Received August 15, 2021, accepted September 1, 2021, date of publication September 7, 2021, date of current version September 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3110868

Analysis of Modified Shell Sort for Fully Homomorphic Encryption

JOON-WOO LEE[1], YOUNG-SIK KIM[2], (Member, IEEE), AND JONG-SEON NO[1], (Fellow, IEEE)
[1]Department of Electrical and Computer Engineering, INMC, Seoul National University, Seoul 08826, Republic of Korea
[2]Department of Information and Communication Engineering, Chosun University, Gwangju 61452, Republic of Korea

Corresponding author: Young-Sik Kim (iamyskim@chosun.ac.kr)

This work was supported by the Samsung Research Funding and Incubation Center of Samsung Electronics under Project SRFC-IT1801-08.

ABSTRACT The Shell sort algorithm is one of the most practically effective in-place sorting algorithms. However, it is difficult to execute this algorithm with its intended running time complexity on data encrypted using fully homomorphic encryption (FHE), because the insertion sort in Shell sort has to be performed by considering the worst-case input data. In this paper, in order for the sorting algorithm to be used on the FHE data, we modify the Shell sort with an additional parameter α, allowing exponentially small sorting failure probability. For a gap sequence of powers of two, the modified Shell sort with input array length n is found to have the trade-off between the running time complexity of $O(n^{3/2}\sqrt{\alpha + \log\log n})$ and the sorting failure probability of $2^{-\alpha}$. Its running time complexity is close to the intended running time complexity of $O(n^{3/2})$ and the sorting failure probability can be made very low with slightly increased running time. Further, the near-optimal window length of the modified Shell sort is also derived via convex optimization. The proposed analysis of the modified Shell sort is numerically confirmed by using randomly generated arrays. For the practical aspect, our modification can be applied to any gap sequence, and we show that Ciura's gap sequence, which is known to have good practical performance, is also practically effective when our modified Shell sort is applied. We compare our modified Shell sort with other sorting algorithms with the FHE over the torus (TFHE) library, and it is shown that this modified Shell sort has the best performance in running time among in-place sorting algorithms on homomorphic encryption scheme.

INDEX TERMS Fully homomorphic encryption (FHE), fully homomorphic encryption over the torus (TFHE), insertion sort, Shell sort, sorting failure probability.

I. INTRODUCTION

Fully homomorphic encryption (FHE) is an encryption scheme that provides encrypted data with an evaluation algorithm, which enables addition or multiplication of plaintext without decryption [1], [2]. The FHE enables specific operations to be performed on encrypted information without leaking any clue to the plaintext. The notion of the FHE was suggested by Rivest *et al.* [1]. Although several cryptography researchers had attempted to construct the FHE scheme because of its effectiveness with respect to operations in cloud systems, no one had been able to successfully construct it until 2009, when Gentry succeeded in developing an FHE scheme using an ideal lattice [2]. Several researchers suggested different types of the FHE algorithms in series using the bootstrapping technique in Gentry's scheme and optimized the FHE schemes [3]–[7]. Recently, FHE schemes have been significantly improved in regard to various performance criteria [8]–[12], which makes this scheme practically applicable. Further, the efficient implementations of the FHE schemes have been proposed actively now [13]–[17].

Further, the algorithms used on the FHE data are expected to demonstrate the oblivious property, i.e., providing the most appropriate outputs without knowing any information about the input. In other words, the behavior of an oblivious algorithm does not depend on the input data. If it depends on the input, it implies the leakage of input information. The oblivious property of an algorithm is essential for the FHE schemes to ensure privacy.

When processing large amounts of ciphertexts in cloud systems, it is frequently required to process the sorted data rather than unaligned data. Thus, one of the most essential

The associate editor coordinating the review of this manuscript and approving it for publication was Cong Pu.

operations on the FHE data is the sorting algorithm, which is generally used as a subroutine algorithm of many algorithms. However, most sorting algorithms are not suitable for the FHE data. For example, because the quick sort algorithm, one of the most popularly used sorting algorithms, is not oblivious, it cannot be used on the FHE data. Although numerous studies have been conducted to render the quick sort algorithm oblivious, its running time complexity becomes $O(n^2)$, where n is the input array length. Its actual running time is even longer than that of the bubble sort, which is considered to have the longest running time among all the known sorting algorithms. Therefore, modifying conventional sorting algorithms to make them suitable for the FHE data is necessary. Several studies have been conducted for this purpose [18]–[20].

A. MOTIVATION FOR SHELL SORT

Since the oblivious sorting algorithm can be applied for encrypted data with the FHE, Emmadi *et al.* [20] compared several oblivious algorithms for sorting the FHE data. We can divide sorting algorithms for the FHE data into two classes of oblivious sorting algorithms: in-place algorithm and recursive algorithm. The bubble sort and insertion sort are basic in-place oblivious algorithms, and the bitonic sort and odd-even merge sort are recursive oblivious algorithms. The recursive oblivious algorithms are much better than the in-place oblivious algorithms in the aspect of both the asymptotic performance and practical performance.

However, the recursive algorithms may have inefficiency in some cases. Since many function calls are caused in the recursive algorithms and the amount of memory for the ciphertext array is quite big, the total transmission of data in the memory bus must be somewhat large. When the bandwidth of the memory bus is restricted, this transmission time can be a bottleneck for sorting with encrypted data. This situation can occur in lightweight IoT devices, whose memory or bandwidth cannot be large enough. For this reason, it is desirable to devise an efficient in-place sorting algorithm for the FHE data. The Shell sort [21], [22], which is one of the oldest sorting algorithms, is the generalized version of the insertion sort. The Shell sort algorithm is an in-place algorithm, which is fast and easy to implement, and thus, many systems use it as a sorting algorithm.

B. MAIN RESEARCH PROBLEM AND PREVIOUS WORKS

It is known that Shell sort uses insertion sort as a subroutine algorithm, and insertion sort can be performed on the FHE data [18], [23]. However, the Shell sort should be modified to be used in the FHE setting. If we do not allow any error in sorting, then insertion sort is expected to be quite conservative, i.e., the number of operations for sorting must be set for the worst case, because the insertion sort algorithm in the FHE setting is an oblivious algorithm. Thus, if we use insertion sort in the Shell sort, the running time complexity of Shell sort in the FHE setting must be $O(n^2)$, which makes the use of Shell sort ineffective. Therefore, it is important to devise a sorting algorithm that is better than the Shell sort on the FHE data in terms of running time complexity.

Goodrich [24] suggested an asymptotically optimal randomized oblivious Shell sort. He proved that its running time complexity is $O(n \log n)$ and sorting failure probability (SFP) is $O(1/n^b)$ for some constant $b \geq 1$, where n is the length of an array. While it is pretty efficient in the asymptotic sense, there are two points to be considered. First, the analytically induced SFP is an inverse polynomial of the length of an array. When we sort the array having a small length with this algorithm, the induced SFP may not be the practically allowable value. Further, the inverse polynomial SFP is not considered a small probability in the asymptotic sense. Since users are often conservative with the SFP in the sorting, the exponentially decaying SFP is more desirable. Second, it can be inefficient for the array of the small length. For lowering the SFP, many processes are required in the randomized Shell sort. This causes rather large additional operations for an array with a small length. Considering that the running time of the homomorphic operations in the FHE is quite large, sorting a large number of encrypted data is not a practical situation yet. Thus, it can be desirable to devise an oblivious variant of Shell sort which is practically efficient and has truly negligible SFP, which is independent of the array length.

On the other hand, it is known [23] that we can reduce the running time of insertion sort on the FHE data by allowing very small sorting failure probability using what is known as the window technique. According to this technique, in each insertion sort, instead of inserting the ith element into the subarray of $(a_1, a_2, \cdots, a_{i-1})$, we insert the ith element into the subarray $(a_{i-k}, a_{i-k+1}, \cdots, a_{i-1})$ of length k, called the window length, immediately to the left of the ith element. We call this subarray a "window" with window length k. This technique is used to reduce the number of bootstrapping operations in [23]. Since the insertion sort is the subroutine algorithm for the Shell sort, the window technique is adequate to be applied to the Shell sort. However, the effective application of the window technique in the Shell sort for homomorphic encryption has not been proposed.

C. MAIN CONTRIBUTION

In this paper, we devise a method to modify the Shell sort in the FHE setting using the window technique, which is proved to be effective in the theoretical aspect and the practical aspect. It is referred to as a "modified Shell sort". The window technique in [18] is applied to each subroutine insertion sort in our modified Shell sort for FHE setting. We note that the role of the window technique in our algorithm is different from the original use of the window technique. Our algorithm does not reduce the bootstrapping itself compared to the number of the homomorphic gates, but we reduce the number of the comparison operations with the window technique. For this reason, the homomorphic comparison operation in our sorting algorithm does not generate a comparison error.

For theoretical view, the running time complexity of the modified Shell sort is $O(n^{3/2}\sqrt{\alpha + \log \log n})$ with SFP $2^{-\alpha}$

when the gap sequence is powers-of-two, which is close to the average-case time complexity $O(n^{3/2})$ of the original Shell sort. The value of α is our additional parameter that controls the trade-off between the running time and the SFP. This trade-off is quite effective because the SFP is decreased exponentially with α but the running time is proportional only to $\sqrt{\alpha}$. To this end, we use the exact distribution of window lengths of subarrays in each gap for successful sorting in the Shell sort. If the length of the subarray for the insertion sort in some gap is s, it is discovered that the average of the required window length for successful sorting is proportional to $\sqrt{s}$, and the right tail of its probability distribution is very thin. In the sorting process, the window length is provided as a constant multiple of $\sqrt{s}$, which ensures a negligible SFP. If the window length is close to $\beta\sqrt{s}$, the SFP decays as $e^{-\beta^2}$, which signifies a very fast-decaying function. Therefore, with a fixed negligible SFP, we can set a small window length so that the running time is asymptotically faster than that of the naive version of the Shell sort on the FHE data.

For the practical view, the running time of the modified Shell sort is effectively reduced even in the small arrays, compared to the basic in-place sorting algorithms on the FHE data, bubble sort, and insertion sort.

In this paper, we address only the gap sequence of powers of two in the analysis of the modified Shell sort, i.e., 2^h, $h = 1, 2, 3, \cdots$. Although this gap sequence is not optimal in terms of running time complexity, we first analyze the running time complexity of the modified Shell sort on the FHE data, which is important for the FHE in cloud systems. The performance of the modified Shell sort is numerically compared with the cases of near-optimal window lengths obtained through convex optimization and Ciura's optimal gap sequence [25], which was evaluated numerically as an optimal gap sequence in the non-FHE settings. Although we do not analyze this case, the method of deriving the near-optimal window length in the modified Shell sort functions well for Ciura's optimal gap sequence.

We also suggest the convex optimization method to derive a tighter window length. In other words, the window length obtained by the convex optimization method makes the running time of the modified Shell sort to be less than that of the case employing the analytical method in the modified Shell sort. The running time of the proposed modified Shell sort is compared with that of the conventional algorithms with the TFHE scheme, and it is shown that this modified Shell sort has the best performance in running time among in-place sorting algorithms on the TFHE scheme.

Thus, our contributions are summarized as follows:

- We propose a modified Shell sort with an additional parameter α on the FHE data, and derive its theoretical trade-off between the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$ and the SFP $2^{-\alpha}$ when the gap sequence is power-of-two sequence.
- The near-optimal window length of each gap in the modified Shell sort is derived via the convex optimization technique.
- The numerical simulation with TFHE homomorphic encryption scheme is performed, and the modified Shell sort with Ciura's gap sequence is proven to have the best running time performance in the practical situation among the in-place sorting algorithms for the FHE setting.

D. OUTLINE

The remainder of this paper is organized as follows. Section II presents the preliminary of the paper, which includes the related sorting algorithms and the notion of the FHE. In Section III, we present the distribution of the required window length for each gap in the Shell sort on the FHE data with the gap sequence of powers of two. Then, we propose a modified Shell sort for the FHE and derive the trade-off between the running time complexity and the SFP. Section IV discusses a method to deduce the near-optimal window length of each gap of the modified Shell sort using the convex optimization technique. Section V shows numerical results that support the proposed analysis in the TFHE setting. From these results, the performance in the case of the optimal gap sequence or the near-optimal window lengths is compared with that of conventional algorithms using the TFHE library. Section VI concludes the study and discusses the scope for future research.

II. PRELIMINARIES

A. NOTATIONS

Let $A[1:n]$ be an array of length n with indices from 1 to n, where the ith element of $A[1:n]$ is denoted by $A[i]$. $\lceil x \rceil$ means the least integer which is larger than or equal to x. $\binom{m}{n}$ means the binomial coefficient which is given by $\frac{m!}{n!(m-n)!}$. $\Pr[A]$ means the probability of an event A, and $\Pr[A|B]$ means the conditional probability of an event A given B. $g(n) = O(f(n))$ means that there is a positive real number C and a real number n_0 such that $|g(n)| \leq Cf(n)$ for all $n \geq n_0$. Plaintext data are denoted by the normal math italic letters like A, and ciphertext data are denoted by the letters of the typewriter type letters like $\mathtt{A}$.

B. FULLY HOMOMORPHIC ENCRYPTION

the FHE is a public-key encryption scheme, which supports an arbitrary number of additions and multiplications of plaintext without decryption so that anyone without the decryption key can operate the circuit with any ciphertext without leaking the information of its plaintext.

Gentry suggested the bootstrapping technique to transform a somewhat homomorphic encryption scheme, which allows only a finite number of operations on the encrypted data, to a fully homomorphic encryption scheme [2]. The bootstrapping operation has enabled several researchers to construct the FHE schemes [2], [26], which involves implementing the decryption circuit on encrypted data using the evaluation algorithm, that is, the addition and multiplication algorithms in the FHE setting. All of the FHE schemes suggested thus far ensure security by adding the plaintext to an LWE sample or a ring-LWE sample, which is known as pseudorandom samples. For security reasons, the LWE sample or the ring-LWE sample includes some errors. As the addition and multiplication operations are repeated, the total number of errors increases, and if the total number of errors exceeds a certain limit, a decryption failure occurs. Thus, the errors need to be removed after a certain number of operations on the encrypted data, so that the ciphertexts can be further evaluated. The purpose of the bootstrapping operation is to reset the errors in the ciphertext when the errors are too large to be decrypted.

As bootstrapping utilizes a considerable amount of computation during the processing of the FHE, the number of bootstrapping operations significantly affects the total operation time of the FHE. In fact, the number of bootstrapping operations depends on the multiplicative depth of the circuit. The lower the depth of a circuit, the fewer the number of bootstrapping operations. Thus, it is crucial to consider the number of the bootstrapping operations for each element, when bootstrapping is implemented in the FHE schemes. If the total number of operations in an algorithm is fixed, it is better to evenly distribute the operations on the inputs. Furthermore, to stably address errors, deterministic algorithms are better than randomized algorithms. This is because we can predict the error size of each element in deterministic algorithms ensuring that these errors are handled easily and error control is optimized adequately.

C. TFHE HOMOMORPHIC ENCRYPTION

The TFHE homomorphic encryption scheme [27] is the most practical bit-wise homomorphic encryption scheme now. There are two types of the TFHE scheme: the leveled homomorphic encryption and the fully homomorphic encryption. Since we use the fully homomorphic encryption version of the TFHE scheme, we deal with only it in this subsection. Its basic elements are the bootstrapped homomorphic gates, which performs each gate followed by the bootstrapping. Although the noise in the ciphertext grows when we perform the homomorphic gate without the bootstrapping, the bootstrapping refreshes the noise independent of the input noise. Hence, any large-depth Boolean circuits can be performed without noise growth of the ciphertext using the TFHE scheme.

The secret key $\mathbf{s}$ is a vector of length n in $\{0, 1\}^n$ uniformly sampled, and the ciphertext is formed by $(\mathbf{a}, b) \in \mathbb{T}^n \times \mathbb{T}$, where $b = \mathbf{a} \cdot \mathbf{s} + e + \mu$ and $\mu \in \{-\frac{1}{8}, \frac{1}{8}\}$ is encoded by $\mu = \frac{1}{4}(b - \frac{1}{2})$ with the message bit $b \in \{0, 1\}$. The bootstrapping procedure makes the encoded message $\frac{1}{8}$ when the input encoded message is in $(0, \frac{1}{2})$ and makes the encoded message $-\frac{1}{8}$ when the input encoded message is in $(\frac{1}{2}, 1)$. Before the bootstrapping for each bootstrapped homomorphic gate, each matched linear operation is processed so that the encoded message is in $(0, \frac{1}{2})$ when the output bit is 1 and is in $(\frac{1}{2}, 1)$ when the output bit is 0. The linear operations can easily be performed homomorphically since the LWE ciphertext has the linear property. For example, the homomorphic NAND gate performs $\frac{1}{8} - a - b$ homomorphically where a, b are the encoded message of the two input ciphertexts before the bootstrapping. All Boolean gates can be designed by this method, and thus we can compose any Boolean circuits with these bootstrapped homomorphic gates. Each linear operation for each homomorphic gate and the detailed bootstrapping procedure can be referred to in [27].

D. SORTING ALGORITHMS

Although there exist several sorting algorithms [28], we consider only the insertion sort and Shell sort in this paper. These are comparison-based sorting algorithms, which do not rely on the divide-and-conquer method.

The insertion sort is an iterative sorting algorithm that sorts from the leftmost element. In each iteration, we define an element to be sorted into its left-side subarray as the pivot element. It is assumed that the elements to the left of the pivot element are already sorted. We then compare the already sorted elements with the pivot element, deduce its proper position, and insert it into this position. Its worst-case and average-case running time complexities are both $O(n^2)$. It is known that the insertion sort is slightly faster than the bubble sort in practical cases.

The operations in the conventional insertion sort require the knowledge of its input, and this is not allowed in the case of the FHE data. Therefore, we cannot determine the correct position of a pivot element in the already sorted subarray in the FHE setting, and thus, the operation and behavior of the insertion sort need to be modified. It is known [18] that we can perform an insertion sort on the FHE data by sequentially swapping the pivot element with the elements in the already sorted subarray to its left, from left to right. In fact, the FHE version of the insertion sort has already been proposed, and its performance has been assessed numerically in the previous works [18], [20]. This operation, however, is inefficient, as the number of operations is always the same as that in the worst case, that is, its average-case running time complexity is estimated to be $O(n^2)$.

The Shell sort is a generalized version of the insertion sort [21]. It requires a gap sequence, which is a decreasing sequence of positive integers ending with 1. For each gap h and each integer j, $0 \le j \le h - 1$, the $(hi + j)$-th elements $i = 0, 1, 2, \cdots$ are sorted using insertion sort. As the gap sequence ends with 1, we can finally obtain the correctly sorted array. Algorithm 1 shows the specific algorithm for classical Shell sort.

Even though the running time complexity of the Shell sort varies depending on the gap sequences [22], it is asymptotically better than that of insertion sort. To the best of our knowledge, a trial of the Shell sort on the FHE data has not been performed thus far.

E. COMPARISON OPERATION IN FHE

In the sorting algorithms in the FHE setting, the swap operation is performed by comparing two encrypted elements. Although it is not possible to determine the larger element

Algorithm 1: ShellSort$(A[1:n])$

```
Input : An array $A[1:n]$ with $n$ elements and gap
        sequence $G[1:p]$ with decreasing order
Output: Sorted array $A[1:n]$
for $\ell \leftarrow 1$ to $p$ do
    $g \leftarrow G[\ell]$
    for $i \leftarrow g+1$ to $n$ do
        $k \leftarrow i - g$
        $T \leftarrow A[i]$
        while $k \geq 1$ and $T < A[k]$ do
            $A[k+g] \leftarrow A[k]$
            $k \leftarrow k - g$
        end
        $A[k+g] \leftarrow T$
    end
end
```

in the FHE setting, it has been established that computing the maximum and minimum elements out of the two elements is possible in the FHE setting, even though these elements are encrypted.

Bit-wise encrypted numbers can be compared by homomorphically computing the maximum or the minimum using Boolean circuits. Algorithm 2 shows the algorithm sorting the two encrypted numbers with HomXNOR gate and HomMUX gate, which can be supported with any FHE scheme. HomXNOR and HomMUX are the bootstrapped homomorphic gates for XNOR gate and MUX gate, respectively, where $\mathsf{Dec}(\mathsf{HomXNOR}(\mathtt{a},\mathtt{b})) = \overline{\mathsf{Dec}(\mathtt{a}) \oplus \mathsf{Dec}(\mathtt{b})}$ and $\mathsf{Dec}(\mathsf{MUX}(\mathtt{a},\mathtt{b},\mathtt{c})) = \mathsf{Dec}(\mathtt{a})\ ?\ \mathsf{Dec}(\mathtt{b}) : \mathsf{Dec}(\mathtt{c})$.

Algorithm 2: SortTwo$(\mathtt{X}, \mathtt{Y})$

```
Input : An encrypted integer
        $\mathtt{X} = \mathtt{x}[1:m]$, $\mathtt{Y} = \mathtt{y}[1:m]$ with $m$ bits
Output: $\mathtt{X}$ and $\mathtt{Y}$ are sorted with increasing order.
$\mathtt{u}[1:m] = \mathtt{x}[1:m]$
$\mathtt{v}[1:m] = \mathtt{y}[1:m]$
$\mathtt{c} = \mathsf{Enc}(0)$
for $i = 1$ to $m$ do
    $\mathtt{t} = \mathsf{HomXNOR}(\mathtt{u}[i], \mathtt{v}[i])$
    $\mathtt{c} = \mathsf{HomMUX}(\mathtt{t}, \mathtt{c}, \mathtt{u}[i])$
end
for $i = 1$ to $m$ do
    $\mathtt{x}[i] = \mathsf{HomMUX}(\mathtt{c}, \mathtt{v}[i], \mathtt{u}[i])$
    $\mathtt{y}[i] = \mathsf{HomMUX}(\mathtt{c}, \mathtt{u}[i], \mathtt{v}[i])$
end
```

III. ANALYSIS OF MODIFIED SHELL SORT OVER FHE

In this section, we propose a modified Shell sort using the window technique suggested in [23], and the probability distribution of the required window length for the successful sorting is also obtained when the powers-of-two gap sequence is used. Finally, the running time complexity of the modified Shell sort in each gap for the successful sorting of each subarray is determined for the FHE, considering the trade-off with the SFP when the powers-of-two gap sequence is used.

Algorithm 3: FHEShellSort$(\mathtt{A}[1:n])$

```
Input : An encrypted array $\mathtt{A}[1:n]$ with $n$ elements,
        gap sequence $G[1:p]$ with decreasing order,
        and $\alpha$
Output: Sorted array of encrypted data $\mathtt{A}[1:n]$ without
        sorting failure
for $\ell \leftarrow 1$ to $p$ do
    $g \leftarrow G[\ell]$
    for $i \leftarrow g+1$ to $n$ do
        for $j \leftarrow \lceil \frac{i}{g} \rceil - 1$ to $1$ do
            $\mathsf{SortTwo}(\mathtt{A}[i-gj], \mathtt{A}[i])$
        end
    end
end
```

A. MODIFIED SHELL SORT OVER FHE

As insertion sort can be performed on the FHE data, the Shell sort, which uses the insertion sort as a subroutine algorithm, can also be performed on the FHE data. The Shell sort using the FHE variant insertion sort as subroutine algorithms is shown in Algorithm 3. However, if the Shell sort is to be employed without any sorting failure in the FHE setting, it is expected to be pretty conservative. In other words, as we need to consider the worst case for each gap, its running time complexity becomes $O(n^2)$, which does not provide any advantage in comparison with a simple insertion sort. This is because the FHE variant insertion sort cannot be performed adaptively with the intermediate situations. Thus, designing the Shell sort with a negligible SFP and a running time complexity close to the original average-case time complexity is necessary.

To this end, we employ the window technique [18], [23] in the Shell sort. During the insertion sort in each gap, instead of searching the position of each element in the whole partially sorted array, we search for its position in the partially sorted subarray of a certain window length, located to the left of the pivot element, as shown in Fig. 1. Fig. 1 shows an example of the modified Shell sort using the window technique, where the gap is 4 and the window length is 2. Subarrays consisting of elements that are separated by the gap are sorted using the insertion sort. To sort each subarray, it is compared only with the elements that are located to its left, within a distance equal to the window length from the pivot element to be inserted, which is called the modified Shell sort.

The proposed modified Shell sort is described in Algorithm 4. As the minimum and maximum functions can be computed without knowing their plaintext in the FHE

Algorithm 4: ModifiedShellSort$(A[1:n], \alpha)$

```
Input  : An encrypted array A[1 : n] with n elements,
         gap sequence G[1 : p] with decreasing order,
         and α
Output: Sorted array of encrypted data A[1 : n] with
         SFP $2^{-\alpha}$
$c \leftarrow \alpha + 1 + \log\log n$
for $\ell \leftarrow 1$ to $p$ do
    $g \leftarrow G[\ell]$
    $k \leftarrow \min\left\{\left\lceil\sqrt{\lceil\frac{n}{2g}\rceil \cdot (c+\ell) \cdot \frac{1}{\log e}}\right\rceil, \left\lceil\frac{n}{2g}\right\rceil\right\}$
    for $i \leftarrow g+1$ to $n$ do
        $u \leftarrow \min\left\{k, \lceil\frac{i}{g}\rceil - 1\right\}$
        for $j \leftarrow u$ to 1 do
            SortTwo(A[$i - gj$], A[$i$])
        end
    end
end
```

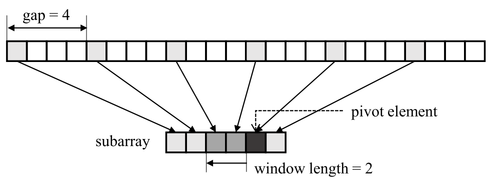


FIGURE 1. Modified Shell sort using the window technique.

setting as we deal with in Section II, neither of the operations in Algorithm 4 require any knowledge of the contents of elements in the array $A[i]$. Thus, Algorithm 4 can be executed in the FHE setting. In designing this algorithm, deciding the window length in each gap for successfully sorting each subarray in the Shell sort for the given SFP $2^{-\alpha}$ is not a trivial problem. Along with the design of the window length for each gap, we propose a modified Shell sort with an additional parameter α.

We prove that the running time complexity of the modified Shell sort is determined to be $O(n^{3/2}\sqrt{\alpha + \log\log n})$ with an SFP of $2^{-\alpha}$ for powers-of-two gap sequence, which consists of all powers of two less than the length of the array. Note that the average-case time complexity of the classical Shell sort with powers-of-two gap sequence is $O(n^{3/2})$ [29]. The parameter α is determined only from the SFP, regardless of the input length n. In fact, α is considerably smaller than n and should be larger than or equal to $\sqrt{6}\log e - 1 \simeq 2.534$, the derivation of which is provided in a subsequent section of this paper. It is noted that the proposed modified Shell sort considers the trade-off between the running time complexity and the SFP.

Before analyzing the running time complexity and the sorting failure probability of the modified Shell sort with power-of-two gap sequence, we introduce the main idea of the analysis to help readers to understand the following theorems and lemmas.

Lemma 1 and 2 deal with the useful properties of the intermediate arrays to make it easy to induce the exact number of acceptable arrays for a certain window length, which is dealt with in Lemma 3 and Theorem 4. Theorem 4 is the special case of Lemma 3 that matches our aim.

While the number of acceptable arrays is represented with some binomial coefficients, the resultant running time complexity has to be represented with some analytic function. To this end, Lemma 5 and 6 relate some formulas with binomial coefficients with some exponential function. With the help of these lemmas, Theorem 7 suggests the running time complexity and the sorting failure probability of the modified Shell sort.

Remark: We assume that the modified Shell sort is performed with the bootstrapped homomorphic gate, a homomorphic Boolean gate followed by the bootstrapping. The bootstrapping always removes the noise in the ciphertext, and thus we do not have to consider the noise amplification in the ciphertext when processing any homomorphic evaluations. In addition, the ciphertext size does not grow in the fully homomorphic encryption scheme, and we also do not have to consider the amplification of the ciphertext size. Thus, we focus on the validity of the modified Shell sort itself in the following analysis. If we use the leveled homomorphic encryption instead of the FHE, the analysis for the noise growth or the ciphertext size growth will be needed, and this analysis becomes an interesting future work.

B. PROBABILITY DISTRIBUTION OF REQUIRED WINDOW LENGTH

In this subsection, we derive the probability distribution of required window length in each gap required for successfully sorting each subarray in the Shell sort. This probability distribution is essential in determining the window length of each gap in the modified Shell sort, because the properties of the tail of the probability distribution must be used to obtain the required window length.

The array of n elements is denoted by its index vector $(a_1, a_2, \cdots, a_n)$, which is a permuted vector of $(1, 2, \cdots, n)$. If we handle the real data, we map each datum to its respective index in $\{1, 2, \cdots, n\}$. Moreover, we assume that n is an even integer. If n is odd, the same analysis can be applied, with an additional dummy element inserted in the rightmost position with the largest element. Several lemmas are needed for devising the main theorem of the probability distribution for the required window length.

Before obtaining the probability distribution, the meaning of the probability distribution in this subsection has to be clarified. For each permutation of $(1, 2, \cdots, n)$, the required window length is defined as the minimum window length such that the insertion sort with the window length returns a perfectly sorted array. The required window length is a random variable when the sample space is the set of all permutations of $(1, 2, \cdots, n)$ or its subset. For analyzing the modified Shell sort, we are interested in the case when the sample space is the set of all permutations $(a_1, a_2, \cdots, a_n)$ of $(1, 2, \cdots, n)$ which satisfy $a_i < a_{i+2}$.

While the analysis of the conventional Shell sort is performed for an average number of operations, the analysis of the window length in the modified Shell sort involves the maximum number of insertion operations for each subarray.

We assume that the gap sequence is powers of two, i.e., $2^{\lfloor \log n \rfloor}, 2^{\lfloor \log n \rfloor - 1}, \cdots, 2^2, 2, 1$. With this gap sequence, each subarray that is sorted using insertion sort has the following structure. The elements in odd positions of the subarray for a gap 2^h are already sorted and the elements in even positions are also sorted for a gap 2^h using the previous insertion sort for a gap 2^{h+1}. We analyze the insertion sort under this special situation.

The following Lemma 1 and Lemma 2 suggest that the required window length of a permuted vector $(a_1, a_2, \cdots, a_{2m})$ of $(1, 2, \cdots, 2m)$ is equal to the maximum distance of the current position and the right position in our situation. After Lemma 1 and Lemma 2, we identify these two notions as equivalent notions.

Lemma 1: Let $\mathbf{a} = (a_1, a_2, \cdots, a_{2m})$ *be a subarray in each gap in the Shell sort with a gap sequence* 2^h*, which is permuted from* $(1, 2, \cdots, 2m)$*, satisfying* $a_i < a_{i+2}$ *for* $i = 1, 2, \cdots, 2m-2$*. Let* $M(\mathbf{a}) = \max_{1 \le i \le 2m} |a_i - i|$*. Then there exists an even integer* j *and an odd integer* k *such that*

$$M(\mathbf{a}) = |a_j - j| = |a_k - k|$$

and $(a_j - j)(a_k - k) \le 0$.

Proof: Let M_1, M_2, M_3, and M_4 be defined as

$$\begin{aligned} M_1 &= \max_{1 \le i \le m} (a_{2i-1} - (2i-1)) \\ M_2 &= -\min_{1 \le i \le m} (a_{2i-1} - (2i-1)) \\ M_3 &= \max_{1 \le i \le m} (a_{2i} - 2i) \\ M_4 &= -\min_{1 \le i \le m} (a_{2i} - 2i). \end{aligned}$$

It is clear that at least one of M_1 and M_2 as well M_3 and M_4 is a non-negative integer. If we establish that $M_1 = M_4$ and $M_2 = M_3$, the lemma can be proved by the following argument. If $M_1 \ge M_2$, we obtain $M(\mathbf{a}) = M_1 = M_4 \ge M_2 = M_3$, and thus, there exist an odd index j and an even index k, such that $M(\mathbf{a}) = a_j - j = -(a_k - k)$. If $M_1 < M_2$, $M(\mathbf{a}) = M_2 = M_3 \ge M_1 = M_4$ holds, then there exist an odd index j and an even index k, such that $M(\mathbf{a}) = -(a_j - j) = a_k - k$. Thus, it is sufficient to prove that $M_1 = M_4$ and $M_2 = M_3$.

To show $M_1 = M_4$ and $M_2 = M_3$, we prove the following four inequalities; $M_1 \ge M_4$, $M_1 \le M_4$, $M_2 \ge M_3$, and $M_2 \le M_3$.

i) Firstly, we show that $M_1 \ge M_4$. Consider an index l, such that $a_{2l} - 2l = \min_{1 \le i \le m}(a_{2i} - 2i)$, which is $-M_4$. We establish this case for $a_{2\ell} = 2m$ or $a_{2\ell} < 2m$.

i)-1 If $a_{2l} = 2m$, l must be m, as $2m$ is the largest element. Thus, we obtain $\min_{1 \le i \le m}(a_{2i} - 2i) = 0$ and $a_{2i} \ge 2i$ for all i, $1 \le i \le m$, which implies that 1 cannot be in the even index and must be in the first index, and $a_1 - 1 = 0$. Therefore, $M_1 = \max_{1 \le i \le m}(a_{2i-1} - (2i-1)) \ge 0 = -\min_{1 \le i \le m}(a_{2i} - 2i) = M_4$.

i)-2 If $a_{2l} < 2m$, we show that $a_{2l} + 1$ must be in the odd index. Let $a_{2l} + 1$ be in the even index; this implies that $a_{2l} + 1 = a_{2l+2}$, because all the elements in the even indices are already sorted. Then, we obtain $a_{2l+2} - (2l+2) = (a_{2l} + 1) - (2l + 2) = a_{2l} - 2l - 1 < a_{2l} - 2l$, which is a contradiction to the assumption that $a_{2l} - 2l$ is the minimum value, and thus, $a_{2l} + 1$ must be in the odd index. Among $\{1, 2, \cdots, a_{2l} - 1\}$, $l - 1$ elements have to be placed in the even indices in the left-side of a_{2l}. The remaining $a_{2l} - l$ elements must be placed in the odd indices in the increasing order from the first index 1. Thus, the index of $a_{2l} + 1$ must be $2(a_{2l} - l) + 1$. As $a_{2(a_{2l}-l)+1} - (2(a_{2l} - l) + 1) = (a_{2l} + 1) - (2(a_{2l} - l) + 1) = 2l - a_{2l}$, we obtain $M_1 = \max_{1 \le i \le m}(a_{2i-1} - (2i-1)) \ge 2l - a_{2l} = -\min_{1 \le i \le m}(a_{2i} - 2i) = M_4$.

ii) We then show that $M_2 \ge M_3$. Consider an index l, such that $a_{2l} - 2l = \max_{1 \le i \le m}(a_{2i} - 2i)$, which is M_3. We establish this case for $a_{2\ell} = 1$ or $a_{2\ell} > 1$.

ii)-1 If $a_{2l} = 1$, l must be 1, as 1 is the smallest element, and therefore, $\max_{1 \le i \le m}(a_{2i} - 2i) = -1$. As $a_{2i} - 2i \le -1$ for all i, $1 \le i \le m$, $2m$ cannot belong to the even index. Thus, $2m$ must be in the $(2m-1)$-th index, and $a_{2m-1} - (2m-1) = 1$. Therefore, $M_2 = -\min_{1 \le i \le m}(a_{2i-1} - (2i-1)) \ge -1 = \max_{1 \le i \le m}(a_{2i} - 2i) = M_3$.

ii)-2 If $a_{2l} > 1$, we show that $a_{2l} - 1$ must be in the odd index. Let $a_{2l} - 1$ be in the even index. We have $a_{2l} - 1 = a_{2l-2}$, because all the elements in the even indices are already sorted. Then, we obtain $a_{2l-2} - (2l-2) = (a_{2l} - 1) - (2l - 2) = a_{2l} - 2l + 1 > a_{2l} - 2l$, which is a contradiction to the assumption that $a_{2l} - 2l$ is the maximum value, and thus, $a_{2l} - 1$ must be in the odd index. Among $\{1, 2, \cdots, a_{2l} - 2\}$, $l - 1$ elements need to be placed in the even indices in the left-side of a_{2l}. The remaining $a_{2l} - l - 1$ elements have to be placed in the odd indices from the first index 1. Thus, the index of $a_{2l} - 1$ must be $2(a_{2l} - l) - 1$. As $a_{2(a_{2l}-l)-1} - (2(a_{2l} - l) - 1) = (a_{2l} - 1) - (2(a_{2l} - l) - 1) = 2l - a_{2l}$, we obtain $M_2 = -\min_{1 \le i \le m}(a_{2i-1} - (2i-1)) \ge -(2l - a_{2l}) = \max_{1 \le i \le m}(a_{2i} - 2i) = M_3$.

Similarly, we can establish that $M_1 \le M_4$ and $M_2 \le M_3$ by swapping the even indices with the odd indices. Therefore, we can prove that $M_3 = \max_{1 \le i \le m}(a_{2i} - 2i) \ge -\min_{1 \le i \le m}(a_{2i-1} - (2i-1)) = M_2$, and $M_4 = -\min_{1 \le i \le m}(a_{2i} - 2i) \ge \max_{1 \le i \le m}(a_{2i-1} - (2i-1)) = M_1$. Therefore, we establish that $M_1 = M_4$ and $M_2 = M_3$. □

Lemma 2: Let $\mathbf{a} = (a_1, a_2, \cdots, a_{2m})$ *be a subarray in the Shell sort with a gap sequence* 2^h*, which is permuted from* $(1, 2, \cdots, 2m)$ *satisfying* $a_i < a_{i+2}$ *for* $i = 1, 2, \cdots, 2m-2$.

Let $W(\mathbf{a})$ be the required minimum window length to sort the subarray successfully. Then, we have

$$W(\mathbf{a}) = \max_{1 \le i \le 2m} (i - a_i).$$

When we insert a_i into the partially sorted subarray, the following scenarios can be given; if $a_i < i$, we require a window length of $i - a_i$, and if $a_i \ge i$, a_i stays in place regardless of the window length.

Consider the first case, where $a_i < i$. First, we assume that i is even. Consider the elements to the left of a_i. From the condition $a_i < a_{i+2}$, it is clear that all the elements in even indices to the left of a_i are less than a_i. As there are $i/2 - 1$ even indices to the left of a_i, the remaining $a_i - i/2$ elements in $\{1, 2, \cdots, a_i - 1\}$ have to be placed in odd indices in increasing order from the leftmost odd index. As the number of odd indices to the left of a_i is $i/2$ and $i/2 > a_i - i/2$, all the elements less than a_i are located to the left of a_i.

We then assume that i is odd. The proof is almost the same as that for the scenario in which i is even. As there are $(i-1)/2$ odd indices to the left of a_i, the remaining $a_i - (i+1)/2$ elements in $\{1, 2, \cdots, a_i - 1\}$ must be placed in even indices from the first even index, in increasing order. As the number of even indices to the left of a_i is $(i-1)/2$ and $(i-1)/2 > a_i - (i+1)/2$, all the elements less than a_i are located to the left of a_i.

Thus, we prove that all the elements less than a_i are located to the left of a_i. The partially sorted subarray, therefore, must include the elements $\{1, 2, \cdots, a_i - 1\}$ in the indices $\{1, 2, \cdots, a_i - 1\}$ in the appropriate order. This implies that a_i moves to the index a_i, and thus, we require a minimum window length of $i - a_i$.

Consider the second case, in which $a_i \ge i$. It is evident that $i/2 \le a_i - i/2$, when i is even, and $(i-1)/2 \le a_i - (i+1)/2$, when i is odd. This implies that all the elements to the left of a_i are less than a_i. Thus, the partially sorted subarray in the indices $\{1, 2, \cdots, i-1\}$ comprises elements smaller than a_i. Therefore, a_i does not move to the left but stays in its position, regardless of the window length.

From Lemma 1, it is noted that $M(\mathbf{a})$ is equal to $W(\mathbf{a})$. Lemma 3 is needed to obtain the exact number of the arrays whose required window length is some non-negative number k. Theorem 4 corresponds to the conclusion of this subsection, and this can be obtained by only considering the special case of Lemma 3.

Lemma 3: Let $p_k(n, m)$ be the number of distinct arrays $(a_1, a_2, \cdots, a_m)$ of length m, whose elements from $\{1, 2, \cdots, n\}$ are sorted in increasing order, $a_i < a_{i+1}$, and $\max_{1 \le i \le m} |a_i - 2i| \le k$ is satisfied for a positive integer k and $n \ge m$. Let $(b_0, b_1, \cdots)$ and $(c_0, c_1, \cdots)$ be the two arrays defined as

$$b_0 = c_0 = 0$$

$$b_{i+1} = \begin{cases} b_i + (k+1) & \text{if } i \text{ is even} \\ b_i + (k+2) & \text{if } i \text{ is odd} \end{cases}$$

$$c_{i+1} = \begin{cases} c_i + (k+2) & \text{if } i \text{ is even} \\ c_i + (k+1) & \text{if } i \text{ is odd.} \end{cases}$$

For $2m - k \le n \le 2m + k$, we obtain

$$p_k(n, m) = \binom{n}{m} - \sum_{1 \le b_i \le m} (-1)^{i+1} \binom{n}{m - b_i} - \sum_{1 \le c_i \le m} (-1)^{i+1} \binom{n}{m + c_i}. \quad (1)$$

Proof: It is clear that $p_k(1, 1) = 1$ for all $k \ge 1$, and $p_k(n, 1) = n$ for $n \le k+2$. As the element a_m in the last index must be $2m - k \le a_m \le 2m + k$ from $\max_{1 \le i \le m} |a_i - 2i| \le k$, the following can be determined from the condition $2m - k \le a_m \le 2m + k$:

i) For $n < 2m - k$,

$$p_k(n, m) = 0,$$

because the minimum possible value of a_m must be $2m - k$.

ii) For $n > 2m + k + 1$,

$$p_k(n, m) = p_k(2m + k, m),$$

because the maximum possible value of a_m must be $2m + k$.

iii) For $2m - k \le n \le 2m + k + 1$,
We derive the recurrence relation of $p_k(n, m)$ using the following three cases:

iii)-1 For $n = 2m + k + 1$,
It is easy to derive that

$$p_k(2m + k + 1, m) = p_k(2m + k, m). \quad (2)$$

Note that this case can be included in ii). Although this separation of the case appears unnatural, it enables us to analyze $p_k(n, m)$ well.

iii)-2 For $2m - k + 1 \le n \le 2m + k$,
If the element in the last index m is n, the elements in the remaining indices should be selected from $\{1, 2, \cdots, n - 1\}$, and thus, there are $p_k(n - 1, m - 1)$ possible arrays. If the element in the last index m is not n, the element n cannot be located in one of the indices $\{1, 2, \cdots, m - 1\}$, because the elements are sorted in increasing order. Thus, $\{1, 2, \cdots, n - 1\}$ should be located in the indices $\{1, 2, \cdots, m\}$, and there are $p_k(n - 1, m)$ possible arrays. Therefore, we obtain

$$p_k(n, m) = p_k(n - 1, m) + p_k(n - 1, m - 1). \quad (3)$$

iii)-3 For $n = 2m - k$,
We obtain

$$p_k(2m - k, m) = p_k(2m - k - 1, m - 1), \quad (4)$$

because the element $2m - k$ must be located in the index m.

Let $q_k(n,m)$ be the right-hand side in (1). Then, we prove that $q_k(2m+k+2,m)=0$ and $q_k(2m-k-1,m)=0$. First, $q_k(2m+k+2,m)$ can be written as

$$\sum_{i=1}^{j}(-1)^i\binom{2m+k+2}{m+c_i}+\sum_{i=1}^{j}(-1)^{i-1}\binom{2m+k+2}{m-b_{i-1}}. \quad (5)$$

As $(m+c_i)+(m-b_{i-1})=2m+k+2$, $\binom{2m+k+2}{m+c_i}=\binom{2m+k+2}{m-b_{i-1}}$ holds, and $q_k(2m+k+2,m)$ is equal to 0.

Next, $q_k(2m-k-1,m)$ can be written as

$$\sum_{i=1}^{j}(-1)^{i-1}\binom{2m-k-1}{m+c_{i-1}}+\sum_{i=1}^{j}(-1)^{i}\binom{2m-k-1}{m-b_i}. \quad (6)$$

As $(m+c_{i-1})+(m-b_i)=2m-k-1$, $\binom{2m-k-1}{m+c_{i-1}}=\binom{2m-k-1}{m-b_i}$ holds, and $q_k(2m-k-1,m)$ is equal to 0.

We now prove that $p_k(n,m)=q_k(n,m)$ when $2m-k\le n\le 2m+k+1$. Using the fact that $q_k(n,m)$ is simply a linear combination of binomial coefficients and the property of the binomial coefficients, we easily know that

$$q_k(n,m)=q_k(n-1,m)+q_k(n-1,m-1).$$

When $n=2m+k+1$, we know that $q_k(n-1,m-1)=0$ from (5). Thus, we have

$$q_k(2m+k+1,m)=q_k(2m+k,m).$$

When $n=2m-k$, we know that $q_k(n-1,m)=0$ from (5). Thus, we have

$$q_k(2m-k,m)=q_k(2m-k-1,m-1).$$

Clearly, $p_k(1,1)=q_k(1,1)$. The recurrence relation of $p_k(n,m)$ is identical to that of $q_k(n,m)$ when $2m-k\le n\le 2m+k+1$ and the initial value is also identical. Therefore, we prove that $p_k(n,m)=q_k(n,m)$ when $2m-k\le n\le 2m+k+1$, which proves it. □

Remark: In order to understand intuitionally the proof of Lemma 3, we add the additional explanation of proof of Lemma 3 in the Appendix B.

From the previous lemmas, we have the following theorem.

Theorem 4: Let $C(2m,k)$ be the number of the permutations **a** *of* $\{1,2,\cdots,2m\}$*, such that $a_i<a_{i+2}$ for all possible i, and $W(\mathbf{a})\le k$. Then, we have*

$$C(2m,k)=\binom{2m}{m}-\sum_{1\le b_i\le m}(-1)^{i+1}\binom{2m}{m-b_i}-\sum_{1\le c_i\le m}(-1)^{i+1}\binom{2m}{m-c_i},$$

where b_i and c_i are defined in Lemma 3.

Proof: As $M(\mathbf{a})$ of the odd indices is equal to that of the even indices from Lemma 1, we consider only the even indices. Thus, we can consider this situation to be equivalent to the following simple situation; we consider distinct m elements from $\{1,2,\cdots,2m\}$ randomly, sort them in increasing order, and consider a_i-2i rather than a_i-i. Then, $C(2m,k)$ is identical to $p_k(2m,m)$ in Lemma 3. This is established as $\binom{2m}{m+b_i}=\binom{2m}{m-b_i}$. □

In fact, $C(2m,k)$ denotes the number of arrays for gap 2^h, which can be successfully sorted using the proposed modified Shell sort with a window length of k. Clearly, the exact number of arrays with $W(\mathbf{a})=k$, such that $a_i<a_{i+2}$ for all i can be obtained by computing $C(2m,k)-C(2m,k-1)$. With this result, we derive the running time complexity of the modified Shell sort in the next subsection.

C. DERIVATION OF RUNNING TIME COMPLEXITY FOR A SPECIFIC SFP

In this subsection, we derive the running time complexity $O(n^{3/2}\sqrt{\alpha+\log\log n})$ of the proposed modified Shell sort with powers-of-two gap sequence, considering the optimal trade-off with the SFP $2^{-\alpha}$, in which α is the parameter that controls the window length of each gap. In the running time complexity, $\log\log n$ increases gradually as n increases. Therefore, the running time complexity is approximately proportional to $n^{3/2}\sqrt{\alpha}$. However, the probability that the output is not successfully sorted decreases exponentially as α increases. It is noted that the SFP $2^{-\alpha}$ is not related to the number of the input data. One of the advantages of the modified Shell sort algorithm is irrespective of the number of the input data, and thus we can obtain a trade-off between the SFP and running time complexity by considering an appropriate α.

It is important to prove the following lemmas to determine the relation between the binomial coefficients and exponential function. It is a well-known fact from the central limit theorem in statistics that the closer n is to infinity, the closer a binomial distribution is to a normal distribution. Even though the binomial and normal distributions are similar, we should establish that some binomial coefficients are upper-bounded by the probability distribution function of the normal distribution. The following Lemma 5 is used in the proof of Lemma 6, and Lemma 6 is used to prove Theorem 7.

Lemma 5: Let $f:[a,\infty)\to\mathbb{R}$ be a function of some real number a satisfying the following;

i) $\lim_{x\to\infty}f(x)=M$ for some real number M.
ii) There exists a positive integer n, such that the n-th order derivative $f^{(n)}(x)$ exists on (a,∞), and $(-1)^nf^{(n)}(x)>0$ for all $x\in(a,\infty)$.

Then, $f(x)>M$ for all $x\in[a,\infty)$.

Proof: It is sufficient to show that $f^{(m)}(x)\to 0$ as $x\to\infty$ and $(-1)^mf^{(m)}(x)$ is a monotonically decreasing function for m, $1\le m\le n-1$. If this is proved, then $f(x)$ is a monotonically decreasing function and is larger than the limit value M from the first condition in Lemma 5, as $f'(x)$ is negative for (a,∞). Since it is true for $m=n$ that $(-1)^mf^{(m)}(x)>0$, we will prove the following: if it is true for $2\le k\le n$ that $(-1)^kf^{(k)}(x)>0$, then we have $\lim_{x\to\infty}f^{(k-1)}(x)=0$, and it is true that $(-1)^{k-1}f^{(k-1)}(x)$ is a monotonically decreasing function.

Let $g_k(x) = (-1)^k f^{(k)}(x)$. As $(-1)^{k-1} f^{(k)}(x) = g'_{k-1}(x) < 0$ on (a, ∞), $g_{k-1}(x)$ is a monotonically decreasing function. As a monotonically decreasing function always converges to a certain value, if it possesses some lower bound, we obtain $\lim_{x\to\infty} g_{k-1}(x) = T$ for some T, or $\lim_{x\to\infty} g_{k-1}(x) = -\infty$. We assume that $\lim_{x\to\infty} g_{k-1}(x) = T$ for some $T \neq 0$, or $\lim_{x\to\infty} g_{k-1}(x) = -\infty$. Then, we can deduce some $N \in (a, \infty)$, $R > 0$, such that $|g_{k-1}(x)| > R$, i.e., $f^{(k-1)}(x) > R$ for all $x > N$, or $f^{(k-1)}(x) < -R$ for all $x > N$.

Consider the case of $f^{(k-1)}(x) > R$. If we integrate both terms from N to $x \in (N, \infty)$ iteratively as

$$\begin{aligned} f^{(k-2)}(x) - f^{(k-2)}(N) &= \int_N^x f^{(k-1)}(t)dt \\ &> \int_N^x R dx = R(x-N) \\ f^{(k-3)}(x) - f^{(k-3)}(N) &= \int_N^x f^{(k-2)}(t)dt \\ &> \int_N^x \left(R(x-N) + f^{(k-2)}(N)\right) dx \\ &= \frac{R}{2}(x-N)^2 + f^{(k-2)}(N)(x-N), \end{aligned}$$

we obtain

$$f(x) > \frac{R}{(k-1)!}(x-N)^{k-1} + \sum_{i=0}^{k-2} \frac{f^{(i)}(N)}{i!}(x-N)^i,$$

whose right-hand side tends to infinity, as $x \to \infty$. In this case, $f(x)$ tends to infinity as well, which contradicts the first condition. If we consider the case of $f^{(m)}(x) < -R$, the inequality is changed to

$$f(x) < -\frac{R}{(k-1)!}(x-N)^{k-1} + \sum_{i=0}^{k-2} \frac{f^{(i)}(N)}{i!}(x-N)^i,$$

whose right-hand side tends to negative infinity, as $x \to \infty$. Then $f(x)$ tends to negative infinity as well, which also contradicts the first condition.

Thus, we obtain $\lim_{x\to\infty} g_{k-1}(x) = 0$. As $g_{k-1}(x)$ is a monotonically decreasing function, $g_{k-1}(x) > 0$ on (a, ∞), which completes the proof. □

Lemma 6 directly uses Lemma 5. To prove the inequality in Lemma 6, we only prove that the condition of Lemma 5 holds for some function.

Lemma 6: For any real number $\alpha \geq \sqrt{6}$ and any positive integer $n \geq \lceil \alpha^2 \rceil$, the following inequality holds

$$\binom{2n}{n - \lceil \alpha\sqrt{n} \rceil} < e^{-\alpha^2}\binom{2n}{n}.$$

Proof: It can be derived that

$$\begin{aligned} \frac{\binom{2n}{n}}{\binom{2n}{n-\lceil \alpha\sqrt{n}\rceil}} &= \frac{(n+\lceil \alpha\sqrt{n}\rceil)(n+\lceil \alpha\sqrt{n}\rceil - 1)\cdots(n+1)}{n(n-1)\cdots(n-\lceil \alpha\sqrt{n}\rceil+1)} \\ &= \prod_{k=0}^{\lceil \alpha\sqrt{n}\rceil - 1}\left(1 + \frac{\lceil \alpha\sqrt{n}\rceil}{n-k}\right). \end{aligned}$$

We must prove that

$$\prod_{k=0}^{\lceil \alpha\sqrt{n}\rceil - 1}\left(1 + \frac{\lceil \alpha\sqrt{n}\rceil}{n-k}\right) > e^{\alpha^2}. \tag{7}$$

If we consider the logarithm on the left-hand side and change the form, we obtain

$$\sum_{k=0}^{\lceil \alpha\sqrt{n}\rceil - 1} \ln\left(1+\frac{\lceil \alpha\sqrt{n}\rceil}{n-k}\right) \geq \sum_{k=0}^{\lceil \alpha\sqrt{n}\rceil - 1} \ln\left(1+\frac{\alpha}{\sqrt{n}-\frac{k}{\sqrt{n}}}\right). \tag{8}$$

Let $f(x) = \log\left(1+\frac{\alpha}{x}\right)$. Then, the right-hand side of (8) can be defined as

$$\sqrt{n}\sum_{k=1}^{\lceil \alpha\sqrt{n}\rceil} \frac{1}{\sqrt{n}} f(\sqrt{n} + \frac{1}{\sqrt{n}} - \frac{k}{\sqrt{n}}),$$

which is a type of Riemann sum of $f(x)$. As $f(x)$ is a monotonically decreasing function, the Riemann sum demonstrates its lower bound as the integration of $f(x)$ from $\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil \alpha\sqrt{n}\rceil}{\sqrt{n}}$ to $\sqrt{n}+\frac{1}{\sqrt{n}}$. As $\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil \alpha\sqrt{n}\rceil}{\sqrt{n}} \leq \sqrt{n}+\frac{1}{\sqrt{n}}-\alpha$, we obtain

$$\begin{aligned} &\sum_{k=0}^{\lceil \alpha\sqrt{n}\rceil - 1} \ln\left(1+\frac{\alpha}{\sqrt{n}-\frac{k}{\sqrt{n}}}\right) \\ &\geq \sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\frac{\lceil \alpha\sqrt{n}\rceil}{\sqrt{n}}}^{\sqrt{n}+\frac{1}{\sqrt{n}}} \ln\left(1+\frac{\alpha}{x}\right)dx \\ &\geq \sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\alpha}^{\sqrt{n}+\frac{1}{\sqrt{n}}} \ln\left(1+\frac{\alpha}{x}\right)dx. \end{aligned} \tag{9}$$

To integrate right-hand side of (9), let $g(x) = x\ln x$. We then obtain

$$\begin{aligned} &\sqrt{n}\int_{\sqrt{n}+\frac{1}{\sqrt{n}}-\alpha}^{\sqrt{n}+\frac{1}{\sqrt{n}}} \ln\left(1+\frac{\alpha}{x}\right)dx \\ &\quad = g(n+\alpha\sqrt{n}+1) + g(n-\alpha\sqrt{n}+1) - 2g(n+1). \end{aligned}$$

Let $h(x) = g(x^2+\alpha x+1) + g(x^2-\alpha x+1) - 2g(x^2+1)$ in $[\alpha, \infty)$. If we prove $\lim_{x\to\infty} h(x) = \alpha^2$, and $h^{(3)}(x) < 0$ in (α, ∞), we obtain $h(x) > \alpha^2$ in $[\alpha, \infty)$ using Lemma 5. As $\sqrt{n} \geq \alpha$, we obtain $h(\sqrt{n}) > \alpha^2$, which proves (7). We must establish $\lim_{x\to\infty} h(x) = \alpha^2$, and $h^{(3)}(x) < 0$ in (α, ∞). To prove $\lim_{x\to\infty} h(x) = \alpha^2$, we consider $e^{h(x)}$. Using $g(x) = x\ln x$ and $h(x)$, we obtain

$$e^{h(x)} = \left(1 - \frac{\alpha^2 x^2}{(x^2+1)^2}\right)^{x^2-\alpha x+1}\left(1+\frac{\alpha x}{x^2+1}\right)^{2\alpha x}.$$

From $\lim_{x\to\infty}\left(1+\frac{p}{x}\right)^x = e^p$, we obtain $\lim_{x\to\infty} e^{h(x)} = e^{\alpha^2}$, and thus, $\lim_{x\to\infty} h(x) = \alpha^2$.

Moreover, $h^{(3)}(x)$ can be computed as

$$h^{(3)}(x) = -\frac{4\alpha^2 x(x^2-1)\{\alpha^2(x^4+4x^2+1) - 6(x^2+1)^2\}}{(x^2+1)^2(x^2-\alpha x+1)^2(x^2+\alpha x+1)^2}.$$

As $\alpha \geq \sqrt{6}$, we obtain $h^{(3)}(x) < 0$ in (α, ∞). Thus, we complete the proof. □

We present the following theorem, which is the main theorem of this subsection. The situation in Theorem 4 occurs in Theorem 7, so that we can directly use Theorem 4. Then, we use Lemma 6 to obtain a simple upper bound of the complicated formula.

Theorem 7: The running time complexity of the proposed modified Shell sort algorithm is obtained as $O(n^{3/2}\sqrt{\alpha + \log\log n})$. *For* $\alpha \geq \sqrt{6}\log e - 1$, *its SFP is upper-bounded by* $2^{-\alpha}$.

Proof: As the swapping operation in the modified Shell sort algorithm can be performed within a certain constant time, the running time complexity of the modified Shell sort in Algorithm 4 is determined from the number of swapping operations. Let $S(n)$ be the number of the swapping operations with an input length n. Then, $S(n)$ can be upper-bounded as

$$S(n) \leq n \sum_{\ell=0}^{\lfloor \log n \rfloor} k_\ell$$

where the window length k_ℓ of each gap is defined as

$$\left\lceil \sqrt{\left\lceil \frac{n}{2^{\ell+1}} \right\rceil \cdot (\alpha + 1 + \log\log n + \ell) \cdot \frac{1}{\log e}} \right\rceil.$$

Thus, $S(n)$ can be expressed as

$$S(n) = O\left(n^{\frac{3}{2}} \sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\frac{\alpha + \log\log n + \ell + 1}{2^{\ell+1}}} \right).$$

Using $\sqrt{a+b} \leq \sqrt{a} + \sqrt{b}$, we obtain that $T(n)$ is

$$O\left(n^{\frac{3}{2}} \left(\sqrt{\alpha + \log\log n} \sum_{\ell=1}^{\lfloor \log n \rfloor + 1} \frac{1}{2^{\frac{\ell}{2}}} + \sum_{\ell=1}^{\lfloor \log n \rfloor + 1} \frac{\sqrt{\ell}}{2^{\frac{\ell}{2}}} \right) \right).$$

Thus, we obtain $S(n) = O(n^{3/2}\sqrt{\alpha + \log\log n})$, because $\sum_{\ell=1}^{\infty} \frac{1}{2^{\frac{\ell}{2}}}$ and $\sum_{\ell=1}^{\infty} \frac{\sqrt{\ell}}{2^{\frac{\ell}{2}}}$ are both finite.

At this point, we consider the SFP. Let B denote the event that the output of the sorting algorithm is not successfully sorted and let B_ℓ denote the event that at least one subarray for the gap 2^ℓ is not successfully sorted. As $\mathsf{B} \subseteq \bigcup_{\ell=0}^{\lfloor \log n \rfloor} \mathsf{B}_\ell = \bigcup_{\ell=0}^{\lfloor \log n \rfloor} \left(\mathsf{B}_\ell \cap \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right)$, we obtain

$$\begin{aligned} \Pr[\mathsf{B}] &\leq \sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \cap \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] \\ &\leq \sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right], \end{aligned}$$

where $\bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c$ implies the event that the sorting is successful for the gaps $2^{\ell+1}, \cdots, 2^{\lfloor \log n \rfloor}$. All of the subarrays satisfy the condition $a_i < a_{i+2}$ in Theorem 4, before we perform the insertion sort for the gap 2^ℓ. Clearly, there are 2^ℓ subarrays when the gap is 2^ℓ, and the length of subarray is less than or equal to $2\lceil \frac{n}{2^{\ell+1}} \rceil$. Let $m_\ell = \lceil \frac{n}{2^{\ell+1}} \rceil$, and $\beta_\ell = \sqrt{(\alpha + 1 + \log\log n + \ell) \cdot \frac{1}{\log e}}$. As $\beta_\ell \geq \sqrt{6}$, the probability that one subarray of length $2m_\ell$ is not successfully sorted can be upper-bounded as

$$1 - \frac{C(2m_\ell, \beta_\ell\sqrt{m_\ell})}{\binom{2m_\ell}{m_\ell}} \leq 2\frac{\binom{2m_\ell}{m_\ell - \beta_\ell\sqrt{m_\ell}}}{\binom{2m_\ell}{m_\ell}} \leq 2e^{-\beta_\ell^2}, \tag{10}$$

where the second inequality is obtained from Lemma 6 if $m_\ell \geq \lceil \beta_\ell^2 \rceil$. If $m_\ell < \lceil \beta_\ell^2 \rceil$, the left term of (10) is 0, and thus (10) trivially holds. We then obtain

$$\begin{aligned} \sum_{\ell=0}^{\lfloor \log n \rfloor} \Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] &\leq \sum_{\ell=0}^{\lfloor \log n \rfloor} 2^\ell \cdot 2e^{-\beta_\ell^2} \\ &= \frac{2^{-\alpha} \lfloor \log n \rfloor}{\log n} \leq 2^{-\alpha}, \end{aligned}$$

and thus, the theorem is proved. □

Remark: Theorem 7 states that the asymptotic running time complexity of Algorithm 4 is lower than the trivially modified Shell sort in Algorithm 3, which is $O(n^2)$. The reduction of the running time in a concrete sense is rather clear, in that the number of iterative steps in the last **for** statement in Algorithm 4 is lower than that in Algorithm 3. On the other hand, the asymptotic running time of the modified Shell sort is lower than that of the insertion sort for the FHE setting, but the concrete comparison for them in a practical situation is not clear in this theoretical analysis. In Section V, we numerically compare their running time using TFHE homomorphic encryption scheme.

IV. NEAR-OPTIMAL WINDOW LENGTH BY CONVEX OPTIMIZATION

It is necessary to find the shortest window length for the SFP so that the least running time complexity of the modified Shell sort is obtained. Generally, it is not easy to derive the optimal window length in closed form. In this section, we obtain the near-optimal window length using convex optimization [30]. Let $\beta_\ell\sqrt{\lceil n/2^{\ell+1} \rceil}$ be the window length for the gap 2^ℓ, and $\Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right]$ be the SFP for the gap 2^ℓ, when sorting is successful for the gaps $2^{\ell+1}, \cdots, 2^{\lfloor \log n \rfloor}$. From Theorem 4 and Lemma 6, we obtain

$$\Pr\left[\mathsf{B}_\ell \middle| \bigcap_{u=\ell+1}^{\lfloor \log n \rfloor} \mathsf{B}_u^c \right] \leq 2^\ell e^{-\beta_\ell^2}.$$

The objective function that needs to be minimized is the total number of swap operations, which determines the running time. As the exact running time formula is rather complicated, we consider a tight upper bound of the running time, $n\sum_{\ell=0}^{\lfloor \log n \rfloor} \beta_\ell\sqrt{\lceil n/2^{\ell+1} \rceil}$, which is used in the proof of Theorem 7. Let $p_\ell = 2^\ell e^{-\beta_\ell^2}$. Then, we have

$\beta_\ell = \sqrt{(\ell + \log(1/p_\ell))/\log e}$. As it is sufficient to minimize $\sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))}$, the problem of the near-optimal window length can be formulated as follows;

$$\textbf{minimize} \sum_{\ell=0}^{\lfloor \log n \rfloor} \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))}$$

$$\textbf{s.t.} \sum_{\ell=1}^{k-1} p_\ell \le p_{\text{err}}.$$

This formulation implies that the total running time with SFP upper-bounded by p_{err} needs to be minimized. We can validate that $\sqrt{c + \log \frac{1}{x}}$ is a convex function on small positive values, where c is a constant. As the weighted sum of convex functions is also a convex function, the objective function is a convex function, and the constraint is also convex. Thus, this can be termed as a convex optimization problem. As every convex optimization problem can be solved using numerical analysis, it is easy to obtain the near-optimal window length. Then, we can deduce p_ℓ, and the near-optimal window length is determined to be $\lceil \sqrt{\lceil n/2^{\ell+1} \rceil (\ell + \log(1/p_\ell))/\log e} \rceil$ for each gap 2^ℓ. It is noted that the above formulation is not sufficiently tight, because it still uses the union bound. Constructing a tighter formulation, which can be solved easily, can be a focus for future research.

V. SIMULATION RESULTS

A. SIMULATION WITHOUT HOMOMORPHIC ENCRYPTION

The performance of the proposed modified Shell sort is numerically verified using a personal computer with an AMD Ryzen 9 5950X CPU running at 2.04GHz, and 128GB RAM. First, we validate the running time and SFP when the array length varies. Then, the running time and SFP are numerically obtained when the parameter α is varied. Finally, the performance of the modified Shell sort is compared with the cases corresponding to the near-optimal window length, which is obtained using convex optimization, and Ciura's optimal gap sequence, which has been validated numerically as an optimal gap sequence in the non-FHE settings. We firstly simulate these sorting algorithms without homomorphic encryption schemes, i.e., in the plaintext region. Since the use of homomorphic encryption schemes can affect only the running time, the result of SFP values in this simulation has the same meaning in the case of using homomorphic encryption.

Fig. 2 shows the relation between the running time and SFP against various array lengths for $\alpha = 3$. It is observed that the array length increases from 50 to 1000. The input arrays are randomly generated, and 10^5 input arrays are generated for each array length. It is observed from Fig. 2 that the running time increases in proportion to $n^{3/2}$, and the SFP is independent of the array length. This numerical result coincides well with the proposed analysis of the modified Shell sort. Note that the value $c = \alpha + 1 + \log \log n$ increases slightly as the length of the array increases.

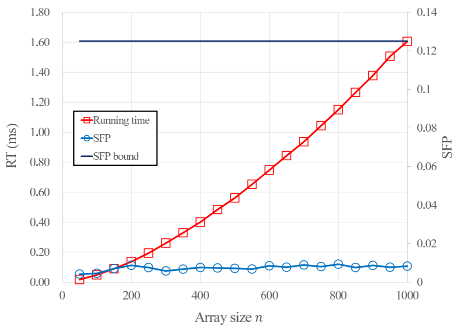


FIGURE 2. Running time and SFP of the modified Shell sort for varied array lengths.

Fig. 3 shows the relation between the running time and SFP for various α values, in which g2p denotes the power of the 2-gap sequence, gop denotes Ciura's optimal gap sequence [25], and a-win and o-win denote the analytically derived window length and near-optimal window length derived by convex optimization, respectively. The input array length is fixed at 1000. Similar to the previous simulation, 10^5 input arrays are randomly generated for each α value. Algorithm 4 and the case corresponding to the Ciura's optimal gap sequence or near-optimal window length are simulated, with the near-optimal window length derived using the convex optimization discussed in Section IV.

From Fig. 3, it is observed that the running time of Algorithm 4 increases as α increases and the growth rate decreases. This observation coincides with the proposed analysis, i.e., the running time is approximately proportional to $\sqrt{\alpha}$. The logarithms of the SFP values of Algorithm 4 are parallel to that of the SFP bounds. This implies that the SFP is proportional to $2^{-\alpha}$ with some small proportional constant.

When the gap sequence is replaced with Ciura's gap sequence, the running time is reduced by approximately 0.5 ms. Sorting failure is not detected in the case of the simulation that uses Ciura's gap sequence. This implies that the order of the SFP of Ciura's optimal gap sequence is less than or equal to 10^{-5}. Although the window lengths of each gap in this paper are analytically derived for the power of the 2-gap sequence, a better result is obtained when Ciura's optimal gap sequence is used.

We numerically find the value $c = \alpha + 1 + \log \log n$ when the SFP value reaches 10^{-5} for some length of an array, and Table 1 shows the values. While the value c increases slightly as the length of the array increases when the gap sequence is powers-of-two and the SFP value is fixed, the value c decreases sharply as the length of the array increases. It suggests that the trade-off in the case of Ciura's gap sequence is asymptotically better than the case of the powers-of-two gap sequence. The exact asymptotical analysis of Ciura's gap sequence is an open problem.

The near-optimal window length is derived using the convex optimization problem described in Section IV. The running time in this case is marginally reduced compared with

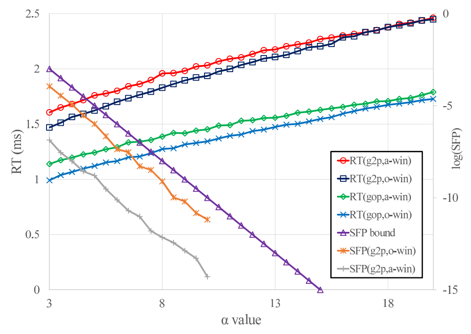


FIGURE 3. **Running time and SFP of the modified Shell sort for varied α values and comparison of these values with those obtained from the cases of Ciura's optimal gap sequence and near-optimal window length derived by convex optimization.**

TABLE 1. **The value $c = \alpha + 1 + \log\log n$ for array of various lengths when the Ciura's gap sequence is used and the SFP is 10^{-5}.**

Array length	Value c
100	4.93
200	2.73
300	2.04
400	1.41
500	1.26

the case using the analytically obtained window length. However, their values become closer as α increases. The SFP of the case using the near-optimal window length for the power of the 2-gap sequence is closer to the SFP bound than that of the case using the analytically obtained window length. Thus, the running time can be reduced, while the SFP remains less than the SFP bound.

B. SIMULATION WITH TFHE SCHEME

In this subsection, we measure the running time of several sorting algorithms on encrypted data, including the modified Shell sort algorithm. We implement each sorting algorithm with the TFHE library [31]. The security parameter in the TFHE scheme is set to be 128, and the number of bits for each data is set to be 10. Table 2 shows the main parameters used in the simulation satisfying 128-bit security. For the modified Shell sort, we set the value of c to make the SFP 10^{-5}, and the Ciura's gap sequence is used rather than the powers-of-two gap sequence. The unit of each running time result is in seconds.

The sorting algorithm to be compared with the modified Shell sort is chosen as follows. Since the modified Shell sort can be the generalized algorithm for the insertion sort, we choose to compare the insertion sort. The randomized Shell sort [24] is the most related sorting algorithm to the proposed modified Shell sort, and thus we also choose it to be compared. These two sorting algorithms are in-place algorithms. For the recursive sorting algorithm, the odd-even merge sort and the bitonic sort are chosen, which are the standard oblivious recursive sorting algorithms. These two

TABLE 2. **TFHE parameters with 128-bit security.**

	Ciphertext dimension	Noise standard deviation
Ciphertext and Key-switching key (LWE)	630	2^{-15}
Bootstrapping key (Ring-LWE)	1024	2^{-25}

recursive algorithms are also used in [20] to compare the sorting algorithm for homomorphic encryption.

As for the odd-even merge sort and the bitonic sort, the length of the input array is originally assumed to be a power of two. However, we cannot generally choose the array length, and thus we perform the simulation with a more general type of numbers rather than the power-of-two array length. Since the length of the input array in our simulation is not a power-of-two integer, we add dummy data in the end of the array to make the input array power-of-two, and these dummy data is assumed to be larger than the data in the input array. Since these dummy data will not be moved in the sorting, we ignore the comparison if dummy data is homomorphically compared to other data, in order to erase the effect of the addition of dummy data. Thus, we can fairly compare the running time of each sorting algorithm in the case of more general array lengths other than the power-of-two length. We specify the whole algorithms used in the simulation in Appendix A.

Table 3 shows the running time of several sorting algorithms required to sort an array of encrypted data of length 500. The running time of the modified Shell sort with Ciura's gap sequence is far lower than the insertion sort, which is the basic in-place sorting algorithm. The use of the modified Shell sort is proved to be efficient not only in the asymptotic sense but also practical sense. Also, although the randomized Shell sort [24] is asymptotically better than our algorithm, the performance of our algorithm is better than that of the randomized Shell sort for an array of length 500.

When we compare the efficient and recursive sorting algorithms, bitonic sort and odd-even merge sort, the running time of the modified Shell sort is yet larger, but it is closer than the original insertion sort. This running time performance will depend on the situation, especially in the IoT device. Since these recursive sorting algorithms make many function calls recursively and the memory of the input array is quite big, the transmission time can be a serious problem when the memory bus bandwidth is not large enough. In this situation, the modified Shell sort will be useful in that it uses no function calls or almost no additional memories. Even though the running times of bitonic sort and odd-even merge sort are smaller than that of the modified Shell sort, the numbers of memory and function calls of the bitonic sort and odd-even merge sort increases to 5121 and 9729, respectively.

Table 4 shows the performance of the modified Shell sort, the insertion sort, and the bitonic sort for an array of various

TABLE 3. Comparison of the running time of several sorting algorithms for array of length 500.

Sorting algorithm	Running time (sec)	Non-recursive/ Recursive	Number of function calls
Insertion sort	97889	Non-recursive	1
Rand Shell sort [24]	58583	Non-recursive	1
Modified Shell sort	**28576**	**Non-recursive**	**1**
Odd-even merge sort	13085	Recursive	5121
Bitonic sort	8753	Recursive	9729

TABLE 4. Comparison of the running time in seconds of several sorting algorithms for array of various lengths.

Array length	Modified Shell sort (in-place)	Insertion sort (in-place)	Bitonic sort (recursive)
100	**2521**	3899	1038
200	**7281**	15636	2745
300	**13352**	34988	5066
400	**20793**	62739	6812

lengths less than 500. While the running time of the insertion sort increases fast as the array length increases, the running time of the modified Shell sort with Ciura's gap sequence increases not very fast as the array length increases. This rate is somewhat similar to the rate of the Bitonic sort, whose running time complexity is better than ours.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a modified Shell sort with an additional parameter α in the FHE setting, and for a gap sequence of powers of two, we derived the running time complexity $O(n^{3/2}\sqrt{\alpha + \log\log n})$, considering a trade-off with the SFP $2^{-\alpha}$. We also established that the running time complexity of the proposed algorithm is almost the same as the average-case running time complexity of the original Shell sort, while the SFP is maintained to be minimal. We then obtained the near-optimal window length of each gap by numerically solving a convex optimization problem. We believe that this study plays a significant role in the foundation of the analysis of the Shell sort in the FHE settings. Using the TFHE encryption scheme, the running time of the proposed modified Shell sort with Ciura's gap sequence was compared with that of the conventional sorting algorithms, and it has the best running time performance among other in-place sorting algorithms in the FHE setting.

The performances of the recursive sorting algorithms on the FHE data, such as bitonic sort and odd-even merge sort, are better than our algorithms but use many function calls in the process of sorting. The detailed analysis of the memory usage in the sorting algorithms for FHE and the simulation in the practical environment with limited memory bus bandwidth is also an important research topic, and we leave it as future work. Designing a practically faster variant of the Shell sort on the FHE data than the recursive sorting algorithms is also our future work. Also, we plan to analyze the modified Shell sort with other gap sequences.

Recently, Hong *et al.* [32] proposed the k-way sorting method extending the conventional 2-way sorting. All previous works used sorting of the two elements as the building block, but they used sorting of the k elements larger than two as the building block. Since we used the 2-way sorting in our modified Shell sort algorithm, analyzing the k-way Shell sort algorithm will be interesting future work.

APPENDIX A
IMPLEMENTATION OF OTHER SORTING ALGORITHMS

We specify the algorithms for other sorting algorithms used in the simulation with TFHE scheme. Algorithm 5, 6, 9, and 12 shows the algorithms used in the simulation for the insertion sort, the randomized Shell sort, the odd-even merge sort, and the bitonic sort, respectively. Other algorithms are the subroutine algorithms for the main algorithm. Note that the randomized Shell sort can be implemented without any function calls by including the subroutine algorithms in the body of the algorithm, but the odd-even merge sort and the bitonic sort should be performed recursively.

Algorithm 5: InsertionSort(A$[1:n]$)

```
Input  : An encrypted array A$[1:n]$ with $n$ elements
Output: Sorted array of encrypted data A$[1:n]$
1 for $i \leftarrow 2$ to $n$ do
2     for $j \leftarrow 1$ to $i-1$ do
3         SortTwo(A$[j]$, A$[i]$)
4     end
5 end
```

Algorithm 6: RandShellSort(A$[1:n]$)

```
Input  : An encrypted array A$[1:n]$ with $n$ elements
Output: Sorted array of encrypted data A$[1:n]$ with
        small sorting failure
1 $m = 2^{\lceil \log n \rceil}$
2 Append $m-n$ dummy elements to A$[1:n]$.
3 RandShellPower2(A$[1:m]$).
4 Remove the $m-n$ dummy elements at the end of A.
```

APPENDIX B
EXPLANATION OF PROOF OF LEMMA 3

The recurrence relations (2)-(4) of $p_k(n,m)$ are similar to the Pascal's triangle $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$ shown in Fig. 4(a), except that the width of the triangle for $p_k(n,m)$ is limited, as shown in Fig. 4(b) as well as (2) and (4). This recursive relation can then be transformed into overlapped Pascal's triangles. Fig. 4(c) shows a part of Fig. 4(b) near the boundary of the lower dotted line. Here, we only consider the lower dotted line. We then establish that this recursive relation near the boundary in Fig. 4(c) is equivalent to the situation of Fig. 4(d), which is two overlapped Pascal's triangles, in which $p_k(0,0) = 1$ and $p_k(0,k+1) = -1$.

```
Algorithm 7: CompareRegions(A[1 : n], s, t, offset)
 Input  : An encrypted array A[1 : n] with n elements
 Output: Sorted array of encrypted data A[1 : n] with
         small sorting failure
1 for count = 1 to count = 4 do
2     Choose the permutation
      σ : {0, ··· , offset − 1} → {0, ··· , offset − 1}
      randomly. for i = 0 to i = offset − 1 do
3         SortTwo(A[s + i], A[t + σ(i)])
4     end
5 end
```

```
Algorithm 8: RandShellPower2(A[1 : n])
 Input  : An encrypted array A[1 : n] with power-of-two
          n = 2^ℓ elements
 Output: Sorted array of encrypted data A[1 : n] with
         small sorting failure
1  for offset = n/2 to offset = 1 with offset ← offset/2
    do
2      for i = 1 to i = n − 2 · offset + 1 with
        i ← i + offset do
3          CompareRegions(A[1 : n], i, i + offset, offset)
4      end
5      for i = n − offset + 1 to i = offset + 1 with
        i ← i − offset do
6          CompareRegions(A[1 : n], i − offset, i, offset)
7      end
8      for i = 1 to i = n − 4 · offset + 1 with
        i ← i + offset do
9          CompareRegions(A[1 :
            n], i, i + 3 · offset, offset)
10     end
11     for i = 1 to i = n − 3 · offset + 1 with
        i ← i + offset do
12         CompareRegions(A[1 :
            n], i, i + 2 · offset, offset)
13     end
14     for i = 1 to i = n − 2 · offset + 1 with
        i ← i + 2 · offset do
15         CompareRegions(A[1 : n], i, i + offset, offset)
16     end
17     for i = offset + 1 to i = n − 3 · offset with
        i ← i + 2 · offset do
18         CompareRegions(A[1 : n], i, i + offset, offset)
19     end
20 end
```

First, it can be obtained that the values on the dotted line in Fig. 4(d) are always 0, because of the symmetry of Pascal's triangles. As adding a 0 does not change the value, the cases of Fig. 4(c) and Fig. 4(d) are equivalent regarding the area to the left of the dotted line.

```
Algorithm 9: OddevenMergeSort(A[1 : n])
 Input  : An encrypted array A[1 : n] with n elements
 Output: Sorted array of encrypted data A[1 : n]
1 m = 2^⌈log n⌉
2 Append m − n dummy elements to A[1 : n].
3 OddevenMergeRange(A[1 : m], 1, m).
4 Remove the m − n dummy elements at the end of A.
```

```
Algorithm 10: OddevenMergeRange(A[1 :
n], init, count)
 Input  : An encrypted array A[1 : n] with n elements,
          initial position init, subarray length count
 Output: Sorted array of encrypted data A[1 : n]
1 if count = 1 then
2     return
3 end
4 ℓ = count/2
5 OddevenMergeRange(A[1 : n], init, ℓ)
6 OddevenMergeRange(A[1 : n], init + ℓ, ℓ)
7 OddevenMergeMerge(A[1 : n], init, count, 1)
```

```
Algorithm 11: OddevenMergeMerge(A[1 :
n], init, count, step)
 Input  : An encrypted array A[1 : n] with n elements,
          initial position init, subarray length count,
          stride step step
 Output: Sorted array of encrypted data A[1 : n]
1  if count = 1 then
2      return
3  end
4  if 2 · step < count then
5      OddevenMergeMerge(A[1 :
        n], init, count, 2 · step)
6      OddevenMergeMerge(A[1 :
        n], init + step, count, 2 · step)
7      i = init + step
8      while i ≤ init + count − step do
9          SortTwo(A[i], A[i + step])
10         i = i + step
11     end
12 end
```

However, the values on both the dotted lines in Fig. 4(b) must be 0. To satisfy the other boundary condition $p_k(2m + k + 2, m) = 0$ on the upper dotted line in Fig. 4(b), we consider another Pascal's triangle translated by $-(k + 2)$ with $p_k(0, -(k+2)) = -1$. If we add these three Pascal's triangles P_{-1}, P_0, and P_1 shown in Fig. 4(e), there are zero boundary values on the lines from Q_1 to Q_2 and from R_1 to R_2. However, the boundary value after Q_2 or R_2 is not equal to 0. To obtain the boundary values on the lines from Q_2 to Q_3 and from

Algorithm 12: BitonicSort(A[1 : n])

```
Input  : An encrypted array A[1 : $n$] with $n$ elements
Output: Sorted array of encrypted data A[1 : $n$]
1 $m = 2^{\lceil \log n \rceil}$
2 Append $m - n$ dummy elements to A[1 : $n$].
3 BitonicRange(A[1 : $m$], 1, $m$, true).
4 Remove the $m - n$ dummy elements at the end of A.
```

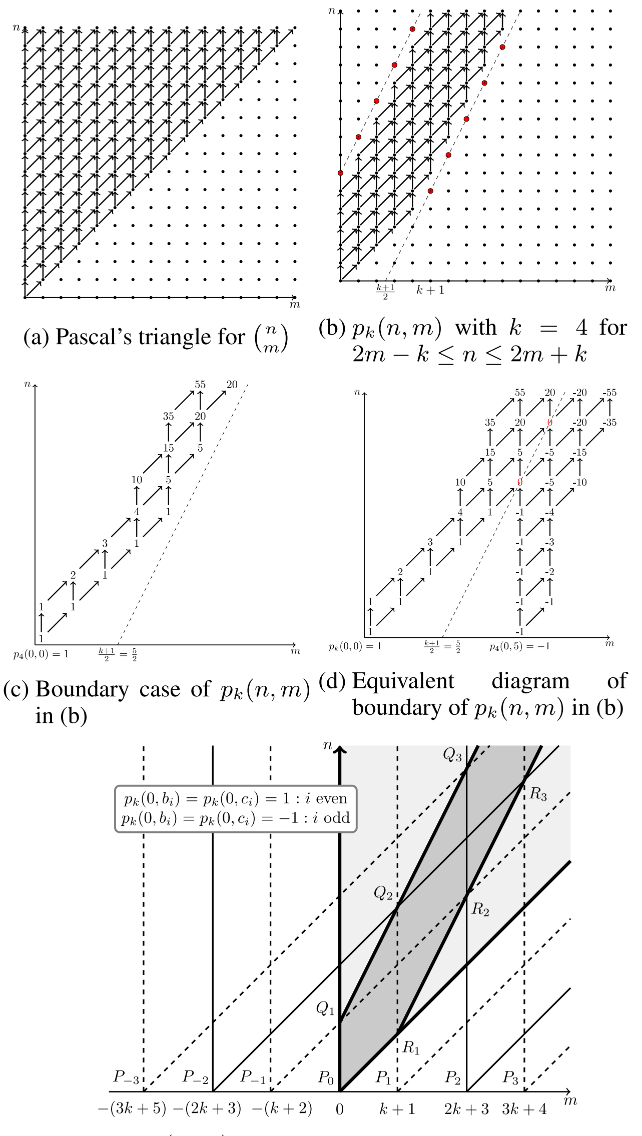
Algorithm 13: BitonicRange(A[1 : n], init, count, ascend)

```
Input  : An encrypted array A[1 : $n$] with $n$ elements,
         initial position init, subarray length count,
         Boolean variable ascend
Output: Sorted array of encrypted data A[1 : $n$]
1 if count = 1 then
2     return
3 end
4 BitonicRange(A[1 : $n$], init, count/2, true)
5 BitonicRange(A[1 : $n$], init + count/2, count/2, false)
6 BitonicMerge(A[1 : $n$], init, count, ascend)
```

Algorithm 14: BitonicMerge(A[1 : n], init, count, ascend)

```
Input  : An encrypted array A[1 : $n$] with $n$ elements,
         initial position init, subarray length count,
         Boolean variable ascend
Output: Sorted array of encrypted data A[1 : $n$]
1  if count = 1 then
2      return
3  end
4  for $i$ = init to $i$ = init + count/2 do
5      if ascend = true then
6          SortTwo(A[$i$], A[$i$ + count/2])
7      end
8      else
9          SortTwo(A[$i$ + count/2], A[$i$])
10     end
11 end
12 BitonicMerge(A[1 : $n$], init, count/2, ascend)
13 BitonicMerge(A[1 :
     $n$], init + count/2, count/2, ascend)
```

R_2 to R_3, we must add the Pascal's triangles P_2 and P_{-2}. Therefore, we repeat this process, as shown in Fig. 4(e). The sequence $\{b_i\}$ in Lemma 3 is the distance from the initial vertex of P_0 to that of P_i, while $\{c_i\}$ is the distance from the initial vertex of P_0 to that of P_{-i}. The initial value at the initial vertex of P_i is 1 if i is even, and -1 if i is odd. Q_i is defined as the intersection of the boundaries of the two Pascal's triangles starting from the initial vertices of P_{i-1} and P_{-i}, and R_i is defined as the intersection of the boundaries of the two Pascal's triangles starting from the initial vertices of P_i and $P_{-(i-1)}$.



n
m
(a) Pascal's triangle for $\binom{n}{m}$
$\frac{k+1}{2}$
$k+1$
(b) $p_k(n,m)$ with $k = 4$ for $2m - k \le n \le 2m + k$
55
20
35
15
5
10
4
3
2
1
$p_4(0,0) = 1$
$\frac{k+1}{2} = \frac{5}{2}$
(c) Boundary case of $p_k(n,m)$ in (b)
0
-20
-55
-35
-5
-15
-10
-1
-4
-3
-2
$p_4(0,5) = -1$
(d) Equivalent diagram of boundary of $p_k(n,m)$ in (b)
$p_k(0,b_i) = p_k(0,c_i) = 1 : i$ even
$p_k(0,b_i) = p_k(0,c_i) = -1 : i$ odd
Q_1
Q_2
Q_3
R_1
R_2
R_3
P_{-3}
P_{-2}
P_{-1}
P_0
P_1
P_2
P_3
$-(3k+5)$
$-(2k+3)$
$-(k+2)$
0
$k+1$
$2k+3$
$3k+4$
(e) $p_k(n,m)$ using overlapped Pascal's triangles

FIGURE 4. $p_k(n,m)$ **using Pascal's triangle.**

We establish that if the Pascal's triangles P_i's, $i = \cdots, -1, 0, 1, \cdots$, are overlapped, all of the integer points on the half-lines of $\overrightarrow{Q_1Q_2}$ and $\overrightarrow{R_1R_2}$ must be 0s. The integer points on the upper half-line of $\overrightarrow{Q_1Q_2}$ exhibit the form $n = 2m + k + 2$ for all non-negative integers m, and those on the lower half-line of $\overrightarrow{R_1R_2}$ exhibit the form $n = 2m - k - 1$ for all $m \ge k + 1$. First, in the case of the points on the half-line of $\overrightarrow{Q_1Q_2}$, we consider the integer points on $\overline{Q_jQ_{j+1}}$, which can be denoted as $n_1 = 2m_1 + k + 2$ and $b_{i-1} \le m_1 \le b_i$. Then, we can only consider Pascal's triangles $P_{-j}, \cdots, P_{j-1}$. Considering the parallel translation of each Pascal's triangle, the overlapped values on the points are defined as

$$\sum_{i=1}^{j}(-1)^i\binom{2m_1+k+2}{m_1+c_i}+\sum_{i=1}^{j}(-1)^{i-1}\binom{2m_1+k+2}{m_1-b_{i-1}}. \quad (11)$$

As $(m_1+c_i)+(m_1-b_{i-1}) = 2m_1+k+2$, $\binom{2m_1+k+2}{m_1+c_i} = \binom{2m_1+k+2}{m_1-b_{i-1}}$ holds, and (5) is equal to 0.

In the case of the points on the half-line of $\overrightarrow{R_1 R_2}$, we consider the integer points on $\overline{R_jR_{j+1}}$, which can be denoted as $n_2 = 2m_2 - k - 1$ and $b_i \le b_{i+1}$. Then, we can only consider Pascal's triangles $P_{j-1}, \cdots, P_j$. The overlapped values on the points are defined as

$$\sum_{i=1}^{j}(-1)^{i-1}\binom{2m_2-k-1}{m_1+c_{i-1}} + \sum_{i=1}^{j}(-1)^{i}\binom{2m_2-k-1}{m_1-b_i}. \tag{12}$$

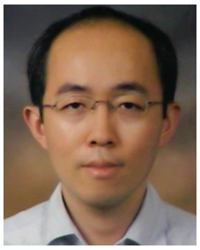
As $(m_2+c_{i-1})+(m_2-b_i) = 2m_2-k-1$, $\binom{2m_2-k-1}{m_1+c_{i-1}} = \binom{2m_2-k-1}{m_1-b_i}$ holds, and (6) is also equal to 0.

Therefore, we establish that with respect to the region between the two dotted lines in Fig. 4(b), Fig. 4(b) is exactly equivalent to the hashed part of Fig. 4(e). We obtain $p_k(n,m)$ by adding the values of points of several Pascal's triangles as in (1), where the first term is from the central Pascal's triangle P_0; the second term is from the right-side Pascal's triangles P_i's for the positive integer i; and the third term is from the left-side Pascal's triangles P_{-i}'s for the positive integer i.

REFERENCES

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.
[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. Symp. Theory Comput.*, vol. 9, 2009, pp. 169–178.
[3] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. EUROCRYPT*. Berlin, Germany: Springer, 2010, pp. 24–43.
[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, p. 13, 2014.
[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," IACR Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Tech. Rep. 2012/144, 2012, p. 144.
[6] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. CRYPTO*. Berlin, Germany: Springer, 2013, pp. 75–92.
[7] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. ASIACRYPT*. Cham, Switzerland: Springer, 2017, pp. 409–437.
[8] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Proc. Int. Symp. Cyber Secur. Cryptogr. Mach. Learn. (CSCML)*. Cham, Switzerland: Springer, 2021, pp. 1–19.
[9] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," IACR Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Tech. Rep. 2021/729, 2021, p. 729.
[10] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Proc. Cryptograph. Track RSA Conf.* Cham, Switzerland: Springer, 2020, pp. 364–390.
[11] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Proc. EUROCRYPT*. Cham, Switzerland: Springer, 2021, pp. 587–617.
[12] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, "High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *Proc. EUROCRYPT*. Cham, Switzerland: Springer, 2021, pp. 618–647.
[13] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 2, pp. 70–95, May 2018.
[14] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 941–956, Apr. 2021.
[15] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2020, pp. 1295–1309.
[16] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 4, pp. 114–148, Aug. 2021.
[17] A. A. Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "PrivFT: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226544–226556, 2020.
[18] A. Chatterjee and I. SenGupta, "Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective?" *IEEE Trans. Services Comput.*, vol. 13, no. 3, pp. 545–558, May/Jun. 2017.
[19] A. Chatterjee, M. Kaushal, and I. Sengupta, "Accelerating sorting of fully homomorphic encrypted data," in *Proc. Int. Conf. Cryptol. India*. Cham, Switzerland: Springer, 2013, pp. 262–273.
[20] N. Emmadi, P. Gauravaram, H. Narumanchi, and H. Syed, "Updates on sorting of fully homomorphic encrypted data," in *Proc. Int. Conf. Cloud Comput. Res. Innov. (ICCCRI)*, Oct. 2015, pp. 19–24.
[21] D. L. Shell, "A high-speed sorting procedure," *Commun. ACM*, vol. 2, no. 7, pp. 30–32, Jul. 1959.
[22] R. Sedgewick, "Analysis of shellsort and related algorithms," in *Proc. Eur. Symp. Algorithms*. Berlin, Germany: Springer, 1996, pp. 1–11.
[23] A. Chatterjee and I. Sengupta, "Windowing technique for lazy sorting of encrypted data," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Sep. 2015, pp. 633–637.
[24] M. T. Goodrich, "Randomized shellsort: A simple data-oblivious sorting algorithm," *J. ACM*, vol. 58, no. 6, p. 27, 2011.
[25] M. Ciura, "Best increments for the average case of shellsort," in *Proc. Int. Symp. Fundam. Comput. Theory*. Berlin, Germany: Springer, 2001, pp. 106–117.
[26] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Proc. EUROCRYPT*. Cham, Switzerland: Springer, 2018, pp. 360–384.
[27] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 1–58, 2018.
[28] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. London, U.K.: Pearson, 1997.
[29] T. O. Espelid, "Analysis of a shellsort algorithm," *BIT*, vol. 13, no. 4, pp. 394–400, Dec. 1973.
[30] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[31] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. (Aug. 2016). *TFHE: Fast Fully Homomorphic Encryption Library*. [Online]. Available: https://tfhe.github.io/tfhe/.
[32] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, "Efficient sorting of homomorphic encrypted data with k-way sorting network," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 4389–4404, Aug. 2021.

JOON-WOO LEE received the B.S. degree in electrical and computer engineering from Seoul National University, Seoul, South Korea, in 2016, where he is currently pursuing the Ph.D. degree. His current research interests include homomorphic encryption and lattice-based cryptography.



YOUNG-SIK KIM (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer science from Seoul National University, in 2001, 2003, and 2007, respectively. He joined the Semiconductor Division, Samsung Electronics, where he worked in the research and development of security hardware IPs for various embedded systems, including modular exponentiation hardware accelerator (called Tornado 2MX2) for RSA and elliptic-curve cryptography in smart-card products and mobile application processors of Samsung Electronics, until 2010. He is currently a Professor with Chosun University, Gwangju, South Korea. He is also a Submitter of two candidate algorithms (McNie and pqsigRM) in the first round for the NIST Post Quantum Cryptography Standardization. His research interests include post-quantum cryptography, the IoT security, physical-layer security, data hiding, channel coding, and signal design. He is selected as one of the 2025's 100 Best Technology Leaders (for crypto-systems) by the National Academy of Engineering of Korea.

JONG-SEON NO (Fellow, IEEE) received the B.S. and M.S.E.E. degrees in electronics engineering from Seoul National University, Seoul, South Korea, in 1981 and 1984, respectively, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 1988. He was a Senior MTS with Hughes Network Systems, from 1988 to 1990. He was an Associate Professor with the Department of Electronic Engineering, Konkuk University, Seoul, from 1990 to 1999. He joined the Faculty of the Department of Electrical and Computer Engineering, Seoul National University, in 1999, where he is currently a Professor. His research interests include error-correcting codes, cryptography, sequences, LDPC codes, interference alignment, and wireless communication systems. He became an IEEE Fellow through the IEEE Information Theory Society, in 2012. He became a member of the National Academy of Engineering of Korea (NAEK), in 2015, where he served as the Division Chair of Electrical, Electronic, and Information Engineering, from 2019 to 2020. He was a recipient of the IEEE Information Theory Society Chapter of the Year Award, in 2007. From 1996 to 2008, he served as the Founding Chair of the Seoul Chapter of the IEEE Information Theory Society. He was the General Chair of Sequence and Their Applications 2004 (SETA 2004), Seoul. He also served as the General Co-Chair of the International Symposium on Information Theory and Its Applications 2006 (ISITA 2006) and the International Symposium on Information Theory 2009 (ISIT 2009), Seoul. He served as the Co-Editor-in-Chief of the IEEE Journal of Communications and Networks, from 2012 to 2013.