Efficient Sorting of Homomorphic Encrypted Data with k-way Sorting Network

Seungwan Hong[1], Seunghong Kim[1], Jiheon Choi[2], Younho Lee[3], and Jung Hee Cheon[1]

[1] Seoul National University, South Korea
{swanhong, 0815sy, jhcheon}@snu.ac.kr
[2] Hanyang University, South Korea
2005cjh@naver.com
[3] Seoul National University of Science and Technology, South Korea
younholee@seoultech.ac.kr

Abstract. In this study, we propose an efficient sorting method for encrypted data using fully homomorphic encryption (FHE). The proposed method extends the existing 2-way sorting method by applying the k-way sorting network for any prime k to reduce the depth in terms of comparison operation from $O(\log_2^2 n)$ to $O(k \log_k^2 n)$, thereby improving performance. We apply this method to approximate FHE which is widely used due to its efficiency of homomorphic arithmetic operations. In order to build up the k-way sorting network, the k-sorter, which sorts k-numbers with a minimal comparison depth, is used as a building block. The approximate homomorphic comparison, which is the only type of comparison working on approximate FHE, cannot be used for the construction of the k-sorter as it is because the result of the comparison is not binary, unlike the comparison in conventional bit-wise FHEs. To overcome this problem, we propose an efficient k-sorter construction utilizing the features of approximate homomorphic comparison. Also, we propose an efficient construction of a k-way sorting network using cryptographic SIMD operations.
To use the proposed method most efficiently, we propose an estimation formula that finds the appropriate k that is expected to reduce the total time cost when the parameters of the approximating comparisons and the performance of the operations provided by the approximate FHE are given. We also show the implementation results of the proposed method, and it shows that sorting $5^6 = 15625$ data using 5-way sorting network can be about 23.3% faster than sorting $2^{14} = 16384$ data using 2-way.

1 Introduction

The importance of data privacy and the prevalence of the delegation-of-computation paradigm have necessitated the processing of encrypted data. Fully homomorphic encryption (FHE), an innovative method that facilitates operations on encrypted inputs without decryption, has been developed to turn privacy-preserving computation into reality [26]. FHE is especially useful in situations where analysis of

large-scale or privacy-sensitive data (e.g., patient DNA analyses) is delegated to a cloud server, without the data themselves being disclosed to the cloud server [39, 31].

However, building applications using FHE is difficult because of the special characteristics of FHE. First, FHE schemes support only a limited set of basic operations, which are mainly addition and multiplication. When we design a computational algorithm that takes plaintexts as input, we can use a variety of computational tools and functions. However, when designing computational algorithms that take FHE-encrypted data as inputs, we cannot use them unless they can be implemented using the basic operations supported by FHE.

Second, because the intermediate result of a computational algorithm implemented by FHE is in an encrypted state, it is difficult to improve the performance of the algorithm by designing it to run based on the intermediate result. This is in contrast to classic algorithms that take unencrypted inputs. For instance, if a conditional statement is executed in an algorithm implemented using FHE operations, the evaluation result of the conditional statement is in an encrypted state. Thus, our algorithm cannot be designed to utilize the evaluation result to improve the performance.

Lastly, for the sake of security, a small error value is added to a ciphertext in FHE. The error value grows in proportion to the number of operations performed on the ciphertext. If this error exceeds the limit, the error overwrites the message hidden in the ciphertext. Consequently, the decryption cannot recover the message anymore.

After a fresh ciphertext is generated as a result of encryption, we can perform an operation supported by FHE with the ciphertext, and the result is again used as an input for another operation, allowing repeated operations. Unfortunately, there is a limit to the number of possible operations because the amount of error inside the result increases each time an operation is performed. In order to overcome this limitation and enable to perform more operations, a new ciphertext can be created that contains the same message as the original ciphertext but has a smaller error. This operation is called *bootstrapping*. Because the computational cost of bootstrapping is overwhelmingly high compared to other basic operations in FHE, to save the computational cost, it is essential to use bootstrapping as infrequently as possible when implementing algorithms with FHE operations.

Recently, many FHE schemes have been developed. They can be classified into two types according to the message storage method: bit-wise and word-wise. Bit-wise FHEs [19, 22, 30] encrypt data in a bit-by-bit manner. They support fast logical operations, such as comparison. However, bit-wise FHEs are inefficient for arithmetic operations between multi-bit numbers (or word-sized numbers) because only bit-wise operations are supported by them. Thus, a lot of operations are needed to process multi-bit data, and the computational cost of the FHE operation in bit-wise FHEs is very close to that in word-wise FHEs. On the contrary, word-wise FHEs [5, 25, 28], which store messages as their word-sized numbers, support high-speed arithmetic operations between messages. Moreover,

an efficient *approximate* comparison algorithm for word-wise encrypted data with Single-Instruction Multiple-Data (SIMD) method was recently developed, which enhanced the usability of the word-wise FHEs [15].

In this study, based on this approximate comparison algorithm, we target efficient sorting of large numbers encrypted by a word-wise FHE. Among various data processing techniques used as tools in many applications, sorting is one of the most promising techniques that can be used for numerous applications, such as in k-means clustering, top-k data operations, binning, and exploratory statistical analysis of data. To build up a sorting method on encrypted data, two elementary algorithms are used repeatedly as building blocks: one that compares two or more data elements and the other that replaces the positions of the data elements based on the comparison result. Unlike the comparison operation provided by the order-preserving encryption (OPE) [1] where its comparison result displays which of two input ciphertexts has a greater plaintext value than the other, the FHE comparison operation produces a new ciphertext that has the maximum value[4] among the two plaintext values in the input ciphertexts. However, due to the randomization performed on the comparison operation, it is difficult to know which input ciphertext contains the same plaintext value as the result of the comparison. This type of comparison does not compromise the security of the encryption as opposed to OPE.

For that reason, in the case of Quick Sort, which is a repetition of classifying the data to be sorted based on a selected pivot value, it is difficult to apply to the encrypted data by FHE: If the ciphertext of a pivot value is compared with a ciphertext containing a value, the comparison algorithm does not tell if the value is greater than the pivot value or not. For the same reason, it is also difficult to apply well-known sorting algorithms with $O(n \log n)$ comparisons such as Merge Sort. In particular, research has claimed that Quick Sort is impossible to implement on top of secure FHEs [9]. Therefore, among the classical sorting algorithms, researchers succeeded in constructing only inefficient sorting methods such as Bubble Sort or Insertion Sort, which requires $\mathrm{O}(n^2)$ comparison operations [8], for the encrypted data by FHE.

To avoid such disadvantages of classical sorting algorithms, recent studies have mainly used the (2-way) Sorting Network algorithms, such as Batcher's Odd-Even Sort (BOES) or Bitonic Sort [24, 36]. Unlike most efficient classical algorithms that determine the next comparison target based on the intermediate result, the sorting network is an algorithm that pre-determines the order of computations. Although the sorting network has $O(n \log^2 n)$ complexity, which is more inefficient than optimal comparison-based sorting, it is considered to be more friendly for sorting the data encrypted by FHE mainly because of the two reasons: first, there is no need to know from which input ciphertext the ciphertext of the larger (smaller) value came, and second, the sorting is completed by performing the same operations of the same order regardless of the result of the intermediate comparisons. Moreover, the sorting network is suitable for paral-

[4] The ciphertext of the minimum value can be derived by the subtracting the output ciphertext from the sum of both input ciphertexts.

lel computations so the depth of the algorithm is $O(\log^2 n)$ when $O(n)$ parallel structure is given.

From the viewpoint of reducing the depth of the algorithm, Shi *et al.* proposed the k-way sorting network [40], which is a generalized algorithm of the sorting network. The k-way sorting network is an algorithm for a prime k that can sort numbers using j-sorters, the algorithms that can sort j number of input values, for $j \leq k$, as a building block. A k-way sorting network consists of a number of *stages*, each of which consists of some unit operations where a unit operation can be divided into a comparison step and a swap step. In the comparison step, for each of j data that would be sorted in j -sorter, a comparison operation is performed for all possible pairs of them. Then, in the swap phase, each j-sorter uses the result of the comparison to sort j data without using any more comparison operations. In the case of the 2-way sorting network, the comparison step performs a comparison between predetermined data pairs, and the swap step computes the min-max function with one multiplication using the comparison result. On the other hand, for k-way sorting network, the comparison step performs $\binom{j}{2}$ number of comparisons for each pair in group with j elements in parallel for $j \leq k$. After that, in the swap step, each of j elements must be sorted using the comparison result.

The number of steps in the k-way sorting network is asymptotically $O(k \log_k^2 n)$, and more precisely, 5-way sorting network requires about 44% fewer stages compared to 2-way when n is large. Since the number of steps is the number of times to perform the parallel comparison operation, which affects the number of bootstrapping, a larger k is more advantageous in the aspect of reducing depth in FHE-based sorting. In order to implement k-way sorting network over FHE, we need j-sorter algorithm for $j \leq k$ as a building block, which sorts j elements using additions and multiplications under the condition that all possible pairs comparisons are given.

1.1 Our Contribution

In this paper, we propose a new sorting algorithm that sorts small elements efficiently with one comparison depth. Using our algorithm as a building block, we enable exploiting k-way sorting network algorithm [40] to sort over large-scale encrypted data by FHE. The sorting algorithm that we have devised can be applied to FHE based on approximate operations and the exploitation of k-way sorting network algorithm is the idea that can be generally applied to all FHE. As we use k larger than 2 for large n, the number of stages required for k-way sorting network is reduced while the cost for computing k-sorter is increased. Therefore, if an appropriate k is used for a given n and an implementation environment for FHE, the time cost required for sorting can be reduced compared to the conventional 2-way method. To the best of our knowledge, our work is the first attempt in the literature to apply k-way sorting network over encrypted data.

For more details, we first propose a new k-sorter algorithm which aims to sort k elements using addition and multiplication only, in the condition that comparisons of all possible pairs of k elements are given. The main purpose

of the algorithm is to perform sorting even if the comparison result is given approximately. To achieve this, we exploit the computation method of finding a maximum of two variables. Let θ_{ab} be the comparison value of variables a, b, i.e. $\theta_{ab} = 1$ if $a > b$ and 0 otherwise. Then, the max value of (a, b) can be computed with addition and multiplication by $\mathsf{L}_{a>b}(a, b) = \theta_{ab} \cdot a + (1 - \theta_{ab}) \cdot b$. In the above expression, we interpret the max function as a function that outputs the internally dividing point between two numbers according to the comparison result.

To be precise, suppose that the comparison value is given approximately, which means that θ_{ab} is close to 1 or 0 if $|a - b|$ is large and outputs a number close to 0.5 if $|a - b|$ is small. If $|a - b|$ is large so that the error between θ_{ab} and true comparison value is small enough, then $\mathsf{L}_{a>b}(a, b)$ outputs the value close to max. Otherwise, then a and b are too close so θ_{ab} is close to 0.5, but in this case, since L outputs the internally dividing point between a and b, the output would still close to both a and b, the candidates of max value. Consequently, we can build a sorting algorithm with approximate comparison by the compositions of L's. In particular, our algorithm is made so that the sorting result is less than a certain error compared to the true value even if an approximate comparison is used. Because of that, our algorithm is useful for sorting over word-wise FHE in which the approximate comparison is more efficient than the exact one.

Secondly, we propose a method of implementing the k-way sorting network for FHE that supports SIMD operations and message rotation and provide the time cost estimation formula of the proposed method that enables finding the expected time cost for various k's. In order to operate between two messages stored in the same ciphertext, it is necessary to store them in the same location for SIMD operation in the different ciphertext. This process can be done by multiplying the ciphertext by a plain vector of 0s and 1s, called the masking vector, to generate a new ciphertext containing only the desired message and then rotating it to match the position. Using our SIMD-friendly method for the k-way sorting network algorithm, it can be efficiently computed for when data are packed in one ciphertext.

Since the depth consumed by the algorithm and the number of required multiplications can be determined in advance, we can analyze the total expected computation cost in terms of the number of bootstrapping and multiplications in FHE. In a practical situation, the implementation speed of an algorithm is closely related not only to the algorithm itself but is also dependent on how well the FHE library has been implemented or on the efficiency of the algorithm used for bootstrapping. Therefore, we estimate the time cost of our algorithm for each k under the condition of FHE implementation as well as the input size. As a result, we propose an explicit formula for computing required depth and the number of multiplications from the parameters n, k, the time cost of multiplication and bootstrapping, and the number of messages that can be contained in one ciphertext. Using our estimation, we can predict the expected time cost for various k so that the appropriate k can be estimated that works efficiently with our method on the fixed implementation environment.

At last, we verify the performance of the proposed sorting method by implementing it with CKKS FHE scheme [17]. Using our method, we show the results of sorting $n = k^m$ elements for some m in our concrete parameter setting for implementation. Then, based on the implementation, we measure the elapsed time for sorting various numbers in a PC environment. As a result of the performance analysis, we confirm that it takes about 4.9 hours to sort 5^6=15625 multi-precision real numbers using 5-way sorting network, which is about 23.3% times faster than the result for $2^{14} = 16384$ elements using 2-way sorting network that takes about 6.4 hours in the same environment. We report implementation results of our sorting algorithm over encrypted data for $k \leq 5$ in Section 5.

1.2 Related Works

The problem of sorting over encrypted data was first addressed in [8]. In that work, the authors defined HE-based comparison functions and swapping algorithms through bit-wise encoding, and implemented Bubble Sort and Insertion Sort with the hcrypt library [6]. They also proposed the LazySort algorithm, which combines Bubble Sort and Insertion Sort to reduce the number of Recrypt operations in FHE.

Working to obtain a more efficient sorting algorithm for FHE, Chatterjee and SenGupta [9] proved that, if partition-based sort can be be performed on FHE data, the FHE scheme is insecure against indistinguishability under chosen-plaintext attack (IND-CPA). Thus, to perform partition-based sorting such as that achieved with Quick Sort, not only the plaintext but also the array indices should be encrypted. For that case, however, the authors of [9] showed that implementation of Quick Sort on an encrypted array yields no better performance than those of other comparison-based sorting methods. In [24], Emmadi *et al.* reported results for well-known sorting algorithms implemented on encrypted data. Hence, they showed that BOES yields the best performance among the various sorting techniques, such as Bubble Sort, Insertion Sort, and Bitonic Sort, over integer-based HE [41].

In terms of 2-way sorting network, Kim *et al.* [36] implemented the Bitonic Sort algorithm with SIMD operations on the Homomorphic Encryption Library (HElib) [30]. They performed SIMD bit-wise operations on HElib to implement reverse and bitonic merge algorithms for bitonic sequences. Recently, Lu *et al.* [38] also implemented Bitonic Sort using their optimized framework PEGASUS. This work uses comparison operation by evaluating look-up tables on ciphertexts and perform sorting 64 data in 409.09 seconds with 4 threads.

The main limitation of the above methods is that the operation depth for sorting is too large, and thus, it is hard to obtain implementation results for large data sizes. Cetin *et al.* [7] overcame this obstacle using an algorithm that requires a log-scale depth corresponding to the number of messages and bit lengths. However, that algorithm compares all message pairs, which becomes infeasible if the number of messages exceeds thousands. Moreover, the algorithm expresses sorting as a logical operation of comparison results, which is difficult to apply for approximate FHE.

2 Preliminaries

2.1 Homomorphic Encryption (HE)

HE is a special type of encryption that supports arithmetic operations between ciphertexts without decryption. HE is very useful in cases involving private data as it allows analysis of encrypted data without information leakage from messages. HE was first proposed in a blueprint by Gentry and Boneh [26], and a number of HE schemes have been suggested [2, 4, 14, 20, 21, 42, 23, 25, 27, 28] and applied to various fields [3, 11, 29, 34].

In a HE scheme, a ciphertext is created with a small degree of noise to ensure security. When certain operations are performed using ciphertexts, the resultant ciphertext has greater noise than the input ciphertexts. Thus, the noise "grows" whenever operations are performed using these ciphertexts. If the noise exceeds certain bounds, however, the decryption algorithm does not function properly. Hence, we can estimate the remaining number of possible consecutive operations that can be performed on a ciphertext while preserving its correct decryptability. We can say this is the ciphertext level in HE. To overcome this problem of excessive ciphertext noise, Gentry suggested a so-called bootstrapping technique to refresh the noise. Given a ciphertext with large noise, the bootstrapping algorithm converts it to another ciphertext, which is the encryption of the same message with small noise. This allows evaluation of functions with an arbitrary number of levels without decryption. A HE scheme with bootstrapping is called an FHE scheme. The main idea of bootstrapping is a creation of the new ciphertext that is the output of the decryption function implemented with FHE operations, which takes the encrypted decryption key and the target ciphertext to be refreshed. Most FHE schemes support elementary operations only, but decryption requires complex non-polynomial operations such as modulus reduction; therefore, the cost of bootstrapping is incomparably larger than other operations. Thus, one of the main research topics pertaining to FHE application is reduction of the number of bootstrapping operations for improved efficiency.

Approximate Homomorphic Encryption Most of FHE schemes have a message space as $\mathbb{Z}_p$ or a vector space $\mathbb{Z}_p^n$ for some integers p and n. For $p = 2$, such a message space is efficient for bit-wise computation. For general integer arithmetic, however, bit-wise operations generate heavy cost and are, thus, unsuitable in terms of efficiency. Selection of larger p increases the FHE operation cost.

In 2017, Cheon *et al.* proposed an approximate homomorphic encryption scheme called HEaaN [14]. In HEaaN, the small amount of noise added to the plaintext during the encryption process for ciphertext security is not removed during decryption. The magnitude of this noise is very low compared to the decryption result; therefore, the decryption result retaining the noise is regarded as an "approximate" decryption result.

HEaaN also provides a rescaling operation to remove the least significant bits of error, which causes the linear growth of the bit length of error proportionally to the number of levels consumed. In this sense, we can represent a ciphertext's level by the bit-length of its modulus. The efficiency of HEaaNhas been proven

for many real-world applications, such as machine learning [34, 35] and cyber physical system [11], and is still being improved with better bootstrapping algorithms [10, 12]. We omit details of HEaaN scheme construction from this paper, and use the algorithms listed below as black boxes. Note that, a fresh HEaaN ciphertext is equipped with a modulus Q, which is reduced by $1/\Delta$ for every multiplication, where Δ is a message scaling factor. If the minimal Q is reached (related to bootstrapping), we use the bootstrapping algorithm to raise the modulus to some fixed value $q_0 < Q$. We also remark that parameters N, Δ, Q, and s in setup should be the power of 2. Definitions of each algorithm in below. For the details, we refer [14], and especially [10, 12] for bootstrapping.

- $\mathsf{Setup}(1^\lambda, N, \Delta, Q, h, s)$: Given a security parameter λ, a polynomial ring of degree N, a rescaling factor Δ, ciphertext modulus Q, and a number of slots $s \leq N/2$, the algorithm outputs a secret key sk with hamming weight h, a public key pk, a evaluation key evk, and a bootstrapping key bk.
- $\mathsf{Enc}(\mathsf{pk}, \mathbf{m})$: Given a public key pk and a message vector $\mathbf{m} \in \mathbb{C}^s$, the algorithm outputs a ciphertext ct with modulus Q, which is the encryption of $\Delta \cdot \mathbf{m} + \mathbf{e}$, i.e., the message is scaled by a factor Δ and a small error vector $\mathbf{e} \in \mathbb{C}^n$ is added. The vector structure of $\mathbf{m}$ is preserved after the encryption. We say the position where each element is $\mathbf{m}$ is placed in ct as *slot*.
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$: Given a secret key sk and a ciphertext ct which is the encryption of $\Delta \cdot \mathbf{m} + \mathbf{e}$, the algorithm outputs the message vector with errors $\mathbf{m} + \Delta^{-1} \cdot \mathbf{e}$.
- $\mathsf{Add/Sub}(\mathsf{ct}_1, \mathsf{ct}_2)$: Given two m q-modulus ciphertexts ct_1 and ct_2, which are encryptions of message vectors $\Delta \cdot \mathbf{m}_1 + \mathbf{e}_1$ and $\Delta \cdot \mathbf{m}_2 + \mathbf{e}_2$, respectively, the algorithms outputs a q-modulus ciphertext $\mathsf{ct}_{add}/\mathsf{ct}_{sub}$ which is the encryption of $\Delta \cdot (\mathbf{m}_1 \pm \mathbf{m}_2) + \mathbf{e}'$ for some small error vector $\mathbf{e}'$, respectively.
- $\mathsf{Mult}(\mathsf{evk}, \mathsf{ct}_1, \mathsf{ct}_2)$: Given two q-modulus ciphertexts ct_1 and ct_2, which are encryptions of message vectors $\Delta \cdot \mathbf{m}_1 + \mathbf{e}_1$ and $\Delta \cdot \mathbf{m}_2 + \mathbf{e}_2$, respectively, the algorithms outputs a q/Δ-modulus ciphertext ct_{mult} which is the encryption of $\Delta \cdot \mathbf{m}_1 \odot \mathbf{m}_2 + \mathbf{e}'$ for some small error vector $\mathbf{e}'$.
- $\mathsf{LeftRotate}(\mathsf{evk}, \mathsf{ct}, d)$: Given the q-modulus ciphertext ct, which is an encryption of $\Delta \cdot \mathbf{m} + \mathbf{e}$ and $d > 0$, the algorithm outputs a q-modulus ciphertext ct' which is an encryption of $\Delta \cdot (m_d, \cdots, m_{s-1}, m_0, m_1, \cdots, m_{d-1}) + \mathbf{e}'$ for some small error vector $\mathbf{e}'$.
- $\mathsf{RightRotate}(\mathsf{evk}, \mathsf{ct}, d)$: Given the q-modulus ciphertext ct, which is an encryption of $\Delta \cdot \mathbf{m} + \mathbf{e}$ and $d > 0$, the algorithm outputs a q-modulus ciphertext ct' which is an encryption of $\Delta \cdot (m_{s-d}, \cdots, m_{n-1}, m_0, m_1, \cdots, m_{s-1-d}) + \mathbf{e}'$ for some small error vector $\mathbf{e}'$.
- $\mathsf{BootStrapping}(\mathsf{bk}, \mathsf{ct})$: Given a ciphertext ct with small modulus, the algorithm outputs a ciphertext q_0-modulus ct' which is an encryption of the same message vector.

We omit the public key as input when denoting the operations between ciphertexts (or ciphertext and plain vector) by common symbols, such as $\mathsf{Enc}(\mathsf{pk}, \mathsf{ct}) = \mathsf{Enc}(\mathsf{ct})$, $\mathsf{Add}(\mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{ct}_1 + \mathsf{ct}_2$, $\mathsf{Mult}(\mathsf{evk}, \mathsf{ct}_1, \mathsf{ct}_2) = \mathsf{ct}_1 \cdot \mathsf{ct}_2$, if it does not make any confusion.

2.2 Approximate Algorithms for Comparison Function

As noted above, current approximate HE schemes support addition and multiplication only. This has the advantage that polynomial operations can be performed on encrypted data. However, logical operations are difficult to implement with the operations of HE. Therefore, to compute a comparison function, an approximated version of a comparison function was proposed using polynomials.

Hereafter, we denote the approximate comparison for two inputs x, y by $(x > y)$ or $(y < x)$. We define $(x > y)$ by taking two inputs $x, y \in (0, 1)$ and outputs 1 if $x \gg y$ or 0 if $x \ll y$. If x and y are close compared to iteration number, then $(x > y)$ outputs the value between 0 and 1.

Cheon *et al.* [15] suggested an efficient formula to compute approximate comparison function by compositing the same polynomials. More precisely, those researchers have found $(2d + 1)$-degree polynomial $f_d(x)$ and $g_d(x)$ that satisfy certain conditions. Further, repeated compositions of $f_d(x)$ and $g_d(x)$ output the approximate value of the sign function.

The sorting method in this work employs the approximate comparison method where f_d and g_d are chosen by $d = 3$, where

$$f_3(x) = (35x - 35x^3 + 21x^5 - 5x^7)/2^4,$$
$$g_3(x) = (4589x - 16577x^3 + 25614x^5 - 12860x^7)/2^{10}.$$

To minimize the multiplication depth, we compute the multiplication with coefficient integers through repeated addition, and perform division with a power of two by divByPo2 algorithm. For the reference, graphs of f_3, g_3 and their compositions are shown in Fig. 1.

Note that, in [15], the authors recommended $n = 4$ for the optimal complexity. However, since the leading coefficient of $f_4(= 35/128)$ and $g_4(= 46623/1024)$ are positive, the f_4 and g_4 compositions may diverge if the absolute value of input x exceeds 1. When this comparison algorithm is implemented on top of the HEaaN library, if x is close to 1, the comparison result may diverge because of the attached error, yielding an incorrect output. The authors mentioned this problem for the max algorithm in their previous work [16], but the same problem occurs for f_n when n is even. Hence we select $n = 3$ in this work, which is the optimal odd-n value.

In conclusion, in this paper, $(x > y)$ is computed as follows for given iteration numbers d_f and d_g:

$$(x > y) := (f_3^{(d_f)} \circ g_3^{(d_g)}(x - y) + 1)/2. \tag{1}$$

Here $f^{(d)}$ means $f \circ f \circ \cdots \circ f$, performed for d times.

For the analysis of error of approximate comparison, we need following definition and theorem. To avoid confusion, we denote sign functions for intervals $[-1, 1]$ and $[0, 1]$ by $\mathsf{sgn}_{[-1,1]}$ and $\mathsf{sgn}_{[0,1]}$, respectively. $\mathsf{sgn}_{[-1,1]}(x)$ outputs ± 1 depending on whether $x \geq 0$ or not, and $\mathsf{sgn}(x)_{[0,1]}(x)$ outputs 1 or 0 depending on whether $x \geq 0.5$ or not, respectively.

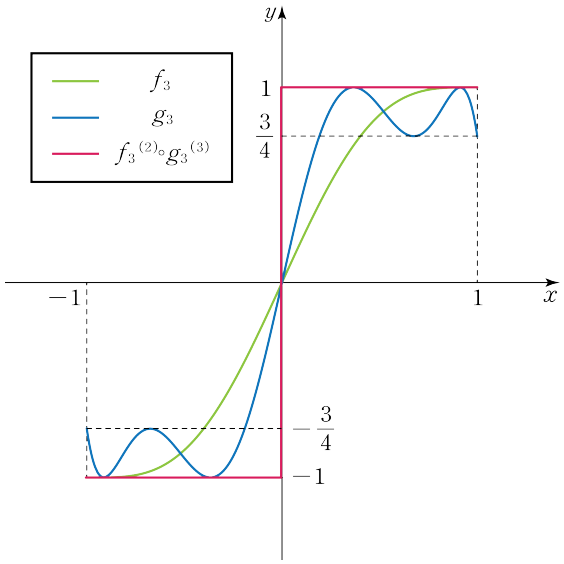

Fig. 1: Illustration of f_3, g_3, and their compositions. The compositions of g_3 make the absolute value of the output greater than $3/4$. Then it becomes close to the sign value of the input after compositions of f_3.

Definition 1 *[15] For $\alpha > 0$ and $0 < \epsilon < 1$, we say a polynomial p is $(\alpha, \epsilon)-$*close *to sign function $\mathsf{sgn}_{[-1,1]}(x)$ over $[-1, 1]$ if it satisfies*

$$||p(x) - \mathsf{sgn}_{[-1,1]}(x)||_{\infty,[-1,-\epsilon]\cup[\epsilon,1]} \leq 2^{-\alpha}.$$

*Similarly, p is $(\alpha, \epsilon)-$*close *to sign function $\mathsf{sgn}_{[0,1]}(x)$ over $[0, 1]$ if it satisfies*

$$||p(x) - \mathsf{sgn}_{[0,1]}(x)||_{\infty,[0,(1-\epsilon)/2]\cup[(1+\epsilon)/2,1]} \leq 2^{-\alpha}.$$

Theorem 2 *If $d_g \geq \frac{1/2+O(1)}{\log c_d} \cdot \log(1/\epsilon) + O(1)$ and $d_f \geq \frac{\log(\alpha-3)}{\log d} + O(1)$, the approximate comparison $(a > b) = (f_d^{(d_f)} \circ g_d^{(d_g)}(a - b) + 1)/2$ is (α, ϵ)-close to $\mathsf{sgn}_{[0,1]}$, i.e. it outputs an approximate value of comparison output within $2^{-\alpha}$ error for $a, b \in [0, 1]$ satisfying $|a - b| \geq \epsilon$.*

Proof. If (d_f, d_g) satisfies the above condition, then the polynomial $f_d^{(d_f)} \circ g_d^{(d_g)}(x)$ is $(\alpha - 1, \epsilon)$-close to $\mathsf{sgn}_{[-1,1]}$ from [15, Theorem 2]. Then the result is directly follows by scaling this polynomial to interval $[0, 1]$.

2.3 Security Models

Our scenario consists of two parties, data owner and service provider, who behave as honest-but-curious model. Data owners want to share their data to obtain

the results of certain operations with the data, but they don't want to leak any information of data to the service provider while the service provider perform the computation, denoted by a function f, to provide its service.

In our protocol, data owner encrypt her data m by her own FHE secret key and provide the ciphertexts $\mathsf{Enc}(m)$ to service provider. Then, the service provider manipulates the ciphertext to obtain the result of the own model $\mathsf{Enc}(f(m))$ using operations that FHE supports. Finally, the service provider give back the result $\mathsf{Enc}(f(m))$ to the data owner, and the data owner can receive the desired output $f(m)$ by decrypting the ciphertext with her own key.

Since IND-CPA security of FHE scheme guarantees that any adversary who has the ciphertexts and the results homomorphic operations between them cannot extract any information about the messages in ciphertexts, the security in our scenario is secure under the security of FHE, regardless of what f is supported by the service provider since the service provider can only access to the ciphertexts.

Recently, Li *et al.* [37] have noticed that the protocol using the approximate FHE including CKKS requires more attention than the general FHE for security. In short, we have to clear the lsb bits where only error value exists to preserve the security. Otherwise, the secret key can be extracted from both ciphertext and its decryption result. Therefore, users should be careful not to leak the decrypted values or consider other scheme modifications that are currently being researched [13].

2.4 Notations

All logarithms are of base 2 unless it is indicated. The set of finite non-negative numbers is denoted by $[n] = \{1, \cdots, n\}$. We call the ordered set of elements by array and denote it by capital letter. The index of array starts with 1. For instance, we denote arrays as $A = (a_1, \cdots, a_n)$ or $B^i = (b_1^i, \cdots, b_n^i)$. The number of elements in array X is denoted by $|X|$. We use slice notation to represent part of an array. Precisely, given s, f which are starting and final index of given array, we define $A[s{:}f] = (a_s, a_{s+1}, \cdots, a_f)$. If the starting index s is 1 or the final index f is $|A|$, the index in slice notation may be omitted, as $A[{:}f]$ or $A[s{:}]$. For an element x and an array A, the set of comparisons between x and all elements in A is denoted by $\{(x > A)\} = \{(x > a_i)\}_{a_i \in A}$ or $\{(A > x)\} = \{1 - (x > a_i)\}_{a_i \in A}$ Similarly, we denote the set of comparisons between all possible pairs of arrays A and B by $\{(A > B)\} = \{(a_i > b_j)\}_{a_i \in A, b_j \in B}$. Also, every sorted array in this paper is assumed to be written in decreasing order.

3 Our k-sorter Algorithm for encrypted data by FHE

The main purpose of our k-sorter algorithm is to sort encrypted data with the comparison results that come after comparing all k numbers with each other in a pair-wise manner using the approximate comparison algorithm in order to minimize the computation depth in terms of the comparison operation.

The main difference between the exact comparison and the approximate comparison is that the result of the exact comparison is clearly binary therefore the logical operations with the comparison results can be implemented using the multiplication and addition operations. However, in the approximate comparison, the result is not binary: the result is just closed to either 0 or 1. For instance, suppose three data a, b, c is given and let θ_{ab}, θ_{bc} be the exact or approximate comparison result for a, b and b, c, respectively. In order to obtain θ_{abc} which represents whether $a > b > c$ or not, we just need to calculate $\theta_{ab} \cdot \theta_{bc}$ if both are from exact comparison operations.

However, if the comparison is approximate, then θ_{ab} and θ_{bc} are in the interval $(0, 1)$ and we only know that $\theta_{ab} > 0.5$ if and only if $a > b$. Suppose it satisfies that $a > b > c$ but two numbers in each pair of (a, b) and (b, c) are too close, respectively, so that the comparison results are close to 0.5, say $\theta_{ab} = 0.7$ and $\theta_{bc} = 0.6$. In this case, $\theta_{abc} = \theta_{ab} \cdot \theta_{bc} = 0.42 < 0.5$, which implies $a > b > c$ is false. Therefore, in order to perform sorting using approximate comparison, it is necessary to have a new approach that requires further operations than just using the logical operation.

To devise sorting algorithm based on not logical operations but approximate comparisons, we recall trivial approach to find maximum between two elements a, b with their comparison. Using our comparison notation, 2-max function can be computed using the following formula:

$$\max(a, b) = (a > b) \cdot a + (a < b) \cdot b.$$

Since the equation $(a > b) + (a < b) = 1$ is always satisfied due to the characteristics of the method [15], $\max(a, b)$ is approximately equal to $(a > b) \cdot a + (a < b) \cdot a = a$ when a and b are very close. Therefore, it outputs approximately correct value even when $|a - b|$ is small and the approximate comparison value is close to 0.5.

Exploiting this property, we define the function L as

$$\mathsf{L}_{(a>b)}(F, G) = (a > b) \cdot F + (a < b) \cdot G.$$

Note that the definition of L is inspired by max function $\max(a, b) = \mathsf{L}_{a>b}(a, b)$. This function can be interpreted as returning F if $a > b$ and G if $a < b$, respectively. We propose a method of obtaining m-max function (which outputs m-th maximum of given n inputs) for $m \leq n$ as a composition of L's. Since L is multivariate function, the definition of the term "composition of L" needs to be clarified. We define a composition of L as $\mathsf{L}(\mathsf{L}, \mathsf{L})$. One potential problem of $\mathsf{L}_{(a>b)}(F, G)$ is that if $|a - b|$ is too small, the comparison results may output the values closed to 0.5 and the error can be large. To find the m-th max, we overcome this problem by selecting the functions that appropriately satisfy $F \approx G$ if $a \approx b$. More detail regarding this can be found in Lemma 2. Note that, in each step, we can always select F and G that satisfy this condition, as the m-max algorithm is symmetric. Equipping this condition, if $a \approx b$, then $\mathsf{L}_{a>b}(F, G) \approx F \approx G$ since $(a > b) + (a < b)$ is always 1. In the rest of the section, we develop our new k-sorter algorithm by the compositions of L function

only, and determine total error of the algorithm by showing that each part of compositions forming as $\mathsf{L}_{(a>b)}(F,G)$ satisfies $F \approx G$ if $a \approx b$.

3.1 Our Divide-and-Conquer Algorithm

Before stating full structure of our algorithm, we first explain the main ideas in our algorithm, which is to apply divide-and-conquer method.

To be precise, assume that arrays $A(A')$ can be divided into two disjoint sets $B(B')$ and $C(C')$, respectively, where $B = \{b_1, \cdots, b_s\}, C = \{c_1, \cdots, c_t\}, B'$ and C' are sorted arrays. In addition, the approximate comparisons $\{(B > C)\}$, $\{(B' > C')\}$ and $\{(A > A')\}$ are also given (recall that we denote the set of comparisons of two arrays X, Y by $\{(X > Y)\} = \{(x_i > y_j)\}_{(x_i,y_j)\in X\times Y}$. In this setting, our goal here is to obtain two results:

1. a sorted array of Z where $Z = A$ as a set
   (identically, Z' is a sorted array where $Z' = A'$).
2. $\{(Z > Z')\}$, which is necessary when computing a sorted sequence by merging Z and Z'.

First, assuming that B, C and $\{(B > C)\}$ are given, we aim to compute $Z = \{z_1, \cdots, z_{s+t}\}$ which is the sorted array of $B \cup C$. For $i \in [s]$ and $j \in [t]$, the key idea is that the number of candidates of z_{i+j} in $B \cup C$ can be reduced by using the comparison result between b_i and c_j. If $b_i > c_j$, then the candidates that are larger than b_i are elements in $B[:i-1]$ and $C[:j-1]$ which means there are at most $i + j - 2$ elements that are larger than b_i. Since the number of elements larger than z_{i+j} is exactly $i + j - 1$, b_i cannot be z_{i+j}, and under the same reason, $b_1, \cdots, b_{i-1}$ also cannot be z_{i+j}. Likewise, since all elements in $B[:i]$ and $C[:j]$ are larger than c_{j+1}, the number of elements larger than c_{j+1} is at least $i + j$. Hence c_{j+1} cannot be z_{i+j}, and under the same reason, $c_{j+2}, \cdots, c_t$ also cannot be z_{i+j}. Finally, the remaining candidates are in $B[i+1:] \cup C[:j]$. Since $B[:i]$ should be larger than z_{i+j}, z_{i+j} is the j-th maximal element in remaining candidates. If $b_i < c_j$, then symmetrically z_{i+j} is the i-th maximal element of $B[:i] \cup C[j+1:]$.

In conclusion, let $\max_m(B, C)$ be the m-th maximal element in the union of two sorted arrays B, C for $1 \le m \le \lfloor \frac{s+t}{2} \rfloor$. Then our merging algorithm can be summarized by

$$\max_m(B, C) = \mathsf{L}_{(b_i > c_j)}\,(X, Y) \tag{2}$$

where $(i, j) = (\lfloor \frac{m}{2} \rfloor, \lceil \frac{m}{2} \rceil)$ and X, Y are defined recursively as

$$X = \max_j(B[i+1:], C[:j]), Y = \max_i(B[:i], C[j+1:]).$$

Under this condition, the length of arrays to be merged is reduced into approximately half of the original length. Therefore, $\max_m(B, C)$ can be computed for any m by recursively repeating the above process with about $\log m$ recursive

Algorithm 1 m-th Max/Min Algorithms for 2 Sorted Arrays

function MAX($m, B, C, \{(B > C)\}$)
  ▷ sorted arrays $B = \{b_1, \cdots, b_s\}, C = \{c_1, \cdots, c_t\}$
  ▷ $(B > C)$, a set of comparisons between B and C
  if $|B| = 0$ or $|C| = 0$ **then**
    if $|B| = 0$ **then return** c_m **else return** b_m
  else
    $(i, j) \leftarrow (\lfloor \frac{m}{2} \rfloor, \lceil \frac{m}{2} \rceil)$
    left $\leftarrow$ MAX$(j, B[i{+}1{:}], C[{:}j], \{(B[i{+}1{:}] > C[{:}j])\})$
    right$\leftarrow$MAX$(i, B[{:}i], C[j{+}1{:}], \{(B[{:}i] > C[j{+}1{:}])\})$
    return $\mathsf{L}_{(x_i > y_j)}$ (left, right)
  end if
end function

function MIN($m, B, C, \{(B > C)\}$)
  ▷ sorted arrays $B = \{b_1, \cdots, b_s\}, C = \{c_1, \cdots, c_t\}$
  ▷ $(B > C)$, a set of comparisons between B and C
  if $|B| = 0$ or $|C| = 0$ **then**
    if $|B| = 0$ **then return** c_{t-m+1} **else return** b_{s-m+1}
  else
    $(i', j') \leftarrow (s - \lfloor \frac{m}{2} \rfloor, t - \lceil \frac{m}{2} \rceil)$
    left $\leftarrow$ MIN$(j', B[{:}i'{-}1], C[j'{:}], \{(B[{:}i'{-}1] > C[j'{:}])\})$
    right$\leftarrow$MIN$(i', B[i'{:}], C[{:}j'{-}1], \{(B[i'{:}] > C[{:}j'{-}1])\})$
    return $\mathsf{L}_{(x_{i'} < y_{j'})}$ (left, right)
  end if
end function

depth. The m-th minimal element in $B \cup C$ can be computed in similar logic. The explicit algorithms to find m-th maximal/minimal element are stated in Algorithm 1.

Secondly, assume that we have sorted A and A' into Z and Z', respectively. Our next goal is to compute $\{(Z > Z')\}$ using comparisons results $\{(A > A')\}$. Note that the result in this step will be used for merging two sorted arrays Z and Z'. Our key idea is that if X and Y are sorted arrays, the comparisons $\{(X > r)\}$(or $\{(r > Y)\}$) are also sorted(or reversely sorted, resp) for any element $r \in (0, 1)$. Precisely, $(a_{i_1} > a'_j) > (a_{i_2} > a'_j)$ if and only if $(a_{i_1} > a_{i_2}) > 0.5$ and $(a_i > a'_{j_1}) > (a_i > a'_{j_2})$ if and only if $(a'_{j_1} > a'_{j_2}) < 0.5$. Hence, we can use $\{(B > C)\}$ when merging two sorted arrays $\{(B > a'_j)\}$ and $\{(C > a'_j)\}$ as if it is the comparison output $((b_{i_1} > a'_j) > (c_{i_2} > a_j))$, or similarly use it reversely to merge $\{(a_i > B')\}$ and $\{(a_i > C')\}$.

In conclusion, to compute $(z_i > z'_j)$ (or equivalently $(z'_j > z_i) = 1 - (z_i > z'_j)$), we basically need to apply the above merging process exactly twice. The first step is to merge two sorted arrays of $\{(B > a'_j)\}$ and $\{(C > a'_j)\}$ for each $a'_j \in A'$. The comparison between them are given as $\{(B > C)\}$ as explained in

previous paragraph. The output of the first step is $\{(Z > a'_j)\}$ and by repeating this process for all $a'_j \in A'$, we get $\{(Z > A')\}$=$\{(Z > B')\} \cup \{(Z > C')\}$. Along with this result, the following step is to merge $\{(z_i > B')\}$ and $\{(z_i > C')\}$ for $z_i \in Z$ in which the comparison result between $\{(z_i > B')\}$ and $\{(z_i > C')\}$ is $\{(C' > B')\}$. This step is also same with computing z_{i+j} by replacing b_i, c_i with $(z_i > b'_i), (z_i > c'_i)$. The result of the second step is $\{(z_i > Z')\}$ and by repeating this process for all $z_i \in Z$, we obtain $\{(Z > Z')\}$.

Combining above two steps, we state the precise k-sorter algorithm, which sorts array with k elements when the comparison of all possible pairs are already given, in Algorithm 2. We also illustrate the whole progress of our algorithm in the case of $k = 2^m$ (in this case the algorithm can be easily understood as divide-by-half and merging method) in Figure 2.

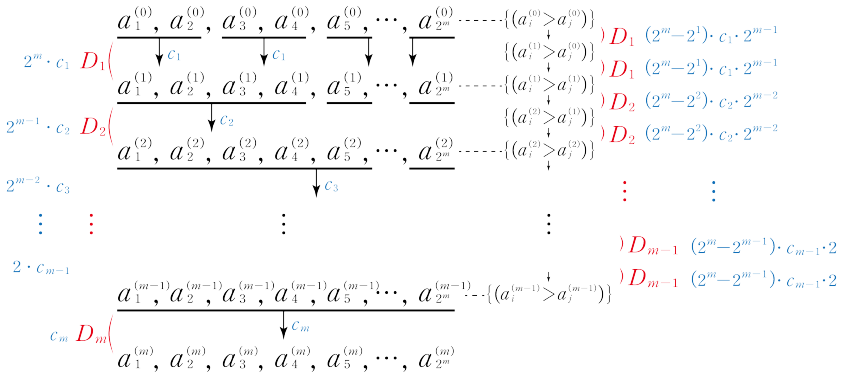

Fig. 2: Diagram that summarizes the entire algorithm of k-sorter and depicts the multiplication and depth consumed in each step

3.2 Complexity of k-Sorter

In this section, we calculate the asymptotic depth and multiplication complexity of our k-sorter algorithm. In fact, the depth and complexity of an algorithm that consists of polynomial operations can be dependent on how and in what order the multiplication is done. We show that our k-sorter algorithm requires log square depth and cubic complexity over input size k.

Prior to the examination, let's define the term stage i as $\{a_1^{(i)}, \cdots, a_{2^m}^{(i)}\}$ where the elements are grouped into sorted arrays of size 2^i. For the better understanding of the computation process, we assume $k = 2^m$. If k is not a power of 2, then k-sorter can be regarded as part of 2^m-sorter where $2^m > k$.

As in the illustration in Figure 2, the whole algorithm consists of the compositions of function MERGE. Let the multiplication complexity and depth of the MERGE algorithm in merging two sorted arrays of length 2^{n-1} be C_n and D_n,

Algorithm 2 k-Sorter Algorithm

function SORTER($A = \{a_1, \cdots, a_k\}, \{(A > A)\}$)
  ▷ an array A and pairwise comparisons $\{(A > A)\}$
  if $k = 1$ **then**
    return A
  end if
  $(s, t) \leftarrow (\lfloor \frac{k}{2} \rfloor, \lceil \frac{k}{2} \rceil)$
  $B \leftarrow$ SORTER($A[:s], \{(A[:s] > A[:s])\}$)
  ▷ $A[:s]$ is sorted to $B = \{b_1, \cdots, b_s\}$
  $C \leftarrow$ SORTER($A[s{+}1:], \{(A[s{+}1:] > A[s{+}1:])\}$)
  ▷ $A[s+1:]$ is sorted to $C = \{c_1, \cdots, b_t\}$
  for $j = s+1, \cdots, k$ **do**
    $\{(B{>}a_j)\}\leftarrow$
   SORTER($\{(A[:s]{>}a_j)\}, \{(A[:s]{>}A[:s])\}$)
    ▷ $\cup_{j=s+1}^{k}\{(b_i > a_j)\} = \{(b_i > A[s{+}1:])\}$
  end for
  for $i = 1, \cdots, s$ **do**
    $\{(b_i{>}C)\}\leftarrow$
   SORTER($\{(b_i{>}A[s{+}1:]\}, \{(A[s{+}1:]{>}A[s{+}1:])\}$)
    ▷ $\cup_{i=1}^{s}\{(b_i > C)\} = \{(B > C)\}$
  end for
  return MERGE($B, C, \{(B > C)\}$)
end function

function MERGE($B, C, \{(B > C)\}$)
  ▷ sorted arrays $B = \{b_1, \cdots, b_s\}, C = \{c_1, \cdots, c_t\}$
  ▷ $\{(B > C)\}$, a set of comparisons between B and C
  $k \leftarrow \lfloor \frac{s+t}{2} \rfloor$
  for $i = 1, \cdots, k$ **do**
    $z_i \leftarrow$ MAX($i, B, C, \{(B > C)\}$)
  end for
  for $j = k+2, \cdots, s+t$ **do**
    $z_j \leftarrow$ MIN($s + t - j, B, C, \{(B > C)\}$)
  end for
  $z_k \leftarrow \sum_{i=1}^{s} b_i + \sum_{j=1}^{t} c_j - \sum_{m\in[k], m\neq k+1} z_m$
  return $\{z_1, \cdots, z_{s+t}\}$
end function

respectively. MERGE is a repetition of MAX/MIN algorithm so the complexity and depth of MERGE comes from that of MAX algorithm as MIN is almost same with MAX.

We note that the MAX algorithm computes L function and the inputs of L are recursively L's. As in Algorithm 1, the composition occurs $\lceil \log i \rceil$ times when computing i-th maximal element of the merged array. Since L function consists

of one multiplication, the depth of computing i-th element is $\lceil \log i \rceil$. The depth of merging sorted arrays of length 2^n is the maximum of the depth of computing i-th element for $1 \leq i \leq 2^n$. Therefore, $D_n = \lceil \log 2^n \rceil = n$. From the Figure 2, we can observe that when merging sorted arrays of length 2^{n-1} in pairs to obtain sorted arrays of length 2^n, the comparison result $\{(a_i^{(n-1)} > a_j^{(n-1)})\}$ needs to be merged simultaneously, which requires depth of $2D_n$. If we let $\mathcal{F}_i$ as the depth necessary to obtain $\{(a_i^{(i)} > a_j^{(i)})\}$ from $\{(a_i^{(0)} > a_j^{(0)})\}$ and $\mathcal{T}_d^{(m)}$ as the total depth required for sorting $k = 2^m$ elements,

$$\mathcal{F}_i = \mathcal{F}_{i-1} + 2D_i,\ \mathcal{T}_d^{(m)} = \mathcal{F}_{m-1} + D_m.$$

From the recurrence relation, the exact formula of $\mathcal{T}_d^{(m)}$ can be computed as

$$\mathcal{T}_d^{(m)} = 2\sum_{i=1}^{m-1} D_i + D_m = 2\sum_{i=1}^{m-1} i + m = m^2 = O(\log^2 k).$$

For $2^{i-1} < l \leq 2^i$, let the complexity of calculating the l-th maximal element of the merged array as $C_m^{\max}(l)$. The l-th maximal element can be expressed as i compositions of functions. Within one composition operation, L takes two L as its inputs($\mathsf{L}(\mathsf{L}(\cdots), \mathsf{L}(\cdots))$) so in total, L emerges $1 + 2 + \cdots + 2^i = 2^{i+1} - 1$ times when computing l-th maximal element and so $C_m^{\max}(l) = 2^{i+1} - 1$ for $2^{i-1} < l \leq 2^i$. Since $C_m^{\min}(l)$ can be computed identically with $C_m^{max}(l)$, C_m can be computed as

$$\begin{aligned} C_m &= \sum_{i=1}^{2^{m-1}} (C_m^{\max}(i) + C_m^{\min}(i)) \\ &= 2\sum_{i=0}^{m-1} (2^{i+1} - 1)(2^i - 2^{i-1}) \\ &= \frac{2}{3} \cdot (4^m - 1) - (2^m - 1) = O(2^{2m}). \end{aligned}$$

Let $\mathcal{T}_c^{(m)}$ be the total complexity of sorting $k = 2^m$ elements. Figure 2 shows that the number of multiplications required for obtaining $A^i = \{a_j^{(i)}\}$ is $2^{m-i}C_i$. In order to compute $(A^i > A^i)$, the multiplication is operated $2(2^m - 2^i)2^{m-i} \cdot C_i$ times. It is because there are 2^{m-i} sorted arrays in stage i and for each sorted array $A_{2k}^{i-1} \cup A_{2k+1}^{i-1}$ of size 2^i, the merge of $(A_{2k}^{i-1} > r)$ and $(A_{2k+1}^{i-1} > r)$ of which its cost is C_i occurs $2^m - 2^i$ times as the number of candidates of r is $2^m - 2^i$. Also, this process is repeated twice when obtaining $\{(a_k^{(i+1)} > a_l^{(i+1)})\}$ from $\{(a_k^{(i)} > a_l^{(i)})\}$ as depicted in Figure 2. Based on this analysis, we can see that the following recurrence relation holds

$$\mathcal{T}_c^{(m)} = 2\mathcal{T}_c^{(m-1)} + C_m + C_{m-1} \cdot 2(2^m - 2^{m-1}).$$

From the recurrence relation, the exact formula of $\mathcal{T}_c^{(m)}$ can be computed as

$$\begin{aligned}\mathcal{T}_c &= \sum_{i=1}^{m} 2^{m-i}(1+2^{m+1}-2^{i+1})\cdot C_i \\ &= O(2^{2m}\sum_{i=1}^{m-1} 2^i - 2^m \sum_{i=1}^{m-1} 2^{2i}) = O(2^{3m}) = O(k^3).\end{aligned}$$

3.3 Error Analysis of Algorithm

In this work, we assume that the encrypted data are compared by approximate comparison algorithm [15]. Approximation comparison algorithms may not output exact values, and most of them will output erroneous values. In this section, we show that our algorithm works well with approximate comparison. Moreover, we estimate the final error in the output of our sorter algorithm based on the error in the approximate comparison that has already been analyzed. In the reminder of this section, we assume that the parameters (d, d_f, d_g) are chosen to satisfy that our approximate comparison is $(\alpha, \epsilon)-$close to sign function $\mathsf{sgn}_{[0,1]}$ as in Theorem 2.

At first, we state the error analysis of the function L. To do this, we define the desired true value of $\mathsf{L}_{(x>y)}(B,C)$(or L is it is not confused) by $\mathsf{T}_{(x>y)}(B,C)$(or T), i.e. $\mathsf{T} = \mathsf{T}_{(x>y)}(B,C) = B$ if $x > y$ and C otherwise. Also, the true values of B, C, and $(x > y)$ are denoted by T_B, T_C, and $t_{(x>y)}$. Let the error of L be $\mathcal{E} = \mathcal{E}_{(x>y)}(B,C) = |\mathsf{T}_{(x>y)}(B,C) - \mathsf{L}_{(x>y)}(B,C)|$ and the error of $(x > y)$ be $\epsilon_{(x>y)} = |t_{(x>y)} - (x > y)|$.

Lemma 1 *For $x, y, B, C \in (0,1)$, the error of L is bounded by $\mathcal{E}_{(x>y)}(B,C) \leq |B-C|/2$. Moreover, if $|x-y| \geq \epsilon$, then $\mathcal{E}_{(x>y)}(B,C) \leq 2^{-\alpha}$.*

Proof. $\mathsf{T}_{(x>y)}(B,C) = t_{x>y}\cdot B + (1-t_{(x>y)})\cdot C$. By the definition of $\mathcal{E}$,

$$\mathcal{E}_{(x>y)}(B,C) = |((x>y) - t_{x>y})\cdot(B-C)| \leq |B-C|/2.$$

Here the last inequality comes from that the max error of approximate comparison is $1/2$. If $|x-y| \geq \epsilon$, then by the definition of $(\alpha,\epsilon)-$closeness, $|t_{(x>y)} - (x > y)| \leq 2^{-\alpha}$. Since $|B-C| \leq 1$, the second inequality is obtained.

From the above lemma, we can observe that if it is guaranteed that B, C are close if x, y are close, then $\mathcal{E} = |\mathsf{L} - \mathsf{T}|$ can be bounded by ϵ or $2^{-\alpha}$ regardless of $|x-y|$. It is because if $|x-y| \geq \epsilon$, $\mathcal{E} \leq 2^{-\alpha}$ and if $|x-y| \leq \epsilon$, $\mathcal{E} \leq \frac{|B-C|}{2} \leq \frac{|x-y|}{2} \leq \frac{\epsilon}{2}$. The following lemma proves that $|B-C| \leq |x-y|$ within our divide-and-conquer algorithm.

Lemma 2 *Suppose two sorted arrays B, C are given and define $X = \max_j(B[i+1:], C[:j]), Y = \max_i(B[:i], C[j+1:])$ for some i, j. Assume that the $(i+j)$-th maximal element is computed by $\max_{i+j}(B,C) = \mathsf{L}_{(b_i>c_j)}(X,Y)$ as in Equation (2). Then, $|X-Y| \leq |b_i - c_j|$. Moreover, the error of L for $\max_{i+j}(B,C)(= \mathcal{E}_{(b_i>c_j)}(X,Y))$ satisfies $\mathcal{E}_{(b_i>c_j)}(X,Y) \leq \max\{\epsilon/2, 2^{-\alpha}\}$.*

Proof. Because $c_j \leq c_1, \cdots, c_j$, $X \geq c_j$. If $b_i < c_j$, $X = c_j$ and if $b_i > c_j$, then the number of elements that are larger than b_i is at most $j-1$ in $B[i+1:] \cup C[:j]$. Therefore, $c_j \leq X \leq \max(b_i, c_j)$. Symmetrically, $b_i \leq Y \leq \max(b_i, c_j)$. Thus, we can conclude that $|X - Y| \leq |b_i - c_j|$. Under the results in Lemma 1, if $|b_i - c_j| \geq \epsilon$, then $\mathcal{E} \leq 2^{-\alpha}$. Otherwise, $\mathcal{E} \leq |B - C|/2 \leq |b_i - c_j|/2 < \epsilon/2$. Therefore, $\mathcal{E} \leq \max\{\epsilon/2, 2^{-\alpha}\}$.

Next, we analyze the error of the composition of L in the setting of Algorithm 1. Firstly, we check that the error increases linearly with the composition of L. For instance, an L is given as $\mathsf{L}_{(x>y)}(A, B)$ where A, B are given as $A = \mathsf{L}_{(x_1>y_1)}(A_1, B_1), B = \mathsf{L}_{(x_2>y_2)}(A_2, B_2)$. If we assume that $x > y$ and $x_1 > y_1$, the true value is A_1 and the error can be computed as

$$\begin{aligned} |A_1 - \mathsf{L}_{(x>y)}(A, B)| &\leq |A - \mathsf{L}_{(x>y)}(A, B)| + |A - A_1| \\ &\leq 2 \cdot \max\{\epsilon/2, 2^{-\alpha}\}. \end{aligned}$$

In Algorithm 1, we remark that the inputs of L function are also L. To be more specific, in Equation 2, X and Y can be expressed at $\mathsf{L}_{(b_\alpha>c_\beta)}(P, Q)$ and $\mathsf{L}_{(b_\gamma>c_\delta)}(R, S)$, respectively, and with further decomposition, $\max_{i+j}(B, C)$ can be expressed completely by L as $\mathsf{L}(\mathsf{L}(\cdots, \cdots), \mathsf{L}(\cdots, \cdots))$. Let's classify them by the number of compositions and denote $\mathcal{S}_m$ as the set of functions in which the number of compositions is m. More precisely, $S_m = \{\mathsf{L}(A, B) | A, B \in S_{m-1}\}$. Denote ϵ_m as the maximum of error of functions in $\mathcal{S}_m$. We can bound the error of the composition of L through the following lemma.

Lemma 3 *$\epsilon_m \leq \epsilon_{m-1} + \frac{\epsilon}{2}$ or $\epsilon_m \leq 2^{-\alpha}$. Thus, $\epsilon_m \leq \max(2^{-\alpha}, \frac{(m+1)}{2}\epsilon)$.*

Proof. Choose $f \in \mathcal{S}_m, f = \mathsf{L}_{(x_i>y_i)}(A, B)$. If $|x_i - y_i| \geq \epsilon$, $\epsilon_m \leq (t_{(x_i>y_i)} - (x_i > y_i)) \leq 2^{-\alpha}$ by Lemma 1. If $|x_i - y_i| < \epsilon$,on the other hand, $|A - B| \leq |T_A - A| + |T_B - B| + |T_A - T_B| \leq \epsilon_{m-1} + |x_i - y_j| + \epsilon_{m-1}$ where T_A, T_B refers to true value of A, B, respectively. The second inequality holds by Lemma 2 and by the definition of ϵ_{m-1}. Therefore, $\epsilon_m \leq \frac{|A-B|}{2} \leq \epsilon_{m-1} + \frac{\epsilon}{2}$ holds.

Theorem 3 *Let $\mathcal{E}^k$ be the error of k-sorter. Then, it satisfies*

$$\mathcal{E}^k \leq \max(O(\log k) \cdot 2^{-\alpha}, O(\log^2 k) \cdot \epsilon)$$

regardless of inputs for sort.

Proof. From Figure 2, we can see that ϵ_i refers to the error occurred by computing $\{a^{(i)}\}$ from $\{a^{(i-1)}\}$. If we denote the true value of $a_l^{(i)}$ based on the assumption that $\{a^{(i-1)}\}$'s are true values as $T_{a_l^{(i)}}$, the error of k-sorter becomes $|T_{a_l^{(m)}} - a_l^{(0)}| \leq |T_{a_l^{(0)}} - a_l^{(0)}| + |T_{a_l^{(1)}} - a_l^{(1)}| + \cdots + |T_{a_l^{(m)}} - a_l^{(m)}|$ where $m = \lceil \log k \rceil$. It holds due to Equation 3. $|T_{a_l^{(i)}} - a_l^{(i)}| \leq \max(2^{-\alpha}, \frac{(i+1)}{2}\epsilon)$ for all l by Lemma 3. Therefore, $\mathcal{E} = \max\{|T_{a_l^{(i)}} - a_l^{(i)}|\} \leq \max(2^{-\alpha}, \frac{(i+1)}{2}\epsilon)$.

There exists j such that $\frac{j}{2}\epsilon \leq 2^{-\alpha} < \frac{j+1}{2}\epsilon$. If $j > m$, $\mathcal{E} \leq (m+1)2^{-\alpha} = O(\log k) \cdot 2^{-\alpha}$. If $j \leq m$. $\mathcal{E} \leq j \cdot 2^{-\alpha} + \sum_{i=j+1}^{m+1} \frac{i}{2}\epsilon < j \cdot \frac{j+1}{2}\epsilon + \sum_{i=j+1}^{m+1} \frac{i}{2}\epsilon \leq \frac{(m+1)^2}{2}\epsilon$. Therefore, $\mathcal{E} \leq \max((m+1)2^{-\alpha}, \frac{(m+1)^2}{2}\epsilon)$.

4 Proposed *k*-way Sorting Network for encrypted data using our *k*-Sorter

Our sorting algorithm introduced in section 3 has the advantage that it can be used even with approximate comparison results, but sorting n elements using only this algorithm needs $O(n^2)$ comparisons and $O(n^3)$ time complexity, which is still impractical. In this section, we propose a method of combining our algorithm with k-way sorting network to reduce the required number of comparisons and complexity for sorting the encrypted data. The k-way sorting network is an algorithm that sorts n elements using j-sorter for $j \leq k$, which sorts j elements in unit time, as a building block. By using k-way sorting network, we can reduce the total cost of sorting encrypted data compared to the previous works, which have only used 2-way sorting networks. In addition, we provide a formula that estimates which k-value is appropriate for a given implementation environment. Throughout this section, let n be the number of elements and $m = \lceil \log_k n \rceil$.

4.1 *k*-way Sorting Network

k-way sorting network algorithm for plaintext data was suggested from [40]. In this section, we introduce the outline of the algorithm and analysis on the number of stages briefly. For a prime k, the k-way sorting network is a recursive algorithm which repeats the process of forming a sorted array of length k^l by merging k number of sorted arrays of length k^{l-1} for $1 \leq l \leq m$. When an array of length k^l consists of k sorted arrays of length k^{l-1}, the algorithm can be understood by the following three steps. Firstly, the process begins with dividing each sorted array of length k^{l-1} into k sorted arrays of k^{l-2}, regrouping each i-th array for $1 \leq i \leq k$ into arrays of k^{l-1} and merging each group of them. Since each divided array of length k^{l-2} is also sorted, this merging algorithm can be done by recursive formula. The next step is to align newly sorted arrays of k^l into a matrix of $k \times k^l$ and to sort along the lines of slope i for $1 \leq i \leq \lfloor \frac{k}{2} \rfloor$. The final step is to apply $(k-1)$-sorter on the first $\frac{k-1}{2}$ elements of r-th row and the latter $\frac{k-1}{2}$ elements of $r+1$-th row for $1 \leq r \leq k^m - 1$. We categorize each stage as the step of applying sorters in parallel by *type* i for $0 \leq i \leq \lceil \frac{k}{2} \rceil$. Type $\lceil \frac{k}{2} \rceil$ refers the third step and denote a stage as type i if the slope of sorters is i ($0 \leq i \leq \lceil \frac{k}{2} \rceil$). We note that according to the algorithm, the size of sorters are inconsistent when sorting along diagonals of same slope since the size of sorters are smaller near the corner. For the purpose of efficient implementations, we will regard all the sorters of slope i to be j-sorter ($j = \lfloor \frac{k-1}{i} \rfloor + 1$), the maximum length among them.

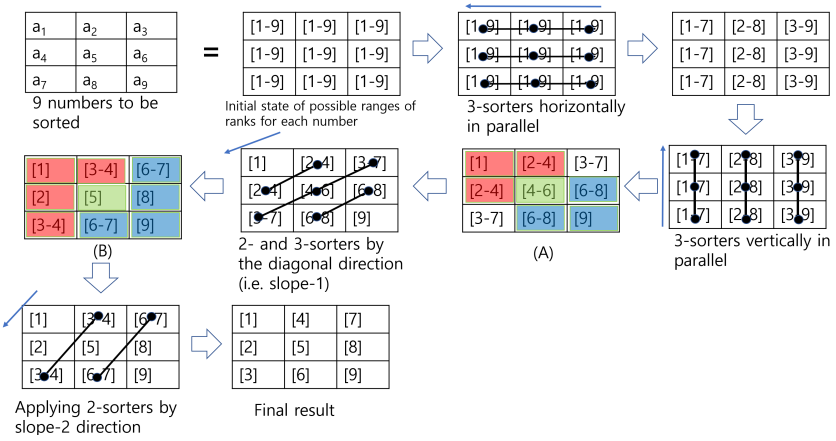

Fig. 3: An example of sorting 3^2 numbers using 3-way sorting network. Blue arrows refer to the direction of the sorting indicating to which the larger values are moving. The green cell has a greater value than the blue cells and a less one than the red.

Figure 3 exemplifies the procedure of sorting 3^2 numbers with 3 and 2-sorter. As seen in the figure, 3-sorters are applied in parallel horizontally then vertically. Then 2 and 3-sorters are applied along the line of slope 1 and 2, respectively. From the figure, it can be seen that the range of rank values, or the number of elements equal or less than the element, that each cell value can have decreases as sorting progresses. For example, in part (A) of the figure, the rank of the center cell (green) ranges from 4 to 6. However, in (B), after sorting in the slope 1 direction, the rank value of the corresponding position is fixed as 5. Finally, the sorting is terminated when all cell's ranks are fixed to a single one, respectively. Also, we can easily find that the depth of this sorting network is $4(= (\log_3 9)^2)$. We denote the detail in Section 4.2.

Analyzing the recursive algorithm explicitly, the number of stages of type 0, i.e. where k-sorter is applied in parallel is m since it emerges once per merging step. In addition, the number of stages of type i for $i \geq 1$ is $\frac{m(m-1)}{2}$ because when merging sorted arrays of k^i into an array of k^{i+1}, sorting along lines of slope i emerges i times and thus the total frequency is $1+\cdots+(m-1) = \frac{m(m-1)}{2}$. The final step of the algorithm also shows up $\frac{m(m-1)}{2}$ times for the same reason and therefore, the total number of stages of k-way sorting network for length k^m is $N_{m,k} = m + \lceil k/2 \rceil \cdot m(m-1)/2$.

4.2 FHE-friendly Algorithms for Sorting

This section describes exploitation of our k-sorter algorithm to construct a FHE-friendly k-way sorting network algorithm. To accelerate the running time of sorting, we propose a method of exploiting our k-sorter algorithm to imple-

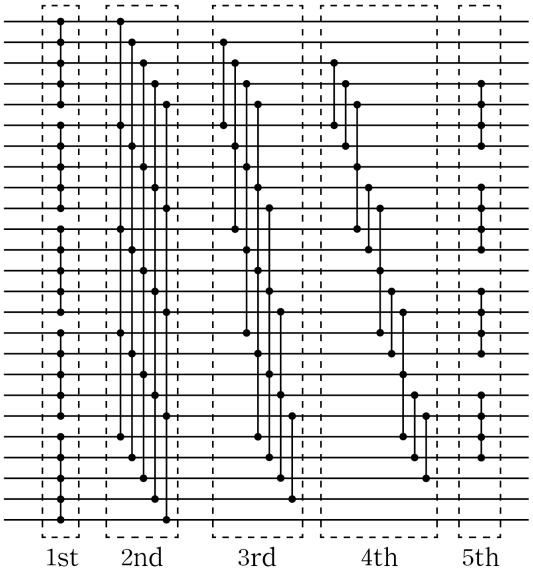

Fig. 4: 5-way sorting network for 5^2 numbers. Each line represents a k-sorter and the multiple dots on each line represent input numbers. The 5 arrows on the left depict the primary stage in which 5 neighboring numbers are sorted and the arrow on the right indicates the arrangement direction, from minimum to maximum.

ment k-way sorting network using SIMD operations and message rotations. Our implementation method can be applied to the general k, but for the sake of understanding, we describe the algorithm through an example in the case of $k = 5$.

In each stage, the algorithm consists of two main steps: first, computing approximate comparison in parallel that are necessary to run k-sorter are computed, and second, the k-sorter algorithms described in Section 3 using the comparison outputs. Recall that we may need to use k-sorters for multiple k values in the same stage. In the case of $k = 5$, there are stages of four types, from type 0 to 4. Therefore, an SIMD algorithm must be constructed for each type. However, if a method of running the 2-, 3-, 4-, and 5-sorter algorithms in parallel were devised, that method can be applied to stages regardless of types. Thus, in this section, we assume that j-sorters for $j \leq k$ are executed at the same time. Note that the k-way sorting network used here has two important features in each stage: first, the distance between the input messages for each k-sorter algorithm is always the same, and second, these inputs never overlap. For instance, consider the third stage for sorting of 5^2 elements as shown in Figure 4. In that stage, all 2-, 3-, 4-, and 5-sorters appear. We denote 25 elements by $x_1, x_2 \cdots, x_{25}$ in order, check that the 2-sorter inputs are (x_2, x_6) and (x_{20}, x_{24}), the 3-sorter inputs

are (x_3, x_7, x_{11}) and (x_{15}, x_{19}, x_{23}), the 4-sorter inputs are $(x_4, x_8, x_{12}, x_{16})$ and $(x_{10}, x_{14}, x_{18}, x_{22})$, and 5-sorter inputs are $(x_5, x_9, x_{13}, x_{17}, x_{21})$. These inputs are illustrated in Figure 5.

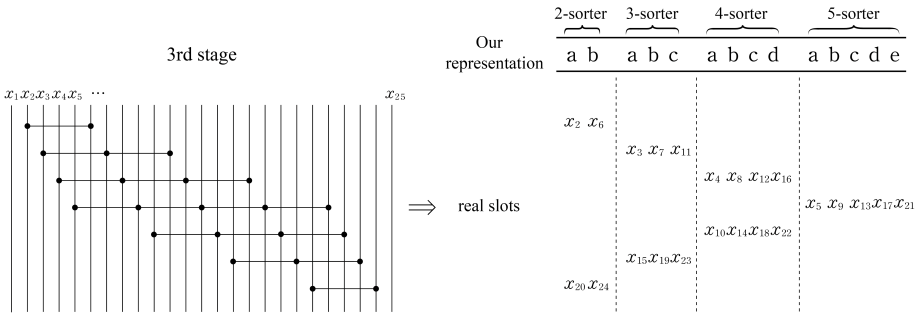

Fig. 5: Example of 25-sorter third stage. This stage consists of all 2-, 3-, 4-, and 5-sorters, and the distances between neighboring slots are all 4. None of the inputs overlap and, thus, parallel operation is possible.

Parallel Comparisons The first step of each stage of k-way sorting network for SIMD operations is the computation of the comparison function. Because the comparison function is applied to the same slots in each ciphertext, $\binom{k}{2}$ pairs of numbers can be compared simultaneously.

As each number is contained in a different slot and the comparison algorithm runs for each slot in SIMD-wise sense, we use rotation operations to match the slots to be compared. When the ciphertext is rotated, the slots that are not of interest also move. Therefore, we first multiply the ciphertext by a scalar vector, referred to as the *masking vector*, to extract our target slots. In this case, the masking vector consists of the value 1 for our target slot for extraction and 0 for the other slots.

In detail, the comparison part consists of two steps. First, the messages are rotated to match the slots for comparison and subtraction is performed. Second, the approximate comparison algorithm is implemented on the ciphertext, so that each slot contains the appropriate comparison output, SIMD-wisely. The comparison algorithm simply implements Equation 1; thus only the rotation step requires explanation here.

Focusing on slots of our interest only, we denote a vector with slots which are the inputs of 3-sorters only and omit others as

$$[a, b, c] := (\cdots, a, \cdots, b, \cdots, c, \cdots)$$

where a, b, and c are three inputs of 3-sorter. We also similarly define vectors with slots corresponding to the 4- and 5- sorter inputs as $[a, b, c, d]$ and $[a, b, c, d, e]$, respectively.

Let us begin with the 3-sorter case. We simply denote the ciphertext as $\mathsf{ct} = \mathsf{Enc}([a, b, c])$, and let the distance between the slot locations of a and b(or b and c) be ℓ. We wish to rotate ct by ℓ to obtain ct_1=$\mathsf{Enc}([b, c, a])$ and to implement a comparison algorithm on ct and ct_1. For the rotation, we use the two mask vectors $\mathsf{mask}_{3,\mathsf{left}} = [1, 1, 0]$ and $\mathsf{mask}_{3,\mathsf{right}} = [0, 0, 1]$. We compute $\mathsf{ct}_{3,\mathsf{left}} = \mathsf{ct} \cdot \mathsf{mask}_{3,\mathsf{left}}$ and $\mathsf{ct}_{3,\mathsf{right}} = \mathsf{ct} \cdot \mathsf{mask}_{3,\mathsf{right}}$, which are $\mathsf{Enc}([a, b, 0])$ and $\mathsf{Enc}([0, 0, c])$, respectively. This facilitates retrieval of the desired ciphertext $\mathsf{ct}_{\mathsf{rot}} = \mathsf{RightRotate}(\mathsf{ct}_{3,\mathsf{left}}, \ell) + \mathsf{LeftRotate}(\mathsf{ct}_{3,\mathsf{right}}, 2\ell) = \mathsf{Enc}([c, a, b])$, which is the desired output. Finally, we compute comparison algorithm $\textsc{comp}(\mathsf{ct}, \mathsf{ct}_{\mathsf{rot}})$. The output $\mathsf{comp}_1 = \mathsf{Enc}([(a > c), (b > a), (c > b)])$ is returned.

More generally, let the inputs of a k-sorter be $\mathsf{ct} = \mathsf{Enc}([a_1, \cdots, a_k])$. Multiply $\mathsf{mask}_{k,\mathsf{left}} = [1, \cdots, 1, 0]$ and $\mathsf{mask}_{k,\mathsf{right}} = [0, \cdots, 0, 1]$ to ct to obtain $\mathsf{ct}_{k,\mathsf{left}}$ and $\mathsf{ct}_{3,\mathsf{right}}$, respectively. Then, we are able to compute $\mathsf{ct}_1 = \mathsf{RightRotate}(\mathsf{ct}_{k,\mathsf{left}}, \ell) + \mathsf{LeftRotate}(\mathsf{ct}_{k,\mathsf{right}}, (k-1)\ell)$, which is the encryption of $\{a_{i+1} - a_i\}$ (regarding $a_{k+i} = a_i$). In a similar way, $\mathsf{ct}_1, \cdots, \mathsf{ct}_{\lfloor \frac{k}{2} \rfloor}$ where $\mathsf{ct}_j = \{a_{i+j} - a_i\}$ can be computed. The final step is to run comparison algorithm for ct_i's to obtain the encrypted comparison results $\mathsf{comp}_1, \cdots, \mathsf{comp}_{\lfloor \frac{k}{2} \rfloor}$.

In the case that the number of slots s is large enough, then we can store 2 or more ciphertexts into one ciphertext and thus reduce the number of computations for comparison. For instance, if $n < \frac{s}{2}$, then it means that ct_i contains n nonzero elements and the rest of the slots are filled with zeros, as $\mathsf{ct}_i = \mathsf{Enc}(m_i||0)$ for $m_i \in \mathbb{C}^{s/2}$ (here $||$ means the concatenation of two vectors). Then, we rotate ciphertext ct_{2i} and add to ct_{2i-1} to get $\mathsf{ct}'_i = \mathsf{Enc}(m_{2i-1}||m_{2i})$ containing two plain vectors. Now we run comparison function for ct_{2i} for $1 \le i \le \frac{1}{2}\lfloor \frac{k}{2} \rfloor$ to get comp'_i, which reduces the number of comparison operations by half. Finally, we multiply masking vector $(1||0), (0||1)$ to comp'_i and by rotating the latter one, we get comp_{2i-1} and comp_{2i}. Generalizing this step, we can reduce the number of comparisons by concatenating n_0 number of ciphertexts when $n < \frac{s}{n_0}$, which maximizes the efficiency of parallel computing.

Slot Alignment In the next step, the ciphertext and comparison results, which are positioned in the $\mathsf{ct}, \mathsf{comp}_1$, and comp_2 slots, should be collected into a single slot through rotation. In 5-sorter, for instance, let $\mathsf{ct} = \mathsf{Enc}([a, b, c, d, e])$, and let comp_1 and comp_2 be the encrypted comparison results. For the iMax computation, ten comparison results distributed in five comp_1 and comp_2 slots must be gathered and computed in one slot. Precisely, we wish to obtain $\mathsf{ct}_1 = \mathsf{Enc}([a, 0, 0, 0, 0]), \cdots, \mathsf{ct}_5 = \mathsf{Enc}([e, 0, 0, 0, 0])$ and $\mathsf{ct}_{a>b} = \mathsf{Enc}([(a > b), 0, 0, 0, 0])$, $\cdots$, $\mathsf{ct}_{d>e} = \mathsf{Enc}([(d > e), 0, 0, 0, 0])$. To achieve this, we multiply each ciphertext by a masking vector for each slot, and then rotate the ciphertext so that the desired slots are located in the first slot. For instance, to obtain $\mathsf{ct}_3 = \mathsf{Enc}([c, 0, 0, 0, 0])$, we perform the masking vector multiplication $\mathsf{ct}'_3 = \mathsf{ct} \cdot [0, 0, 1, 0, 0] = \mathsf{Enc}([0, 0, c, 0, 0])$, and then rotate 2ℓ to the left. The mandatory input values for iMax computation are then stored separately in the first slot of the 5 sorter. Similarly we match the slots for each comparison results by multiplying the matching masking vector and performing rotation. By running the 5-sorter algorithm, we finally obtain the outputs in the first slots of $\mathsf{ct}_1, \cdots, \mathsf{ct}_5$

and $\mathsf{ct}_{a>b}, \cdots, \mathsf{ct}_{d>e}$. Similarly, for the k-sorter algorithm, we obtain k and $\binom{k}{2}$ ciphertexts that contains each element and their comparison, respectively.

Running k-sorter After slot alignment, now we can compute k-sorter through SIMD operation. The important point here is that in the case of slots where we actually need to compute j-sorter for j less than k, we have to assume that there are zeros in the remaining $k - j$ slots that we don't consider or else, the inputs of 2-sorter can be pushed back while running k-sorter and overlap with inputs of other sorters. This can be solved by adding a dummy comparison to the ciphertext corresponding to the comparison. Since our k-sorter algorithm sorts only with the comparison value regardless of the given elements, if we add 1 to the empty comparison slot that is not used in j-sorter, it works as the slots we don't consider is smaller than the j elements we want to sort, so the j elements will sorted in the first j ciphertexts. To explain through an example, suppose we want to run 2- and 3-sorters in one message vector as $[(a_2, b_2), (a_3, b_3, c_3)]$. Here, (a_2, b_2) and (a_3, b_3, c_3) are the 2- and 3-sorter inputs, respectively. After comparison and slot alignment , we obtain the following ciphertexts:

$$\begin{aligned}
\mathsf{ct}_1 &= \mathsf{Enc}([(a_2, 0), (a_3, 0, 0)] \\
\mathsf{ct}_2 &= \mathsf{Enc}([(b_2, 0), (b_3, 0, 0)] \\
\mathsf{ct}_3 &= \mathsf{Enc}([(0, 0), (c_3, 0, 0)] \\
\mathsf{ct}_{(a>b)} &= \mathsf{Enc}([((a_2 > b_2), 0), ((a_3 > b_3), 0, 0)]) \\
\mathsf{ct}_{(a>c)} &= \mathsf{Enc}([(1, 0), ((a_3 > c_3), 0, 0)]) \\
\mathsf{ct}_{(b>c)} &= \mathsf{Enc}([(1, 0), ((b_3 > c_3), 0, 0)])
\end{aligned}$$

Here we add 1 to the first slots for 2 sorter in $\mathsf{ct}_{(a>c)}$ and $\mathsf{ct}_{(b>c)}$, which are originally not used for 2-sorter. Note that the process works as if the hidden third input c_2 in ct_3 is smaller than a_2 and b_2, so parallel computation of the 3-sorter works well for sorting of two items (this case is treated as if $(a_2, b_2, 0)$ is being sorted).

After all above steps, we operate the slot alignment reversely (only for outputs for k-sorter) to get the ciphertext that contains the array that sorting operation for one stage is completed. We illustrate the example of whole method in Figure 6.

4.3 Time Cost Estimation of k-way Sorting Network

Our goal in this section is to estimate the ideal time of sorting n elements through k-way sorting network for each k depending on the implementation environment. Since k-way sorting network operates only on inputs of size power of k, we append 0's to the given array to make the length to k^m which is the smallest power of k larger than n.

First, let's define the terms to be used throughout this subsection: we denote the depth and multiplication complexity of the j-sorter and a comparison operation by $(\mathcal{D}_{\mathsf{sorter}(j)}, \mathcal{C}_{\mathsf{sorter}(j)})$ and $(\mathcal{D}_{\mathsf{comp}}, \mathcal{C}_{\mathsf{comp}})$, respectively. From Section 4.1, we can observe that k-way sorting network is the recursive algorithm of

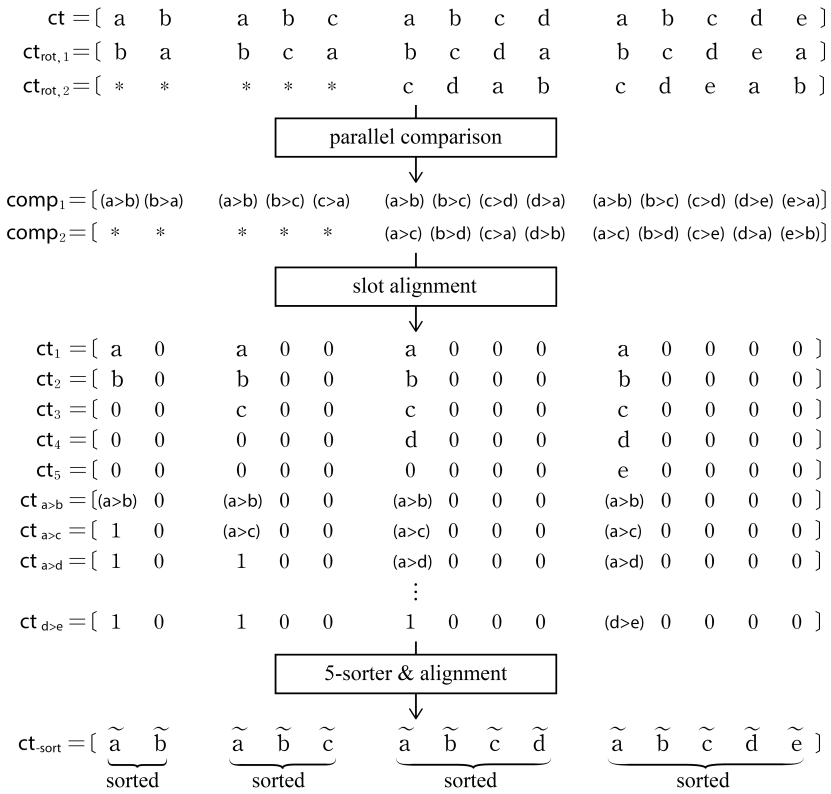

Fig. 6: Illustration of proposed parallel algorithm for the stage with slope 1 in 5-way sorting network. Assuming that all 2- to 5-sorters are run simultaneously, this figure shows the values of each slot in each parallel algorithm step.

forming a sorted array of length k^l by merging sorted arrays of length k^{l-1}. Thus, each stage in k-way sorting network can be classified according to i, the stage type and k^l, the size of merging. Using these variables, we denote (i, l) as the characteristic of a stage. If the characteristic of stage is (i, l), let j_i be the maximum size of the sorter of this stage, $S_{(i,l)}$ be the number of j_i sorters being implemented in the stage.

Prior to the examination, we make two assumptions to simplify the computation. First, since the cost of multiplying masking vector is comparably smaller than other operations, we ignore the multiplication cost and the depth for masking vectors. Second, we assume that the sorters in a stage of type i are all j_i-sorters as the implementation is following our method in Section 4.2.

To compute j_i and $S_{(i,l)}$ based on above assumptions, we consider each cases for slope i. If $i = 0$, the slope of sorters is 0, so $S_{(i,l)}$ is equal to $\frac{k^m}{k} = k^{m-1}$. If $i = \lceil \frac{k}{2} \rceil$, according to Section 3, $j_i = k - 1$. When we merge k number of sorted arrays of length k^{l-1}, $(k-1)$-sorter is implemented $k^{l-1} - 1$ times in total. Since

such merging occurs k^{m-l} times, we get $S_{(i,l)} = k^{m-l} \cdot (k^{l-1} - 1) = k^{m-1} - k^{m-l}$. For other i's, j_i is $\lfloor \frac{k-1}{i} \rfloor + 1$ since the slope is i and $0 \leq y < k$. To count $S_{(i,l)}$, we regard $k^m = k \cdot k^{m-1}$ elements as a matrix $A = (a_{x,y})$ where $0 \leq x < k^{m-1}$ and $0 \leq y < k$. Then, instead of counting $S_{(i,l)}$, we can count the number of starting points of each j_i-sorters. If $a_{x,y} \in A$ is a starting entry of a j_i-sorter, it should satisfy two conditions: $a_{x-i,y+1} \in A$ and $a_{x+i,y-1} \notin A$. We can count the number of (x, y) satisfying the conditions to be $i \cdot k^{l-1} + k - 3i$. Such feature also occurs k^{m-l} times, the number of sorters is $k^{m-l} \cdot (i \cdot k^{l-1} + k - 3i)$. In short, $S_{(i,l)}$ and j_i are computed as

$$S_{(i,l)} = \begin{cases} k^{m-1} & \text{if } i = 0 \\ k^{m-1} - k^{m-l} & \text{if } i = \lceil \frac{k}{2} \rceil \\ i \cdot k^{m-1} + k^{m-l} \cdot (k - 3i) & \text{otherwise} \end{cases},$$

$$j_i = \begin{cases} k & \text{if } i = 0 \\ k-1 & \text{if } i = \lceil \frac{k}{2} \rceil \text{ and } k > 2 \\ \lfloor \frac{k-1}{i} \rfloor + 1 & \text{otherwise} \end{cases}.$$

Next we examine the expected time by counting the number of multiplications and bootstrappings. Assuming that the characteristic of stage is (i, l), we can count them by consider the number of required ciphertexts for packing in comparison and swap step, denote by $N^{\mathsf{comp}}_{(i,l)}$ and $N^{\mathsf{swap}}_{(i,l)}$, respectively. In each stage, assuming that the characteristic of stage is (i, l), let $d_{\mathsf{comp},(i,l)}$ and $d_{\mathsf{sorter},(i,l)}$ be the required depth for the comparison and j_i-sorter steps in the k-sorting network, respectively. Since our algorithm computes all sorters in a stage with the j_i-sorter, the number of comparisons in the stage is $\binom{j_i}{2} \cdot S_{(i,l)}$. Since we have s number of slots to compute comparisons in one ciphertext, the required number of ciphertexts is $N^{\mathsf{comp}}_{(i,l)} = \lceil \binom{j_i}{2} \cdot S_{(i,l)}/s \rceil$. Also, in the swap step, we uses j slots for computing j-sorter, the number of slots for computing swap step is at most $j_i \cdot S_{(i,l)}$. To simplify our estimation, we choose $N^{\mathsf{swap}}_{(i,l)}$ by its upper bound, so that $N^{\mathsf{swap}}_{(i,l)} = \lceil j_i \cdot S_{(i,l)}/s \rceil$ by packing s elements in one ciphertexts.

Using above number of ciphertexts, the number of multiplications and bootstrappings are followed directly. First, the number of multiplications $\mathcal{M}_{(i,l)}$ is

$$\mathcal{M}_{(i,l)} = N^{\mathsf{comp}}_{(i,l)} \cdot \mathcal{C}_{\mathsf{comp}} + N^{\mathsf{swap}}_{(i,l)} \cdot \mathcal{C}_{\mathsf{sorter}(j_i)}.$$

Since the number of boots depends much more on the implementation method than the number of multiplications, it is difficult to estimate the exact number. Therefore, we first count the depth required for the entire algorithm, then divide it by d, the restored depth per one bootstrapping, to find the expected number instead. At this time, by multiplying the depth required at each step by the number of ciphertexts required for the operation at that step, the expected number of bootstrappings is made close to the exact value. Also, we ignore the required depth for multiplying by masking vectors, this is because the required number of depths for the masking vector is very small compared to other processes, and when implemented using the CKKS scheme, the consumed depth can

be further reduced by choosing the scaling factor for the masking vector smaller than that used in the ciphertext. Using the above method for expected number of bootstrapping $\mathcal{B}_{(i,l)}$, the formula for computing $\mathcal{B}_{(i,l)}$ is almost as same as one for $\mathcal{M}_{(i,l)}$ with additional division by d. In short, $\mathcal{B}_{(i,l)}$ is computed as

$$\mathcal{B}_{(i,l)} = (N_{(i,l)}^{\mathsf{comp}} \cdot \mathcal{D}_{\mathsf{comp}} + N_{(i,l)}^{\mathsf{swap}} \cdot \mathcal{D}_{\mathsf{sorter}(j_i)})/\mathsf{d}.$$

The total time cost can be estimated by using the ratio between the time cost of multiplication and bootstrapping, denoted by γ. In conclusion, the total estimated time is the sum of $\mathcal{M}_{(i,l)} + \gamma \cdot \mathcal{B}_{(i,l)}$ for every stages.

Remark 1. Our estimation shows the optimal implementation in theory and it is useful to see the tendency of expected time cost from various k and the implementation environment. However, the estimated results do not directly refer to the real time cost. The crucial reason of the expected difference is that the maximum number of depths cannot always be obtained from each bootstrapping. To compute the k-sorter algorithm, we create several ciphertexts from a given ciphertext to place the values to be sorted and their comparisons results in the same slot. At this time, if the remaining depth of the initial ciphertext was not enough for the k-sorter algorithm, bootstrapping operation should be applied to all ciphertexts in the process of the k-sorter algorithm. Instead, it is practically more efficient to bootstrap the initial ciphertext and then multiplying masking vectors to process the k-sorter algorithm, which significantly reduces the number of bootstrapping. In addition, our estimation ignores the cost of multiplication and depth for masking vectors, which is relatively small compared to the cost of comparisons, but it could increase as k becomes larger. In short, we notice that our estimation does not show the exact difference of the time cost for various k in real time, and recommend to use it to estimate the cost difference in the *ideal* implementation.

5 Experimental Results

5.1 Parameter Selection

For HEaaN, we fix the dimension $N = 2^{17}$, $\Delta = 2^{40}$, and hamming weight $h = 128$ and the secret key is chosen from the ring with ternary secret distribution, i.e. all nonzero coefficients of secret key are ± 1. Then we can set the largest modulus Q to 2^{2900} by considering the state-of-the-art attacks for ring LWE encryption schemes for sparse ternary secret [18]. To achieve larger modulus for ciphertexts, we need additional parameter dnum, which refers the number of evaluation keys used for key-switching algorithm in multiplication or rotation of HEaaN. Using large dnum, the number of evaluation keys increased, but it enables to set larger initial ciphertext modulus Q_0, which can be computed as $Q_0 = Q \cdot \mathsf{dnum}/(\mathsf{dnum}+1)$. We set $\mathsf{dnum} = 3$ and so that the ciphertext modulus Q_0 is 2^{2175}. We refer [32] for the detail of relationship between dnum and security level.

For the approximate comparisons, we set d_f, d_g from fixed error size. Since the elements are natural numbers less than n, the minimum difference between different numbers is $\frac{1}{n}$ if the numbers are normalized. If $\mathcal{E}^k$ is bounded by $\frac{1}{2n}$, different numbers can be well sorted by k-sorter as $\frac{1}{n} = \frac{1}{2n} + \frac{1}{2n} \geq \mathcal{E}^k + \mathcal{E}^k$. According to Theorem 3, $\mathcal{E}^k \leq \frac{1}{2n}$ if $\max((\log k+1)2^{-\alpha}, \frac{(\log k+1)^2}{2}\epsilon) \leq \frac{1}{2n}$ holds. Thus, ϵ and α can be determined and so does d_f and d_g through Theorem 2.

5.2 Time Cost Estimation

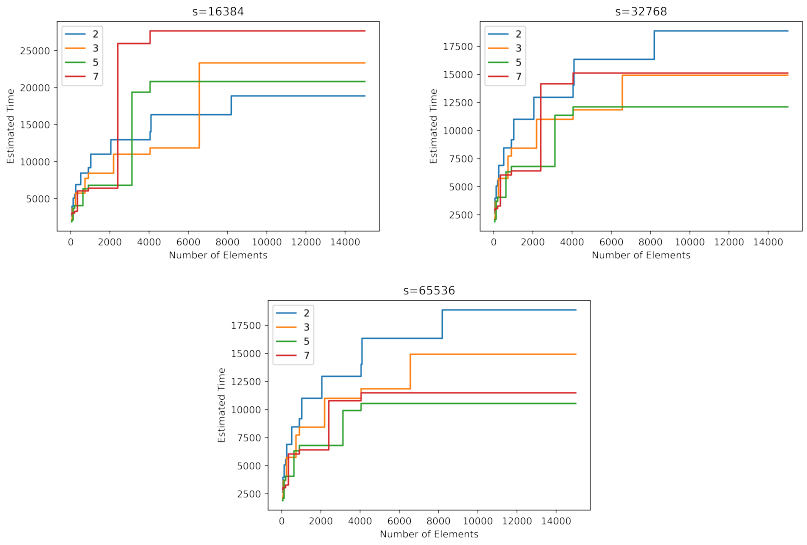

Fig. 7: Graph of time estimation for sorting with k-way sorting network for each $k = 2, 3, 5, 7$ in different number of SIMD operation(or number of slots) environments, especially $s = 2^{14}$, 2^{15}, and 2^{16}. The parameters for approximate comparison are chosen to satisfy desired error bound as $1/2n$ for k-sorter in each stage.

We show the result of time cost estimation from our formula in Section 4.3 for k = 2,3,5,7 and $s = 2^{14}$, 2^{15} and 2^{16}. Based on the experiment result in our implementation environment, we have set $\gamma = 84.67$ as the multiplication and bootstrapping takes 66.3 and 0.783 seconds on average and we have set $\mathsf{d} = 27$ as we are able to obtain up to 27 modulus through one bootstrapping. Figure 7 displays the time required for sorting $n \leq 2^{15}$ numbers. From the graphs in Figure 7, we can observe that the decrease in time is greater as s increases if k is large: If $k = 2$, there is almost no change in the time for different s. On the other hand, if $k = 5$ or 7, the time for sorting decreases as s gets larger. Such relation

k	size	(d_f, d_g)	Time (min)	Errors (log) Max	Avg	Memory (GB)
2	512	(3, 6)	150.57	-15.97	-17.37	23.39
	1024	(4, 7)	186.00	-14.61	-16.93	24.20
	2048	(4, 7)	223.25	-14.92	-16.78	24.75
	4096	(4, 8)	276.53	-13.60	-16.06	24.93
	8192	(4, 8)	329.13	-13.00	-15.79	25.26
	16384	(4, 8)	384.43	-11.62	-14.82	25.29
3	729	(4, 7)	129.38	-14.48	-16.88	26.85
	2187	(4, 8)	182.11	-14.46	-16.31	28.01
	6561	(4, 9)	263.73	-12.39	-14.97	28.39
	19683	(4, 10)	314.28	-10.95	-13.78	28.35
5	625	(4, 8)	114.32	-13.29	-16.82	37.84
	3125	(4, 9)	192.83	-13.53	-16.93	46.11
	15625	(4, 10)	294.81	-13.66	-16.54	55.40

Table 1: Performance of sorting over encrypted data with k-way sorting network for various k, fixing the number of slots $s = 2^{15}$.

between k and s exists because the number of ciphertexts used in comparison step is $\lceil \binom{k}{2} \cdot S_{(0,0)}/s \rceil = \lceil (k-1)/2 \cdot k^m/s \rceil$. Therefore, more comparison operations can be parallelized when s exceeds $2 \cdot 5^m$ for $k = 5$ or $3 \cdot 7^m$ for $k = 7$.

5.3 Performance of k-way Sorting Networks over Encrypted Data

For the purpose of verifying our estimation, we have implemented our algorithm using CKKS FHE scheme with an improved bootstrapping technique [10, 12]. Our artifact is publicly available at [33], and the experiment was performed in Ubuntu 18.04.2 LTS environment, run on an AMD Ryzen(TM) 9 3950X CPU with 128GB memory and 32 threads.

Table 1 lists the performance results of the proposed sorting algorithm for $k = 2, 3, 5$ and various parameters, visualized in Figure 8. The selected data for the experiment were random real numbers between 0 and 1. The time was measured as the total run time of sorting algorithm over encrypted data only, with the encryption or decryption time being omitted. After the proposed sorter was executed over encrypted data, the original data were sorted in plain and compared with decrypted data in terms of the average of absolute values of error.

In the result, there are cases where the maximum error between the encrypted sort result and the true value exceeds the error bound we wanted. This is because errors are basically included in the operations in the CKKS scheme, which is based on approximate computation. Since we concretely analyzed the error of

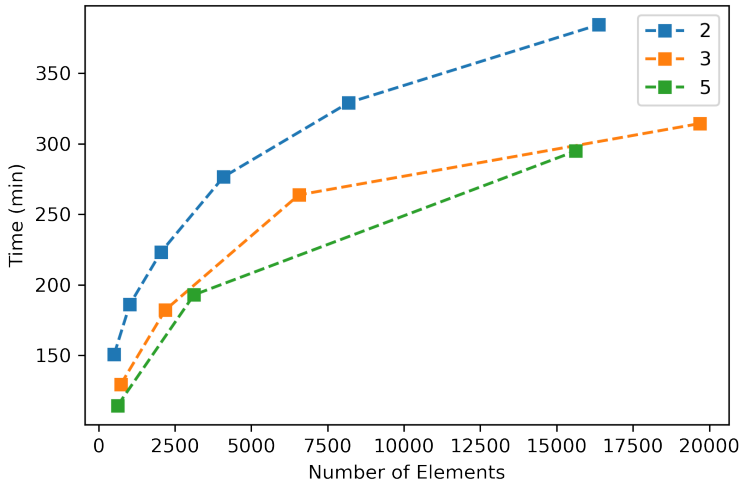

Fig. 8: The illustration of Time and data size from Table 1 with the number of slots $s = 2^{15}$. Recall that the number of stage for sorting n elements using k-sorting network is $O(k \cdot \log_k^2 n)$.

our k-sorter algorithm, as the depth of the algorithm increases, errors from the FHE operation may cause to increase the maximal error after sorting.

Compared to our estimation in Section 5.2, It can be seen that our implementation result is almost as same as the estimation result for $s = 2^{15}$. In particular, takes 294.81 minutes to sort $5^6 = 15625$ elements using 5-way, which is reduced by 23.3% compared to the time cost for sorting $2^{14} = 16384$ which are 384.43 minutes.

References

1. A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill, "Order-preserving symmetric encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009, pp. 224–241.
2. J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *IMA International Conference on Cryptography and Coding*. Springer, 2013, pp. 45–64.
3. C. Boura, N. Gama, and M. Georgieva, "Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning," Cryptology ePrint Archive, Report 2018/758, Tech. Rep., 2018.
4. Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.
5. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, p. 13, 2014.

6. M. Brenner and H. Perl, "Sm: libscarab software library."
7. G. S. Çetin, Y. Doröz, B. Sunar, and E. Savaş, "Depth optimized efficient homomorphic sorting," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2015, pp. 61–80.
8. A. Chatterjee, M. Kaushal, and I. Sengupta, "Accelerating sorting of fully homomorphic encrypted data," in *International Conference on Cryptology in India*, vol. 13, no. 3. Springer, 2013, pp. 262–273.
9. A. Chatterjee and I. SenGupta, "Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective?" *IEEE Transactions on Services Computing*, vol. 13, no. 5, pp. 545–558, 2020.
10. H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Report 2018/1043, 2018. [Online]. Available: https://eprint.iacr.org/2018/1043
11. J. H. Cheon, K. Han, S. Hong, H. J. Kim, J. Kim, S. Kim, H. Seo, H. Shim, and Y. Song, "Toward a secure drone system: Flying with real-time homomorphic authenticated encryption," *IEEE Access*, vol. 6, pp. 24 325–24 339, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2819189
12. J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, 2018, pp. 360–384. [Online]. Available: https://doi.org/10.1007/978-3-319-78381-9_14
13. J. H. Cheon, S. Hong, and D. Kim, "Remark on the security of ckks scheme in practice," *IACR Cryptol. ePrint Arch*, vol. 2020, p. 1581, 2020.
14. J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.
15. J. H. Cheon, D. Kim, and D. Kim, "Efficient homomorphic comparison methods with optimal complexity," in *Advances in Cryptology – ASIACRYPT 2020*. Cham: Springer International Publishing, 2020, pp. 221–256.
16. J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2019, pp. 415–445.
17. J. H. Cheon, D. Kim, Y. Kim, and Y. Song, "Ensemble method for privacy-preserving logistic regression based on homomorphic encryption," *IEEE Access*, vol. 6, pp. 46 938–46 948, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2866697
18. J. H. Cheon, Y. Son, and D. Yhee, "Practical fhe parameters against lattice attacks," 2021.
19. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.
20. ——, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.
21. A. Costache and N. P. Smart, "Which ring based somewhat homomorphic encryption scheme is best?" in *Cryptographers' Track at the RSA Conference*. Springer, 2016, pp. 325–340.

22. Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic aes evaluation using ntru." *IACR Cryptology ePrint Archive*, vol. 2014, p. 39, 2014.
23. L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.
24. N. Emmadi, P. Gauravaram, H. Narumanchi, and H. Syed, "Updates on sorting of fully homomorphic encrypted data," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE, 2015, pp. 19–24.
25. J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.
26. C. Gentry and D. Boneh, *A fully homomorphic encryption scheme*. Stanford University Stanford, 2009, vol. 20, no. 09.
27. C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Annual Cryptology Conference*. Springer, 2012, pp. 850–867.
28. C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.
29. R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*, 2016, pp. 201–210.
30. S. Halevi and V. Shoup, "Helib-an implementation of homomorphic encryption," *Cryptology ePrint Archive, Report 2014/039*, 2014.
31. K. Han, S. Hong, J. H. Cheon, and D. Park, "Efficient logistic regression on large encrypted data." *IACR Cryptology ePrint Archive*, vol. 2018, p. 662, 2018.
32. K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.
33. S. Hong, "k_way_sort_enc." [Online]. Available: https://hub.docker.com/repository/docker/swanhong/k_way_sort_enc
34. A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2018, p. 254, 2018. [Online]. Available: http://eprint.iacr.org/2018/254
35. M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, "Secure logistic regression based on homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2018, p. 74, 2018. [Online]. Available: http://eprint.iacr.org/2018/074
36. P. Kim, Y. Lee, and H. Yoon, "Sorting method for fully homomorphic encrypted data using the cryptographic single-instruction multiple-data operation," *IEICE Transactions on Communications*, vol. 99, no. 5, pp. 1070–1086, 2016.
37. B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," *IACR Cryptol. ePrint Arch*, vol. 2020, p. 1533, 2020.
38. W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption."
39. S. Park, M. Kim, S. Seo, S. Hong, K. Han, K. Lee, J. H. Cheon, and S. Kim, "A secure snp panel scheme using homomorphically encrypted k-mers without snp calling on the user side," *BMC genomics*, vol. 20, no. 2, p. 188, 2019.
40. F. Shi, Z. Yan, and M. Wagh, "An enhanced multiway sorting network based on n-sorters," in *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2014, pp. 60–64.

41. N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *International Workshop on Public Key Cryptography*. Springer, 2010, pp. 420–443.
42. M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 24–43.