Numerical Methods for Comparison on Homomorphically Encrypted Numbers

Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, Keewoo Lee

Department of Mathematical Sciences, Seoul National University

Abstract. We propose a new method to compare numbers which are encrypted by Homomorphic Encryption (HE). Previously, comparison and min/max functions were evaluated using Boolean functions where input numbers are encrypted bit-wisely. However, the bit-wise encryption methods require relatively expensive computation of basic arithmetic operations such as addition and multiplication.

In this paper, we introduce iterative algorithms that approximately compute the min/max and comparison operations of several numbers which are encrypted word-wisely. From the concrete error analyses, we show that our min/max and comparison algorithms have $\Theta(\alpha)$ and $\Theta(\alpha \log \alpha)$ computational complexity to obtain approximate values within an error rate $2^{-\alpha}$, while the previous minimax polynomial approximation method requires the exponential complexity $\Theta(2^{\alpha/2})$ and $\Theta(\sqrt{\alpha} \cdot 2^{\alpha/2})$, respectively. We also show the (sub-)optimality of our min/max and comparison algorithms in terms of asymptotic computational complexity among polynomial evaluations to obtain approximate min/max and comparison results. Our comparison algorithm is extended to several applications such as computing the top-k elements and counting numbers over the threshold in encrypted state.

Our new method enables word-wise HEs to enjoy comparable performance in practice with bit-wise HEs for comparison operations while showing much better performance on polynomial operations. Computing an approximate maximum value of any two ℓ-bit integers encrypted by HEAAN, up to error $2^{\ell-10}$, takes only 1.14 milliseconds in amortized running time, which is comparable to the result based on bit-wise HEs.

1 Introduction

Homomorphic Encryption (HE) is a cryptographic primitive which allows arithmetic operations over encrypted data without any decryption process. From this distinctive property, HE has received lots of attention in many privacy preserving applications. The HE schemes can be classified as word-wise HEs [8, 14, 25, 29] and bit-wise HEs [17, 22] according to the basic operations provided by them. Basic operations of word-wise HEs are component-wise addition and multiplication of an encrypted array over $\mathbb{Z}_p$ for a positive integer $p > 2$ [8, 25] or the field $\mathbb{C}$ of complex numbers [14], and all other operations are built upon two basic operations. Contrary to word-wise HEs, basic operations of bit-wise HEs are logical gates such as NAND gate [22] and look-up table based operations [17, 18].

When input numbers are encrypted word-wisely, polynomial operations consisting of additions and multiplications are quite natural, but it is rather hard to carry out non-polynomial operations such as comparison and min/max functions. On the other hand, when each bit of ℓ-bit integers is encrypted separately (e.g., $a = \sum_{i=0}^{\ell-1} a_i 2^i$ is encrypted as $\texttt{Enc}(a_0), \texttt{Enc}(a_1), ..., \texttt{Enc}(a_{\ell-1})$), comparing two ℓ-bit integers is done by evaluating a Boolean function in $\Theta(\ell)$ homomorphic multiplications with depth $\log \ell$ [16]. However, this bit-wise encryption method is rather inefficient for homomorphic addition and multiplication since it requires sequential computation of each carry bit transferred from lower-bit operations.

In this paper, we propose an efficient numerical approach for comparison and min/max functions, which can be efficiently exploited for word-wise HEs. Instead of evaluating a Boolean function over bit-wisely encrypted inputs, we homomorphically evaluate iterative algorithms to obtain approximate max/min values and the comparison result over word-wisely encrypted inputs. We note that our methods are especially effective in real-world applications which requires several min/max or comparison operations between a large amount of polynomial operations, such as neural network and clustering algorithms.

1.1 Our Idea

To perform non-polynomial operations over word-wise HEs, the previous works [13, 30, 36] utilized general polynomial approximation methods (e.g., Taylor, least square, minimax approximation). To obtain the desired error bound in the given interval, they choose an appropriate polynomial degree of approximation. As the degree grows, the lower error is guaranteed but the higher computational cost is required.

To obtain an approximate value within $2^{-\alpha}$ relative error through general polynomial approximations, the approximate polynomial should have the degree at least $\Theta(2^\alpha)$ (see Section 6). However, the evaluation of a general polynomial of degree $\Theta(2^\alpha)$ requires at least exponential computational complexity $\Theta(2^{\alpha/2})$ [39]. In this respect, the general polynomial approximation methods, which mainly consider the optimality of polynomial degree rather than computational complexity, may not be the best solution for HE applications.

To this end, our main idea is to use some special polynomials, whose structure can be exploited by iterative algorithms, rather than general polynomials obtained by polynomial approximation methods. The evaluation of these well-structured polynomials requires less complexity than the evaluation of general polynomials of the same degree. To be precise, the well-structured polynomial of degree $2^{\Theta(d)}$ can be evaluated with only $\Theta(d)$ multiplications while the evaluation of a general polynomial of the same degree requires exponentially many multiplications.

We devise iterative (approximate) algorithms for min/max and comparison focusing on their numerical properties and exploiting classical iterative approximation for inverse and square root as building blocks. We first consider a problem of approximating the maximum value of two numbers without bit operations.

The max function can be simply expressed as following equation:

$$\max(a,b) = \frac{a+b}{2} + \frac{|a-b|}{2} = \frac{a+b}{2} + \frac{\sqrt{(a-b)^2}}{2}$$

Thus, we can reduce the problem of homomorphically evaluating the maximum function to that of efficiently evaluating the square root function.

For comparison of several distinct positive numbers, we may use another identity:

$$\lim_{k\to\infty} \frac{a_i^k}{a_1^k + \cdots + a_n^k} = \begin{cases} 1 & \textit{if } a_i \textit{ is maximal, and} \\ 0 & \textit{otherwise.} \end{cases}$$

That is, if we can efficiently evaluate the inverse function, we can exploit this to evaluate the comparison function. For $k = 2$, the equation can be interpreted as a sigmoid approximation of the step function which corresponds to the comparison operation (see Section 5). To sum up, we can reduce the problems obtaining comparison and min/max results to efficiently approximating square root and inverse functions by arithmetic operations.

In our algorithms, the size of intermediate values a_i^k grow exponentially as k increases, so they are not easy to be computed only with additions and multiplications in the bounded plaintext space. Instead, we remark that several most significant bits of a_i^k are sufficient for the approximate computation of our algorithms, and they can be obtained by an efficient bit-extraction [27, 32] or the rounding-off operation. We utilize the (approximate) rounding-off operation which is supported by an approximate HE scheme HEAAN [12, 14] almost for free computational cost.

1.2 Our Result

We introduce iterative algorithms for min/max and comparison with numerical approaches, which can be efficiently used for word-wise HEs. We further apply these algorithms to evaluate top-k maximum values and threshold counting. Through the rigorous analysis on the error compared to the true value, we compute the minimal depth and computational complexity of our algorithms, and provide the strategies to choose the number of iterations.

Compared to general polynomial approximation methods which have been used to deal with non-polynomial operations in HE, our method requires much less computation to obtain min/max and comparison result within a certain level of error. In fact, our `Min`/`Max` (resp. `Comp`) algorithms achieve (nearly) minimal asymptotic computational complexity among the polynomial evaluations to obtain approximate min/max (resp. comparison) results.

To show the practicality of our algorithms, we present some experimental results of our algorithms implemented on HEAAN. Specific results on our algorithms are summarized as follows:

First, for min/max algorithm,

– To obtain an approximate min/max value of two ℓ-bit integers a and b up to error $2^{\ell-\alpha}$ for $\alpha > 0$, our max algorithm denoted by `Max` requires $\Theta(\alpha)$ depth and complexity.
– Under the condition $|a-b| \geq c$ for some small $c > 0$, the required depth and complexity are reduced to $\Theta(\log \alpha + 2\log(1/c))$.
– The homomorphic evaluation of `Max` for two 32-bit integers a and b preserving top-10 most significant bits takes 75 seconds with 2^{16} plaintext slots (1.14 milliseconds as the amortized running time).

Second, for comparison algorithm,

– To obtain an approximate value of $\text{comp}(a, b) = (a > b?)$ with error bounded by $2^{-\alpha}$ where $\max(a, b)/\min(a, b) \geq c$ for some fixed $c > 1$, our comparison algorithm denoted by `Comp` requires $\Theta(\log(\alpha/\log c) \cdot \log(\alpha + \log(\alpha/\log c)))$ depth and complexity.
– The homomorphic evaluation of `Comp` for two 32-bit integers (whose ratio is larger than 1.01) with 14-bit precisions takes about 230 seconds with 2^{16} plaintext slots (3.5 milliseconds as the amortized running time).

We additionally provide some implementation results as applications of the comparison algorithm. For example, we can compute the index of the maximum element among 16 encrypted 7-bit integers (where the maximum is at least twice larger than the others) with 7-bit precisions with amortized running time of about 75.9 milliseconds. We also propose an efficient solution to the so-called "threshold counting" problem, which aims to count the number of data exceeding a certain value. For any 32 encrypted 7-bit integers, the amortized running time of our solution is 135 milliseconds.

1.3 Related Works

There are a lot of works that consider comparison-related operations in HE schemes [5, 6, 10, 16, 18, 20, 23, 37, 43]. Most of these works deal with min/max, equality test, and sorting based on the bit-wise encryption approach. In other words, to provide bit-wise access they encrypt each bit of numbers separately that can be optimized using plaintext batching technique.

Chillotti et al. [18] calculate the maximum of two numbers of which each bit is encrypted into a distinct ciphertext by a bit-wise HE scheme [17, 18]. They express the max function by controlled Mux gates via weighted finite automata approach, and the implementation of their max algorithm on 8-bit integers took approximately a millisecond. Some other works [16, 20, 37, 43] implemented a Boolean function corresponding to the comparison operation, where input numbers are still encrypted bit-wisely. Cheon et al. [16] calculate a comparison operation over two 10-bit integers in 307 milliseconds using the plaintext space $\mathbb{Z}_{2^{14}}$. More recent work of Crawford et al. [20] takes a few seconds to compute a comparison result of 8-bit integers. Since the comparison operation can be simultaneously done in 1800 plaintext slots, the amortized running time becomes just a few milliseconds. These bit-wise encryption methods show very nice performance on comparison operations as described above, but polynomial operations including addition and multiplication of large numbers are significantly inefficient compared to word-wise encryption methods.

On the other hand, Boura et al. [5] compute absolute function and sign function, which correspond to min/max and comparison respectively, over word-wisely encrypted numbers by approximating the functions via Fourier series over a target interval. This method has an advantage on numerical stability compared to general polynomial approximation methods: Since Fourier series is a periodic function, the approximate function does not diverge to ∞ outside of the interval, while approximate polynomials obtained by polynomial approximation methods diverge. The homomorphic evaluation of the sign function over wide-wisely encrypted inputs is also described in [6], which implemented the evaluation phase of discretized neural network based on HE. It utilizes the bootstrapping technique of [17] to homomorphically extract the sign value of the input number and bootstrap the corresponding ciphertext in the same time.

When applying min/max and comparison functions on real-world applications such as machine learning, there have been some attempts to detour these functions by substituting them with other HE-friendly operations. For example, Gilad-Bachrah et al. [30] expressed the max of positive numbers $a_1, ..., a_n$ as $\lim_{k\to\infty}(\sum_{i=1}^n a_i^k)^{1/k}$; however, they substituted the max function by the simple summation $\sum_{i=1}^n a_i$ due to the hardness of evaluating $x^{1/k}$ for large k in HE.

In general, polynomial evaluation methods including Horner's method and Paterson-Stockmeyer method [39] can also be viewed as iterative algorithms, but they require $\Theta(d)$ or $\Theta(\sqrt{d})$ computational complexity for the polynomial degree d. In the rest of our paper, the context "iterative algorithm" refers to an algorithm with a *log-degree* complexity. In [15], for example, Cheon et al. apply an iterative method that efficiently computes a trigonometric function for bootstrapping of HEAAN. Since evaluating an approximate polynomial of trigonometric function over the large interval $[-B, B]$ requires very large computational complexity $\Theta(\sqrt{B})$, they first compute an approximate polynomial of the trigonometric function over the sufficiently small interval $[-B/2^r, B/2^r]$, and then apply the double-angle formula r times iteratively. From this iterative method, they could reduce the computational complexity for the evaluation of trigonometric functions to the log-scale complexity $\Theta(\log B)$.

1.4 Road Map

In Section 2, we introduce some notations used in our paper and basics of HE. In Section 3, we introduce iterative algorithms for inverse and square root operations with analyses on the approximation error. In Section 4 and 5, we propose iterative algorithms to compute the approximate min/max and comparison results with concrete error analyses. In Section 6, we compare our methods with general polynomial approximation methods verifying the computational efficiency and the optimality of our methods. In Section 7, we apply the proposed algorithms to solve the threshold counting problem and obtain the top-k max, and show some implementation results on our algorithms based on HEAAN in Section 8.

2 Preliminaries

2.1 Notations

All logarithms are base 2 unless otherwise indicated. For a real-valued function f defined over $\mathbb{R}$ and a domain $I \subset \mathbb{R}$, we denote the infinite norm of f over the domain I by $||f||_{\infty,I} := \max_{x \in I} |f(x)|$. If $I = \mathbb{R}$, then we omit the second term of the subscript.

For a power-of-two integer N, we define a polynomial ring $R := \mathbb{Z}[X]/(X^N + 1)$. For an integer $q \geq 0$, a quotient polynomial ring R/qR is denoted by R_q. A positive integer d denotes the number of iterations in inverse and square root algorithms, and d' and t denote the numbers of iterations in the comparison algorithm.

2.2 Homomorphic Encryption

Homomorphic Encryption (denoted as HE afterwards) is a cryptographic primitive which allows arithmetic operations such as an addition and a multiplication over encrypted data without decryption process. HE is regarded as a promising solution which prevents private information leakage during analyses on sensitive data such as biomedical data and financial data. A number of HE schemes [4, 7, 8, 14, 17, 19, 21, 22, 25, 28, 29] have been suggested following Gentry's blueprint [26], and achieving successes in various applications [5, 11, 15, 30, 35].

An HE scheme consists of the following algorithms:

- $\underline{\texttt{Setup}(1^\lambda, L)}$. For a level parameter L and a security parameter λ, output the parameters params for the given HE scheme to achieve λ-bit security and be able to evaluate a depth-L circuit.
- $\underline{\texttt{KeyGen}(\mathsf{params})}$. For parameters params, output a public key pk, a secret key sk, and an evaluation key evk.
- $\underline{\texttt{Enc}_{\mathsf{pk}}(m)}$. For a message m, output the ciphertext ct of m.
- $\underline{\texttt{Dec}_{\mathsf{sk}}(\mathsf{ct})}$. For a ciphertext ct of m, output the message m.
- $\underline{\texttt{Add}_{\mathsf{evk}}(\mathsf{ct}_1, \mathsf{ct}_2)}$. For ciphertexts ct_1 and ct_2 of m_1 and m_2, output the ciphertext $\mathsf{ct}_{\mathtt{add}}$ of $m_1 + m_2$.
- $\underline{\texttt{Mult}_{\mathsf{evk}}(\mathsf{ct}_1, \mathsf{ct}_2)}$. For ciphertexts ct_1 and ct_2 of m_1 and m_2, output the ciphertext $\mathsf{ct}_{\mathtt{mult}}$ of $m_1 \cdot m_2$.

3 Iterative Algorithms for Inverse and Square root

In this section, we first introduce approximate algorithms computing the inverse and the square root of a real number through additions and multiplications, so that they can be efficiently computed based on word-wise HEs. We additionally analyze the error rate of each algorithm to measure the quality of the approximation.

3.1 Inverse Algorithm

One of the most common iterative algorithms to compute the inverse of a (positive) real number is Goldschmidt's division algorithm [31]. For $x \in (0,2)$, the main idea of Goldschmidt's algorithm $\texttt{Inv}(x;d)$ is

$$\frac{1}{x} = \frac{1}{1-(1-x)} = \prod_{i=0}^{\infty}\left(1+(1-x)^{2^i}\right) \approx \prod_{i=0}^{d}\left(1+(1-x)^{2^i}\right).$$

Since $x \in (0,2)$, the value $1+(1-x)^{2^i}$ converges to 1 as $i \to \infty$ so that the approximation holds for sufficiently large $d > 0$. The inversion algorithm for an input x and the number of iterations d, denoted by $\texttt{Inv}(x;d)$, is described in Algorithm 1.

Algorithm 1 $\texttt{Inv}(x;d)$

Input: $0 < x < 2$, $d \in \mathbb{N}$
Output: an approximate value of $1/x$ (refer Lemma 1)

1: $a_0 \leftarrow 2 - x$
2: $b_0 \leftarrow 1 - x$
3: **for** $n \leftarrow 0$ **to** $d-1$ **do**
4: $\quad b_{n+1} \leftarrow b_n^2$
5: $\quad a_{n+1} \leftarrow a_n \cdot (1 + b_{n+1})$
6: **end for**
7: **return** a_d

Lemma 1. *For $x \in (0,2)$ and a positive integer d, the error rate of the output of $\texttt{Inv}(x;d)$ compared to the true value $1/x$ is bounded by $(1-x)^{2^{d+1}}$. In fact, the error is always negative, i.e., the output of $\texttt{Inv}(x;d)$ is always smaller than $1/x$.*

Proof. We can simply compute $|\frac{a_d - 1/x}{1/x}| = 1 - x \cdot a_d = (1-x)^{2^{d+1}}$. □

Remark 1. Lemma 1 implies that if we have tighter lower/upper bound of x, then it guarantees an exponential convergence in the number of iteration d. For example, assuming that $x \in [2^{-n}, 1)$ for some $n \in \mathbb{N}$, the error rate of $\texttt{Inv}(x;d)$ is bounded by $(1-2^{-n})^{2^{d+1}}$ which implies that only $d = \Theta(\log \alpha + n)$ number of iterations suffice for Algorithm 1 to achieve the error bound $2^{-\alpha}$.

3.2 Square Root Algorithm

In order to compute the square root of a positive real number, we exploit a two-variable iterative method proposed by Wilkes in 1951 [44]. The algorithm consists of simple addition and multiplication operations for each iteration, and it has an exponential convergence rate depending on the input value.

Algorithm 2 Sqrt$(x; d)$

Input: $0 \le x \le 1$, $d \in \mathbb{N}$
Output: an approximate value of $\sqrt{x}$ (refer Lemma 2)

```
1: a_0 ← x
2: b_0 ← x − 1
3: for n ← 0 to d − 1 do
4:     a_{n+1} ← a_n (1 − b_n/2)
5:     b_{n+1} ← b_n^2 ((b_n − 3)/4)
6: end for
7: return a_d
```

Lemma 2. *For $x \in (0, 1)$ and a positive integer d, the error rate of the output of* Sqrt$(x; d)$ *compared to the true value $\sqrt{x}$ is bounded by $(1 - \frac{x}{4})^{2^{d+1}}$. In fact, the error is always negative, i.e., the output of* Sqrt$(x; d)$ *is always smaller than $\sqrt{x}$.*

Proof. Since $-1 \le b_0 \le 0$, we can easily check that $-1 \le b_n \le 0$ for all $n \in \mathbb{N}$. Then, $|b_{n+1}| = |b_n| \cdot |\frac{b_n(b_n-3)}{4}| \le |b_n|$, gives $|b_{n+1}| \le |b_n|^2 \cdot (1 - \frac{x}{4})$, and it holds that $|b_d| \le |b_0|^{2^d} \cdot (1 - \frac{x}{4})^{2^d - 1} < (1 - \frac{x}{4})^{2^{d+1}}$.

From the definition of a_n and b_n, the equality $x(1 + b_n) = a_n^2$ can be obtained by simple induction. Hence, the error rate is

$$\left| \frac{a_n - \sqrt{x}}{\sqrt{x}} \right| = 1 - \sqrt{1 + b_n} < |b_n| ,$$

which implies the result of the lemma. □

Remark 2. Similarly to Remark 1, Lemma 2 implies that if we have tighter lower/upper bound of x, it guarantees an exponential convergence rate, e.g., if $x \in [2^{-n}, 1)$, then $d = \Theta(\log \alpha + n)$ iterations are sufficient for Algorithm 2 to achieve the error bound $2^{-\alpha}$.

Absolute value. By observing $|x| = \sqrt{x^2}$, we can also compute the absolute value of $-1 \le x \le 1$ by Sqrt$(x^2; d)$ for some sufficiently large $d > 0$. By Lemma 2, the error rate compared to the true value $|x|$ is bounded by $\left(1 - \frac{x^2}{4}\right)^{2^{d+1}}$.

4 Approximate min/max Algorithms

In this section, we describe approximate algorithms for min/max operations applying the square root algorithm described in the previous section. Our main goal is to obtain the min/max value and the comparison result between ℓ-bit positive integers (or ℓ-bit precision positive real numbers) for some given integer $\ell > 0$. Since our inverse and square root algorithms require input value to be contained in a prefixed interval (e.g., $[0,1]$), we need to scale down the large input values into small range. For this reason, when two inputs $\bar{a}, \bar{b} \in [0, 2^\ell)$ are given, we first scale down

$$(a,b) \leftarrow \left(\frac{\bar{a}}{2^\ell}, \frac{\bar{b}}{2^\ell}\right)$$

so that $a, b \in [0,1)$. After running the algorithms we desired, we will scale up the output value by the factor 2^ℓ. For example, after we obtain an approximate value x of $\max(a,b)$, then we can compute $2^\ell \cdot x \approx \max(\bar{a}, \bar{b})$. Note that this scaling procedure preserves the error rate compared to the true value.

4.1 min/max Algorithm for two numbers

In this subsection, we describe the Min and Max algorithms which approximately compute the minimum and maximum values of given two inputs contained in $[0,1)$, respectively. The approximate min/max algorithms, which we denote by `Min` and `Max`, respectively, can be directly obtained from the following observations.

$$\min(a,b) = \frac{a+b}{2} - \frac{|a-b|}{2} = \frac{a+b}{2} - \frac{\sqrt{(a-b)^2}}{2} \text{ and}$$
$$\max(a,b) = \frac{a+b}{2} + \frac{|a-b|}{2} = \frac{a+b}{2} + \frac{\sqrt{(a-b)^2}}{2}.$$

For the square root part of the formula we will use the square root algorithm described in Section 3.2 as a subroutine, which leads us to the algorithms:

$$\texttt{Min}(a,b;d) = \frac{a+b}{2} - \frac{\texttt{Sqrt}((a-b)^2;d)}{2}, \text{ and}$$
$$\texttt{Max}(a,b;d) = \frac{a+b}{2} + \frac{\texttt{Sqrt}((a-b)^2;d)}{2}.$$

Algorithm 3 $\texttt{Min}(a,b;d)$, $\texttt{Max}(a,b;d)$

Input: $a, b \in [0,1)$, $d \in \mathbb{N}$
Output: an approximate value of $\min(a,b)$ and $\max(a,b)$ (refer Theorem 1,2)
1: $x = \frac{a+b}{2}$ and $y = \frac{a-b}{2}$
2: $z \leftarrow \texttt{Sqrt}(y^2; d)$
3: **return** $x - z$ for $\texttt{Min}(a,b;d)$
$x + z$ for $\texttt{Max}(a,b;d)$

Assume that one would like to obtain a good enough approximate values of min/max of $a, b \in [0,1)$. Roughly speaking, we can obtain an approximate min/max value with an error up to $2^{-\alpha}$ in about 2α iterations.

Theorem 1. *If $d \geq 2\alpha - 3$ for some $\alpha > 0$, then the error of* $\texttt{Max}(a, b; d)$ *(resp.* $\texttt{Min}(a, b; d)$*) from the true value* $\max(a, b)$ *(resp.* $\min(a, b)$*) is bounded by* $2^{-\alpha}$ *for any* $a, b \in [0, 1)$.

Proof. By Lemma 2, we obtain $\left|\texttt{Sqrt}((a-b)^2; d) - |a-b|\right| < \left(1 - \frac{(a-b)^2}{4}\right)^{2^{d+1}} \cdot |a-b|$. Therefore, the error of $\texttt{Max}(a, b; d)$ (resp. $\texttt{Min}(a, b; d)$) from $\max(a, b)$ (resp. $\min(a, b)$) is bounded by $\frac{1}{2} \cdot \left(1 - \frac{(a-b)^2}{4}\right)^{2^{d+1}} \cdot |a - b|$.

Considering $|a - b|$ as a variable x, let us find the maximal value of $f(x) = (1 - \frac{x^2}{4})^{2^{d+1}} \cdot x$ for $x \in [0, 1)$. By a simple computation, one can check that $f'(x) = (1 - \frac{x^2}{4})^{2^{d+1}-1} \cdot \left(1 - \left(\frac{1}{4} + 2^d\right) x^2\right) = 0$ has a unique solution $x_0 = 1/\sqrt{2^d + \frac{1}{4}}$ in $[0, 1)$ so that x_0 is the maximal point of $f(x)$. Hence, we obtain the following inequality

$$\begin{aligned}\left(1 - \frac{(a-b)^2}{4}\right)^{2^{d+1}} \cdot |a - b| &\leq \left(1 - \frac{1}{2^{d+2} + 1}\right)^{2^{d+1}} \cdot \frac{1}{\sqrt{2^d + \frac{1}{4}}} \\ &< \frac{1}{\left(1 + \frac{1}{2^{d+2}}\right)^{2^{d+1}}} \cdot 2^{-\frac{d}{2}} < 2^{-\frac{d+1}{2}},\end{aligned}$$

using the fact that $(1 + x)^{1/x} \geq 2$ for $x \in [0, 1)$. Therfore, under the condition $d > 2\alpha - 3$, the error of $\texttt{Max}(a, b; d)$ (and $\texttt{Min}(a, b; d)$) is upper bounded by $2^{-\alpha}$. □

By Theorem 1, we can select an appropriate parameter d depending on α, i.e., the quality of the approximation. For example, let $\ell = 64$ so that $\bar{a}$ and $\bar{b}$ are 64-bit positive integers. If one aims to obtain exact maximum value between $\bar{a}$ and $\bar{b}$, then one can set $d = 2 \cdot 64 - 3 = 125$. But if one only aims to obtain an approximate value within an error less than 2^{48}, i.e., obtain the top 16 bits of the maximum value in 64-bit representation, one can set much smaller d as $d = 2 \cdot 16 - 3 = 29$. In this case, the output would be a 64-bit integer of which top-16 bits coincide with those of the true maximum value.

Parameter Reduction over the Restricted Domain. We can improve the condition on the parameter d in Theorem 1 from $\Theta(\alpha)$ to $\Theta(\log \alpha)$ by adding some conditions on a and b: $|a - b| \geq c$ for some constant $0 < c < 1$. In other words, $d = \Theta(\log \alpha)$ provides appropriate min/max results with probability $(1 - c)^2$ for uniform randomly chosen a and b from $[0, 1)$.

Theorem 2. *If $d \geq \log \alpha + 2 \log(1/c) + 1$ for some $\alpha > 0$ and $0 < c < 1$, then the error of* $\texttt{Max}(a, b; d)$ *(resp.* $\texttt{Min}(a, b; d)$*) from the true value* $\max(a, b)$ *(resp.* $\min(a, b)$*) is bounded by* $2^{-\alpha}$ *for any* $a, b \in [0, 1)$ *satisfying* $|a - b| \geq c$.

Proof. We resume at the upper bound $\frac{1}{2} \cdot \left(1 - \frac{(a-b)^2}{4}\right)^{2^{d+1}} \cdot |a - b|$ of the error of $\texttt{Max}(a, b; d)$ (resp. $\texttt{Min}(a, b; d)$) from $\max(a, b)$ (resp. $\min(a, b)$) as in the proof of Theorem 1.

Since $|a-b| \geq c$, we obtain

$$\frac{1}{2} \cdot \left(1 - \frac{(a-b)^2}{4}\right)^{2^{d+1}} \cdot |a-b| \leq \left(1 - \frac{c^2}{4}\right)^{2^{d+1}}.$$

Since $(1-x)^{1/x} < \frac{1}{e} < \frac{1}{2}$ for $0 < x < 1$, if $d \geq \log \alpha + 2\log(1/c) + 1$, it holds that

$$\left(1 - \frac{c^2}{4}\right)^{2^{d+1}} = \left(\left(1 - \frac{c^2}{4}\right)^{4/c^2}\right)^{2^{(d+2\log c-1)}} < 2^{-2^{(d+2\log c-1)}} \leq 2^{-\alpha},$$

which is the conclusion we wanted. □

Note that the area of the bad region $\{(a,b) \in [0,1) \times [0,1) : |a-b| \leq c\}$, where the theorem does not hold, is $1-(1-c)^2$ ($\approx 2c$ if c is very small). Consider a, b as a uniform random variable in $[0,1)$, and assume that we want to obtain an appropriate output of $\texttt{Max}(a,b;d)$ and $\texttt{Min}(a,b;d)$ with probability $1-\epsilon$ for $0 < \epsilon < 1$. Then by combining the results from Theorem 1 and Theorem 2, it suffices to set $d \approx \min(2\alpha - 3, \log \alpha + 2\log(1/c) + 1)$.

Depth and Complexity of `Min`/`Max` Algorithms. Since the depth of the $\texttt{Sqrt}(\cdot;d)$ algorithm is $2d+1$, the depth of $\texttt{Min}(\cdot,\cdot;d)$ and $\texttt{Max}(\cdot,\cdot;d)$ algorithms is also $2d+1$. Since the algorithm is iterative, the complexity is indeed $\Theta(d)$.

4.2 Min/max Algorithm for Several Numbers

With a basic min/max algorithm for two numbers in Section 4.1, we are able to construct a min/max algorithm for several numbers. Let $a_{1,0}, a_{2,0}, ..., a_{n,0}$ be given numbers contained in $[0,1)$, and our aim is to obtain an approximate value of the maximum value among them. For convenience of analysis, assume that n is a power-of-two integer. For some positive integer $d > 0$, we first run $\texttt{Max}(a_{2i-1,0}, a_{2i,0}; d)$ for $1 \leq i \leq n/2$ and denote the outputs by $a_{i,1}$, respectively. Repeatedly, we obtain the outputs $a_{i,2}$ of $\texttt{Max}(a_{2i-1,1}, a_{2i,1})$ for $1 \leq i \leq n/4$. Then, we can inductively construct a binary tree structure $\{a_{i,j}\}_{0 \leq j \leq \log n, 1 \leq i \leq n/2^j}$, and $a_{1,\log n}$ would be the desired approximate maximum value. The same argument can be applied to the case of `Min` algorithm.

The following theorem shows the error bound of `ArrayMax` and `ArrayMin` algorithms.

Theorem 3. *Let n be a power-of-two integer. The numbers $a_1, a_2, ..., a_n \in [0,1)$ satisfying $|a_i - a_j| \geq c > 0$ for any $1 \leq i < j \leq n$ are given. When $d \geq \log(\alpha + \log\log n) + 2\log(1/c) + 1$, the error of the output of $\texttt{ArrayMax}(a_1, a_2, ..., a_n; d)$ (resp. $\texttt{ArrayMin}(a_1, a_2, ..., a_n; d)$) from the true value $\max(a_1, a_2, ..., a_n)$ (resp. $\min(a_1, a_2, ..., a_n)$) is bounded by $2^{-\alpha}$. Note that the error is always negative, i.e., the output value is always smaller than the true value.*

Proof. Refer to Appendix A. □

Algorithm 4 ArrayMax$(a_1, a_2, ..., a_n; d)$

Input: $a_1, a_2, ..., a_n \in [0, 1)$, $d \in \mathbb{N}$
Output: an approximate value of $\max(a_1, a_2, ..., a_n; d)$ (refer Theorem 3)

$(a_{1,0}, a_{2,0}, ..., a_{n,0}) \leftarrow (a_1, a_2, ..., a_n)$
$d \leftarrow n$
for $j \leftarrow 0$ **to** $\lfloor \log n \rfloor$ **do**
  if d is odd **then**
    $a_{\lceil d/2 \rceil, j+1} \leftarrow a_{d,j}$
  end if
  $d \leftarrow \lfloor n/2 \rfloor$
  for $i \leftarrow 1$ **to** d **do**
    $a_{i,j+1} \leftarrow \texttt{Max}(a_{2i-1,j}, a_{2i,j}; d)$
  end for
end for
return $a_{1,\lceil \log n \rceil}$

Theorem 2 was applied in this theorem for the good region $\{(a_i)_{1 \le i \le n} \in [0, 1)^n : |a_i - a_j| \ge c$ for any $1 \le i < j \le n$ and some $c > 0\}$. Note that we can also apply Theorem 1 to obtain the worst-case analysis: In this case, d should be set as $d = 2(\alpha + \log\log n) - 3$. The area of the good region, is exactly $(1 - (n-1)c)^n$ ($\approx 1 - n(n-1)c$ when c is very small) referring to [9]. Therefore, if one want to obtain an output of ArrayMax or ArrayMin within error $2^{-\alpha}$ with probability $1 - \epsilon$ for $0 < \epsilon < 1$, then by Theorem 3 it suffices to set $d \approx \min(2(\alpha + \log\log n) - 3, \log(\alpha + \log\log n) + 2\log(1/c) + 1)$.

Remark 3. We set n be a power-of-two integer for convenience of the error analysis, but the theorem still holds for a non-power-of-two integer n.

Depth and Complexity of ArrayMin/ArrayMax Algorithms. Since we constructed a binary tree of depth $\log n$ with the number of nodes n, the depth is $\log n \cdot (2d + 1)$ and the complexity is $\Theta(nd)$.

5 Approximate Comparison Algorithms

In this section, we propose approximate comparison algorithms for various purposes. The core idea of algorithms starts with a simple fact that the comparison result of two numbers a and b can be evaluated as $\mathrm{comp}(a, b) := \chi_{(0,\infty)}(a - b)$ where $\chi_{(0,\infty)}$ is a step function over $\mathbb{R}$ defined as

$$\chi_{(0,\infty)}(x) := \begin{cases} 1 & \text{when } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

However, it is challenging to evaluate discontinuous functions such as $\chi_{(0,\infty)}$ in word-wise HE. To overcome this problem, we first approximate the step function by a globally smooth function called sigmoid $\sigma(x) = 1/(1+e^{-x})$. The error between the sigmoid and $\chi_{(0,\infty)}$ can be controlled by scaling the sigmoid as $\sigma_k(x) := \sigma(kx)$. Following the notation, it holds that

$$\lim_{k\to\infty} ||\chi_{(0,\infty)} - \sigma_k||_{\infty,\mathbb{R}-\{0\}} = 0.$$

In other words, we can approximately evaluate the step function $\chi_{(0,\infty)}$ through the scaled sigmoid function σ_k for sufficiently large k.

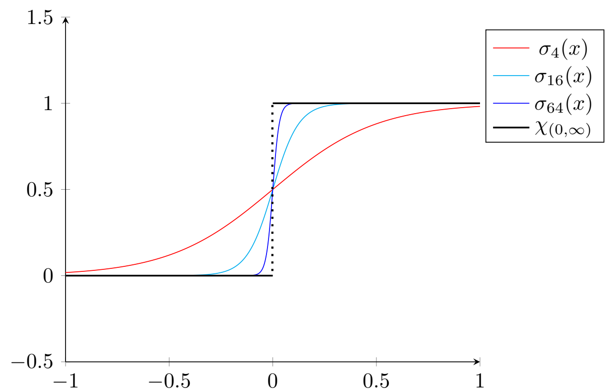

Fig. 1. Approximation of the step function $\chi_{(0,\infty)}$ by scaled sigmoid functions

Though a scaled sigmoid function is a continuous function contrary to $\chi_{(0,\infty)}$, $\sigma_k(a-b) = e^{ka}/(e^{ka}+e^{kb})$ still requires exponential function evaluations which cannot be easily done in HE. This obstacle can be simply overcome by taking logarithm on each input of comparison. Since the log function is a strictly increasing function, it does not reverse the order, i.e., $\log a > \log b$ if and only if $a > b$. Therefore, the evaluation of $\chi_{(0,\infty)}$ on $x = \log a - \log b$ also outputs the correct comparison result of a and b. As a result, we obtain the following approximation formula:

$$\text{comp}(a,b) \approx \sigma_k(\log a - \log b) = \frac{e^{k\log a}}{e^{k\log a} + e^{k\log b}} = \frac{a^k}{a^k+b^k}.$$

5.1 Comparison between two numbers

In this subsection, we discuss how to efficiently evaluate the approximate comparison equation $a^k/(a^k+b^k) \approx \text{comp}(a,b)$ with basic operations such as addition and multiplication. For given two ℓ-bit positive integers $\bar{a}$ and $\bar{b}$, we first scale them down to $a, b \in \left[\frac{1}{2}, \frac{3}{2}\right)$ via the mapping $\bar{x} \mapsto x := \frac{1}{2} + \frac{x}{2^\ell}$ which is *order-preserving*, i.e., $x > y$ if and only if $\bar{x} > \bar{y}$. We may scale those ℓ-bit integers to

$[0,1)$ as in min/max algorithms, but note that the range $\left[\frac{1}{2},\frac{3}{2}\right)$ is more suitable than $[0,1)$ to exploit Inv algorithm.

From the observation in the beginning of Section 5, the following equations hold:

$$\lim_{k\to\infty}\frac{\max(a,b)^k}{a^k+b^k}=1, \text{ and } \lim_{k\to\infty}\frac{\min(a,b)^k}{a^k+b^k}=0 \text{ if } a\neq b, \tag{1}$$

so that we obtained the approximate values if we set sufficiently large $k>0$. Our comparison algorithm denoted by Comp is described as Algorithm 5.

Algorithm 5 $\texttt{Comp}(a,b;d,d',t,m)$

Input: distinct numbers $a,b\in\left[\frac{1}{2},\frac{3}{2}\right)$, $d,d',t,m\in\mathbb{N}$
Output: an approximate value of $\text{comp}(a,b)$ (refer Theorem 4)

$a_0 \leftarrow \frac{a}{2}\cdot\texttt{Inv}\left(\frac{a+b}{2};d'\right)$
$b_0 \leftarrow 1-a_0$
for $n\leftarrow 0$ **to** $t-1$ **do**
  $inv \leftarrow \texttt{Inv}(a_n^m+b_n^m;d)$
  $a_{n+1}\leftarrow a_n^m\cdot inv$
  $b_{n+1}\leftarrow 1-a_{n+1}$
end for
return a_t

The first preparatory stage of the algorithm is to (1-norm) normalize the given input into the new pair (a,b) with $a,b\in[0,1]$ satisfying $a+b=1$. This normalization provides lower and upper bounds $1/2^{k-1}\le a^k+b^k\le 1$ so that a^k+b^k can be an appropriate input of Inv algorithm. The next step is to approximate the value of $a^k/(a^k+b^k)$. One naive approach could be to compute $a^k\cdot\texttt{Inv}(a^k+b^k;d)$ for some positive integer $d>0$. However, since the a^k+b^k could be as small as $1/2^{k-1}$, it requires a large parameter d for sufficiently nice approximation of $1/(a^k+b^k)$ with Inv algorithm (See Remark 1). Since the level parameter L of HE linearly grows to the parameter d, this large d is definitely an obstacle to the performance of the comparison algorithm based on HE.

In order to overcome this bottleneck we approximate the value of $a^k/(a^k+b^k)$ by performing the operation $a^m\cdot\texttt{Inv}(a^m+b^m;d)$ repeatedly for small m. The additional parameter m, which we normally choose as a power-of-two integer, satisfies $m^t=k$. As an illustration, let us take the two steps of the iteration. We first compute $(a_1,b_1)=(\frac{a^m}{a^m+b^m},\frac{b^m}{a^m+b^m})$ applying $\texttt{Inv}(a^m+b^m;d)$, and then compute $(a_2,b_2)=(\frac{a_1^m}{a_1^m+b_1^m},\frac{b_1^m}{a_1^m+b_1^m})=(\frac{a^{2m}}{a^{2m}+b^{2m}},\frac{b^{2m}}{a^{2m}+b^{2m}})$ again using $\texttt{Inv}(a_1^m+b_1^m;d)$. Then, in t steps we arrive at $\frac{a^{m^t}}{a^{m^t}+b^{m^t}}=\frac{a^k}{a^k+b^k}$.

This modification requires more Inv algorithms to be used, but it allows us to set much smaller d for Inv algorithm, because a^m+b^m at each steps is in the range $[1/2^{m-1},1]$ while a^n+b^n is in the range $[1/2^{n-1},1]$. Therefore, it makes a trade-off between the number of iterations t and the parameter d.

Theorem 4. *Let* $a, b \in \left[\frac{1}{2}, \frac{3}{2}\right)$ *satisfying* $\max(a,b)/\min(a,b) \geq c$ *for some fixed* $1 < c < 3$. *When* $t \geq \frac{1}{\log m}[\log(\alpha+1) - \log\log c]$, $d \geq \log(\alpha+t+2)+m-2$, *and* $d' \geq \log(\alpha+2)-1$, *the error of (the vector)* $\texttt{Comp}(a,b;d,d',t,m)$ *compared to the true value* $comp(a,b)$ *is bounded by* $2^{-\alpha}$. *Note that the error is always toward 1/2, i.e., the output value is always in between 1/2 and the true value.*

Proof. Without loss of generality we may assume that $a > b$. Note that the step 1 and 2 of our algorithm scales a, b to non-negative numbers a_0, b_0 with $a_0 + b_0 = 1$. Let us execute the first round of iteration. Note that

$$\left| a_0^m \texttt{Inv}(a_0^m + b_0^m; d) - \frac{a_0^m}{a_0^m + b_0^m} \right| = a_0^m \cdot |\texttt{Inv}(a_0^m + b_0^m; d) - (a_0^m + b_0^m)^{-1}|$$

$$\leq (1 - (a_0^m + b_0^m)^{-1})^{2^{d+1}} \cdot \frac{a_0^m}{a_0^m + b_0^m}.$$

Since $(1 - (a_0^m + b_0^m)^{-1})^{2^{d+1}} < e^{-2^{d+1}/2^{m-1}} < 2^{-2^{d-m+2}}$ from the lower bound estimate $a_0^m + b_0^m \geq 2^{-m+1}$, we can conclude that the error rate for one iteration is bounded by $K = 2^{-2^{d-m+2}}$. Thus, the error rate for t iterations is bounded by $1 - (1-K)^t \leq tK < 2^t K$. Since we want this bound to be smaller than $2^{-\alpha-2}$ we get the desired lower bound for d, namely $d \geq \log(\alpha + t + 2) + m - 2$.

Now we wish to bound the difference

$$\left| 1 - \frac{a^{m^t}}{a^{m^t} + b^{m^t}} \right| = 1 - \frac{1}{1 + (b/a)^{m^t}} \leq \left(\frac{b}{a}\right)^{m^t} \leq c^{-m^t}$$

by $2^{-\alpha-1}$, which leads us to the condition $t \geq \frac{1}{\log m}[\log(\alpha+1) - \log\log c]$.

Finally, we examine the step 1 and 2 of our algorithm, whose error rate is bounded by $2^{-2^{d'+1}}$. If we require this bound to be smaller than $2^{-\alpha-2}$, we get the condition $d' \geq \log(\alpha+2) - 1$, which is implied by our assumption on d'.

Summing up all the error rate, we get the conclusion we wanted.

□

Remark 4. We note that introducing the condition on the ratio of inputs with the constant c is not unrealistic or harsh. In the case of n-bit integers, setting the lower bound $c = a/b \geq \left(\frac{1}{2} + \frac{2^n-1}{2^n}\right) / \left(\frac{1}{2} + \frac{2^n-2}{2^n}\right)$ allows us to compare *any* two n-bit integers. Similar argument also applies to the case of real numbers, if we consider finite precision and input bounds. To sum up, an appropriate c generally exists in real-world applications.

Depth and Complexity of `Comp` Algorithm. The depth and complexity of `Comp` is $d'+1+t(d+\log m+2)$ and $\Theta(d'+t(d+\log m))$ respectively. When we set $m = 2$ which roughly gives $t = \log(\alpha/\log c)$ and $d = \log(\alpha + \log(\alpha/\log c))$, those depth and complexity are optimized as $\Theta(\log(\alpha/\log c) \cdot \log(\alpha + \log(\alpha/\log c)))$. For $c = 1 + 2^{-\alpha}$, it is simplified as $\Theta(\alpha \log \alpha)$.

5.2 Max Index of several numbers

Given several distinct numbers $a_1, a_2, ..., a_n \in \left[\frac{1}{2}, \frac{3}{2}\right)$, assume that we want to obtain the index of the maximum value. This problem can be easily solved by observing Equation (1) with another point of view. As the exponent k increases, then the gap between $\max(a,b)^k$ and $\min(a,b)^k$ becomes larger so that $\max(a,b)^k$ becomes a dominant term of $a^k + b^k$. This observation is also applicable to the comparison of several numbers, i.e., $\max(a_1, a_2, ..., a_n)^k$ is a dominant term of $\sum_{i=1}^n a_i^k$ when k is large enough. As a result, Equation (1) can be generalized as followings:

$$\lim_{k\to\infty} \frac{a_j^k}{a_1^k + a_2^k + \cdots + a_n^k} = 1 \iff a_j = \max(a_1, ..., a_n),$$
$$\lim_{k\to\infty} \frac{a_j^k}{a_1^k + a_2^k + \cdots + a_n^k} = 0 \iff a_j \neq \max(a_1, ..., a_n).$$

From these properties, we construct the algorithm `MaxIdx` of which the output indicates the index of the maximum value, as a simple generalization of the comparison algorithm `Comp` in the previous section. The output is a vector of length n and contains a unique non-zero (which is very close to 1) component which indicates the index of the maximum value.

Algorithm 6 `MaxIdx`$(a_1, a_2, ..., a_n; d, d', m, t)$

Input: n distinct numbers $(a_1, a_2, ..., a_n)$ with $a_i \in \left[\frac{1}{2}, \frac{3}{2}\right)$, $d, d', m, t \in \mathbb{N}$
Output: $(b_1, b_2, ..., b_n)$ where b_i is close to 1 if a_i is the largest among a_j's and is close to 0 otherwise (refer Theorem 5)

$inv \leftarrow$ `Inv`$(\sum_{j=1}^n a_j/n; d')$
for $j \leftarrow 1$ **to** $n-1$ **do**
  $b_j \leftarrow a_j/n \cdot inv$ // Initial 1-norm normalization
end for
$b_n \leftarrow 1 - \sum_{k=1}^{n-1} b_j$
for $i \leftarrow 1$ **to** t **do**
  $inv \leftarrow$ `Inv`$(\sum_{j=1}^n b_j^m; d)$
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $b_j \leftarrow b_j^m \cdot inv$
  end for
  $b_n \leftarrow 1 - \sum_{k=1}^{n-1} b_j$
end for
return $(b_1, b_2, ..., b_n)$

Theorem 5. *Let $a_1, a_2, \ldots, a_n \in \left[\frac{1}{2}, \frac{3}{2}\right)$ be n distinct elements, and the ratio of maximum value over the second maximum value be $1 < c < 3$. If $t \geq \frac{1}{\log m}[\log(\alpha + \log n + 1) - \log\log c]$ and $\min(d, d') \geq \log(\alpha + t + 2) + (m-1)\log n - 1$, the* *error of the output of* $\texttt{MaxIdx}(a_1, ..., a_n; d, d', m, t)$ *compared to the true value is (component-wisely) bounded by* $2^{-\alpha}$. *Note that the error is always toward 1/2, i.e., the output value is always in between 1/2 and the true value.*

Proof. Refer to Appendix A. □

Depth and Complexity of `MaxIdx` **Algorithm.** The depth and complexity of `MaxIdx` is $d' + 1 + t(d + \log m + 2)$ and $\Theta(n + d' + t(d + n \log m))$ respectively, as that of `Comp`, and is again optimized when $m = 2$ roughly giving $t = \log((\alpha + \log n)/\log c)$, $d = \log(\alpha + \log((\alpha + \log n)/\log c)) + \log n$. Note that when $\log n \leq \alpha$, depth of `MaxIdx` (asymptotically) does not exceed the depth of `Comp`.

Remark 5. Under the same condition on d, d', m and t with Theorem 5, we can obtain an approximate maximal value among n distinct numbers $a_1, a_2, ..., a_n$ by computing $\sum_{i=1}^{n} b_i a_i$ for $(b_1, b_2, ..., b_n) \leftarrow \texttt{MaxIdx}(a_1, .., a_n; d, d', m, t)$. This idea is basically derived from the equality

$$\lim_{k\to\infty} \frac{a_1^{k+1} + a_2^{k+1} + \cdots + a_n^{k+1}}{a_1^k + a_2^k + \cdots + a_n^k} = \max(a_1, a_2, ..., a_n).$$

Let a_1 be the unique maximum element without loss of generality, then $1 - 2^{-\alpha} \leq b_1 \leq 1$ and $0 \leq b_i \leq 2^{-\alpha}$ for $2 \leq i \leq n$. Then, the error of $\sum_{i=1}^{n} b_i a_i$ compared to the true value $\max(a_1, ..., a_n)$ is bounded by $2^{-\alpha} \cdot \max(a_1, \sum_{i=2}^{n} a_i) \leq \frac{3n}{2} \cdot 2^{-\alpha}$.

6 Asymptotic Optimality of our Methods

In this section, we compare the efficiency of our min/max and comparison algorithms with general polynomial approximation methods, in terms of computational complexity. As a result, we prove the (sub-)optimality of our algorithms in terms of asymptotic computational complexity among polynomial evaluations to obtain approximate min/max and comparison results.

There have been various approaches on dealing with non-polynomial homomorphic operations in many applications of word-wise HE [13, 30, 36], and those works commonly use polynomial approximation. Since our algorithms are based on addition and multiplication, they can be also viewed as polynomial evaluations. However, the main difference is that our polynomial evaluations are represented as recursive algorithms so that the complexity is significantly lower than that of general polynomial evaluation of the same degree.

As described in Theorem 1–5, we estimated an approximation error of our methods (Algorithm 3–6) through the infinite norm, i.e, the maximal error over the domain. Therefore, the *minimax polynomial approximation* [40] which targets the (degree-)optimal polynomial approximation with respect to the error measured by the infinite norm should be compared with our methods. The upper bound of the error of minimax polynomial approximation is given by Jackson's inequality [41] which is a well-known result in approximation theory. The inequality originally covers both algebraic and trigonometric polynomial approximation of general functions, but it can be simplified fitting into our case as following [38]. If a function f defined on $[-1,1]$ satisfies L-Lipschitz condition, i,e, $|f(x_1) - f(x_2)| \leq L \cdot |x_1 - x_2|$ for any $x_1, x_2 \in [-1,1]$, then it holds that

$$||f - p_k||_{\infty,[-1,1]} \leq \frac{L\pi}{2(k+1)} \tag{2}$$

where p_k is the degree-k minimax polynomial of f over the interval $[-1,1]$. Namely, the maximal error between the degree-k minimax polynomial and the original Lipschitz function is $O(1/k)$.

6.1 Min/max from Minimax Approximation

As described in Section 4, the min/max functions are simply described with the absolute function as

$$\min(a,b) = \frac{a+b}{2} - \frac{|a-b|}{2}, \quad \max(a,b) = \frac{a+b}{2} + \frac{|a-b|}{2}.$$

Since the absolute function can also be expressed as $|x| = x - 2 \cdot \min(x,0) = 2 \cdot \max(x,0) - x$, the evaluation of min and max functions are actually equivalent to the evaluation of the absolute function with some additional linear factors. Hence it suffices to consider the minimax polynomial approximation of the absolute function $f(x) = |x|$. Here we assume that a and b are scaled numbers contained in $[0,1)$ as discussed in Section 4.

In the case of $f(x) = |x|$, it is proved that the error upper bound $O(1/k)$ of Jackson's inequality is quite *tight* in terms of asymptotic complexity. To be precise, it holds that

$$\lim_{k \to \infty} k \cdot |||x| - p_k||_{\infty,[-1,1]} = \beta$$

for some constant $\beta \approx 0.28$ [3]. For more details of experimental results on the equation above, we refer the readers to [38, p.19]. As a result, to obtain an approximation error at most $2^{-\alpha}$ for $f(x) = |x|$, it requires the degree of the minimax polynomial to be at least $\Theta(2^\alpha)$. Since general polynomial of degree n requires at least $\sqrt{n}$ multiplications [39], the evaluation of the minimax polynomial requires at least $\Theta(2^{\alpha/2})$ multiplications. In contrasts, our min/max algorithms require only $\Theta(\alpha)$ complexity by Theorem 1. Note that the depths of minimax polynomial evaluation and our min/max algorithms are $\alpha + O(1)$ and $4\alpha - 6$, respectively, both of which are $\Theta(\alpha)$.

Even without asymptotic point of view, our method outperforms the minimax approximation in terms of the required number of multiplications when α is larger than 13. Easy computations show that the required number of multiplications in our iterative method and the minimax approximation method to achieve certain error bound $2^{-\alpha}$ are $3 \cdot (2\alpha - 3) = 6\alpha - 9$ and (approximately) $\sqrt{2\beta} \cdot 2^{\alpha/2}$, respectively (refer Figure 2). Here $2\alpha - 3$ is the minimal number of iterations in `Min`/`Max`, and 3 is the number of multiplications in each iteration.

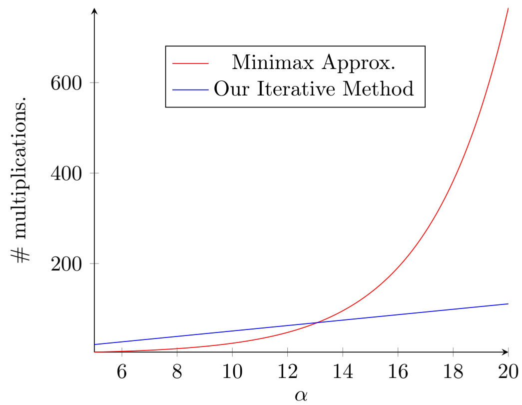

Fig. 2. The actual number of multiplications in minimax approximation and our iterative method for `Max`

6.2 Comparison from Minimax Approximation

Since the comparison equation is expressed as $\mathrm{comp}(a, b) = \chi_{(0,\infty)}(a - b)$, one needs to find a minimax polynomial of the step function $\chi_{(0,\infty)}$. Note that the evaluations of comp and $\chi_{(0,\infty)}$ are equivalent since the step function can also be expressed as $\chi_{(0,\infty)}(x) = \texttt{Comp}(x, 0)$. Let a and b be scaled numbers contained in $\left[\frac{1}{2}, \frac{3}{2}\right)$ as discussed in Section 5. Then the range of $(a - b)$ is $(-1, 1)$, so we can still consider the approximation over the interval $[-1, 1]$.

Contrary to the absolute function $|x|$, the minimax polynomial approximation of $\chi_{(0,\infty)}$ over an interval $[-1, 1]$, which contains 0, *never* gives a nice error bound $||\chi_{(0,\infty)} - p_k||_{\infty,[-1,1]}$ since the step function is discontinuous on $x = 0$. Therefore, it is inevitable to abandon a good polynomial approximation of $\chi_{(0,\infty)}$ over an interval $(-\epsilon, \epsilon)$ for some small $\epsilon > 0$, and our goal should be reduced to find an approximate polynomial p of $\chi_{(0,\infty)}$ which minimizes $||\chi_{(0,\infty)} - p||_{\infty,[-1,-\epsilon]\cup[\epsilon,1]}$. Namely, we should aim to obtain a nice approximate result of comparison on a and b satisfying $|a - b| \geq \epsilon$, not for all $a, b \in \left[\frac{1}{2}, \frac{3}{2}\right)$.

Let us denote by $q_{k,\epsilon}$ the degree-k approximate polynomial which minimizes $||\chi_{(0,\infty)} - p||_{\infty,[-1,-\epsilon]\cup[\epsilon,1]}$. For the step function $\chi_{(0,\infty)}$, there exists a tighter upper bound on the approximation error than Jackson's inequality as following:

$$\lim_{k\to\infty} \sqrt{\frac{k-1}{2}} \cdot \left(\frac{1+\epsilon}{1-\epsilon}\right)^{\frac{k-1}{2}} \cdot ||\chi_{(0,\infty)} - q_{k,\epsilon}||_{\infty,[-1,-\epsilon]\cup[\epsilon,1]} = \frac{1-\epsilon}{2\sqrt{\pi\epsilon}},$$

which was proved by Eremenko and Yuditskii [24]. Assume that k is large enough so that $\sqrt{\frac{k-1}{2}} \cdot \left(\frac{1+\epsilon}{1-\epsilon}\right)^{\frac{k-1}{2}} \cdot ||\chi_{(0,\infty)} - q_{k,\epsilon}||_{\infty,[-1,-\epsilon]\cup[\epsilon,1]}$ is sufficiently close to the limit value. To obtain an approximation error at most $2^{-\alpha}$ for $\chi_{(0,\infty)}$ over

$[-1, -\epsilon] \cup [\epsilon, 1]$, the degree k should be chosen to satisfy

$$\sqrt{\frac{k-1}{2}} \cdot \left(\frac{1+\epsilon}{1-\epsilon}\right)^{\frac{k-1}{2}} \cdot \frac{2\sqrt{\pi\epsilon}}{1-\epsilon} > 2^{\alpha}.$$

Let us consider those two cases: $\epsilon = \omega(1)$ and $\epsilon = 2^{-\alpha}$. In the case of $\epsilon = \omega(1)$, i.e., ϵ is a constant with respect to α, the polynomial degree k should be at least $\Theta(\alpha)$. Therefore, the required depth and computational complexity of q_k evaluation considering the Paterson-Stockmeyer method are $\Theta(\log \alpha)$ and $\Theta(\sqrt{\alpha})$, respectively. In the case of $\epsilon = 2^{-\alpha}$, the polynomial degree k should be at least $\Theta(\alpha \cdot 2^{\alpha})$, needing $\Theta(\alpha)$ depth and $\Theta(\sqrt{\alpha} \cdot 2^{\alpha/2})$ multiplications with the Paterson-Stockmeyer method.

For a fair comparison between the above polynomial approximation and our comparison method, we set $c = \frac{3}{3-2\epsilon}$ where $1 < c < 3$ is a constant defined in Theorem 4 so that the domain $D_1 := \{(a,b) \in \left[\frac{1}{2}, \frac{3}{2}\right)^2 : |a-b| \geq \epsilon\}$ for the above polynomial approximation is completely contained in the domain $D_2 := \{(a,b) \in \left[\frac{1}{2}, \frac{3}{2}\right)^2 : \max(a,b)/\min(a,b) \geq c\}$ for our method. In this setting, the depth and complexity $\Theta(\log(\alpha/\log c) \cdot \log(\alpha + \log(\alpha/\log c)))$ of our `Comp` algorithm becomes $\Theta(\log^2 \alpha)$ if $\epsilon = \omega(1)$ and $\Theta(\alpha \log \alpha)$ if $\epsilon = 2^{-\alpha}$.

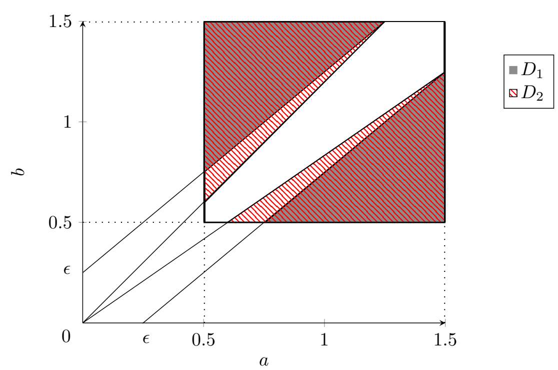

Fig. 3. Regions $D_1 \subset D_2$ for $\epsilon = \frac{3}{2} \cdot \left(1 - \frac{1}{c}\right)$

As a result, the comparison results on the complexity of our methods and minimax polynomial approximation are summarized as following Table 1. As discussed above, we set two cases $\epsilon = \omega(1)$ and $\epsilon = 2^{-\alpha}$ for the comparison operation.

(Sub-)optimality of our Methods. The comparison of computational complexity on our method and minimax approximation method implies the *(sub-)optimality* of our `Min`/`Max` and `Comp` algorithms in terms of asymptotic computational complexity. What Jackson's inequality implies is that *any* polynomial evaluation to obtain an absolute value (hence a min/max result) within $2^{-\alpha}$

		Minimax Approx.	**Our Method**
min/max		$\Theta(2^{\alpha/2})$	$\boldsymbol{\Theta(\alpha)}$
comparison	$\epsilon = \omega(1)$	$\Theta(\sqrt{\alpha})$	$\boldsymbol{\Theta\left(\log^2 \alpha\right)}$
	$\epsilon = 2^{-\alpha}$	$\Theta\left(\sqrt{\alpha} \cdot 2^{\alpha/2}\right)$	$\boldsymbol{\Theta\left(\alpha \log \alpha\right)}$

Table 1. Complexity of our methods and minimax approximation method

error requires $\omega(2^\alpha)$ degree. Regardless of how the polynomial of degree $\omega(2^\alpha)$ is well-structured, the complexity of the polynomial evaluation should be at least the depth $\omega(\alpha)$. In this respective, our `Min`/`Max` algorithm is optimal in asymptotic complexity among the polynomial evaluations to obtain an approximate min/max result. In the same manner, any polynomial evaluation to obtain a comparison result within $2^{-\alpha}$ error requires at least $\omega(\log \alpha)$ and $\omega(\alpha)$ complexity for the cases $\epsilon = \omega(1)$ and $\epsilon = 2^{-\alpha}$, respectively. Therefore, our `Comp` algorithm achieves a kind of sub-optimal asymptotic complexity with an additional factor $\log \alpha$.

Remark 6. In [5], Boura, Gama and Georgieva proposed a different approach for evaluating the absolute function and the step function applying Fourier approximation, and the evaluations can be efficiently done in HEAAN which supports operations of complex numbers. For the fair comparison with our method, we look into the theoretical upper bound of errors in Fourier approximation. By Jackson's inequality for Fourier approximation [33], the upper bound for error of the Fourier approximation of an Lipschitz function f is given as

$$||f - S_k f||_\infty \leq K \cdot \frac{\log k}{k}$$

for some $K > 0$ where $S_k f(x) := \sum_{n=-k}^{k} \hat{f}(n) \cdot e^{inx}$ is the k-th Fourier approximation of f, which can be viewed as a polynomial of $e^{ix} = \cos x + i \cdot \sin x$ and e^{-ix}.

We note that the upper bound of the Fourier approximation error for the absolute function can be reduced to $\Theta(1/k)$. As a result, to make the error upper bound less than $2^{-\alpha}$ following theoretical results, one needs at least $\Theta(2^\alpha)$-th (resp.$\Theta(\alpha \cdot 2^\alpha)$-th) Fourier approximation for the absolute function (resp. step function). Moreover, exponential functions e^{ix} and e^{-ix} should be also approximately evaluated which derives an additional error. Therefore, this Fourier approximation approach still requires exponential computational complexity with respect to α. To sum up, in asymptotic complexity sense, the Fourier approximation approach in [5] requires more computations than our method to obtain the result within a certain level of error.

7 Applications of Comparison Algorithms

In this section, we exploit our comparison algorithms proposed in Section 5 for several applications: Threshold Counting and Top-k Max.

7.1 Threshold Counting

In this subsection, we give a solution to the problem asked at the very beginning of HE. In 1978, Rivest et al. [42] first proposed the concept of HE and listed some problems to be solved with HE:

$\cdots$ *This organization permits the loan company to utilize the storage facilities of the time—sharing service, but generally makes it difficult to utilize the computational facilities without compromising the privacy of the stored data. The loan company, however, wishes to be able to answer such questions as:*

- *What is the size of the average loan outstanding?*
- *How much income from loan payments is expected next month?*
- *How many loans over $5,000 have been granted?*

While the first two problems can be answered with simple arithmetic operations, the last problem requires comparison-like operation intrinsically. We propose a solution to the third problem with our `Comp` algorithm. First, we abstract the problem to "Threshold Counting" problem. The goal of threshold counting problem is to find the number of a_i's larger than b for given $(a_1, a_2, ..., a_n)$ and b. The algorithm is rather simple. We compare a_i's with b and sum up the values $\text{comp}(a_i, b)$. We can use usual packing method of HE to compare several elements in a single operation. We remark that if $a_i = b$ then a_i is counted as $1/2$, not 0 or 1, but in real-world applications this error may be ignored or adjusted by subtracting a very small constant to the threshold b.

Algorithm 7 `Threshold`$(a_1, a_2, .., a_n; b; d, d', t, m)$

Input: n numbers $(a_1, a_2, ..., a_n)$ with $a_i \in [0, 1)$, $b \in [0, 1)$, $d, d', m, t \in \mathbb{N}$
Output: an approximate value of the number of a_i's larger than b

for $i \leftarrow 1$ **to** n **do**
  $c_i \leftarrow$ `Comp`$(a_i, b; d, d', m, t)$ // Can be done in a SIMD manner via HE.
end for
$sum \leftarrow 0$
for $j \leftarrow 1$ **to** k **do**
  $sum \leftarrow sum + c_i$
end for
return sum

7.2 Top-k Max

Applying the `MaxIdx` algorithm in Section 5.2 recursively, we can obtain top-k maximum values which we call top-k max algorithm. For given distinct numbers $a_1, a_2, ..., a_n \in \left[\frac{1}{2}, \frac{3}{2}\right)$ and some positive integers $d, d', m, t \geq 0$, let $(b_1, b_2, ..., b_n) \leftarrow$ `MaxIdx`$(a_1, a_2, ..., a_n; d, d', m, t)$. Then as noted in Remark 5, $\sum_{i=1}^{n} b_i a_i$ is an approximate maximum value of $a_1, ..., a_n$ since $b_i \approx 1$ if and only if a_i is the maximum. Now, to compute the second maximum value, let a_j be the (unique) maximum value, and define $c_i := (1-b_i)a_i$ for $1 \leq i \leq n$. Then $c_i = (1-b_i)a_i \approx a_i$ for all $i \neq j$ and $c_j = (1-b_j)a_j \approx 0$. Since we assume that a_i's are positive numbers, the output of `MaxIdx`$(c_1, c_2, ..., c_n; d, d', m, t)$ indeed indicates the index of the second maximum value. This algorithm can be generalized as following.

Algorithm 8 `Top-k-Max`$(a_1, a_2, .., a_n; d, d', m, t)$

Input: n distinct numbers $(a_1, a_2, ..., a_n)$ with $a_i \in [0, 1)$, $d, d', m, t \in \mathbb{N}$

Output: $(m_1, m_2, ..., m_k)$ where m_i denotes an approximate value of the i^{th} largest number among $\{a_1, a_2, ..., a_n\}$

```
for i ← 1 to n do
    c_i ← a_i
end for
for j ← 1 to k do
    (b_1, b_2, ..., b_n) ← MaxIdx(c_1, c_2, ..., c_n; d, d', m, t)
    m_j ← Σ_{i=1}^{n} b_i c_i
    (c_1, c_2, ..., c_n) ← ((1 − b_1)c_1, (1 − b_2)c_2, ..., (1 − b_n)c_n)
end for
return (m_1, m_2, ..., m_k)
```

Theorem 6. *Let $a_1, a_2, \ldots, a_n \in [1/2, 3/2]$ be n distinct elements, and let the ratio of i-th maximum value over the $(i+1)$-th maximum value $\frac{\max_i}{\max_{i+1}}$ is larger than c_i for $1 \leq i \leq k$. For some $c > 1$ and $\alpha > 0$ satisfying $2^\alpha \cdot (1 - 2^{-\alpha})^{\frac{k(k-1)}{2}} > c^k$, assume that $c_i = c/(1-2^{-\alpha})^{i-1}$ and $\frac{(1-2^{-\alpha})^k \max_{k+1}}{2^{-\alpha} \max_1} > c$. If t, d and d' satisfy the same conditions in Theorem 5, the output $(m_1, ..., m_k)$ of* ***Top-k-Max***$(a_1, ..., a_n; d, d', m, t)$ *satisfies $(1-2^{-\alpha})^j \max_j \leq m_j \leq \max_j$ for $1 \leq j \leq k$.*

Proof. Refer to Appendix A. □

8 Experimental Results

This section illustrates some implementation results of the algorithms we described in the previous sections based on an approximate HE scheme called

HEAAN [14]. Though HEAAN is specialized for iterative (large-depth) computations by supporting approximate calculations, one might wonder if our iterative algorithms proceed well regardless of the errors caused by some operations of HEAAN. We claim that it is indeed so, which can be backed up by the paper [34], which proves that certain classical algorithms for inversion and square root are still stable under floating point rounding as long as we do not experience under/overflow during the computation given the similarity between floating point calculations and HEAAN. In addition, we propose some reasonable parameters and implementation results for each algorithms, and show that they can be carried out with HEAAN very well.

We first show the performance of `Max` algorithm for several setups based on HEAAN. We also implement `Comp` algorithm based on HEAAN and show that it can be exploited to solve the threshold counting problem efficiently. Lastly, we show the performance of our `MaxIdx` algorithm.

8.1 An Approximate Homomorphic Encryption Scheme HEAAN

Cheon et al. [14] proposed a homomorphic encryption scheme HEAAN which supports approximate arithmetic of numbers. That is, an encryption ct of plaintext $m \in R$ by a secret key sk for a ciphertext modulus q will have a decryption structure of the form $\langle \mathsf{ct}, \mathsf{sk} \rangle = m + e \pmod q$ for some small e. By abandoning the exact computation, HEAAN achieved big advantages in ciphertext/plaintext ratio and speed of algorithms. Since many real-world applications require real number computations, HEAAN has a strength in various real-world problems, which usually deal with approximate computation of real numbers, compared to the other HE schemes. Indeed, there have been proposed a lot of works on real-world applications which requires privacy preservation such as machine learning [15, 35, 36] and cyber physical system [11] based on HEAAN. The simple description of HEAAN is as following.

- $\underline{\texttt{KeyGen}(L, 1^\lambda)}$.
  - Given the level parameter L and the security parameter λ, output the ring dimension N and the largest ciphertext modulus Q which are power-of-two.
  - Set the small distributions $\chi_{\texttt{key}}, \chi_{\texttt{err}}, \chi_{\texttt{enc}}$ over R.
  - Sample a secret $s \leftarrow \chi_{\texttt{key}}$, a random $a \leftarrow R_Q$ and an error $e \leftarrow \chi_{\texttt{err}}$. Set the secret key as $\mathsf{sk} \leftarrow (1, s)$ and the public key as $\mathsf{pk} \leftarrow (b, a) \in R_Q^2$ where $b \leftarrow -as + e \pmod Q$.
- $\underline{\texttt{KSGen}_{\mathsf{sk}}(s')}$. For $s' \in R$, sample a random $a' \leftarrow R_{Q^2}$ and an error $e' \leftarrow \chi_{\texttt{err}}$. Output the switching key as $\mathsf{swk} \leftarrow (b', a') \in R_{Q^2}^2$ where $b' \leftarrow -a's + e' + Qs'$ $\pmod{Q^2}$.
  - Set the evaluation key as $\mathsf{evk} \leftarrow \texttt{KSGen}_{\mathsf{sk}}(s^2)$.
- $\underline{\texttt{Enc}_{\mathsf{pk}}(m)}$. For $m \in R$, sample $v \leftarrow \chi_{\texttt{enc}}$ and $e_0, e_1 \leftarrow \chi_{\texttt{err}}$. Output $v \cdot \mathsf{pk} + (m + e_0, e_1) \pmod Q$.
- $\underline{\texttt{Dec}_{\mathsf{sk}}(\mathsf{ct})}$. For $\mathsf{ct} = (c_0, c_1) \in R_{2^\ell}^2$, output $c_0 + c_1 \cdot s \pmod{2^\ell}$.

- $\underline{\texttt{Add}(\mathsf{ct}_1, \mathsf{ct}_2)}$. For $\mathsf{ct}_1, \mathsf{ct}_2 \in R^2_{2^\ell}$, output $\mathsf{ct}_{\texttt{add}} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2 \pmod{2^\ell}$.
- $\underline{\texttt{Mult}_{\mathsf{evk}}(\mathsf{ct}_1, \mathsf{ct}_2)}$. For $\mathsf{ct}_1 = (b_1, a_1), \mathsf{ct}_2 = (b_2, a_2) \in R^2_{2^\ell}$, let $(d_0, d_1, d_2) = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2) \pmod{2^\ell}$. Output $\mathsf{ct}_{\texttt{mult}} \leftarrow (d_0, d_1) + \lfloor Q^{-1} \cdot d_2 \cdot \mathsf{evk} \rceil \pmod{2^\ell}$.
- $\underline{\texttt{ReScale}(\mathsf{ct}; p)}$. For a ciphertext $\mathsf{ct} \in R^2_{2^\ell}$ and a scaling bit p, output $\mathsf{ct}' \leftarrow \lfloor 2^{-p} \cdot \mathsf{ct} \rceil \pmod{2^{\ell - p}}$.

For any power-of-two integer $k \leq N/2$, HEAAN supports the packing method which pack k complex numbers each into different slots in a single ciphertext via some mapping $\phi : \mathbb{C}^k \to R$ derived from the complex canonical embedding. The packing method enables us to perform parallel computations over encryption, yielding a better amortized running time. In addition, HEAAN provides the rotation operation on plaintext slots, i.e., it enables us to securely obtain an encryption of the shifted plaintext vector $(w_r, \ldots, w_{k-1}, w_0, \ldots, w_{r-1})$ from an encryption of $(w_0, \ldots, w_{k-1})$. Using $\log k$ rotations and additions, we can obtain the following operation.

- $\underline{\texttt{RotateSum}(\mathsf{ct})}$. For a ciphertext ct, an encryption of $(w_0, \ldots, w_{k-1})$ via packing method, output ct', an encryption of $\left(\sum_{i=0}^{k-1} w_i, \cdots, \sum_{i=0}^{k-1} w_i \right)$ via packing method.

Refer [14] for the technical details and noise analysis.

8.2 Implementations of various non-polynomial Operations

All experiments on our method were implemented in C++ on Linux with Intel Xeon CPU E5-2620 v4 at 2.10GHz processor with multi-threading (8 threads) turned on for speed acceleration. Note that we checked the security level of HEAAN parameters we used in our implementation through a security estimator constructed by Albrecht [1, 2]. More precisely, we set the level parameter L to be the minimum required considering the depth of algorithms (without bootstrapping), the dimension N to be the minimum ensuring the security parameter $\lambda \geq 128$, and the scaling bit p to be 40 or around.

In the rest of the section, we present both the actual running time and the amortized running time considering the plaintext batching technique of HEAAN. We note that the amortized running time is important as much as the actual running time in various applications which require a number of same operations. For example, even a basic task such as threshold counting can be performed simultaneously with only a single homomorphic comparison. More seriously, k-nearest neighbor algorithm for classification and k-means algorithm for clustering requires substantial numbers of min/max and comparison, which can also be parallelized in the same manner with the above threshold counting.

Max of two integers. We first show the performance of Algorithm 3 (`Max`) which outputs an approximate value of the maximum value given two large integers. Since HEAAN supports at most $N/2$ operations simultaneously in a SIMD manner, the actual experiment is to compute $\max(a_i, b_i)$ for $1 \leq i \leq N/2$. In Table 2, minimal iteration d required for Max to achieve each bit precision α is provided. The number of iterations are chosen empirically considering worst case, instead of considering the theoretical expectation of Theorem 1. We can see that the empirical requirement of d is a bit smaller than theoretical expectation. For example, when $\alpha = 10$, then $d = 14$ suffices while theoretical requirement is $d \geq 17$. The amortized running time is measured by dividing total running time by the number of plaintext slots.

We remark that our performance only depends on the precision α, not on the input bitsize ℓ. It provides us much flexibility when we need only approximate maximum value. For example, our implementation shows that we can obtain an approximate maximum value of any two 32-bit integers with an error up to 2^{22} in 1.14 milliseconds (with amortized time sense). Also, the performance of our algorithm does not linearly depend on the number of threads. When we use only a single thread, the performance is worsened by 2–3 times.

The performance of our Max algorithm is comparable, in amortized running time sense, to the previous results of which input numbers are encrypted bit-wisely. For example, the max algorithm from [18] based on a bit-wise HE, which expressed the max function by a number of logical gates via weighted finite automata, takes about 1 millisecond to compute the maximum of two 8-bit integers.

Comparison of two integers. We also implemented our Comp algorithm for various setups on the number of precision bits α and the lower bound c of the ratio $\frac{\max(a,b)}{\min(a,b)}$. As in the previous subsection, we put integers in full $N/2$ plaintext slots of HEAAN ciphertext so that the Comp algorithm supports $N/2$ simultaneous comparison operations. For each setup, we empirically chose optimal parameters $m = 4$, d, d' and t. Refer to Algorithm 5 for definitions of the parameters.

Algorithm	# precision bits	# iterations	Running time	
	α	d	Total (s)	Amortized (ms)
Max	8	11	48[a)]	0.73
	10	14	75[b)]	1.14
	12	17	127[c)]	1.94
	16	23	237[d)]	3.62

Table 2. HEAAN implementation of Max algorithm for several precision bits. HEAAN parameters $(\log N, Q, \lambda)$ were chosen as [a)] $(17, 930, 192.2)$, [b)] $(17, 1170, 147.0)$, [c)] $(17, 1410, 131.5)$, and [d)] $(17, 1890, 107.7)$.

In Table 3, `Comp` (exact) denotes the comparison experiment considering the worst case, i.e, comparing *any* of two α-bit integers scaled into $\left[\frac{1}{2}, \frac{3}{2}\right)$ with α-bit precision, which corresponds to $c = \left(\frac{1}{2} + \frac{2^\alpha - 1}{2^\alpha}\right) / \left(\frac{1}{2} + \frac{2^\alpha - 2}{2^\alpha}\right)$. For the cases $c = 1.01$ and $c = 1.05$, we took 32-bit integers satisfying the ratio lower bound as input.

As same as `Max`, our empirically chosen parameters d, d' and t and are smaller than the theoretical expectation from Theorem 4. For example, for 7-bit precision of `Comp` (exact), it was expected to be $d, d' > 5.9$ and $t > 5.5$ from the theorem, but we found that a bit smaller parameters were sufficient.

The result shows that when we do not need exact comparison, i.e., when we are given that two inputs has enough difference, we can get more efficient parameters. For example, the same iteration $(d', d, t) = (5, 5, 5)$ guarantees 14, or 24 bit precision when c is 1.01 or 1.05, respectively, while it only guarantees 7-bit precision if we need exact comparison. When c is 1.05, only $(d', d, t) = (5, 4, 4)$ iteration suffices for 8-bit precision. Note that each result shows high performance of `Comp` showing less than 5 milliseconds of amortized running time considering 2^{16} number of plaintext slots in one ciphertext.

In [20], Crawford et al. reported some recent implementation results on the comparison operation based on HElib, where the input integers were bit-wisely encrypted. We referred their comparison experiment on 8-bit integers which uses the 15709-th cyclotomic polynomial, and it took about a second with 8 threads. Considering ciphertexts over 15709-th cyclotomic polynomial have 682 plaintext slots, the amortized running time is around 1.5 milliseconds. This shows that the performance of our word-wise comparison is comparable, in amortized time, to that of a bit-wise comparison which has been regarded to be one of the most natural approaches to compare numbers.

Max Index for several numbers. We present an experimental evaluation of the `MaxIdx` algorithm. For experiment, we compute max index of 16 encrypted 7-bit integers. We assume that the maximum integer has non-zero most significant

Algorithm	# precision bits α	# iterations (d', d, t)	Running time Total (s)	Running time Amortized (ms)
`Comp` (exact)	7	(5, 5, 5)	225 [a)]	3.43
	8	(5, 5, 6)	310 [b)]	4.72
`Comp` ($c = 1.01$)	14	(5, 5, 5)	230 [a)]	3.50
`Comp` ($c = 1.05$)	24	(5, 5, 5)	259 [a)]	3.94

Table 3. Implementation of `Comp` for several precision bits. HEAAN parameters $(\log N, Q, \lambda)$ were chosen as [a)] $(17, 1600, 121, 6)$, and [b)] $(17, 1870, 108.9)$.

bit, while other integers have most and 2nd-most significant bits zero. This condition corresponds to the lower bound $c = \left(\frac{1}{2} + \frac{2^6}{2^7}\right) / \left(\frac{1}{2} + \frac{2^5-1}{2^7}\right) = \frac{128}{95}$.

The parameter chosen by considering worst-case is a little better than the theoretical estimation (Theorem 5) that $t > 2$ and $d > 14$. Total running time is about 311 seconds, and we can run $2^{16}/2^4 = 2^{12}$ number of Max index algorithms with one ciphertext resulting amortized running time to be only about 75 milliseconds.

As an example, assume that we are given many ciphertexts each encrypting vectors, and we want to choose a vector whose first component is maximum among that of the other vectors in encrypted state. This problem can not be resolved by max algorithm, since the output of it does not contain any information about its index. On the other hand, we can extract the desired ciphertext simply multiplying output of `MaxIdx` algorithm to each ciphertext and summing them up.

Threshold Counting. For `Threshold` algorithm, we assume that the threshold b is encrypted. This is because in some scenarios the threshold could be private information or trade secret. If b is not secret, the algorithm shows a better performance since a constant multiplication is faster than a ciphertext multiplication in HE.

For a power-of-two integer $k \leq N/2$, HEAAN supports a packing method which packs k real numbers in a single ciphertext, enabling us to perform parallel computations over encryption. As mentioned in the Section 7.1, we utilize this packing method to solve threshold counting with exactly one `Comp` query and then use `RotateSum` to sum up the results of the `Comp`.

For experimental results, we assume that given 2^5 number of 7-bit integers, we want to calculate the number of elements bigger than an encrypted 7-bit threshold. Then, we can take the lower bound $c = \left(\frac{1}{2} + \frac{2^6-1}{2^6}\right) / \left(\frac{1}{2} + \frac{2^6-2}{2^6}\right) = \frac{191}{190}$, and it suffices to bound error size to be smaller than $2^{-\alpha} = 2^{-6}$ for each result of comparison, since we evaluate the addition of 2^5 comparison results, whose true value is an integer. In Table 3, we can see that it takes about 278 seconds to get the number of elements bigger than the given threshold. Since we

Algorithm	# precision bits α	# iterations (d', d, t)	Running time Total (s)	Running time Amortized (ms)
`MaxIdx`	7	(3, 11, 3)	311	75.9
`Threshold`	6	(3, 5, 5)	278	135

Table 4. Implementation of `MaxIdx` and `Threshold` for 2^4 and 2^5 encrypted 7-bit integers, respectively. HEAAN parameters $(\log N, Q, \lambda)$ were chosen as $(17, 1800, 111.3)$.

can pack at most 2^{16} numbers in one ciphertext, we can manage 2^{11} threshold counting problems for 2^5 numbers with only a single ciphertext, resulting about 135 milliseconds of amortized running time. If we allow some errors in the final result, or we are given that the gap between threshold and other numbers are large, we can get more efficient result than above.

References

1. M. R. Albrecht. A Sage Module for estimating the concrete security of Learning with Errors instances., 2017. https://bitbucket.org/malb/lwe-estimator.
2. M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
3. S. Bernstein. Sur la meilleure approximation de| x| par des polynomes de degrés donnés. *Acta Mathematica*, 37(1):1–57, 1914.
4. J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding*, pages 45–64. Springer, 2013.
5. C. Boura, N. Gama, and M. Georgieva. Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning. Cryptology ePrint Archive, Report 2018/758, 2018. https://eprint.iacr.org/2018/758.
6. F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.
7. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
8. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proc. of ITCS*, pages 309–325. ACM, 2012.
9. K. Brown. Probability of intersecting intervals. https://www.mathpages.com/home/kmath580/kmath580.htm.
10. A. Chatterjee and I. SenGupta. Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective? *IEEE Transactions on Services Computing*, 2017.
11. J. H. Cheon, K. Han, S. M. Hong, H. J. Kim, J. Kim, S. Kim, H. Seo, H. Shim, and Y. Song. Toward a secure drone system: Flying with real-time homomorphic authenticated encryption. *IEEE Access*, 6:24325–24339, 2018.
12. J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.
13. J. H. Cheon, J. Jeong, J. Lee, and K. Lee. Privacy-preserving computations of predictive medical models with minimax approximation and non-adjacent form. In *International Conference on Financial Cryptography and Data Security*, pages 53–74. Springer, 2017.
14. J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.
15. J. H. Cheon, D. Kim, Y. Kim, and Y. Song. Ensemble method for privacy-preserving logistic regression based on homomorphic encryption. *IEEE Access*, 2018.

16. J. H. Cheon, M. Kim, and M. Kim. Search-and-compute on encrypted data. In *International Conference on Financial Cryptography and Data Security*, pages 142–159. Springer, 2015.
17. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–33. Springer, 2016.
18. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 377–408. Springer, 2017.
19. A. Costache and N. P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Cryptographers' Track at the RSA Conference*, pages 325–340. Springer, 2016.
20. J. L. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. Doing real work with fhe: The case of logistic regression. 2018.
21. M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In H. Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
22. L. Ducas and D. Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015*, pages 617–640. Springer, 2015.
23. N. Emmadi, P. Gauravaram, H. Narumanchi, and H. Syed. Updates on sorting of fully homomorphic encrypted data. In *Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on*, pages 19–24. IEEE, 2015.
24. A. Eremenko and P. Yuditskii. Uniform approximation of sgn x by polynomials and entire functions. *Journal d'Analyse Mathématique*, 101(1):313–324, 2007.
25. J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.
26. C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. http://crypto.stanford.edu/craig.
27. C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography–PKC 2012*, pages 1–16. Springer, 2012.
28. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.
29. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.
30. R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.
31. R. E. Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964.
32. S. Halevi and V. Shoup. Bootstrapping for helib. In *Advances in Cryptology–EUROCRYPT 2015*, pages 641–670. Springer, 2015.
33. D. Jackson. *The theory of approximation*, volume 11. American Mathematical Soc., 1930.

34. M. Joldes, O. Marty, J.-M. Muller, and V. Popescu. Arithmetic algorithms for extended precision using floating-point expansions. *IEEE Trans. Computers*, 65(4):1197–1210.
35. A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(4):83, Oct 2018.
36. M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Med Inform*, 6(2):e19, Apr 2018.
37. O. Kocabas and T. Soyata. Utilizing homomorphic encryption to implement secure and private medical cloud computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 540–547. IEEE, 2015.
38. R. Pachón and L. N. Trefethen. Barycentric-remez algorithms for best polynomial approximation in the chebfun system. *BIT Numerical Mathematics*, 49(4):721, 2009.
39. M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.
40. G. M. Phillips. *Best Approximation*, pages 49–118. Springer New York, New York, NY, 2003.
41. M. J. D. Powell. *Approximation theory and methods*. Cambridge university press, 1981.
42. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.
43. M. Togan, L. Morogan, and C. Plesca. Comparison-based applications for fully homomorphic encrypted data. *Proceedings of the Romanian Academy-series A: Mathematics, Physics, Technical Sciences, Information Science*, 16:329, 2015.
44. M. V. Wilkes. *The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press, 1951.

A Proofs

Proof of Theorem 3. By Theorem 2, the error of $\texttt{Max}(\cdot,\cdot;d)$ algorithm from the true value is bounded by $2^{(-\alpha-\log\log n)} = 2^{-\alpha}/\log n$. Note from the proof of Lemma 2 that the output of the square root algorithm $\texttt{Sqrt}(x;d)$ always smaller than the true value $\sqrt{x}$, so that the same holds for the max algorithm $\texttt{Max}(\cdot,\cdot;d)$. This means that $a_{i,1} = \texttt{Max}(a_{2i-1,0}, a_{2i,0}; d)$ can be written $a_{i,1} = \max(a_{2i-1,0}, a_{2i,0}) - \epsilon_i$ for $1 \le i \le n/2$ with $0 \le \epsilon_i \le 2^{-\alpha}/\log n$. Now we have

$$\begin{aligned}\max(a_{2i-1,1}, a_{2i,1}) &= \max(\max(a_{4i-3,0}, a_{4i-2,0}) - \epsilon_{2i-1}, \max(a_{4i-1,0}, a_{4i,0}) - \epsilon_{2i})\\ &\ge \max(a_{4i-3,0}, a_{4i-2,0}, a_{4i-1,0}, a_{4i,0}) - \max(\epsilon_{2i-1}, \epsilon_{2i})\\ &\ge \max(a_{4i-3,0}, a_{4i-2,0}, a_{4i-1,0}, a_{4i,0}) - 2^{-\alpha}/\log n,\end{aligned}$$

which implies that the error of $a_{i,2} = \texttt{Max}(a_{2i-1,1}, a_{2i,1}; d)$ from $\max(a_{2i-1,1}, a_{2i,1})$ is bounded by $2 \cdot 2^{-\alpha}/\log n$ for $1 \le i \le n/4$. We can repeat the above procedure to get the conclusion that the error of $a_{1,\log n}$ from $\max(a_1, ..a_n)$ is bounded by $\log n \cdot 2^{-\alpha}/\log n = 2^{-\alpha}$.

For the case of min algorithm we note that the approximate values are larger than the true values and we can apply a similar approach to the above with reversed inequalities. □

Proof of Theorem 5. Note that `MaxIdx` is a natural generalization of `Comp`. Without loss of generality, we assume that a_1 is the unique maximum element, and we only consider the error between the output b_1 of `MaxIdx` and the real value 1. At Step 1–4, $(a_i)_{i=1}^n$ is scaled to $(b_i)_{i=1}^n$ whose sum is 1. Moreover, every input of `Inv` is bounded by $\frac{n}{2^m}$ since $\sum_{k=1}^n b_j$ is always set to be 1 before the `Inv` algorithm. Note that each b_j from the iterations is nothing but $a_j^{m^t}/\sum_{i=1}^n a_i^{m^t}$ with t being increased by one as the iteration go. The error of `MaxIdx` algorithm is also composed of three parts as theorem 4; an error from the convergence of $\lim_{m\to\infty} a_1^m/\sum_{i=1}^n a_i^m = 1$, and an error from the approximation of $1/(\sum_{i=1}^n b_i^m)$ by our `Inv` algorithm and an error coming from Step 1–4.

Now, the error analysis is almost the same as the proof of Theorem 4 with minor differences in the values of errors. The first part of the error is bounded by $n\cdot(1/c)^{m^t}$ since $1 - \frac{a_1^N}{\sum_{i=1}^n b_i^N} = 1 - \frac{1}{1+\sum_{i=2}^n (b_i/a_1)^N} \le n/c^N$. The second part of the error (from the `Inv` algorithm) is bounded by $(1-n^{-(m-1)})^{2^{d+1}}$ since $n^{-(m-1)}$ is the lower bound of the denominators $\sum_{i=1}^n b_i^m$ by Cauchy-Schwartz inequality. As a result, we can conclude that the conditions $t \ge \frac{1}{\log m}[\log(\alpha + \log n + 1) - \log\log c]$ and $d, d' \ge \log(\alpha + t + 1) + (m-1)\log n - 1$ suffice to make the total error of `MaxIdx` less than $2^{-\alpha}$ by a similar argument as in Theorem 4. □

Proof of Theorem 6. Without loss of generality, let a_i be the i^{th} maximum value $\max_i$ for $1 \le i \le n$.

For $1 \le i < k$, since $(1-2^{-\alpha})^i a_{i+1} > (1-2^{-\alpha})^k a_{k+1}$, we first obtain $\frac{(1-2^{-\alpha})^i a_{i+1}}{2^{-\alpha} a_1} > c$. For $j = 1$, the statement holds directly by Theorem 5. After obtaining m_1, the algorithm takes $(\epsilon_1 a_1, (1-\epsilon_2)a_2, ..., (1-\epsilon_n)a_n)$ as an input of $\texttt{MaxIdx}(\cdots; d, d', m, t)$, where $0 \le \epsilon_i \le 2^{-\alpha}$. Since the following inequalities

$$(1-\epsilon_2)a_2 \ge (1-2^{-\alpha}) \cdot \frac{2^{-\alpha}}{1-2^{-\alpha}} \cdot ca_1 \ge c \cdot \epsilon_1 a_1, \text{ and}$$

$$(1-\epsilon_2)a_2 > (1-\epsilon_2)c_2 a_3 \ge ca_3 \ge c \cdot (1-\epsilon_j)a_j \text{ for } 3 \le j \le n$$

hold, the output m_2 satisfies $(1-2^{-\alpha})^2 a_2 \le m_2 \le a_2$ by Theorem 5.

Inductively, assume that we have obtained $m_1, m_2, ..., m_{j-1}$ satisfying the statement condition. After obtaining an approximate value m_{j-1} of the $(j-1)^{\text{th}}$ maximum value a_{j-1}, the next input of $\texttt{MaxIdx}$ algorithm is $(\delta_1 a_1, \delta_2 a_2, ..., \delta_n a_n)$ where $0 \le \delta_i \le 2^{-\alpha}$ for $i < j$ and $(1-2^{-\alpha})^j \le \delta_i \le 1$ for otherwise. From the following inequalities

$$\delta_j a_j \ge (1-2^{-\alpha})^j \cdot \frac{2^{-\alpha}}{(1-2^{-\alpha})^j} \cdot ca_1 \ge c \cdot \delta_i a_i \text{ for } 1 \le i < j, \text{ and}$$

$$\delta_j a_j > \delta_j c_j a_{j+1} \ge ca_{j+1} \ge c \cdot \delta_i a_i \text{ for } i > j,$$

by Theorem 5 the output m_{j+1} satisfies $(1-2^{-\alpha})\delta_j a_j \le m_j \le \delta_j a_j$ so that the statement also holds for j. Therefore, the theorem is proved by induction. □