SIMD-style Sorting of Integer Sequence in RLWE Ciphertext

Zijing Li, Hongbo Li, and Zhengyang Wang

Abstract—This article discusses fully homomorphic encryption and homomorphic sorting. Homomorphic encryption is a special encryption technique that allows all kinds of operations to be performed on ciphertext, and the result is still decryptable, such that when decrypted, the result is the same as that obtained by performing the same operation on the plaintext. Homomorphic sorting is an important problem in homomorphic encryption. Currently, there has been a volume of work on homomorphic sorting. In these works, each integer in a sequence is encrypted in a separate ciphertext, there is a lack of research on sorting sequences of integers encrypted in a single ciphertext. This paper addresses the sorting problem by utilizing Single Instruction Multiple Data (SIMD) technology to provide new algorithms to improve computational efficiency. The content includes the following aspects. For plaintexts encrypted word-wise, this paper studies sorting an integer sequence stored in one or multiple ciphertexts, and proposes a new SIMD-style homomorphic sorting algorithm. On theoretical complexity, compared with three existing sorting algorithms, namely, homomorphic sorting by polynomial computation over a finite field, by TFHE bootstrapping, or by Liu-Wang parallel bootstrapping, the new algorithm achieves a speedup of $O((\log n)^2)$, $O(n(\log n)^3)$, and $O((\log n)^4)$, respectively, for sorting a plaintext integer sequence of length n. By experimental results, the new algorithm is 1.7-9.2 times faster than the three sorting algorithms. The third situation involves sorting multiple shorter sequences simultaneously, all of which can be stored in a single ciphertext. For this situation, this paper proposes a method for calculating the ord function, and uses this method to provide a new sorting algorithm. On theoretical complexity, if the total number of numbers to be sorted is n and there are n^r numbers in each sequence, the new algorithm is faster than three existing sorting algorithms, with speed-ups of $O(n^{1-r}(\log n)^2)$, $O(n^{2-r}(\log n)^3)$, and $O(n^{1-r}(\log n)^4)$, respectively. By experimental results, the new algorithm is 2.1-6.4 times faster than existing sorting algorithms.

Index Terms—

1 INTRODUCTION

S orting of a sequence of *n* unordered numbers is a basic routine task. Early sorting algorithms such as insertion sort [1], selection sort [1], have time complexity $O(n^2)$. The fastest sorting algorithms in the literature have time complexity $O(n \log n)$, such as quick sort [2], merge sort [3], and heap sort [4]. For example, merge sort is to divide the input sequence into two subsequences, first sort each subsequence recursively by calling merge sort, and then merge the two ordered subsequences into a whole ordered sequence.

There are also algorithms of time complexity $O(n \log^2 n)$. For example, odd-even merge sort [5] is similar to merge sort, with the difference lying in the merge procedure: it runs the merge procedure recursively first to the subsequence of odd-positioned entries, then to the subsequence of even-positioned entries, and finishes the whole merge procedure by re-ordering every odd-positioned entry and its succeeding even-positioned entry.

Th today's digital age, data privacy protection is becoming increasingly important. Fully homomorphic encryption (FHE) is an important cryptological technique that allows encrypted data to be computed just as if they are plaintexts. When each entry of an unordered sequence is encrypted as a

 Zijing Li, Hongbo Li and Zhengyang Wang are with Academy of Mathematics and Systems Science, Chinese Academy of Science, Beijing, China. E-mail: lizijing@amss.ac.cn, hli@mmrc.iss.ac.cn, wangzhengyang22@mails.ucas.ac.cn

Hongbo Li is the corresponding author.

ciphertext in an FHE scheme, the classical sorting algorithms can be converted to homomorphic sorting algorithms on the sequence of ciphertexts.

While in classical sorting, the time complexity is dominated by the number of comparisons between two entries and the number of entry movings, in homomorphic sorting, the time complexity is dominated by the number of calls of homomorphic comparison.

In 2013, Chatterjee et al. [6] extended bubble sort and insertion sort to homomorphic sorting. In 2015, Çetin et al. [7] extended odd-even merge sort to homomorphic sorting. In 2021, Lee and Kim [8] extended shell sort [9] to homomorphic sorting. Among all these extensions, the homomorphic odd-even merge sort requires the least number of comparisons: $O(n(\log n)^2)$.

Different homomorphic comparison algorithms have different time complexities.

plaintext encoding is mainly divided into three categories: the first category is to decompose plaintext integers into bits and then encrypt each bit one by one.

The second category is to encrypt the plaintext integers as a whole. The advantage of this method is that it can directly perform homomorphic algebraic operations, but it is more complex for comparing sizes.

The third category also encrypts the entire plaintext, but the plaintext can be real numbers.

For integers encoded word-wise, in 2016, Kim et al. [10] presented a method of using polynomials over finite fields to represent comparison functions.

In 2020, Tan et al. [11] proposed a method of using bivariate polynomial interpolation for comparison, whose interpolation method generates polynomials with smaller degrees.

In 2020, Çetin et al. [12] used surrogate polynomials to give a better way to calculate comparison function results.

In 2021, Ilia and Zucca [13] improved Tan's method and further reduced the degree of bivariate polynomials.

In addition to using polynomial methods, comparing two numbers using function bootstrapping is feasible. In 2019, Chillotti, Ligier, Orfila, and Tap [14] proposed a method that can homomorphically calculate the sign function using the function bootstrapping. In 2021, Kamil and Leonard [15] improved Chillotti's method with better error control at the cost of more bootstrapping times. In 2022, Micianccio, Polyakov, and Liu [16] improved Kamil's method and Chillotti's method by completing the calculation of the sign function with at most $3\lfloor \frac{\log q}{\log p} \rfloor + 1$ bootstraps for LWE ciphertexts with plaintext modulus p and ciphertext modulus q, while reducing error growth.

For plaintexts encoded bit-wise, comparing two integers only requires comparing them bit by bit, starting from the highest bit. In 2017, Narumanchi et al. [17] compared different sorting algorithms and showed that bitwise encryption of plaintexts makes homomorphic sorting more efficient than wordwise encryption. This encoding method has great advantages in comparing sizes but is not conducive to general algebraic operations such as addition and multiplication. For real number plaintexts, polynomial approximation is the main comparison method.

In homomorphic encryption, there has been a lot of research on ciphertext sorting, but these works have one thing in common: storing the numbers being compared in two different ciphertexts.

if an integer sequence is stored entirely in one ciphertext and then sorted, can the sorting efficiency be improved based on this SIMD encoding?

This paper studies how to perform SIMD homomorphic sorting on plaintext integer sequences to improve algorithm efficiency.

Firstly, for the case of plaintext overall encryption, this article examines how to sort an integer sequence stored in a single ciphertext. This article proposes a new function ord, proves its several properties, gives its homomorphic calculation method, and thus proposes a new SIMD sorting algorithm. From the perspective of theoretical complexity, for sorting a plaintext integer sequence of length n, this article's algorithm is compared with three sorting algorithms based on different comparison methods, namely sorting algorithms based on polynomial comparison on finite fields, sorting algorithms based on TFHE bootstrapping, and sorting algorithms based on Liu-Wang parallel bootstrapping. The speedup is of the order of $O((logn)^2)$, $O(n(logn)^3)$, and $O((logn)^4)$ respectively. From the experimental results, for the setting of conventional parameters, this algorithm is 3.0-8.2 times faster than these sorting algorithms.

For the case of plaintext overall encryption, this article also considers two more complex situations. The second situation is that the sequence is too long and needs to be stored in multiple ciphertexts. For this purpose, this article proposes a new algorithm for reallocating integers stored in two plaintext polynomials so that smaller numbers can be placed in front of ciphertexts according to the size relationship of integers, thus obtaining a SIMD sorting algorithm for this situation. From the perspective of theoretical complexity, the new algorithm has a speedup of $O((logn)^2)$, $O(n(logn)^3)$ and $O((logn)^4)$ respectively compared with the three existing sorting algorithms. From the experimental results, this algorithm is 0.7-2.6 times faster than existing sorting algorithms.

The third situation requires multiple short sequences to be sorted separately, all of which are packed and stored in a single ciphertext. This article uses SIMD technology to give a method for homomorphic calculation of ord function for these short sequences at the same time and gives a new algorithm using this method. From the perspective of theoretical complexity, if the total number of numbers to be sorted is n, each short sequence has n^r numbers. This article's algorithm is compared with three existing sorting algorithms based on different comparison methods. The speedup is of the order of $O(n^{1-r}(\log n)^2)$, $O(n^{2-r}(\log n)^3)$ and $O(n^{1-r}(\log n)^4)$ respectively. From the experimental results, it is 2.1-6.4 times faster than existing sorting algorithms.

Section 2 mainly introduces the homomorphic RLWE ciphertext and its SIMD properties. Section 3 introduces existing plaintext sorting algorithms, the calculation method of ciphertext comparison functions and ciphertext sorting algorithms. Section 4 gives homomorphic sorting algorithms for three different situations under whole encryption. Section 5 gives homomorphic sorting algorithms for three different situations under *p*-bit encryption. Section 6 introduces experimental results. Section 7 summarizes the full text and proposes prospects for future work.

2 BACKGROUND

For some parameters that appear in this paper, in order to measure and compare the complexity between different operations and algorithms, they are listed here and quantified uniformly based on parameter n as follows:

$$q = poly(n), \ p = O(1), \ l = O(n)$$
 (2.1)

The polynomial multiplication mentioned in this paper for measuring complexity refers to the multiplication of two *n*-degree polynomials.

2.1 SIMD

In [18], Smart and Vercauteren gave a way to batch several messages in a finite field into one polynomial. Let m be a positive integer and $n = \varphi(m)$ where φ is Euler's totient function. Let $\mathcal{R} = \mathbb{Z}[X]/\langle \Phi_m(X) \rangle$ where $\Phi_m(X)$ is the m-th cyclotomic polynomial. Let p > 1 be a prime number coprime to m, and d be the order of p modulo m. By Euler's theorem, d|n. Let l = n/d, then $\Phi_m(X)$ splits modulo p into l irreducible factors of same degree d, i.e. $\Phi_m(X) = F_1(X) \cdots F_l(X) \mod p$. In this case, the following ring isomorphism holds:

$$\mathcal{R}_{p} = \mathbb{Z}_{p}[X] / \langle \Phi_{m}(X) \rangle$$
$$\cong \mathbb{Z}_{p}[X] / \langle F_{0}(X) \rangle \bigotimes \cdots \bigotimes \mathbb{Z}_{p}[X] / \langle F_{l-1}(X) \rangle$$
(2.2)

For each $0 \leq i \leq l-1$, the quotient ring $\mathbb{Z}_p[X]/\langle F_i(X) \rangle$ is isomorphic to the finite field \mathbb{F}_{p^d} . Hence, the above isomorphism can be rewritten as $\mathcal{R}_p \cong \mathbb{F}_{p^d}^l$. Thus, for a vector with l messages $\vec{a} = (a_0, \ldots, a_{l-1})$ with each $a_i \in \mathbb{F}_{p^d}$, by applying inverse Chinese Remainder Theorem(CRT), we derive a single message a in \mathcal{R}_p . We write this as:

$$\mathbf{a} = \operatorname{CRT}^{-1}(\vec{a}) = \operatorname{CRT}^{-1}(a_0, \dots, a_{l-1}); \ \vec{a} = \operatorname{CRT}(\mathbf{a})$$
(2.3)

Every copy of \mathbb{F}_{p^d} in this direct product is called a *slot*. Therefore, every element of \mathcal{R}_p contains *l* slots.

Additions and multiplications of \mathcal{R}_p -elements results in the corresponding coefficient-wise operations of their respective slots. For example, given another message vector $\vec{b} = (b_0, \dots, b_{l-1})$ and $\mathbf{b} = \text{CRT}^{-1}(\vec{b})$,

$$CRT(\mathbf{a} + \mathbf{b}) = (a_0 + b_0, \dots, a_{l-1} + b_{l-1})$$

$$CRT(\mathbf{ab}) = (a_0 \times b_0, \dots, a_{l-1} \times b_{l-1})$$
(2.4)

2.2 RLWE ciphertext

RLWE ciphertexts can be constructed as follows:

- 1) Choose plaintexts module p and security parameter λ .
- 2) According to the security parameter λ , choose integers q and n, and a discrete Gaussian distribution χ on \mathcal{R} are generated, where $\mathcal{R} = \mathbb{Z}[X]/\langle \Phi_m(X) \rangle$, $\Phi_m(X)$ is the *m*-th cyclotomic polynomial, and $n = \varphi(m)$.
- Randomly and uniformly select a polynomial s ∈ *R_q* as the symmetric encryption key.
- 4) For $\mathbf{m} \in \mathcal{R}_p$, select $\mathbf{a} \leftarrow \mathcal{R}_q$ according to the uniform distribution, and select the error $\mathbf{e} \leftarrow \chi$ according to the distribution χ . Encrypt it symmetrically as:

$$\begin{aligned} \operatorname{ct}(\mathbf{m}) &= \operatorname{Enc}_{\mathbf{s}}(\mathbf{m}) \\ &= \begin{cases} (\mathbf{a}, \mathbf{as} + p\mathbf{e} + \mathbf{m} \mod q) & (\operatorname{BGV}) \\ (\mathbf{a}, \mathbf{as} + \mathbf{e} + \lfloor q/p \rfloor \mathbf{m} \mod q) & (\operatorname{BFV}) \end{cases} \end{aligned}$$

The first step in decrypting the ciphertext (\mathbf{a}, \mathbf{b}) is to compute:

$$Dec_{\mathbf{s}}(\mathbf{a}, \mathbf{b}) = \mathbf{b} - \mathbf{as} \mod q$$

Then clear the error **e**.

RLWE ciphertext supports addition and multiplication operations. Among them, multiplication requires the operation of changing the key. Specifically, prepare the encryption $\text{Encs}'(\mathbf{s})$ of the key \mathbf{s} under the new key \mathbf{s}' in advance. For the ciphertext $\text{ct}(\mathbf{a}) = \text{Encs}(\mathbf{a})$ encrypted with the key \mathbf{s} , use $\text{Encs}'(\mathbf{s})$ to convert it to ciphertext $\text{ct}(\mathbf{a}) = \text{Encs}'(\mathbf{a})$. As for the multiplication of RLWE ciphertext, its main operation is to convert the key \mathbf{s}^2 back to the original key \mathbf{s} . The specific operation and complexity are given in BGV [19] and BFV [20], [21], which are given as a lemma below:

lemma **1**. For two RLWE ciphertexts $ct(\mathbf{a}) = Encs(\mathbf{a})$ and $ct(\mathbf{b}) = Encs(\mathbf{b})$, their ciphertext multiplication obtains ciphertext $ct(\mathbf{ab}) = Enc_s(\mathbf{ab})$, which requires calculating $O(\log q)$ *n*-degree polynomial multiplications.

Using the setting of (2.1), q = poly(n), so the complexity of calculating an RLWE ciphertext multiplication is equal to

calculating $O(\log n)$ polynomial multiplications. Note that the complexity estimate in this paper counts the number of multiplications of ordinary univariate *n*-degree polynomials.

For the sake of concise expression, this paper uses the usual addition and multiplication notation, that is, $\operatorname{ct}(\mathbf{a}) + \operatorname{ct}(\mathbf{b})$ and $\operatorname{ct}(\mathbf{a}) \cdot \operatorname{ct}(\mathbf{b})$, to represent the addition and multiplication of ciphertexts.

Combining the SIMD technology mentioned earlier, l elements in $\mathbb{F}p^d$ can be encrypted into one ciphertext. The specific operation is as follows: for l elements $a_0, a_1, \ldots, al - 1$ in $\mathbb{F}p^d$, first use SIMD technology to put these elements into the slots of $\mathbf{a} = \operatorname{CRT}^{-1}(a_0, a_1, \ldots, al - 1)$, and then encrypt the polynomial \mathbf{a} with RLWE to obtain $\operatorname{ct}(\mathbf{a})$.

2.3 Permutation among Message Slots

In [22], Gentry, Halevi and Smart gave a method for slot elements of homomorphic permutation plaintext polynomials. Using Galois transformation, for ciphertext ct(**a**), where **a** = $\operatorname{CRT}^{-1}((a_0, \ldots, a_{l-1}))$, for any permutation $\pi \in S_l$, a new ciphertext ct(**a**') can be obtained such that **a**' = $\operatorname{CRT}^{-1}((a_{\pi(0)}, \ldots, a_{\pi(l-1)}))$. Using the key switching technology, it can be ensured that the new ciphertext obtained has the same key as the initial ciphertext, thus supporting subsequent operations.

The paper mainly uses two permutations. One is left shift, which converts ciphertext $ct(\mathbf{a})$ to $ct(\mathbf{a}') = Encs(\mathbf{a}')$, where $\mathbf{a}' = CRT^{-1}((a_1, \ldots, a_{l-1}, a_0))$. The other is right shift, which converts ciphertext $ct(\mathbf{a})$ to $ct(\mathbf{a}'')$, where $\mathbf{a}'' = CRT^{-1}((a_{l-1}, a_0, \ldots, a_{l-3}, a_{l-2}))$. In the algorithm description of this paper, use **LShift**($ct(\mathbf{a})$) to represent left shift and use **RShift**($ct(\mathbf{a})$) to represent right shift. The following lemma is from [22].

lemma **2.** The complexity of one left shift (**LShift**) is $O(\log q)$ polynomial multiplications. The complexity of the right shift (**RShift**) is the same as that of the left shift.

Using the setting of (2.1), q = poly(n), so the complexity of one left shift or right shift is equal to calculating $O(\log n)$ polynomial multiplications.

2.4 Format conversion

For the information in the plaintext polynomial slot, it can be extracted by the Fourier transform, and the reverse operation can be performed by the Fourier inverse transform. This operation can also be performed in the ciphertext case. This format conversion is necessary for bootstrapping. Cheon, Han, Kim, Kim and Song [23] used this transformation in their bootstrapping scheme and gave an estimate of the complexity, see the following lemma.

lemma 3. The information can be converted between slots and coefficients by Fourier transform and its inverse transform, with a complexity of $O(\sqrt{n} \log n)$.

2.5 Bootstrapping

In 2009, Gentry first proposed the bootstrapping framework and used it to construct the first fully homomorphic encryption system. Bootstrapping refers to the operation of homomorphic decryption of the ciphertext corresponding to the private key when the homomorphic algebraic calculation reaches the error threshold and cannot be performed again. After bootstrapping, low-noise ciphertexts that support continued homomorphic operations can be obtained. In 2014, Alperin-Sheriff and Peikert [24] proposed a fast bootstrapping method, which made breakthroughs in security and practicality. In 2015, Ducas and Micciancio [25] used the ideas of Alperin-Sheriff et al. to propose the FHEW bootstrapping scheme, which can bootstrap a singlebit ciphertext in less than one second. In 2019, Chillotti, Ligier, Orfila and Tap [14] designed the TFHE scheme, which further improved the speed of bootstrapping. Currently, TFHE's bootstrapping is a widely used bootstrapping scheme.

In the TFHE bootstrapping, the following parameters are required:

- a large RLWE/RGSW module *Q*;
- the dimension *N*;
- *B_g*, the integer decomposition base required in bootstrapping, decomposes the integer modulo *Q* into *d_g* digits.

The following lemma comes from [16]:

lemma 4. For parameters Q = O(q) = O(n), N = O(n), $d_g = O(\log n)$ and an *n*-dimensional LWE ciphertext input, the complexity of one TFHE bootstrapping is $2n(d_g + 1) = O(n \log n)$ polynomial multiplications.

In 2023, Liu and Wang [26], [27] proposed parallel bootstrapping technology that can bootstrap multiple LWE ciphertexts in parallel.

lemma 5. For parameter settings $\rho = O(1), 2d^{\rho} = r, h = 2d, v = O(n^{0.04}), v' = (2d)^{\rho-1}/v, w = 2d \log q$, when r LWE ciphertexts are input, n bootstrapped LWE ciphertexts can be output through v' VetMatMult and one HomDFT⁻¹. One VetMatMult requires $O((w + vh) \log n)O(\log n)$ polynomial multiplications, and one HomDFT⁻¹ requires $O(\rho(2d)^{\rho}) \log n)O(\log n) = O(r(\log n)^2)$ polynomial multiplications.

Because this bootstrapping can bootstrap r ciphertexts at once, the average number of polynomial multiplications required for bootstrapping one ciphertext is only $O((\log n)^2)$.

3 SORTING

3.1 Sorting for Plaintexts

Sorting algorithms are a fundamental class of computer science algorithms, primarily aimed at arranging data according to specific rules. Common sorting algorithms include insertion sort [1], selection sort [1], quick sort [2], merge sort [3], and odd-even merge sort [5], among others.

The idea behind insertion sort is to insert each unsorted element into its appropriate position within the already sorted sequence, continuing this process until all data elements are inserted. Its advantages include simplicity of implementation and good performance on nearly sorted data. However, for data with a high degree of inversions, the time complexity of insertion sort becomes significantly high. Generally, the time complexity of this algorithm is $O(n^2)$, making it seldom used for sorting particularly large sequences.

Selection sort operates by repeatedly finding the minimum value from the unsorted portion of the sequence and placing it at the end of the sorted portion. Its benefits include simple implementation and low space complexity. However, similar to insertion sort, it has a high time complexity of $O(n^2)$.

Merge sort works by dividing the sequence to be sorted into several subsequences, each of which is already sorted. It then merges these subsequences pairwise until a single sorted sequence is obtained. The advantage of merge sort is its relatively low time complexity of $O(n \log n)$, making it highly efficient for sorting large-scale data. However, its implementation is more complex and requires additional space to store intermediate results.

Quick sort selects a pivot element and partitions the sequence such that elements less than the pivot are placed on the left, and elements greater than the pivot are placed on the right. It then recursively applies quick sort to the left and right subsequences. The advantages of quick sort include its relatively low time complexity of $O(n \log n)$ and often faster actual running times compared to merge sort, making it widely practical. However, in extreme cases, the time complexity of quick sort may degrade.

Odd-even merge sort is an algorithm based on merge sort. It separately compares and swaps elements at odd and even positions, and then performs merge sort. This algorithm has a $O(n \log^2 n)$ complexity but performs better than traditional merge sort when sorting smaller sequences. Additionally, the primary steps of this sorting method involve comparisons and swaps, unlike merge sort and quick sort, which require selecting target numbers based on the results of comparisons.

3.2 Homomorphic comparison by using polynomials on finite fields

This paper set $\mathbb{Z}_p = [-\frac{p}{2}, \frac{p}{2}) \cap \mathbb{Z}$, $\mathbb{Z}_{p^k} = [-\frac{p^k}{2}, \frac{p^k}{2}) \cap \mathbb{Z}$. For $x \in \mathbb{Z}_p$, define:

$$\operatorname{sgn}(x) = \begin{cases} 1, \ x \in [-\frac{p}{2}, 0) \\ 0, \ x \in [0, \frac{p}{2}) \end{cases}$$

For $x, y \in \mathbb{Z}_p$, define **EQ**(equal) as follows:

$$\mathbf{EQ}(x,y) = \begin{cases} 1, \ x = y \\ 0, \ x \neq y \end{cases}$$

In this paper, we define: for $x, y \in \mathbb{Z}_p$, x < y if and only if: either $\operatorname{sgn}(y) = 0$ and $\operatorname{sgn}(x) = 1$, or $\operatorname{sgn}(y) = \operatorname{sgn}(x)$ and $\operatorname{sgn}(y - x) = 1$.

For $x, y \in \mathbb{Z}_p$, define LT(less than) as follows:

$$\mathbf{LT}(x,y) = \begin{cases} 1, \ x < y \\ 0, \ x \ge y \end{cases}$$

For vectors $\vec{x} = (x_0, \dots, x_{l-1})$ and $\vec{y} = (y_0, \dots, y_{l-1})$, we define:

$$LT(\vec{x}, \vec{y}) = (LT(x_0, y_0), \dots, LT(x_{l-1}, y_{l-1}))$$

$$\mathbf{EQ}(\vec{x}, \vec{y}) = (\mathbf{EQ}(x_0, y_0), \dots, \mathbf{EQ}(x_{l-1}, y_{l-1}))$$

For prime p > 5 and $x, y \in \mathbb{Z}_p$, Iliashenko and Zucca [13] gave polynomials to evaluate **LT** and **EQ** over \mathbb{Z}_p :

$$EQ(x,y) = 1 - (x - y)^{p-1}$$
(3.1)

$$\mathbf{LT}(x,y) = y^{p-1} - \frac{p-1}{2} (xy)^{\frac{p-1}{2}} + \sum_{\substack{i,j>0, i\neq j, \ k=0 \\ i+j \le p}} \sum_{k=0}^{p-2} \sum_{l=k+1}^{p-1} k^{p-1-i} l^{p-1-j} x^{i} y^{j}$$
(3.2)

For RLWE ciphertexts, the degree of the polynomial representation of EQ is p - 1. According to Lemma 1, the complexity of each RLWE ciphertext multiplication is $O(\log n)$ polynomial multiplications. According to the parameter selection of the formula (2.1), the complexity of calculating **EQ** is

$$\log(p-1)O(\log n) = O(\log n) \tag{3.3}$$

polynomial multiplications. The degree of the polynomial representation of LT function is p, the complexity of calculating LT is

$$(\log p)O(\log n) = O(\log n) \tag{3.4}$$

polynomial multiplications.

3.3 Hommorphic comparison by using bootstrapping

For $x, y \in \mathbb{Z}_p$, comparing x and y is equivalent to the following operation: first, calculate the sign function of x and y respectively. If they have different signs, if $\operatorname{sgn}(x) = 1$, then x < y; if $\operatorname{sgn}(x) = 0$, then x > y. If they have the same sign, then calculate the sign of x - y. It can be expressed as follows:

$$\mathbf{LT}(x,y) = (1-z) \cdot \operatorname{sgn}(x-y) + z \cdot \operatorname{sgn}(x),$$

$$z = (\operatorname{sgn}(x) - \operatorname{sgn}(y))^2$$
(3.5)

That is to say, three sign function calculations are required in total, plus four homomorphic multiplications.

Chillotti, Gama, Georgieva and Izabach'ene's [28] method of functional bootstrapping can be used to calculate any function. Liu, Micciancio and Polyakov [16] used this method to give a way to homomorphically compute the sign function. The following lemma is from [16].

lemma 6. When the input is an LWE ciphertext $(c, d) \in \mathbb{Z}q^{n+1}$ encrypted with plaintext $\text{pt} \in \mathbb{Z}p$, homomorphically computing the sign function **HomSign** requires at most $3\lfloor \frac{\log q}{\log p} \rfloor + 1$ functional bootstrappings.

Two methods of bootstrapping are introduced in Section 2. The first is TFHE bootstrapping, which can only bootstrap one ciphertext at a time. According to Lemma 4, it requires $O(n \log n)$ polynomial multiplications. Therefore, using this bootstrapping method, the overall complexity of sign function is:

$$3(3\lfloor \frac{\log q}{\log p} \rfloor + 1)O(n\log n) = O(n(\log n)^2)$$
(3.6)

polynomial multiplications. The other is the parallel bootstrapping proposed by Liu and Wang. According to Lemma 5, on average, only $O(\log^2 n)$ polynomial multiplications are required for bootstrapping one ciphertext. Therefore, using this method, the overall complexity of sign function is:

$$3(3\lfloor \frac{\log q}{\log p} \rfloor + 1)O(\log^2 n) = O((\log n)^3)$$
(3.7)

polynomial multiplications.

3.4 Homomorphic Sorting

The method that requires the least number of comparisons for sorting n ciphertexts that encrypt a single plaintext is given by Çetin, Dor"oz, Sunar and Savaş [7]. The method is to directly transplant the odd-even merge sort in plaintext to homomorphic ciphertext. The specific process is shown in the Algorithm 1.

Algorithm 1 Odd-Even Merge Sort (a_0, \ldots, a_{n-1})
Input: $a_0,, a_{n-1}, n \ge 2$ is a power of 2
Output: d_0, \ldots, d_{n-1} , where d_0, \ldots, d_{n-1} is a permutation
of a_0, \ldots, a_{n-1} , and $d_0 \leq d_1 \leq \cdots \leq d_{n-1}$
1: $\{b_i\} \leftarrow \text{Odd-Even Merge Sort}(\{a_0, \dots, a_{\frac{n}{2}-1}\})$
2: $\{c_i\} \leftarrow Odd\text{-Even Merge Sort}(\{a_{\frac{n}{2}}, \dots, a_{n-1}\})$
3: $\{d_i\} \leftarrow Odd\text{-}Even\;Merge(b_i,c_i)^{-2}$

Algorithm 2 Odd-Even Merge(a_0, \ldots, a_{n-1})

- **Input:** a_0, \ldots, a_{n-1} whose two halves $\{a_0, \ldots, a_{\frac{n}{2}-1}\}$ and $\{a_{\frac{n}{2}}, \ldots, a_{n-1}\}$ are non-decreasing
- **Output:** d_0, \ldots, d_{n-1} where d_0, \ldots, d_{n-1} is a permutation of a_0, \ldots, a_{n-1} and $d_0 \le d_1 \le \cdots \le d_{n-1}$
- 1: $\{b_i, i \mod 2 \equiv 0\} \leftarrow \text{Odd-Even Merge}(\{a_0, a_2, \dots, a_{n-2}\})$
- 2: $\{c_i, i \mod 2 \equiv 1\} \leftarrow \text{Odd-Even Merge}(\{a_1, a_3, \dots, a_{n-1}\})$
- 3: for i in $\{1, 3, 5, 7, \dots, n-3\}$ do

4:
$$d_i \leftarrow \mathbf{LT}(b_i, c_{i+1}) \cdot b_i + (1 - \mathbf{LT}(b_i, c_{i+1})) \cdot c_{i+1}$$

5:
$$d_{i+1} \leftarrow \mathbf{LT}(b_i, c_{i+1}) \cdot c_{i+1} + (1 - \mathbf{LT}(b_i, c_{i+1})) \cdot b_i$$

6: end for

The following lemma comes from [7].

lemma 7. Using Algorithm 1 to sort k ciphertexts requires $O(k(\log k)^2)$ LT comparisons.

Three different ciphertext comparison methods were mentioned in the previous section, so there are three different methods for implementing odd-even merge sort on ciphertexts. If formula (3.2) is used to implement the LT function, this method is called a sorting algorithm based on polynomial comparison on finite fields; if TFHE bootstrapping is used to implement the LT function, this method is called the sorting algorithm based on TFHE bootstrapping; if parallel bootstrapping is used to implement the LT function, this method is called the sorting algorithm based on parallel bootstrapping. Next, for these three different sorting algorithms, their complexity for sorting k ciphertexts is given. For the sorting algorithm based on polynomial comparison on finite fields, for plaintext encrypted as a whole, according to the formula (3.4), the complexity of one LT comparison is $O(\log n)$ polynomial multiplications. Combined with Lemma 7, the complexity of sorting *k* ciphertexts using this algorithm can be obtained as:

$$O(k(\log k)^2)O(\log n) = O(k(\log k)^2 \log n)$$
 (3.8)

polynomial multiplications.

For the sorting algorithm based on TFHE bootstrapping, according to formula (3.6), the complexity of one **LT** comparison is $O(n(\log n)^2)$ polynomial multiplications. Combined with Lemma 7, the complexity of sorting *k* ciphertexts using this algorithm can be obtained as:

$$O(k(\log k)^2)O(n(\log n)^2) = O(kn(\log k)^2 \log^2 n)$$
(3.9)

polynomial multiplications.

For the sorting algorithm based on parallel bootstrapping, by formula (3.2), the complexity of one LT comparison is $O((\log n)^3)$ polynomial multiplication. Combined with Lemma 7, it can be obtained that the complexity of this algorithm for sorting *k* ciphertexts is

$$O(k(\log k)^2)O((\log n)^3) = O(k(\log k)^2(\log n)^3)$$
 (3.10)

polynomial multiplications.

4 HOMOMORPHIC SORTING WITH WORD-WISE EN-CRYPTION

4.1 Homomorphic sorting of a single ciphertext and a single sequence with word-wise encryption: special case

Problem Statement: For an input RLWE ciphertext $ct(\mathbf{a})$, $\mathbf{a} = CRT^{-1}((a_0, \ldots, a_{l-1}))$ satisfies $a_i \in \mathbb{Z}p$, and an RLWE ciphertext $ct(\mathbf{b})$ needs to be output, $\mathbf{b} = CRT^{-1}((b_0, \ldots, bl-1))$ and satisfies that b_0, \ldots, b_{l-1} is a permutation of a_0, \ldots, a_{l-1} , and $b_0 \leq b_1 \leq \cdots \leq b_{l-1}$.

To solve this problem, first define the ord of a number in a sequence.

definition **1.** For each number a_i in the sequence $a_0, a_1, \ldots, a_{l-1}$, define $\operatorname{ord}(a_i) = s + k$, if there are exactly *s* numbers in the sequence $a_0, a_1, \ldots, a_{l-1}$ that are less than a_i , and there are exactly *k* numbers in the sequence $a_0, a_1, \ldots, a_{i-1}$ that are equal to a_i .

Take the sequence $a_0 = 7$, $a_1 = 5$, $a_2 = 4$, $a_3 = 5$, $a_4 = 2$ as an example to introduce the calculation of ord.

- For a₁ = 5, in the sequence a₀, a₁,..., a₄, that is, 7, 5, 4, 5, 2, there are exactly two numbers less than 5, namely 4 and 2; in the sequence a₀ = 7, no number is equal to 5, so ord(a₁) = 2 + 0 = 2.
- For a₃ = 5, in the sequence a₀, a₁,..., a₄, that is, 7, 5, 4, 5, 2, there are exactly two numbers less than 5, namely 4 and 2; and in the sequence a₀, a₁, a₂, there is exactly one number equal to 5. Therefore, ord(a₃) = 2 + 1 = 3.

The translation to English is: "It can be seen that even if there are two identical numbers in the sequence, their ord is different. In fact, any two numbers in a sequence will not have the same ord, see the following lemma.

Proof: There are two cases to discuss.

The first case is $a_i = a_j$, without loss of generality, let i < j. Let $\operatorname{ord}(a_i) = s + k$. Since $a_i = a_j$, there are exactly s numbers less than a_j in the sequence $a_0, a_1, \ldots, a_{l-1}$; since i < j, there are at least k + 1 numbers equal to a_j in the sequence $a_0, a_1, \ldots, a_{j-1}$. Therefore, $\operatorname{ord}(a_j) \ge s + k + 1 > \operatorname{ord}(a_i)$.

The second case is $a_i \neq a_j$, without loss of generality, let $a_i < a_j$. In the sequence $a_0, a_1, \ldots, a_{l-1}$, there are at least $\operatorname{ord}(a_i) + 1$ numbers less than a_j . Therefore, $\operatorname{ord}(a_j) \geq \operatorname{ord}(a_i) + 1$.

n summary,
$$\operatorname{ord}(a_i) \neq \operatorname{ord}(a_j)$$
.

The discussion of the second case in Lemma 8 shows that if $a_i < a_j$, then $\operatorname{ord}(a_i) < \operatorname{ord}(a_j)$. From this, the following corollary is obtained:

corollary **1**. If $\operatorname{ord}(a_i) > \operatorname{ord}(a_j)$, then $a_i \ge a_j$.

T

The following lemma gives the range of values for ord:

lemma 9.
$$\{ \operatorname{ord}(a_i), 0 \le i \le l-1 \} = \{0, 1, \dots, l-1 \}.$$

Proof: On the one hand, let $\max_{0 \le i \le l-1} a_i = a$, and there are exactly *t* numbers in the sequence equal to *a*. Let a_j be the one with the largest subscript among these numbers equal to *a*. Then in the sequence $a_0, a_1, \ldots, a_{l-1}$, there are exactly l - t numbers less than a_j , and in the sequence $a_0, a_1, \ldots, a_{j-1}$, there are exactly t - 1 numbers equal to a_j . By the definition of ord, $\operatorname{ord}(a_j) = l - t + t - 1 = l - 1$. Thus, we have $\max_{0 \le i \le l-1} \operatorname{ord}(a_i) \ge l - 1$.

On the other hand, by the definition of ord, for any a_i in the sequence, we must have $\operatorname{ord}(a_i) \leq l - 1$. This is because the number of numbers less than or equal to a_i in this sequence will not exceed the length of the sequence. Excluding itself, we can get $\operatorname{ord}(a_i) \leq l - 1$. Combined with the result of the previous paragraph, we have $\max_{0 \leq i \leq l-1} \operatorname{ord}(a_i) = l - 1$.

By the definition of ord, for any a_i in the sequence, $\operatorname{ord}(a_i) \geq 0$. According to Lemma 8, ord is an injection, so we have $\operatorname{ord}(a_i), 0 \leq i \leq l-1 = 0, 1, \ldots, l-1$.

From Lemmas 8 and 9, we can obtain the following theorem:

Theorem 1. For the sequence $a_0, a_1, \ldots, a_{l-1}$, define the sequence b_0, \ldots, b_{l-1} , where $b_{\text{ord}(a_i)} = a_i$, then $b_0 \leq b_1 \leq \cdots \leq b_{l-1}$, which is a sorting of the sequence $a_0, a_1, \ldots, a_{l-1}$.

Proof: According to Lemma 9, $\operatorname{ord}(a_i)$, $0 \le i \le l-1 = 0, 1, \ldots, l-1$, so for $0 \le i \le l-1$, b_i is assigned a value. According to Lemma 8, for $i \ne j$, $\operatorname{ord}(a_i) \ne \operatorname{ord}(a_j)$, so for $0 \le i \le l-1$, b_i will not be assigned two different values. Next, we prove that for any i < j, $b_i \le b_j$.

Suppose for some $0 \leq s, t < l$, we have $\operatorname{ord}(a_s) = i$, $\operatorname{ord}(a_t) = j$. By definition, $b_i = a_s, b_j = a_t$. Since i < j, that is, $\operatorname{ord}(a_s) < \operatorname{ord}(a_t)$, by Corollary 1, we have $a_s \leq a_t$, that is, $b_i \leq b_j$. The theorem is proved.

Theorem 1 shows that for any number a_i in the sequence $a_0, a_1, \ldots, a_{l-1}$, if we know $\operatorname{ord}(a_i)$, we know its position in the sorted sequence. This discovery is the core of the algorithm in this chapter. Following this idea, we can obtain

a method for sorting a sequence: first, for each number in the sequence $a_0, a_1, \ldots, a_{l-1}$, calculate $\operatorname{ord}(a_i)$; second, place each number a_i at the $\operatorname{ord}(a_i)$ -th position in the sequence, thus obtaining a sorted sequence. For ciphertexts, the above method can also be performed. The first step is to calculate the ord of the plaintexts in the ciphertext. The second step is to use the calculated ord to move the elements in the plaintext polynomial slots to the appropriate positions by left shifting. Algorithm 3 gives how to calculate ord for homomorphic ciphertexts, and Algorithm 4 gives how to complete sorting.

Algorithm 3 CalOrd(ct(a))

Input: $ct(a), a = CRT^{-1}((a_0, ..., a_{l-1}))$ **Output:** $\operatorname{ct}(\mathbf{v})$, $\mathbf{v} = \operatorname{CRT}^{-1}((v_0, \ldots, v_{l-1}))$ satisfies $v_i =$ $\operatorname{ord}(a_i)$ 1: $\operatorname{ct}(\mathbf{c}_0) \leftarrow \operatorname{ct}(\mathbf{a})$ 2: for i = 1 to l - 1 do $ct(\mathbf{c}_i) \leftarrow \mathbf{LShift}(ct(\mathbf{c}_{i-1}))$ 3: $\{\mathbf{c}_i = CRT^{-1}(a_i, a_{i+1}, \dots, a_{i+l-1})\}$ 4: $\operatorname{ct}(\mathbf{r}_i) \leftarrow \operatorname{LT}(\operatorname{ct}(\mathbf{c}_i), \operatorname{ct}(\mathbf{c}_0))$ $\{\mathbf{r}_i = CRT^{-1}(\mathbf{LT}(a_i, a_0), \dots, \mathbf{LT}(a_{i+l-1}, a_{l-1}))\}$ $\mathbf{f}_i \leftarrow \mathrm{CRT}^{-1}(0,\ldots,0,1,\ldots,1)$ 5: $\sum_{l=i}$ $\operatorname{ct}(\mathbf{t}_i) \leftarrow \mathbf{EQ}(\operatorname{ct}(\mathbf{c}_i), \operatorname{ct}(\mathbf{c}_0)) \cdot \mathbf{f}_i$ 6: $\{\mathbf{t}_i = \operatorname{CRT}^{-1}(\underbrace{0,\ldots,0}_{l-i},\underbrace{\mathbf{EQ}(a_0,a_{l-i}),\ldots,\mathbf{EQ}(a_{i-1},a_{l-1})}_i)\}$ 7: end for 8: $\operatorname{ct}(\mathbf{v}) \leftarrow \sum_{i=1}^{l-1} (\operatorname{ct}(\mathbf{r}_i) + \operatorname{ct}(\mathbf{t}_i))$

Below, we take the input $\mathbf{a} = CRT^{-1}(7, 5, 4, 5, 2)$ as an example to illustrate how Algorithm 3 works.

- In step 1 of Algorithm 3, copy the ciphertext ct(a) to ct(c₀).
- In the for loop, for the case of i = 1, step 3 of the Algorithm 3 shifts the slots of c₀ to the left once to obtain ct(c₁), at this time c₁ = CRT⁻¹(5, 4, 5, 2, 7).
- 3) In step 4 of Algorithm 3, the original ciphertext $ct(\mathbf{c}_0)$ and ciphertext $ct(\mathbf{c}_1)$ are compared using the function **LT**, and the result is recorded as $ct(\mathbf{r}_1)$. According to the definition of the comparison function, we have $\mathbf{r}_1 = CRT^{-1}(\mathbf{LT}(5,7), \mathbf{LT}(4,5), \mathbf{LT}(5,4), \mathbf{LT}(2,5), \mathbf{LT}(7,2)) = CRT^{-1}(1,1,0,1,0).$
- 4) In steps 5-6 of Algorithm 3, the original ciphertext ct(c₀) and ciphertext ct(c₁) are compared using the function EQ, and the result is multiplied by the plaintext f₁ = CRT⁻¹(0,0,0,0,1) to obtain the result recorded as ct(t₁). At this time, we have t₁ = CRT⁻¹(EQ(5,7)×0, EQ(4,5)×0, EQ(5,4)×0, EQ(2,5)×0, EQ(7,2)×1) = CRT⁻¹(0,0,0,0,0). Since when calculating ord, we only need to compare whether there are any numbers equal to it in the sequence before each number, we need to multiply by the plaintext f₁ to eliminate unnecessary results. For example, in this step, only the last EQ(a₄, a₀) is needed and the previous results need to be eliminated.

5) For i = 2, 3, 4, the same left shift and comparison are performed in steps 3-6 of Algorithm 3. For \mathbf{c}_0 and $\mathbf{c}_2 = \mathrm{CRT}^{-1}(4, 5, 2, 7, 5)$, calculate **LT** and **EQ** to obtain $\mathbf{r}_2 = \mathrm{CRT}^{-1}(1, 0, 1, 0, 0)$ and $\mathbf{t}_2 = \mathrm{CRT}^{-1}(0, 0, 0, 0, 0)$. For \mathbf{c}_0 and $\mathbf{c}_3 = \mathrm{CRT}^{-1}(5, 2, 7, 5, 4)$, calculate **LT** and **EQ** to obtain $\mathbf{r}_3 = \mathrm{CRT}^{-1}(1, 1, 0, 0, 0)$ and $\mathbf{t}_3 = \mathrm{CRT}^{-1}(0, 0, 0, 1, 0)$. For \mathbf{c}_0 and $\mathbf{c}_4 =$ $\mathrm{CRT}^{-1}(2, 7, 5, 4, 5)$, calculate **LT** and **EQ** to obtain $\mathbf{r}_4 = \mathrm{CRT}^{-1}(1, 0, 0, 1, 0)$ and $\mathbf{t}_4 =$ $\mathrm{CRT}^{-1}(0, 0, 0, 0, 0)$.

Theorem 2 provides a proof of the correctness of the algorithm and an analysis of its complexity.

Theorem 2. Algorithm 3 can correctly calculate the ord of all integers in the sequence, with a complexity of $O(n \log n)$ polynomial multiplications.

Proof: In step 5 of algorithm 3, set $\mathbf{r}_i = \operatorname{CRT}^{-1}(r_i, 0, \dots, r_{i,l-1})$, $\mathbf{t}_i = \operatorname{CRT}^{-1}(t_i, 0, \dots, t_{i,l-1})$, $1 \leq i < l$. In step 3 of the algorithm, set $\mathbf{c}_i = \operatorname{CRT}^{-1}(c_{i,0}, \dots, c_{i,l-1})$, $1 \leq i < l$. Note that \mathbf{c}_i is obtained by shifting \mathbf{c}_0 to the left *i* times, so $c_{i,j} = a_{i+j}$ (subscripts are treated modulo *l*). Therefore, for $1 \leq j \leq l-1$, $r_{j,i} = \operatorname{LT}(a_{i+j}, a_i)$. Hence, when $a_{i+j} < a_i, r_{j,i} = 1$; when $a_i \leq a_{i+j}, r_{j,i} = 0$. Similarly, when i + j - 1 > l and $a_i = ai + j - 1$, $t_{j,i} = 1$; otherwise, $t_{j,i} = 0$. Thus, $\sum_{j=1}^{l-1} r_{j,i}$ is the number of elements in the sequence a_0, a_1, \dots, a_{l-1} that are less than a_i , and $\sum_{j=1}^{l-1} t_{j,i}$ is the number of elements in the sequence a_0, a_1, \dots, a_{l-1} that are equal to a_i . Therefore, $v_{0,i} = \sum_{j=1}^{l-1} (r_{j,i} + t_{j,i}) = \operatorname{ord}(a_i)$.

For the complexity of the algorithm, in the for loop from steps 2 to 7, there are l-1 LShifts, l-1 LQs and l-1 EQs. According to (3.3), (3.4) and Lemma 2, the total complexity is $(l-1)O(\log n)+(l-1)O(\log n)+(l-1)O(\log n) = O(l \log n)$ polynomial multiplications. According to the setting of (2.1), there are a total of $O(l \log n) = O(n \log n)$ polynomial multiplications.

Algorithm 4 Slot-Sort(ct(a))

Input: $ct(a), a = CRT^{-1}(a_0, ..., a_{l-1})$ **Output:** ct(**b**), **b** = CRT⁻¹(b_0, \ldots, b_{l-1}), b_0, \ldots, b_{l-1} is a permutation of a_0, \ldots, a_{l-1} and $b_0 \leq b_1 \leq \cdots \leq b_{l-1}$ 1: $\mathbf{h} \leftarrow \operatorname{CRT}^{-1}(0, 1, \dots, l-1)$ 2: $\operatorname{ct}(\mathbf{c}_0) \leftarrow \operatorname{ct}(\mathbf{a})$ 3: for i = 1 to l - 1 do $ct(c_i) \leftarrow LShift(ct(c_{i-1}))$ 4: 5: end for 6: $ct(\mathbf{v}_0) \leftarrow CalOrd(ct(\mathbf{a}))$ 7: $\operatorname{ct}(\mathbf{b}_0) \leftarrow \mathbf{EQ}(\operatorname{ct}(\mathbf{v}_0), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{c}_0)$ 8: for i = 1 to l - 1 do $\operatorname{ct}(\mathbf{v}_i) \leftarrow \mathsf{LShift}(\operatorname{ct}(\mathbf{v}_{i-1}))$ 9: $\operatorname{ct}(\mathbf{b}_i) \leftarrow \operatorname{ct}(\mathbf{b}_{i-1}) + \mathbf{E}\mathbf{Q}(\operatorname{ct}(\mathbf{v}_i), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{c}_i)$ 10: 11: end for 12: $\operatorname{ct}(\mathbf{b}) \leftarrow \operatorname{ct}(\mathbf{b}_{l-1})$

Similarly, taking the input $\mathbf{a} = \operatorname{CRT}^{-1}(7, 5, 4, 5, 2)$ as an example, the operation of algorithm 4 is explained.

- 1) Steps 1 to 5 of algorithm 4, like algorithm 3, are for copying and shifting the plaintext slots.
- 2) Step 6 of algorithm 4 calls algorithm 3 to obtain the ord of each number as $\mathbf{v}_0 = (4, 2, 1, 3, 0)$.
- 3) In step 7 of algorithm 4, \mathbf{v}_0 is compared with the reference $\mathbf{h} = \text{CRT}^{-1}(0, 1, \dots, l-1)$ using the EQ function. The result is multiplied by \mathbf{c}_0 to obtain (0, 0, 0, 5, 0).
- 4) Steps 9-10 of algorithm 4 are executed a total of l 1 = 4 times. The first time, \mathbf{v}_0 is shifted to the left to obtain $\mathbf{v}_1 = (2, 1, 3, 0, 4)$. The result of comparing \mathbf{v}_1 with \mathbf{h} is multiplied by \mathbf{c}_1 to obtain (0, 4, 0, 0, 7). The second time, \mathbf{v}_1 is shifted to the left to obtain $\mathbf{v}_2 = (1, 3, 0, 4, 2)$. The result of comparing \mathbf{v}_2 with \mathbf{h} is multiplied by \mathbf{c}_2 to obtain (0, 0, 0, 0, 0, 0). The third time, \mathbf{v}_2 is shifted to the left to obtain $\mathbf{v}_3 = (3, 0, 4, 2, 1)$. The result of comparing \mathbf{v}_3 with \mathbf{h} is multiplied by \mathbf{c}_3 to obtain (0, 0, 0, 0, 0). The fourth time, \mathbf{v}_3 is shifted to the left to obtain $\mathbf{v}_4 = (0, 4, 2, 1, 3)$. The result of comparing \mathbf{v}_4 with \mathbf{h} and multiplying by \mathbf{c}_4 is (2, 0, 5, 0, 0).
- 5) Step 12 of algorithm 4 adds up the above results to get (2, 4, 5, 5, 7).

Below is a proof of the correctness of algorithm 4 and an analysis of its complexity.

Theorem 3. Algorithm 4 can correctly solve the special case of homomorphic sorting of a single ciphertext and a single sequence with word-wise encryption, with a complexity of $O(n \log n)$ polynomial multiplications.

Proof: In steps 6 and 9 of algorithm 4, set $\mathbf{v}_i = \operatorname{CRT}^{-1}(v_i, 0, \dots, v_{i,l-1}), 0 \leq i < l$. By Theorem 2, $v_{0,i} = \operatorname{ord}(a_i), 0 \leq i \leq l-1$. In step 12 of algorithm 4, set $\mathbf{b} = \operatorname{CRT}^{-1}(b_0, \dots, b_{l-1})$. Below we prove that $b_{\operatorname{ord}(a_i)} = a_i$.

In step 1 of algorithm 4, set $\mathbf{h} = \operatorname{CRT}^{-1}(h_0, \ldots, h_{l-1})$. By definition, $h_i = i$. In steps 7 and 10 of algorithm 4, for $0 \le i < l$, set $\mathbf{c}i = \operatorname{CRT}^{-1}(ci, 0, \ldots, c_{i,l-1})$. For any $0 \le j \le l-1$, $\operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, h_{\operatorname{ord}(a_i)}) = \operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, \operatorname{ord}(a_i)) =$ 1 if and only if $v_{j,\operatorname{ord}(a_i)} = \operatorname{ord}(a_i)$. This is because $v_{0,i} = \operatorname{ord}(a_i)$ and $\mathbf{v}j$ is obtained by shifting $\mathbf{v}0$ to the left j times, so $vj, \operatorname{ord}(a_i) = \operatorname{ord}(a \operatorname{ord}(a_i) + j)$. Therefore, $\operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, h_{\operatorname{ord}(a_i)}) = 1$ if and only if $\operatorname{ord}(a_{\operatorname{ord}(a_i)+j}) =$ $\operatorname{ord}(a_i)$, i.e. $\operatorname{ord}(a_i) + j = i$. As a corollary, $b_{\operatorname{ord}(a_i)} =$ $\sum_{j=0}^{l-1} \operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, h_{\operatorname{ord}(a_i)}) \cdot c_{j,\operatorname{ord}(a_i)} = a_{\operatorname{ord}(a_i)+j} = a_i$. According to Theorem 1, algorithm 4 is correct.

For the complexity of the algorithm, the first for loop uses l - 1 LShifts, and step 6 calls CalOrd. Step 7 and the second for loop use l EQs. Finally, in the second for loop, there are l - 1 LShifts. Therefore, there are a total of 2l - 2 LShifts, l EQs and one CalOrd. According to Lemma 2, the complexity of one CalOrd is $O(n \log n)$ polynomial multiplications; according to (3.3), the complexity of one EQ is $O(\log n)$ polynomial multiplications; according to (2.1), the complexity of one LShift is $O(\log n)$ polynomial multiplications. Adding these results together and according to the setting of (2.1), the total complexity is $(2l - 2)O(\log n) + l \cdot O(\log n) + O(n \log n) = O(n \log n)$ polynomial multiplications.

Section 3.4 presents three sorting algorithms based on different comparison methods. Their complexity is compared with that of algorithm 4 below.

When using a sorting algorithm based on polynomials over finite fields, the number of ciphertexts to be sorted is *l*. By (3.8) and (2.1), its complexity is $O(l(\log l)^2 \log n) = O(n(\log n)^3)$ polynomial multiplications. The complexity of algorithm 4 is $O(n \log n)$ polynomial multiplications, with an advantage of $O((\log n)^2)$ polynomial multiplications.

When using a sorting algorithm based on TFHE bootstrapping, the number of ciphertexts to be sorted is *l*. By (3.9) and (2.1), its complexity is $O(ln(\log l)^2(\log n)^2) = O(n^2(\log n)^4)$ polynomial multiplications. The complexity of algorithm 4 is $O(n \log n)$ polynomial multiplications, with an advantage of $O(n(\log n)^3)$ polynomial multiplications.

When using a sorting algorithm based on parallel bootstrapping, the number of ciphertexts to be sorted is *l*. By (3.10) and (2.1), its complexity is $O(l(\log l)^2(\log n)^3) = O(n(\log n)^5)$ polynomial multiplications. The complexity of algorithm 4 is $O(n \log n)$ polynomial multiplications, with an advantage of $O((\log n)^4)$ polynomial multiplications.

For this case, even if the input ciphertexts need to be converted, according to Lemma 3, the complexity of these operations is at most $O(\sqrt{n} \log n)$, which does not affect the comparison of complexity.

4.2 Homomorphic sorting of a single ciphertext and a single sequence with word-wise encryption: normal case

Problem Statement:Let $1 \leq k < l$. For an input RLWE ciphertext $ct(\mathbf{a})$, $\mathbf{a} = CRT^{-1}((a_0, \ldots, a_{l-1}))$ satisfies $a_i \in \mathbb{Z}p$. The output is an RLWE ciphertext $ct(\mathbf{b})$, $\mathbf{b} = CRT^{-1}((b_0, \ldots, bl-1))$ and satisfies that b_0, \ldots, b_{k-1} is a permutation of a_0, \ldots, a_{k-1} , and $b_0 \leq b_1 \leq \cdots \leq b_{k-1}$; when $k \leq i < l$, $b_i = a_i$.

For this case, the parameters are first set. For the parameter k, according to (2.1), this section sets:

$$k = O(n^a), \ 0 \le a \le 1$$
 (4.1)

In this case, only the first k positions of the plaintext polynomial slots need to be sorted. This case can still be done by calculating the ord of each number in the sequence and then sorting them by shifting left and right. As in the previous section, a simple example is used to illustrate the sorting method.

- 1) For **a** = $CRT^{-1}(2, 1, 3, 5, 3)$, the sequence to be sorted is S = (2, 1, 3). Multiply **a** by **f** = $CRT^{-1}(1, 1, 1, 0, 0)$ to get $\mathbf{u}_0 = CRT^{-1}(2, 1, 3, 0, 0)$. Multiply **a** by **g** = $CRT^{-1}(0, 0, 0, 1, 1)$ to get **b**' = $CRT^{-1}(0, 0, 0, 5, 3)$.
- 2) Shift \mathbf{u}_0 to the left once and multiply the result by $\mathbf{f}_1 = \operatorname{CRT}^{-1}(1, 1, 0, 0, 0)$ to get $\mathbf{c}_1 = \operatorname{CRT}^{-1}(1, 3, 0, 0, 0)$. Shift \mathbf{u}_0 to the right twice and multiply the result by $\mathbf{g}_2 = \operatorname{CRT}^{-1}(0, 0, 1, 0, 0)$ to get $\mathbf{d}_2 = \operatorname{CRT}^{-1}(0, 0, 2, 0, 0)$. Add \mathbf{c}_1 and \mathbf{d}_2 to get $\mathbf{u}_1 = \operatorname{CRT}^{-1}(1, 3, 2, 0, 0)$. In the first three slots, the result of shifting sequence *S* to the left once is obtained.

- 3) Left shift a twice and multiply the result by $f_2 =$ $CRT^{-1}(1, 0, 0, 0, 0)$ to get $c_2 = CRT^{-1}(3, 0, 0, 0, 0)$. Right shift \mathbf{u}_0 once and multiply the result by $\mathbf{g}_1 =$ $CRT^{-1}(0, 1, 1, 0, 0)$ to get $d_1 = CRT^{-1}(0, 2, 1, 0, 0)$. Add c_2 and d_1 to get $u_2 = CRT^{-1}(3, 2, 1, 0, 0)$. In the first three slots, the result of shifting sequence Sto the left twice is obtained.
- Calculate $LT(\mathbf{u}_1, \mathbf{u}_0)$ to get $\mathbf{r}_1 = CRT^{-1}(1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0)$ 4) 0). Calculate $LT(u_2, u_0)$ to get $r_2 = CRT^{-1}(0, 0, 1, 1)$ (0,0). Calculate **EQ**($\mathbf{d}_1, \mathbf{u}_0$) and multiply the result by \mathbf{g}_1 to get $\mathbf{t}_1 = \operatorname{CRT}^{-1}(0, 0, 0, 0, 0)$. Calculate $EQ(d_2, u_0)$ and multiply the result by g_2 to get $\mathbf{t}_2 = CRT^{-1}(1,0,1,0,0)$. Add \mathbf{r}_1 , \mathbf{r}_2 , \mathbf{t}_1 and \mathbf{t}_2 to get $\mathbf{v}_0 = CRT^{-1}(1, 0, 2, 0, 0)$. In the first three slots, the ord of each number in sequence S is obtained.
- Shift \mathbf{v}_0 to the left once and multiply the re-5) sult by $f_1 = CRT^{-1}(1, 1, 0, 0, 0)$ to get $x_1 =$ $CRT^{-1}(0,2,0,0,0)$. Shift \mathbf{v}_0 to the right twice and multiply the result by $\mathbf{g}_2 = \operatorname{CRT}^{-1}(0, 0, 1, 0, 0)$ to get $y_2 = CRT^{-1}(0, 0, 1, 0, 0)$. Add x_1 and y_2 to get $v_1 = CRT^{-1}(0, 2, 1, 0, 0)$. In the first three slots, the result of shifting the ord of each number in sequence S to the left once is obtained.
- 6) Shift \mathbf{v}_0 to the left twice and multiply the result by $f_2 = CRT^{-1}(1,0,0,0,0)$ to get $x_2 =$ $CRT^{-1}(2,0,0,0,0)$. Shift \mathbf{v}_0 to the right once and multiply the result by $\mathbf{g}_1 = CRT^{-1}(0, 1, 1, 0, 0)$ to get $y_1 = CRT^{-1}(0, 1, 0, 0, 0)$. Add x_2 and y_1 to get $\mathbf{v}_2 = \mathrm{CRT}^{-1}(2, 1, 0, 0, 0)$. In the first three slots, the result of shifting the ord of each number in sequence S to the left twice is obtained.
- Calculate EQ for v_0 and the reference h =7) $CRT^{-1}(0, 1, 2, 0, 0)$, multiply the result by \mathbf{u}_0 to get $\mathbf{b}_0 = \operatorname{CRT}^{-1}(0, 0, 3, 0, 0)$. Calculate EQ for \mathbf{v}_1 and the reference h, multiply the result by u_1 to get $\mathbf{b}_1 = CRT^{-1}(0, 0, 0, 0, 0)$. Calculate EQ for \mathbf{v}_2 and the reference **h** and multiply by \mathbf{u}_2 to get $\mathbf{b}_2 = \operatorname{CRT}^{-1}(1, 2, 0, 0, 0).$
- Finally, add \mathbf{b}_0 , \mathbf{b}_1 , \mathbf{b}_2 and \mathbf{b}' to get the final sorted 8) result $\mathbf{b} = CRT^{-1}(1, 2, 3, 5, 3).$

For the general input of this problem, algorithm 5 gives the method for calculating ord, and algorithm 6 gives the entire sorting process.

Theorem 4 provides a proof of the correctness and complexity analysis of algorithm 5.

Theorem 4. Algorithm 5 can correctly calculate the ord of each integer in the sequence, with a complexity of $O(n^a \log n)$ polynomial multiplications.

Proof: The proof follows the same line of reasoning as Theorem 2. In step 8 of algorithm 5, ci is obtained by shifting $\mathbf{a}i$ to the left *i* times and then multiplying by $\mathbf{f}i$, so $\mathbf{c}i =$ $CRT^{-1}(ai, \ldots, ak - 1, 0, \ldots, 0)$. Similarly, in step 9 of algorithm 5, $\mathbf{d}i = CRT^{-1}(0, \dots, 0, a0, \dots, ak - i - 1, 0, \dots, 0).$ Then in step 12 of algorithm 5, $\mathbf{u}i$ = $CRT^{-1}(ai, \ldots, ak - 1, a_0, \ldots, a_{i-1}, 0, \ldots, 0)$ can be obtained. Therefore, in step 13 of algorithm 5, ri = $CRT^{-1}(LT(a_i, a_0), \dots, LT(a_i + k - 1, a_{k-1}), 0, \dots, 0)$ can be obtained. Similarly, in step 14 of algorithm 5, ti = $CRT^{-1}(0,\ldots,0, \mathbf{EQ}(a_0,a_i),\ldots,\mathbf{EQ}(a_{k-i-1},a_{k-1}),0,\ldots,0)$ $CRT^{-1}(ord(a_i),\ldots,ord(a_{k-1+i}),0,\ldots,0)$, where the subcan be obtained.

Algorithm 5 PartOrd(ct(a))

Input: Let $1 \le k < l$, ct(**a**), **a** = CRT⁻¹((a_0, \ldots, a_{l-1})), for $0 \leq j < k, a_j \in \mathbb{Z}_p$; for $k \leq j < l, a_j = 0$

Output: $ct(\mathbf{v}), \mathbf{v} = CRT^{-1}((v_0, \ldots, v_{l-1}))$ satisfies for $0 \leq 1$ $j < k, v_j = \operatorname{ord}(a_j)$

1:
$$\mathbf{f} \leftarrow \operatorname{CRT}^{-1}(\underbrace{1,\ldots,1}_{k},\underbrace{0,\ldots,0}_{l-k})$$

2: $\operatorname{ct}(\mathbf{u}_{0}) \leftarrow \operatorname{ct}(\mathbf{a}) \cdot \mathbf{f}$
3: $\operatorname{ct}(\mathbf{c}_{0}) \leftarrow \operatorname{ct}(\mathbf{a})$
4: $\operatorname{ct}(\mathbf{d}_{0}) \leftarrow \operatorname{ct}(\mathbf{a})$
5: $\mathbf{for} i = 1 \ \mathbf{to} \ k - 1 \ \mathbf{do}$
6: $\mathbf{f}_{i} \leftarrow \operatorname{CRT}^{-1}(\underbrace{1,\ldots,1}_{k-i},0,\ldots,0)$
7: $\mathbf{g}_{i} \leftarrow \operatorname{CRT}^{-1}(\underbrace{0,\ldots,0}_{i},\underbrace{1,\ldots,1}_{k-i},0,\ldots,0)$
8: $\operatorname{ct}(\mathbf{c}_{i}) \leftarrow \operatorname{LShift}(\operatorname{ct}(\mathbf{c}_{i-1})) \cdot \mathbf{f}_{i}$
9: $\operatorname{ct}(\mathbf{d}_{i}) \leftarrow \operatorname{RShift}(\operatorname{ct}(\mathbf{d}_{i-1})) \cdot \mathbf{g}_{i}$
10: end for
11: for $i = 1 \ \mathbf{to} \ k - 1 \ \mathbf{do}$
12: $\operatorname{ct}(\mathbf{u}_{i}) \leftarrow \operatorname{ct}(\mathbf{c}_{i}) + \operatorname{ct}(\mathbf{d}_{k-i})$
 $\{\mathbf{u}_{i} = \operatorname{CRT}^{-1}(a_{i},\ldots,a_{k-1},a_{0},\ldots,a_{i-1},0,\ldots,0)\}$
13: $\operatorname{ct}(\mathbf{r}_{i}) \leftarrow \operatorname{LT}(\operatorname{ct}(\mathbf{u}_{i}),\operatorname{ct}(\mathbf{u}_{0}))$
 $\{\mathbf{r}_{i} = \operatorname{CRT}^{-1}(\operatorname{LT}(a_{i},a_{0}),\ldots,\operatorname{LT}(a_{i+k-1},a_{k-1}),0,\ldots,0)\}$
14: $\operatorname{ct}(\mathbf{t}_{i}) \leftarrow \operatorname{EQ}(\operatorname{ct}(\mathbf{d}_{i}),\operatorname{ct}(\mathbf{u}_{0})) \cdot \mathbf{g}_{i}$
 $\{\mathbf{t}_{i} = \operatorname{CRT}^{-1}(0,\ldots,0, \underbrace{\operatorname{EQ}(a_{0},a_{i}),\ldots,\operatorname{EQ}(a_{k-i-1},a_{k-1})}_{k-i},0,\ldots,0)\}$
15: end for
16: $\operatorname{ct}(\mathbf{v}) \leftarrow \sum_{i=1}^{k-1}(\operatorname{ct}(\mathbf{r}_{i}) + \operatorname{ct}(\mathbf{t}_{i}))$

In the last step of algorithm 5, let $\mathbf{v} = \operatorname{CRT}^{-1}(v_0, \ldots, v_{l-1})$, then $v_i = \sum_{j=0}^{k-1} \operatorname{LT}(a_j, a_i) +$ $\sum_{j=0}^{i-1} EQ(a_j, a_i)$, according to the definition of ord, that is $v_i = \operatorname{ord}(a_i)$. The correctness of algorithm 5 is proven.

For the complexity of algorithm 5, the first for loop (steps 5 to 10) uses k - 1 LShifts and k - 1 RShifts. The second for loop (steps 11 to 15) uses k-1 LTs and k-1 EQs. According to (3.4), (3.3), (4.1) and Lemma 2, the total complexity is $(2k-2)O(n\log n) + (2k-2)O(n\log n) = O(k\log n) =$ $O(n^a \log n).$

Theorem 5 provides a proof of the correctness and complexity analysis of algorithm 6.

Theorem 5. Algorithm 6 can correctly solve the general case of the whole encrypted single ciphertext single sequence problem, with a complexity of $O(n^a \log n)$ polynomial multiplications.

Proof: In step 3 of algorithm 6, algorithm 5 is called. So $\mathbf{v}_0 = \operatorname{CRT}^{-1}(\operatorname{ord}(a_0), \ldots, \operatorname{ord}(ak-1), 0, \ldots, 0).$ Steps 8 to 13 of algorithm 6 are the same as steps 5 to 10 of algorithm 5, so in step 13, $\mathbf{u}i$ = $CRT^{-1}(ai, ..., a_{k-1}, a_0, ..., a_{i-1}, 0, ..., 0)$ is also obtained. Steps 16, 17 and 21 of algorithm 6 shift the elements in the first k slots of \mathbf{v}_0 to the left. Therefore, \mathbf{v}_i = script is processed modulo k.

Input: Let $1 \le k < l$, ct(**a**), **a** = CRT⁻¹((a_0, \ldots, a_{l-1})), for $0 \leq j < k, a_j \in \mathbb{Z}_p$ **Output:** $ct(\mathbf{b})$, $\dot{\mathbf{b}} = CRT^{-1}((b_0, \dots, bl-1))$ and satisfies that b_0, \ldots, b_{k-1} is a permutation of a_0, \ldots, a_{k-1} , and that b_0, \ldots, b_{k-1} is a permutation of a_0, \ldots, c_k $b_0 \le b_1 \le \cdots \le b_{k-1}$; when $k \le i < l, b_i = a_i$ 1: $\mathbf{f} \leftarrow \operatorname{CRT}^{-1}(\underbrace{1, \ldots, 1}_k, \underbrace{0, \ldots, 0}_{l-k})$ 2: $\mathbf{g} \leftarrow \operatorname{CRT}^{-1}(\underbrace{0, \ldots, 0}_k, \underbrace{1, \ldots, 1}_{l-k})$ 3: $\mathbf{h} \leftarrow \operatorname{CRT}^{-1}(0, 1, \ldots, k-1, \underbrace{0, \ldots, 0}_{l-k})$ 4: $\operatorname{ct}(\mathbf{u}_0) \leftarrow \operatorname{ct}(\mathbf{a}) \cdot \mathbf{f}$ 5: $ct(\mathbf{v}_0) \leftarrow PartOrd(ct(\mathbf{a}))$ 6: $\operatorname{ct}(\mathbf{x}_0) \leftarrow \operatorname{ct}(\mathbf{v}_0)$ 7: $\operatorname{ct}(\mathbf{y}_0) \leftarrow \operatorname{ct}(\mathbf{v}_0)$ 8: $\operatorname{ct}(\mathbf{c}_0) \leftarrow \operatorname{ct}(\mathbf{u}_0)$ 9: $\operatorname{ct}(\mathbf{d}_0) \leftarrow \operatorname{ct}(\mathbf{u}_0)$ 10: for i = 1 to k - 1 do $\mathbf{f}_{i} \leftarrow \operatorname{CRT}^{-1}(\underbrace{1,\ldots,1}_{k-i},0,\ldots,0)$ $\mathbf{g}_{i} \leftarrow \operatorname{CRT}^{-1}(\underbrace{0,\ldots,0}_{k-i},\underbrace{1,\ldots,1}_{i},0,\ldots,0)$ 11: 12: $\operatorname{ct}(\mathbf{c}_i) \leftarrow \mathsf{LShift}(\operatorname{ct}(\mathbf{c}_{i-1})) \cdot \mathbf{f}_i$ 13: $\operatorname{ct}(\mathbf{d}_i) \leftarrow \operatorname{\textbf{RShift}}(\operatorname{ct}(\mathbf{d}_{i-1})) \cdot \mathbf{g}_i$ 14:15: end for 16: for i = 1 to k - 1 do $\operatorname{ct}(\mathbf{u}_i) \leftarrow \operatorname{ct}(\mathbf{c}_i) + \operatorname{ct}(\mathbf{d}_{k-i})$ 17: $\{\mathbf{u}_i = CRT^{-1}(a_i, \dots, a_{k-1}, a_0, \dots, a_{i-1}, 0, \dots, 0)\}$ $\operatorname{ct}(\mathbf{x}_i) \leftarrow \mathsf{LShift}(\operatorname{ct}(\mathbf{x}_{i-1})) \cdot \mathbf{f}_i$ 18: 19: $\operatorname{ct}(\mathbf{y}_i) \leftarrow \operatorname{RShift}(\operatorname{ct}(\mathbf{y}_{i-1})) \cdot \mathbf{g}_i$ 20: end for 21: $\operatorname{ct}(\mathbf{b}_0) \leftarrow \mathbf{E}\mathbf{Q}(\operatorname{ct}(\mathbf{v}_0), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{u}_0)$ 22: for i = 1 to k - 1 do $\operatorname{ct}(\mathbf{v}_i) \leftarrow \operatorname{ct}(\mathbf{x}_i) + \operatorname{ct}(\mathbf{y}_{k-i})$ 23: $\left\{\mathbf{v}_i = \operatorname{CRT}^{-1}(\operatorname{ord}(a_i), \dots, \operatorname{ord}(a_{k-1+i}), 0, \dots, 0)\right\}$ $\operatorname{ct}(\mathbf{b}_i) \leftarrow \mathbf{EQ}(\operatorname{ct}(\mathbf{v}_i), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{u}_i)$ 24: 25: end for 26: $\operatorname{ct}(\mathbf{b}') \leftarrow \operatorname{ct}(\mathbf{a}) \cdot \mathbf{g}$ 27: $\operatorname{ct}(\mathbf{b}) \leftarrow \sum_{i=0}^{k-1} \operatorname{ct}(\mathbf{b}_i) + \operatorname{ct}(\mathbf{b}')$

In step 25 of algorithm 6, let $\mathbf{b} = \operatorname{CRT}^{-1}(b_0, \ldots, b_{l-1})$. Below we prove that $b_{\operatorname{ord}(a_i)} = a_i$.

In step 15 of algorithm 6, set $\mathbf{u}_i = \operatorname{CRT}^{-1}(u_i, 0, \dots, u_{i,l-1})$; in step 21 of algorithm 6, set $\mathbf{v}_i = \operatorname{CRT}^{-1}(v_i, 0, \dots, v_{i,l-1})$. By the definition of **b**, we have $b_{\operatorname{ord}(a_i)} = \sum_{j=0}^{k-1} \operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, h_{\operatorname{ord}(a_i)}) \cdot u_{j,\operatorname{ord}(a_i)}$. And $v_{j,\operatorname{ord}(a_i)} = \operatorname{ord}(a_{\operatorname{ord}(a_i)+j}), u_{j,\operatorname{ord}(a_i)} = a_{\operatorname{ord}(a_i)+j}$, $h_{\operatorname{ord}(a_i)} = \operatorname{ord}(a_i)$. Hence, if and only if $\operatorname{ord}(a_i) = \operatorname{ord}(a_{\operatorname{ord}(a_i)+j})$, i.e., $i = \operatorname{ord}(a_i) + j$, then $\operatorname{EQ}(v_{j,\operatorname{ord}(a_i)}, h_{\operatorname{ord}(a_i)}) = 1$. Therefore, $b_{\operatorname{ord}(a_i)} = a_{\operatorname{ord}(a_i)+j} = a_i$. According to theorem 1, the correctness of algorithm 6 is proved.

Regarding the complexity of algorithm 6, in step 3, algorithm 5 is called. In the first for loop (steps 8 to 13), k - 1 LShift and k - 1 RShift are used. In the second for

loop (steps 14 to 18), k-1 LShift and k-1 RShift are used. In step 19 and the third for loop (steps 20 to 23), k EQ are used. According to (3.3), lemma 2 and theorem 4, the overall complexity is $(4k-4)O(\log n) + kO(\log n) + O(n^a \log n) = O(n^a \log n)$ polynomial multiplications.

In section 3.4, three sorting algorithms based on different comparison methods are given. Next, we compare their complexity with that of algorithm 6.

When using a sorting algorithm based on polynomials over finite fields, the number of ciphertexts to be sorted is k. By (3.8), (4.1) and (2.1), its complexity is $O(k(\log k)^2 \log n) = O(n^a(\log n)^3)$ polynomial multiplications. The complexity of algorithm 4 is $O(n^a \log n)$ polynomial multiplications, with an advantage of $O((\log n)^2)$ polynomial multiplications.

When using a sorting algorithm based on TFHE bootstrapping, the number of ciphertexts to be sorted is k. By (3.9), (4.1) and (2.1), its complexity is $O(kn(\log k)^2(\log n)^2) = O(n^{a+1}(\log n)^4)$ polynomial multiplications. The complexity of algorithm 4 is $O(n^a \log n)$ polynomial multiplications, with an advantage of $O(n(\log n)^3)$ polynomial multiplications.

When using a sorting algorithm based on parallel bootstrapping, the number of ciphertexts to be sorted is k. By (3.10), (4.1) and (2.1), its complexity is $O(k(\log k)^2(\log n)^3) = O(n^a(\log n)^5)$ polynomial multiplications. The complexity of algorithm 4 is $O(n^a \log n)$ polynomial multiplications, with an advantage of $O((\log n)^4)$ polynomial multiplications.

4.3 Homomorphic sorting of a single sequence with multiple ciphertexts encrypted word-wise

Problem Statement:Let k > 0, the sequence $a_0, a_1, \ldots, a_{kl-1}$ is stored in k RLWE ciphertexts $\operatorname{ct}(\mathbf{a}_i), 0 \le i < k$, where $\mathbf{a}_i = \operatorname{CRT}^{-1}(a_{il}, a_{il+1}, \ldots, a_{il+l-1})$. The output is k RLWE ciphertexts $\operatorname{ct}(\mathbf{b}i), 0 \le i < k$, where $\mathbf{b}i = \operatorname{CRT}^{-1}(b_{il}, b_{il+1}, \ldots, b_{il+l-1})$, such that the sequence $b_0, b_1, \ldots, b_{kl-1}$ is a permutation of $a_0, a_1, \ldots, a_{kl-1}$, satisfying $b_0 \le b_1 \le \cdots \le b_{kl-1}$.

First, let's set the parameters. For the number of input ciphertexts k, since it is a parameter independent of n, this paper sets:

$$k = O(n^a), \ a \ge 0. \tag{4.2}$$

For the case of a single sequence with multiple ciphertexts, the integers in a sequence are placed in different ciphertexts. To sort correctly, it is necessary to exchange elements in different plaintext polynomial slots. Since the plaintext is encrypted, it is not clear which slot's element should be moved to other plaintext polynomials. Therefore, the method given by Gentry, Halevi and Smart [22] for known permutations cannot be applied to this problem. To solve this problem, we first define the size relationship between two plaintext polynomials.

definition 2. For polynomials **a** and **b**, where **a** = $(a_0, a_1, \ldots, a_{l-1})$ and **b** = $(b_0, b_1, \ldots, b_{l-1})$, we say that polynomial **a** \leq **b** if $\max_{0 \leq i < l} a_i \leq \min_{0 \leq j < l} b_j$.

The redistribution of two polynomials is defined as follows:

definition 3. For the input of two polynomials **a** and **b**, where **a** = $\operatorname{CRT}^{-1}(a_0, a_1, \ldots, a_{l-1})$ and **b** = $\operatorname{CRT}^{-1}(b_0, b_1, \ldots, b_{l-1})$, the **redistribution** of **a** and **b** means obtaining two new polynomials **a'** and **b'**, where **a'** = $(a'_0, a'_1, \ldots, a'_{l-1})$ and **b'** = $(b'_0, b'_1, \ldots, b'_{l-1})$, satisfying that $a'_0, \ldots, a'_{l-1}, b'_0, \ldots, b'_{l-1}$ is a permutation of $a_0, \ldots, a_{l-1}, b_0, \ldots, b_{l-1}$, and **a'** \leq **b'**.

Next, we present the core algorithm of this section. The following algorithm, for the input of two RLWE ciphertexts $ct(\mathbf{a})$ and $ct(\mathbf{b})$, can redistribute the plaintext polynomials **a** and **b**. The specific process is shown in algorithm 7.

Algorithm 7 SlotRelocate(ct(a), ct(b))

Input: ct(a) and ct(b), $a = CRT^{-1}(a_0, a_1, ..., a_{l-1})$, b = $CRT^{-1}(b_0, b_1, \ldots, b_{l-1})$ Output: $\operatorname{ct}(\mathbf{a}')$ and $\operatorname{ct}(\mathbf{b}')$, $\mathbf{a}' = (a'_0, a'_1, \dots, a'_{l-1})$, $\mathbf{b}' =$ $(b'_0, b'_1, \dots, b'_{l-1})$ satisfied $a'_0, \dots, a'_{l-1}, b'_0, \dots, b'_{l-1}$ is a permutation of $a_0, \ldots, a_{l-1}, b_0, \ldots, b_{l-1}$ and $\mathbf{a}' \leq \mathbf{b}'$ 1: $\mathbf{\tilde{h}} \leftarrow \operatorname{CRT}^{-1}(1, 1, \dots, 1)$ 2: $ct(\mathbf{r}_0) \leftarrow LT(ct(\mathbf{b}), ct(\mathbf{a}))$ 3: $\operatorname{ct}(\mathbf{c}_0) \leftarrow (\mathbf{h} - \operatorname{ct}(\mathbf{r}_0)) \cdot \operatorname{ct}(\mathbf{a}) + \operatorname{ct}(\mathbf{r}_0) \cdot \operatorname{ct}(\mathbf{b})$ 4: $\operatorname{ct}(\mathbf{d}_0) \leftarrow \operatorname{ct}(\mathbf{r}_0) \cdot \operatorname{ct}(\mathbf{a}) + (\mathbf{h} - \operatorname{ct}(\mathbf{r}_0)) \cdot \operatorname{ct}(\mathbf{b})$ 5: for i = 1 to l - 1 do $ct(\mathbf{x}) \leftarrow LShift(ct(\mathbf{c}_{i-1}))$ 6: $\operatorname{ct}(\mathbf{r}_i) \leftarrow \operatorname{LT}(\operatorname{ct}(\mathbf{d}_{i-1}), \operatorname{ct}(\mathbf{x}))$ 7: $\operatorname{ct}(\mathbf{c}_i) \leftarrow (\mathbf{h} - \operatorname{ct}(\mathbf{r}_i)) \cdot \operatorname{ct}(\mathbf{c}_{i-1}) + \operatorname{ct}(\mathbf{r}_i) \cdot \operatorname{ct}(\mathbf{d}_{i-1})$ 8: $\operatorname{ct}(\mathbf{d}_i) \leftarrow \operatorname{ct}(\mathbf{r}_i) \cdot \operatorname{ct}(\mathbf{c}_{i-1}) + (\mathbf{h} - \operatorname{ct}(\mathbf{r}_i)) \cdot \operatorname{ct}(\mathbf{d}_{i-1})$ 9: 10: end for 11: $\operatorname{ct}(\mathbf{a}') \leftarrow \operatorname{ct}(\mathbf{c}_{l-1})$ 12: $\operatorname{ct}(\mathbf{b}') \leftarrow \operatorname{ct}(\mathbf{d}_{l-1})$

Similarly, a simple example is used to illustrate the operation of algorithm 7.

- 1) Input ct(a) and ct(b), where $\mathbf{a} = \operatorname{CRT}^{-1}(2, 4, 7, 9)$ and $\mathbf{b} = \operatorname{CRT}^{-1}(6, 5, 8, 3)$. Steps 2-4 of algorithm 7 compare a and b according to the slot number, synchronously comparing the slots with the same number, putting the smaller plaintext integer into the slot of a and the larger one into the slot of b, resulting in $\mathbf{c}_0 = \operatorname{CRT}^{-1}(2, 4, 7, 3)$ and $\mathbf{d}_0 = \operatorname{CRT}^{-1}(6, 5, 8, 9)$.
- 2) Shift c_0 to the left once to get $\mathbf{c}'_0 = CRT^{-1}(4,7,3,2)$. Compare \mathbf{c}'_0 with \mathbf{d}_0 , put the smaller one into the slot of \mathbf{c}'_0 and the larger one into the slot of \mathbf{d}_0 , resulting in $\mathbf{c}_1 = CRT^{-1}(4,5,3,2)$ and $\mathbf{d}_1 = CRT^{-1}(6,7,8,9)$.
- 3) Continue the above steps two more times to get $\mathbf{c}_3 = \mathrm{CRT}^{-1}(3, 2, 4, 5)$ and $\mathbf{d}_3 = \mathrm{CRT}^{-1}(6, 7, 8, 9)$. As can be seen, at this point the maximum number in the slot of \mathbf{c}_3 is smaller than the minimum number in the slot of \mathbf{d}_3 , thus completing the redistribution of the plaintext polynomials.

To prove the correctness of this algorithm, we first cite a classic result as a lemma.

lemma 10. [1] If a sorting algorithm can correctly sort all sequences whose entries are in \mathbb{Z}_2 , then it can sort any

sequence whose entries are in \mathbb{Z}_q , where q > 1 is an integer.

Using this lemma, the correctness proof and complexity estimation of algorithm 7 are given in theorem 6.

Theorem 6. Algorithm 7, for the input of two RLWE ciphertexts $ct(\mathbf{a})$ and $ct(\mathbf{b})$, can redistribute the plaintext polynomials \mathbf{a} and \mathbf{b} , with a complexity of $O(n \log n)$ polynomial multiplications.

Proof: By lemma 10, for **a** and **b**, where **a** = $\operatorname{CRT}^{-1}(a_0, a_1, \ldots, a_{l-1})$ and **b** = $\operatorname{CRT}^{-1}(b_0, b_1, \ldots, b_{l-1})$, it is only necessary to prove that when $a_0, \ldots, a_{l-1}, b_0, \ldots, b_{l-1}$ is a 0-1 sequence, algorithm 7 can correctly output.

In step 2 of algorithm 7, a and b are compared according to the slot number, synchronously comparing the slots with the same number, putting the smaller plaintext integer into the slot of a and the larger one into the slot of b, resulting in two new polynomials c0 and d0. Then, shift c0 to the left once to get x, and continue to perform the above comparison and exchange operation on x and d0. Such left shift, comparison and exchange operations are performed a total of l-1 times. In steps 2-8 of algorithm 7, let $\mathbf{c}_i = \text{CRT}^{-1}(c_{i,0}, c_{i,1}, \dots, c_{i,l-1})$ and $\mathbf{d}_i = \operatorname{CRT}^{-1}(d_{i,0}, d_{i,1}, \dots, d_{i,l-1})$. Note that for any $0 \leq j < l$, if and only if $c_{i,j} = 1$ and $d_{i,j} = 0$, then $LT(d_{i,j}, c_{i,j}) = 1$, otherwise $LT(d_{i,j}, c_{i,j}) = 0$. Therefore, for any $0 \leq j < l$, if and only if $c_{i,j} = 1$ and $d_{i,j} = 0$, will the slots with sequence number j of \mathbf{c}_i and \mathbf{d}_i exchange plaintext integers with each other. And if there is a 1 in the sequence $a_0, \ldots, al - 1$ and a 0 in the sequence $b_0, \ldots, bl-1$, it must appear in the same slot during the left shift process in step 6 of algorithm 7, thus causing an exchange. In summary, in steps 11 and 12 of algorithm 7, let $\mathbf{a}' = (a'_0, a'_1, \dots, a'_{l-1})$ and $\mathbf{b}' = (b'_0, b'_1, \dots, b'_{l-1})$, either the sequence a'_0, \ldots, a'_{l-1} is all 0s or the sequence b'_0, \ldots, b'_{l-1} is all 1s. In either case, we have $\mathbf{a}' \leq \mathbf{b}'$. The correctness of the algorithm is proved.

Regarding the complexity of algorithm 7, a total of l LT and l-1 LShift are used. By formulas (3.4), (2.1) and lemma 2, the overall complexity is $(l-1)O(\log n) + lO(\log n) = O(n \log n)$ polynomial multiplications.

By replacing steps 8-10 of algorithm 1 with algorithm 7, while keeping the other steps unchanged, this algorithm is called **Odd-Even Merge Sort based on SlotRelocate**. Combined with algorithm 4, we can obtain an algorithm to solve the sorting problem of a single sequence with multiple ciphertexts encrypted as a whole. The specific algorithm is listed in algorithm 8.

Theorem 7. Algorithm 8 can correctly solve the sorting problem of a single sequence with multiple ciphertexts encrypted word-wise. When a = 0, its complexity is $O(n \log n)$ polynomial multiplications; when a > 0, its complexity is $O(n^{a+1} \log^3 n)$ polynomial multiplications.

Proof: The correctness of algorithm 8 is derived from the correctness of algorithm **SlotRelocate** and algorithm **Slot Sort**. For the complexity, step 1 of algorithm 8 uses Odd-Even Merge Sort based on SlotRelocate to sort k ciphertexts, so a total of $O(k(\log k)^2)$ calls to algorithm 7 are

Algorithm 8 Sort LongSeq($ct(\mathbf{a}_0), ct(\mathbf{a}_1), \ldots, ct(\mathbf{a}_{k-1})$)

Input: k ciphertexts $ct(\mathbf{a}_0), ct(\mathbf{a}_1), \dots, ct(\mathbf{a}_{k-1})$, for $0 \le i < k, \mathbf{a}_i = CRT^{-1}(a_{il}, a_{il+1}, \dots, a_{il+l-1})$

- **Output:** k ciphertexts $ct(\mathbf{b}_0), ct(\mathbf{b}_1), \dots, ct(\mathbf{b}_{k-1})$, for $0 \le i < k$, $\mathbf{b}_i = CRT^{-1}(b_{il}, b_{il+1}, \dots, b_{il+l-1})$ satisfy that $b_0, b_1, \dots, b_{kl-1}$ is a permutation of $a_0, a_1, \dots, a_{kl-1}$ and $b_0 \le b_1 \le \dots \le b_{kl-1}$
- 1: ct(\mathbf{a}'_i), $0 \le i < k \leftarrow$ Odd-Even Merge Sort based on SlotRelocate(ct(\mathbf{a}_i), $0 \le i < k$)
- 2: for i = 0 to k 1 do
- 3: $\operatorname{ct}(\mathbf{b}_i) \leftarrow \operatorname{SortSlot}(\operatorname{ct}(\mathbf{a}'_i))$
- 4: end for

made. In the for loop of algorithm 8, k **Slot Sort** are used. By (4.2), theorem 6, and theorem 3, the overall complexity is $O(k(\log k)^2)O(n\log n) + kO(n\log n) = O(kn(\log k)^2\log n)$ polynomial multiplications. When a = 0, i.e., k = O(1), its complexity is $O(kn(\log k)^2\log n) = O(n\log n)$ polynomial multiplications; when a > 0, its complexity is $O(kn(\log k)^2\log n) = O(an^{a+1}(\log n)^3)$ polynomial multiplications.

Section 3.4 presents three sorting algorithms based on different comparison methods. Next, we compare their complexity with that of algorithm 8.

When using a sorting algorithm based on polynomials over finite fields, a sequence of length kl needs to be sorted. By (3.8), (2.1) and (4.2), its complexity is $O(kl(\log kl)^2 \log n) = O(n^{a+1}(\log n)^3)$ polynomial multiplications. It can be seen that when a = 0, i.e., k = O(1), the complexity of algorithm 8 is $O(n \log n)$ polynomial multiplications, with an advantage of $O((\log n)^2)$ polynomial multiplications; while when a > 0, the complexity of algorithm 8 is $O(n^{a+1}(\log n)^3)$ polynomial multiplications, which is equivalent to the sorting algorithm based on polynomial comparison over finite fields.

When using a sorting algorithm based on TFHE bootstrapping, a sequence of length kl needs to be sorted. By (3.9), (2.1) and (4.2), its complexity is $O(kln(\log kl)^2(\log n)^2) = O(n^{a+2}(\log n)^4)$ polynomial multiplications. It can be seen that when a = 0, i.e., k = O(1), the complexity of algorithm 8 is $O(n \log n)$ polynomial multiplications, with an advantage of $O(n(\log n)^3)$; while when a > 0, the complexity of algorithm 8 is $O(n^{a+1}(\log n)^3)$ polynomial multiplications, with an advantage of $n \log n$.

When using a sorting algorithm based on parallel bootstrapping, a sequence of length kl needs to be sorted. By (3.10), (2.1) and (4.2), its complexity is $O(kl(\log kl)^2(\log n)^3) = O(n^{a+1}(\log n)^5)$ polynomial multiplications. It can be seen that when a = 0, i.e., k = O(1), the complexity of algorithm 8 is $O(n \log n)$ polynomial multiplications, with an advantage of $O((\log n)^4)$; while when a > 0, the complexity of algorithm 8 is $O(n^{a+1}(\log n)^3)$ polynomial multiplications, with an advantage of $O((\log n)^2)$.

For this situation, even if the input ciphertexts need to be converted in format, according to lemma 3, the complexity of these operations is at most only $O(n^{a+1/2} \log n)$, which

12

does not affect the comparison of complexity. For this problem, it can be generalized to a more general situation. Suppose the length of the sequence to be sorted is kl + r, where 0 < r < l, then at this time, k + 1 plaintext polynomials are needed to store the entire sequence, and only the first r slots in the last polynomial are integers that need to be sorted. For this situation, the largest element $\frac{p-1}{2}$ in \mathbb{Z}_p can be filled in the last l-r slots of the last polynomial, and then the encryption of these k + 1 plaintext polynomials can be substituted into algorithm 8 for sorting. Note that the elements filled in the last polynomial are the largest elements in \mathbb{Z}_p , so after sorting, the last l - rslots in the last plaintext polynomial are still these elements. Using this method, since only one more ciphertext is sorted compared to sorting k ciphertexts, it has no effect on the final complexity. Therefore, for this extended situation, the advantage of algorithm 8 still exists.

4.4 Homomorphic sorting of a ciphertext and multiple sequences with word-wise encryption

problem Statement:Let tk = l, there are k integer sequences of length t, for $0 \le i < k$, the *i*-th sequence is $a_{it}, a_{it+1}, \ldots, a_{it+t-1}$. For an input RLWE ciphertext ct(**a**), **a** = $\operatorname{CRT}^{-1}((a_0, \ldots, a_{l-1}))$, $a_i \in \mathbb{Z}p$, an output ct(**b**) is needed, where **b** = $\operatorname{CRT}^{-1}((b_0, \ldots, bl - 1))$ satisfies that for $\forall 0 \le i < k$, $b_{it}, b_{it+1}, \ldots, b_{it+t-1}$ is a permutation of the sequence $a_{it}, a_{it+1}, \ldots, a_{it+t-1}$, and $b_{it} \le b_{it+1} \le \cdots \le b_{it+t-1}$.

For this situation, we first set the parameters. For the parameters t, k, according to (2.1), this section sets:

$$t = O(l^{r}) = O(n^{r})$$

$$k = O(l^{1-r}) = O(n^{1-r})$$
(4.3)

where $0 \le r \le 1$.

In this problem, since all sequences are encrypted in the same ciphertext, all sequences can be operated on in parallel. As with the sorting problem of a single sequence with a single ciphertext, the ord value of each number in each subsequence is calculated, and sorting is performed accordingly. However, in the current situation, when a plaintext polynomial is shifted to the left once, the first element of each sequence will be moved to the slot where other sequences are located, resulting in errors in subsequent comparison function operations. To avoid such errors, a plaintext needs to be multiplied to clear these erroneous elements. The following is an example to illustrate the specific operation process.

- 1) Input $\mathbf{a} = \text{CRT}^{-1}(3, 1, 2, 5, 6, 5)$, where subsequence $S_1 = 3, 1, 2$ and subsequence $S_2 = 5, 6, 5$, the task is to simultaneously sort S_1 and S_2 separately.
- 2) Following the approach of Chapter 4, ord is calculated by left shifting. After shifting **a** to the left once, we get $\mathbf{c}'_1 = \operatorname{CRT}^{-1}(1, 2, 5, 6, 5, 3)$.
- 3) In the third slot of c'_1 , the element 5 that should have been in S_2 appears, and in the sixth slot, the element 3 that should have been in S_1 appears. Therefore, we need to multiply by a plaintext $g_1 =$

 $CRT^{-1}(1, 1, 0, 1, 1, 0)$ to eliminate these erroneous elements and get $c_1 = CRT^{-1}(1, 2, 0, 6, 5, 0)$.

- 4) If S_1, S_2 are shifted to the left by one position at the same time, then in the third slot, there should be element 3 from S_1 , and in the sixth slot, there should be element 5 from S_2 . We can shift **a** to the right twice to move 3 and 5 to the third and sixth slots, respectively, and then multiply the result by a plaintext $\mathbf{f}_2 = \operatorname{CRT}^{-1}(0,0,1,0,0,1)$ to get $\mathbf{d}_2 = \operatorname{CRT}^{-1}(0,0,3,0,0,5)$. Adding \mathbf{c}_1 and \mathbf{d}_2 together gives the result of shifting subsequence S_1 and subsequence S_2 to the left once each: $\mathbf{u}_1 = \operatorname{CRT}^{-1}(1,2,3,6,5,5)$.
- Calculate LT for and get 5) \mathbf{u}_1 а to $CRT^{-1}(1, 0, 0, 0, 1, 0)$. Calculate =EQ \mathbf{x}_1 \mathbf{u}_1 and a and multiply the result for = CRT⁻¹(0, 0, 1, 0, 0, 1) \mathbf{f}_2 to by get $\mathbf{y}_1 = CRT^{-1}(0, 0, 0, 0, 0, 1).$
- 6) Shift \mathbf{c}_1 to the left once again and multiply by $\mathbf{g}_2 = \operatorname{CRT}^{-1}(1,0,0,1,0,0)$ to get $\mathbf{c}_2 = \operatorname{CRT}^{-1}(2,0,0,5,0,0)$. Shift **a** to the right once and multiply by $\mathbf{f}_1 = \operatorname{CRT}^{-1}(0,1,1,0,1,1)$ to get $\mathbf{d}_2 = \operatorname{CRT}^{-1}(0,1,2,0,5,6)$. Adding \mathbf{c}_2 and \mathbf{d}_1 together gives the result of shifting subsequence S_1 and subsequence S_2 to the left twice each: $\mathbf{u}_2 = \operatorname{CRT}^{-1}(2,3,1,5,5,6)$.
- Calculate LT for \mathbf{u}_2 and 7) а to get $CRT^{-1}(1, 0, 1, 0, 1, 0)$. Calculate \mathbf{x}_2 =EQ for \mathbf{u}_2 and a and multiply the result $CRT^{-1}(0, 1, 1, 0, 1, 1)$ to by \mathbf{f}_1 = get $\mathbf{y}_2 = CRT^{-1}(0, 0, 0, 0, 0, 0).$
- 8) Add \mathbf{x}_1 , \mathbf{x}_2 , \mathbf{y}_1 , and \mathbf{y}_2 together to get $\mathbf{w}_0 = \text{CRT}^{-1}(2,0,1,0,2,1)$. It is the ord value of all the numbers in all subsequences in their corresponding subsequences.
- 9) Calculate **EQ** for \mathbf{w}_0 and the baseline $\mathbf{h} = CRT^{-1}(0, 1, 2, 0, 1, 2)$, and multiply the result by $\mathbf{a} = CRT^{-1}(3, 1, 2, 5, 6, 5)$ to get $\mathbf{b}_0 = CRT^{-1}(0, 0, 0, 5, 0, 0)$.
- 10) Following the previous process, shift \mathbf{w}_0 to the left once and multiply by \mathbf{g}_1 , shift \mathbf{w}_0 to the right twice and multiply by \mathbf{f}_2 , add the two results together to get $\mathbf{w}_1 = \operatorname{CRT}^{-1}(0, 1, 2, 2, 1, 0)$. Calculate **EQ** for \mathbf{w}_1 and the baseline **h** and multiply the result by $\mathbf{u}_1 = \operatorname{CRT}^{-1}(1, 2, 3, 6, 5, 5)$ to get $\mathbf{b}_1 = \operatorname{CRT}^{-1}(1, 2, 3, 0, 5, 0)$.
- 11) Shift \mathbf{w}_0 to the left twice and multiply by \mathbf{g}_2 , shift \mathbf{w}_0 to the right once and multiply by \mathbf{f}_1 , add the two results together to get $\mathbf{w}_2 = \text{CRT}^{-1}(1, 2, 0, 1, 0, 2)$. Calculate **EQ** for \mathbf{w}_2 and the baseline **h** and multiply the result by $\mathbf{u}_2 = \text{CRT}^{-1}(2, 3, 1, 5, 5, 6)$ to get $\mathbf{b}_2 = \text{CRT}^{-1}(0, 0, 0, 0, 0, 6)$.
- 12) Adding $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$ together gives the sorted result $\mathbf{b} = CRT^{-1}(1, 2, 3, 5, 5, 6).$

For the general case, Algorithm 9 gives how to calculate ord, and Algorithm 10 gives the sorting method.

Before proving the correctness of Algorithm 9, let's briefly summarize the content of each block of the algorithm. The first for loop (steps 3 to 8) of Algorithm 9 is to obtain the result of the left-shifted subsequence

Algorithm 9 SubSeqOrd(ct(a))

Input: ct(a), $\mathbf{a} = CRT^{-1}(a_0, ..., a_{l-1})$ $\operatorname{CRT}^{-1}(\operatorname{ord}(a_0),\ldots,$ **Output:** ct(w) satisfies w = $\operatorname{ord}(a_{l-1})$ 1: $\operatorname{ct}(\mathbf{c}_0) \leftarrow \operatorname{ct}(\mathbf{a})$ 2: $\operatorname{ct}(\mathbf{d}_0) \leftarrow \operatorname{ct}(\mathbf{a})$ 3: for i = 1 to t - 1 do $\mathbf{g}_i \leftarrow \operatorname{CRT}^{-1}(\underbrace{1,\ldots,1},\underbrace{0,\ldots,0},\ldots)$ 5: $\operatorname{ct}(\mathbf{c}_i) \leftarrow \mathsf{LShift}(\operatorname{ct}(\mathbf{c}_{i-1})) \cdot \mathbf{g}_i$ 6: $ct(\mathbf{d}_i) \leftarrow \mathbf{RShift}(ct(\mathbf{d}_{i-1})) \cdot \mathbf{f}_i$ 7: 8: end for 9: for i = 1 to t - 1 do $\operatorname{ct}(\mathbf{u}_i) \leftarrow \operatorname{ct}(\mathbf{c}_i) + \operatorname{ct}(\mathbf{d}_{t-i})$ 10: $\Big\{\mathbf{u}_i \ = \ \mathrm{CRT}^{-1}(a_i, \dots, a_{t-1}, a_0, \dots, a_{i-1}, \dots, a_{(k-1)t+i}, \dots, a_{kt-1}, a_{kt-1}, \dots, a_{kt-1}, \dots, a_{kt-1}, a_{kt-1}, \dots, a_{kt$ $a_{(k-1)t}, \ldots, a_{(k-1)t+i-1}\}$ 11: $\operatorname{ct}(\mathbf{x}_i) \leftarrow \operatorname{LT}(\operatorname{ct}(\mathbf{u}_i), \operatorname{ct}(\mathbf{c}_0))$ $\operatorname{ct}(\mathbf{y}_i) \leftarrow \mathbf{EQ}(\operatorname{ct}(\mathbf{c}_0), \operatorname{ct}(\mathbf{u}_i)) \cdot \mathbf{f}_{t-i}$ 12: 13: end for 14: $\operatorname{ct}(\mathbf{w}) \leftarrow \sum_{i=1}^{t-1} (\operatorname{ct}(\mathbf{x}_i) + \operatorname{ct}(\mathbf{y}_i))$

 $a_{it}, a_{it+1}, \ldots, a_{it+t-1}$ for $\forall 0 \leq i < k$. The second for loop (steps 9 to 13) of Algorithm 9 is to calculate the number of smaller numbers for each number in the subsequence, as well as the number of equal numbers that appear before it in the subsequence, in order to calculate the ord of each number in step 14. The proof of correctness and complexity of the Algorithm 9 is given in the theorem below.

Theorem 8. Algorithm 9 can correctly calculate the ord of each number in the sequence, with a complexity of $O(n^r \log n)$ polynomial multiplications.

Proof: First, we prove that in step 10 of Algorithm 9, let $\mathbf{u}_i = \operatorname{CRT}^{-1}(u_{i,0}, \ldots, u_{i,l-1})$, for $0 \le s < k, 0 \le j < t$, when i + j < t, we have $u_{i,st+j} = a_{st+j+i}$; otherwise, we have $u_{i,st+j} = a_{st+j+i-t}$. That is, we need to prove that $u_{i,st}, \ldots, u_{i,st+t-1}$ is obtained by shifting $a_{st}, \ldots, a_{st+t-1}$ to the left by *i* times.

In step 6 of Algorithm 9, \mathbf{c}_i is obtained by shifting \mathbf{c}_0 to the left by *i* times, obtaining $\operatorname{CRT}^{-1}(a_i, a_{i+1}, \dots, a_{l-1}, a_0, \dots, a_{i-1})$, and then multiplying by \mathbf{g}_i , i.e., $\mathbf{c}_i = \operatorname{CRT}^{-1}(a_i, a_{i+1}, \dots, a_{t-1}, 0, \dots, 0, a_{t+i}, a_{t+i+1}, \dots, a_{2t-1}, 0, \dots, 0, \dots, a_{(k-1)t+i}, a_{(k-1)t+i+1}, \dots, a_{(k-1)t-1}, 0, \dots, 0)$. Similarly, in step 7, $\mathbf{d}_i = \operatorname{CRT}^{-1}(0, \dots, 0, a_0, a_1, \dots, a_{t-i-1}, 0, \dots, 0, a_t, a_{t+1}, \dots, a_{2t-i-1}, \dots, 0, \dots, 0, a_{(k-1)t}, a_{(k-1)t+1}, \dots, a_{kt-i-1})$. Therefore, we have $\mathbf{u}_i = \mathbf{c}_i + \mathbf{d}_{t-i} = \operatorname{CRT}^{-1}(a_i, a_{i+1}, \dots, a_{t-1}, a_0, a_1, \dots, a_{t-1}, a_{t-$

 $\begin{array}{l} \ldots, a_{i-1}, \ldots, a_{(k-1)t+i}, a_{(k-1)t+i+1}, \ldots, a_{kt-1}, a_{(k-1)t}, \ldots, \\ a_{(k-1)t+i-1}), \text{ i.e., for } 0 \leq s < k, \ 0 \leq j < t, \text{ if } i+j < t, \\ \text{we have } u_{i,st+j} = a_{st+j+i}, \text{ if } i+j > t, \text{ we have } \\ u_{i,st+j} = a_{st+j+i-t}. \end{array}$

With the above conclusion, to prove the correctness of the Algorithm 9, in step 14 of Algorithm 9, let $\mathbf{w} = \text{CRT}^{-1}(w_0, \ldots, w_{l-1})$, we only need to prove that for any $0 \le s < t$, $w_s = \text{ord}(a_s)$. Since every t slots is a sequence,

we only need to prove that this algorithm is correct for the first sequence, and then it can be proved for the subsequent sequences in the same way.

In step 11 of Algorithm 9, let $\mathbf{x}_i = \operatorname{CRT}^{-1}(x_{i,0}, \ldots, x_{i,l-1})$; in step 12 of Algorithm 9, let $\mathbf{y}_i = \operatorname{CRT}^{-1}(y_{i,0}, \ldots, y_{i,l-1})$. Where $x_{i,s} = \operatorname{LT}(u_{i,s}, a_s) = \operatorname{LT}(a_{i+s}, a_s)$, the subscript of a_{i+s} is calculated modulo t. Therefore, as in the proof of Theorem 2, $\sum_{i=0}^{t-1} x_{i,s}$ is the number of numbers in the sequence a_0, \ldots, a_{t-1} that are less than a_s . Similarly, for any $0 \le i < s$, $\sum_{i=0}^{t-1} y_{i,s}$ is the number of numbers in the sequence a_0, \ldots, a_{s-1} that are equal to a_s . Hence $w_{0,s} = \sum_{i=0}^{t-1} (x_{i,s}+y_{i,s}) = \operatorname{ord}(a_s)$. This completes the proof of the correctness of Algorithm 9.

For the complexity of Algorithm 9, in the first for loop, a total of t - 1 **LShift** and t - 1 **RShift** are used, and in the second for loop, t - 1 **EQ** and t - 1 **LT** are used. According to (4.3), (3.4), (3.3) and Lemma 2, the overall complexity is $(2t-2)O(\log n) + (2t-2)O(\log n) = O(n^r \log n)$ polynomial multiplications.

Algorithm 10 MultipleSort(ct(a))

Input: kt - l, $ct(a), a = CRT^{-1}((a_0, ..., a_{l-1}))$ **Output:** $ct(\mathbf{b})$, $\mathbf{b} = CRT^{-1}((b_0, \dots, b_{l-1}))$, $\forall 0$ \leq $ar{i}$ < k, $b_{it}, b_{it+1}, \ldots, b_{it+t-1}$ is a permutation of $a_{it}, a_{it+1}, \ldots, a_{it+t-1}$ and $b_{it} \leq b_{it+1} \leq \cdots \leq b_{it+t-1}$ 1: $\mathbf{h} \leftarrow \operatorname{CRT}^{-1}(0, 1, \dots, t-1, \dots, 0, 1, \dots, t-1)$ (0 to t-1repeat k times) 2: $ct(\mathbf{w}_0) \leftarrow SubSeqOrd(ct(\mathbf{a}))$ 3: $\operatorname{ct}(\mathbf{v}_0) \leftarrow \operatorname{ct}(\mathbf{w}_0)$ 4: for i = 1 to t - 1 do $\operatorname{ct}(\mathbf{v}_i) \leftarrow \mathsf{LShift}(\operatorname{ct}(\mathbf{v}_{i-1})) \cdot \mathbf{g}_i$ 5: $\operatorname{ct}(\mathbf{w}_i) \leftarrow \operatorname{RShift}(\operatorname{ct}(\mathbf{w}_{i-1})) \cdot \mathbf{f}_i$ 6: 7: end for 8: $\operatorname{ct}(\mathbf{b}_0) \leftarrow \mathbf{EQ}(\operatorname{ct}(\mathbf{v}_0), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{c}_0)$ 9: for i = 1 to t - 1 do $\operatorname{ct}(\mathbf{b}_i) \leftarrow \operatorname{ct}(\mathbf{b}_{i-1}) + \mathbf{E}\mathbf{Q}(\operatorname{ct}(\mathbf{v}_i) + \operatorname{ct}(\mathbf{w}_{t-i}), \mathbf{h}) \cdot \operatorname{ct}(\mathbf{u}_i)$ 10: 11: end for 12: $\operatorname{ct}(\mathbf{b}) \leftarrow \operatorname{ct}(\mathbf{b}_{t-1})$

Before proving the correctness of Algorithm 10, let's briefly summarize the content of each block of Algorithm 10. The first for loop (steps 19 to 22) of Algorithm 10 is to shift the calculated ord in each subsequence to the left, in order to compare them with the benchmark h. The second for loop of Algorithm 10 is to add the comparison results and put each number in its sorted position. Next, we give the proof of the correctness of Algorithm 10.

Theorem 9. Algorithm 10 can correctly solve the sorting problem of multiple sequences with a single ciphertext in global encryption, with a complexity of $O(n^r \log n)$ polynomial multiplications.

Proof: By Theorem 1, we only need to prove that for $\mathbf{b} = \operatorname{CRT}^{-1}(b_0, \ldots, b_{l-1})$ in step 12 of Algorithm 10, for any $0 \le s < k, 0 \le j < t$, we have $b_{\operatorname{ord}(a_{st+j})+st} = a_{st+j}$. Since every *t* slots of **b** corresponds to a sequence, we only need to prove for the first sequence b_0, \ldots, b_{t-1} , i.e., prove that

for any $0 \le j < t$, $b_{\text{ord}(a_j)} = a_j$, and the other sequences can be proved in the same way.

In step 5 of Algorithm 10, let $\mathbf{v}_i = \text{CRT}^{-1}(v_{i,0}, \ldots, v_i)$

 $v_{i,l-1}$), \mathbf{v}_i is obtained by shifting \mathbf{v}_0 to the left by *i* times and then multiplying by g_i ; in step 6 of Algorithm 10, let $\mathbf{w}_i = \operatorname{CRT}^{-1}(w_{i,0}, \ldots, w_{i,l-1})$, \mathbf{w}_i is obtained by shifting \mathbf{w}_0 to the right by *i* times and then multiplying by f_i . This is the same as the construction of \mathbf{c}^i and \mathbf{d}^i in Algorithm 9, so we have $\mathbf{v}_{i,j} + \mathbf{w}_{t-i,j} = \operatorname{ord}(a_{i+j}), 0 \leq j < t$, in step 12 of the algorithm, let $\mathbf{b} = \operatorname{CRT}^{-1}(b_0, b_1, \ldots, b_{l-1})$. By steps 8 and 10 of Algorithm 10, we have $b_j = \sum_{i=0}^{t-1} \operatorname{EQ}(\operatorname{ct}(\mathbf{v}_{i,j}) + \operatorname{ct}(\mathbf{w}_{t-i,j}), \mathbf{h}_j) \cdot u_{i,j} = \sum_{i=0}^{t-1} \operatorname{EQ}(\operatorname{ord}(a_{i+j}), j) \cdot a_{i+j}$ (the last equality uses Theorem 8). It can be seen that when and only when $\operatorname{ord}(a_{i+j}) = j$, $\operatorname{EQ}(\operatorname{ord}(a_{i+j}), j) = 1$, so for any $0 \leq j < t$, $b_{\operatorname{ord}(a_{i+j})} = a_{i+j}$, where the subscript of a_{i+j} is processed modulo *t*. This completes the proof of the correctness of Algorithm 10.

For the complexity of Algorithm 10, Algorithm 9 is called in step 2, a total of t-1 **LShift** and t-1 **RShift** are used in the first for loop, and t-1 **EQ** are used in step 8 and the second for loop. According to (4.3), (3.3), Lemma 2 and Lemma 8, the overall complexity is $(2t-2)O(\log n) + (t-1)O(\log n) +$ $O(n^r \log n) = O(n^r \log n)$ polynomial multiplications.

Section 3.4 presents three sorting algorithms based on different comparison methods. Next, we compare their complexity with that of Algorithm 10 for sorting k sequences of length t.

When using the sorting algorithm based on polynomial over finite fields, by (3.8), (2.1) and 4.3, its complexity is $kO(t(\log t)^2 \log n) = O(n(\log n)^3)$ polynomial multiplications. The complexity of Algorithm 10 is $O(n^r \log n)$ polynomial multiplications, with an advantage of $O(n^{1-r}(\log n)^2)$ polynomial multiplications.

When using the sorting algorithm based on TFHE bootstrapping, by (3.9), (2.1) and (4.3), its complexity is $O(ktn(\log t)^2(\log n)^2) = O(n^2(\log n)^4)$ polynomial multiplications. The complexity of Algorithm 10 is $O(n^r \log n)$ polynomial multiplications, with an advantage of $O(n^{2-r}(\log n)^3)$ polynomial multiplications.

When using the sorting algorithm based on parallel bootstrapping, we need to sort k sequences of length t, where kt = l. By formula (3.10), formula (2.1) and formula (4.3), its complexity is $O(kt(\log t)^2(\log n)^3) = O(n(\log n)^5)$ polynomial multiplications. The complexity of Algorithm 10 is $O(n^r \log n)$ polynomial multiplications, with an advantage of $O(n^{1-r}(\log n)^4)$ polynomial multiplications.

For this situation, even if the input ciphertext needs to be converted in format, according to Lemma (3), the complexity of these operations is at most only $O(\sqrt{n} \log n)$. For Algorithm 10, if the input is LWE ciphertext and format conversion is required, then the complexity of these operations is $O(\sqrt{n} \log n)$ which occupies the main part, but even so, the complexity of Algorithm 10 is only $O(\sqrt{n} \log n)$. Compared to these three methods, it still has at least an advantage of $O(\sqrt{n}(\log n)^2)$ polynomial multiplications.

5 IMPLEMENT

This paper implements the new algorithm in a program and puts the source code in the appendix. The platform for code implementation is PALISADE (v1.11.9) [29]. In order to make a fair comparison with previous work, for the sorting algorithm based on polynomial comparison over finite fields and the sorting algorithm based on TFHE bootstrapping (because there is no code implementation for parallel bootstrapping so far, so no actual machine comparison has been made), the PALISADE code uploaded by others was downloaded and run on the same machine. All experiments were carried out on a laptop with an Intel® Core i5-9300H CPU and 16 GB of memory. Multithreading has been turned off.

The following parameter symbols are used in the display of subsequent experimental results:

- *p*: the input plaintext modulus;
- *q*: the input ciphertext modulus;
- *m*: the cyclotomic order of the ring *R*;
- *n*: the degree of the ring \mathcal{R} ($n = \varphi(m)$);
- *l*: the number of SIMD slots;
- Q: the ciphertext modulus when using bootstrapping;
- N: the degree of the ring when using bootstrapping;

This paper sets parameters according to the fully homomorphic standard [30].

- 1) Set security parameter $\lambda = 128$;
- 2) Two different $n = \Theta(\lambda)$ are selected for the experiment. This paper chooses two types, n = 612 and n = 480;
- 3) After *n* is determined, the corresponding *q* needs to be selected to ensure security. According to the fully homomorphic standard, it is necessary to ensure that $\log q \leq 25$. This paper chooses $q = 2^{12}$ and $q = 2^{14}$ to meet different plaintext modulus choices;
- After *p* and *n* are determined, *l* is uniquely determined;
- 5) For the selection of bootstrapping parameters, according to the suggestions of Micciancio, Polyakov, and Liu [16], choose N = 2048;
- 6) When N is determined, it is necessary to make $\log Q \le 54$ to ensure security. We choose $\log Q = 29$ for the experiment;

For the algorithms 4, 8, and 10in this paper, each randomly generated different sequences, stored in one or more RLWE ciphertexts, and repeated homomorphic sorting many times to obtain the average running time. For the sorting problem of a single ciphertext and a single sequence encrypted by *p*-bits, when using the sorting algorithm based on TFHE bootstrapping, this paper only considers the case where integers are encrypted bit by bit after being represented in binary, so in this case, for the selected plaintext modulus p = 17 and p = 61, no experiments were conducted on the *p*-ary sorting algorithm based on TFHE bootstrapping. In the following text, algorithm P is used to represent the sorting algorithm based on polynomial comparison over finite fields, and algorithm B is used to represent the sorting algorithm based on TFHE bootstrapping.

• Table 1 shows the results of experiments on the sorting problem of single ciphertext single sequence with word-wise encryption. From the table, it can be seen that Algorithm 4 is 3.0-8.2 times faster.

$(p,m,n,\ l,\log q,N)$	Algorithm	Total time(second)	Comparing to our method
(17, 1000, 010	Р	20893	9.2
(17, 1228, 612, 204, 12, 2048)	В	10034	4.4
201, 12, 2010)	4	2249	1
(61, 1240, 480, 240, 14, 2048)	Р	51941	6.4
	В	32987	4.0
	4	8061	1
TABLE 1			

Sorting problem of single ciphertext single sequence with word-wise encryption

$(p, m, n, l, k, \log q, N)$	Algorithm	Total time(second)	Comparing to our method
(17, 1228, 612, 204, 4, 12, 2048)	Р	40377	3.6
	В	19051	1.7
	8	10949	1
(61, 1240, 480, 240, 3, 14, 2048)	Р	73841	3.3
	В	43305	1.9
	8	22073	1

TABLE 2

Sorting problem of multiple ciphertexts single sequence with word-wise encryption

- Table 2 shows the results of experiments on the sorting problem of multiple ciphertexts single sequence with word-wise encryption. The experimental results show that Algorithm 8 is 0.7-2.6 times faster.
- Table 3 shows the results of experiments on the sorting problem of single ciphertext multiple sequences with word-wise encryption. The experiment shows that Algorithm 10 is 2.1-6.4 times faster.

In order to eliminate the impact of format conversion, we separately tested the time required for conversion between LWE and RLWE ciphertexts, and the experimental results are listed in Table 4. It can be seen that the time for format conversion is much smaller compared to the time for sorting, and does not affect the final comparison.

6 CONCLUSION

This paper mainly examines how to use SIMD technology to improve the computational efficiency of sequence sorting. For an integer sequence, this paper proposes a new ord function, uses its properties to give a SIMD sorting algorithm for the case of plaintext overall encryption of a single ciphertext and a single sequence, and proves that the new algorithm has more advantages in such problems. In addition, it is also extended to two other more complex situations, and new algorithms are proposed for these situations, and compared with previous methods; whether from theoretical complexity or experimental results, the new methods have advantages.

In particular, in the case of overall encryption of multiple ciphertexts and a single sequence, algorithm 7 in this paper can not only be used in sorting sequences, but also has more uses. For example, for the top-k recommendation system in machine learning, its purpose is to select the most suitable projects for users from a large number of recommended

$(p,m,n,\ l,k,\log q,N)$	Algorithm	Total time(second)	Comparing to our method	[1
(17, 1228, 612, 204, 17, 12, 2048)	Р	7258	7.4]
	В	3094	3.1	[1
	10	975	1	1
(61, 1240, 480, 240, 12, 14, 2048)	Р	18061	6.6	٦ ١
	В	9839	3.5	1
	10	2736	1	1

TABLE 3 Sorting problem of single ciphertext multiple sequences with word-wise encryption

	$(n, l, \log q)$	Total time(second)	
<i>l</i> LWE ciphertexts to one RLWE ciphertexts	(612, 204, 12)	10.166	
	(480, 240, 14)	23.291	
one RLWE ciphertexts to <i>l</i> LWE ciphertexts	(612, 204, 12)	10.166	
	(480, 240, 14)	23.291	
TABLE 4			

Running time of format transform

projects and display them to users. If privacy-protected machine learning is used, after calculating the weights of a large number of recommended projects, these weights are encrypted, and a ciphertext often encrypts the weights of many projects. It is necessary to select the k recommended projects with the highest weight from many ciphertexts. At this time, one can use algorithm 7 in this paper to rearrange the calculation results instead of using conventional sorting, and then only need to sort the k weights in the first ciphertext and send them. Algorithm 7 in this paper will have a certain application value in this situation.

REFERENCES

- [1] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.
- [2] C. A. Hoare, "Quicksort," The computer journal, vol. 5, no. 1, pp. 10–16, 1962.
- [3] C. Bron, "Merge sort algorithm [m1]," Communications of the ACM, vol. 15, no. 5, pp. 357–358, 1972.
- [4] R. Schaffer and R. Sedgewick, "The analysis of heapsort," Journal of Algorithms, vol. 15, no. 1, pp. 76–100, 1993.
- [5] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, 1968, pp. 307–314.
- [6] A. Chatterjee, M. Kaushal, and I. Sengupta, "Accelerating sorting of fully homomorphic encrypted data," in *Progress in Cryptology– INDOCRYPT 2013: 14th International Conference on Cryptology in India, Mumbai, India, December 7-10, 2013. Proceedings 14.* Springer, 2013, pp. 262–273.
- [7] G. S. Çetin, Y. Doröz, B. Sunar, and E. Savaş, "Low depth circuits for efficient homomorphic sorting," *Cryptology ePrint Archive*, 2015.
- [8] J.-W. Lee, Y.-S. Kim, and J.-S. No, "Analysis of modified shell sort for fully homomorphic encryption," *IEEE Access*, vol. 9, pp. 126198–126215, 2021.
- [9] D. L. Shell, "A high-speed sorting procedure," Communications of the ACM, vol. 2, no. 7, pp. 30–32, 1959.
- [10] M. Kim, H. T. Lee, S. Ling, and H. Wang, "On the efficiency of fhebased private queries," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 2, pp. 357–363, 2016.
- [11] B. H. M. Tan, H. T. Lee, H. Wang, S. Ren, and K. M. M. Aung, "Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2861–2874, 2020.

- [2] G. S. Cetin, E. Savaş, and B. Sunar, "Homomorphic sorting with better scalability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 760–771, 2020.
- 13] I. Iliashenko and V. Zucca, "Faster homomorphic comparison operations for bgv and bfv," *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 3, pp. 246–264, 2021.
- [4] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe," in Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27. Springer, 2021, pp. 670–699.
- [15] K. Kluczniak and L. Schild, "Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption," arXiv preprint arXiv:2109.02731, 2021.
- [16] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping," in Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II. Springer, 2023, pp. 130–160.
- [17] H. Narumanchi, D. Goyal, N. Emmadi, and P. Gauravaram, "Performance analysis of sorting of fhe data: integer-wise comparison vs bit-wise comparison," in 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). IEEE, 2017, pp. 902–908.
- [18] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," Designs, Codes and Cryptography, vol. 71, pp. 57–81, 2014.
- [19] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," ACM Transactions on Computation Theory (TOCT), vol. 6, no. 3, pp. 1–36, 2014.
- [20] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in Advances in Cryptology-CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Springer, 2012, pp. 868–886.
- [21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, 2012.
- [22] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead." in *Eurocrypt*, vol. 7237. Springer, 2012, pp. 465–482.
- [23] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37. Springer, 2018, pp. 360–384.
- [24] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34. Springer, 2014, pp. 297–314.
- [25] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in Advances in Cryptology– EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34. Springer, 2015, pp. 617– 640.
- [26] F.-H. Liu and H. Wang, "Batch bootstrapping i: A new framework for simd bootstrapping in polynomial modulus." Eurocrypt, 2023.
- [27] ——, "Batch bootstrapping ii: Bootstrapping in polynomial modulus only requires $\tilde{o}(1)$ fhe multiplications in amortization." Eurocrypt, 2023.
- [28] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.
- [29] "Palisade lattice cryptography library," https://palisade-crypto. org/, 2022.
- [30] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter *et al.*, "Homomorphic encryption standard," *Protecting privacy through homomorphic encryption*, pp. 31–62, 2021.