ibrahim Al-Nahhal, and Octavia A Dobre. Quantum partial sorting for efficient signal decoding in spatially modulated wireless systems. In *2025 IEEE 26th International Workshop on Signal Processing and Artificial Intelligence for Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2025.
7. Sahin Emrah Amrahov, Yilmaz Ar, Bulent Tugrul, Bekir Emirhan Akay, and Nermin Kartli. A new approach to mergesort algorithm: Divide smart and conquer. *Future Generation Computer Systems*, 157:330–343, 2024.
8. Sofiane Azogagh, Marc-Olivier Killijian, and Félix Larose-Gervais. A non-comparison oblivious sort and its application to private k-nn. *Cryptology ePrint Archive*, 2024.
9. Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022.
10. Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.
11. Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Tfhe gets real: an efficient and flexible homomorphic floating-point arithmetic. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):126–162, 2025.
12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
13. Gizem S Çetin, Yarkın Doröz, Berk Sunar, and Erkay Savaş. Depth optimized efficient homomorphic sorting. In *International conference on cryptology and information security in latin america*, pages 61–80. Springer, 2015.
14. Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. Accelerating sorting of fully homomorphic encrypted data. In *International Conference on Cryptology in India*, pages 262–273. Springer, 2013.
15. Ayantika Chatterjee and Indranil Sengupta. Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective? *IEEE Transactions on Services Computing*, 13(3):545–558, 2017.
16. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

17. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
18. Heewon Chung, Myungsun Kim, Ahmad Al Badawi, Khin Mi Mi Aung, and Bharadwaj Veeravalli. Homomorphic comparison for point numbers with user-controllable precision and its applications. *Symmetry*, 12(5):788, 2020.
19. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
20. Muhammad Dawood, Shanshan Tu, Chuangbai Xiao, Hisham Alasmary, Muhammad Waqas, and Sadaqat Ur Rehman. Cyberattacks and security of cloud computing: a complete guideline. *Symmetry*, 15(11):1981, 2023.
21. Thanh Do, Goetz Graefe, and Jeffrey Naughton. Efficient sorting, duplicate removal, grouping, and aggregation. *ACM Transactions on Database Systems*, 47(4):1–35, 2023.
22. Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. Parallel integer sort: Theory and practice. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 301–315, 2024.
23. Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. High-performance and flexible parallel algorithms for semisort and related problems. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 341–353, 2023.
24. Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.
25. Nitesh Emmadi, Praveen Gauravaram, Harika Narumanchi, and Habeeb Syed. Updates on sorting of fully homomorphic encrypted data. In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 19–24. IEEE, 2015.
26. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
27. Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: A universal graph data structure. *ACM SIGMOD Record*, 52(1):17–25, 2023.
28. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.
29. Shai Halevi and Victor Shoup. Helib design principles. *Tech. Rep.*, 2020.
30. Gabriel Hévr, Ivan Šimeček, and Ondřej Voroneckỳ. Ppqsort: Pattern-defeating parallel quicksort. *Concurrency and Computation: Practice and Experience*, 38(7):e70673, 2026.
31. 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.
32. Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for bgv and bfv. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.
33. Hriday Jain, Ketan Sabale, Aditya Shastri, Hiren Kumar Thakkar, and Ashutosh Londhe. zsort: Stable distribution sort using z-score partitioning. *arXiv preprint arXiv:2605.14419*, 2026.
34. Seunghu Kim, Yongseok Son, Jeongyeup Paek, Sungrae Cho, and Hyung Tae Lee. A survey of blind sorting techniques: Approaches using fully homomorphic encryption. In *2024 15th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 480–485. IEEE, 2024.
35. Seunghu Kim, Eymen Ünay, Ayse Yilmazer-Metin, and Hyung Tae Lee. Optimized rank sort for encrypted real numbers. *Cryptology ePrint Archive*, 2025.
36. Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison-Wesley Professional, 1998.
37. Surendra Kumar, Mridula Dwivedi, Mohit Kumar, and Sukhpal Singh Gill. A comprehensive review of vulnerabilities and ai-enabled defense against ddos attacks for securing cloud services. *Computer Science Review*, 53:100661, 2024.
38. Antonio Mallia, Torsten Suel, and Nicola Tonellotto. Faster learned sparse retrieval with block-max pruning. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2411–2415, 2024.
39. Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.
40. 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)*, pages 8541–8558, 2025.
41. Harika Narumanchi, Dishant Goyal, Nitesh Emmadi, and Praveen Gauravaram. Performance analysis of sorting of fhe data: integer-wise comparison vs bit-wise comparison. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 902–908. IEEE, 2017.

42. Takumi Nishimura, Kazunari Tozawa, and Kunihiko Sadakane. Surpassing the word size limitation of tfhe with noise calibration. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 438–452, 2025.
43. Sulin Pang, Jiaqi Wang, and Xiaoshuang Yi. Application of loan lost-linking customer path correlated index model and network sorting search algorithm based on big data environment. *Neural Computing and Applications*, 35(3):2129–2156, 2023.
44. Dijana Peras and Renata Mekovec. A conceptualization of the privacy concerns of cloud users. *Information & computer security*, 30(5):653–671, 2022.
45. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
46. Lorenzo Rovida, Alberto Leporati, and Simone Basile. Lightweight sorting in approximate homomorphic encryption. *Cryptology ePrint Archive*, 2025.
47. Pranith K Roy and Krishnendu Shaw. An integrated fuzzy credit rating model using fuzzy-bwm and new fuzzy-topsis-sort-c. *Complex & Intelligent Systems*, 9(4):3581–3600, 2023.
48. Hamid Sarbazi-Azad. Stupid sort: A new sorting algorithm. *Newsletter of Computing Science Department, Univ. of Glasgow*, 599:4–10, 2000.
49. Atsuki Sato and Yusuke Matsui. Pcf learned sort: a learning augmented sort algorithm with o (n log log n) expected complexity. *arXiv preprint arXiv:2405.07122*, 2024.
50. Feng Shi, Zhiyuan Yan, and Meghanad Wagh. An enhanced multiway sorting network based on n-sorters. In *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 60–64. IEEE, 2014.
51. Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.
52. Damien Stehlé and Ron Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 27–47. Springer, 2011.
53. Chunping Wang, Jiahui Chen, Xiaoli Zhang, and Hongbing Cheng. An efficient fully homomorphic encryption sorting algorithm using addition over tfhe. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 226–233. IEEE, 2023.
54. Po Yang, Congmin Yang, Vitaveska Lanfranchi, and Fabio Ciravegna. Activity graph based convolutional neural network for human activity recognition using acceleration and gyroscope data. *IEEE Transactions on Industrial Informatics*, 18(10):6619–6630, 2022.
55. Hamoud Younes, Ali Ibrahim, Mostafa Rizk, and Maurizio Valle. An efficient selection-based knn architecture for smart embedded hardware accelerators. *IEEE Open Journal of Circuits and Systems*, 2:534–545, 2021.
56. Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.
57. Enrico Zardini, Enrico Blanzieri, and Davide Pastorello. A quantum k-nearest neighbors algorithm based on the euclidean distance estimation. *Quantum Machine Intelligence*, 6(1):23, 2024.