Hindawi
Security and Communication Networks
Volume 2020, Article ID 1250295, 18 pages
https://doi.org/10.1155/2020/1250295

Research Article

An Efficient Encrypted Floating-Point Representation Using HEAAN and TFHE

Subin Moon[1] **and Younho Lee**[2]

[1]*Department of Software Analysis and Design, SeoulTech, Seoul 18011, Republic of Korea*
[2]*ITM Division, Department of Industrial Engineering, SeoulTech, Seoul 18011, Republic of Korea*

Correspondence should be addressed to Younho Lee; younholee@seoultech.ac.kr

Received 30 October 2019; Revised 17 January 2020; Accepted 3 February 2020; Published 2 March 2020

Guest Editor: Veljko Milutinovic

As a method of privacy-preserving data analysis (PPDA), a fully homomorphic encryption (FHE) has been in the spotlight recently. Unfortunately, because many data analysis methods assume that the type of data is of real type, the FHE-based PPDA methods could not support the enough level of accuracy due to the nature of FHE that fixed-point real-number representation is supported easily. In this paper, we propose a new method to represent encrypted floating-point real numbers on top of FHE. The proposed method is designed to have analogous range and accuracy to 32-bit floating-point number in IEEE 754 representation. We propose a method to perform arithmetic operations and size comparison operations. The proposed method is designed using two different FHEs, HEAAN and TFHE. As a result, HEAAN is proven to be very efficient for arithmetic operations and TFHE is efficient in size comparison. This study is expected to contribute to practical use of FHE-based PPDA.

1. Introduction

Fully homomorphic encryption (FHE) is a technology that supports computation on ciphertexts without decrypting them. Using these properties, we can process and analyze sensitive medical and financial information without exposing them if we use FHE. Therefore, FHE is expected to be an indispensable core technology for the prevalence of artificial intelligence and data mining, which is the core technology of the 4th industrial revolution [1, 2].

Most existing FHEs manage ciphertext on a bit-by-bit basis. Some methods have a number of slots in a ciphertext and support SIMD operations. Even in this case, only one bit of plaintext can be inserted into a single slot. This is because FHE schemes are defined on particular mathematical structures. If multiple bits occupy a slot, it is unclear as to whether or not all efficient computation algorithms can be implemented using the operations provided naturally on the mathematical structures.

On the other hand, in the case where only a 1-bit plaintext is inserted in the slot, any efficient algorithm can be implemented with the bitwise-AND and bitwise-XOR operations. Thus, if these operations are implemented with the operations on the mathematical structures, and the ciphertexts are enabled to work with them indefinite number of times, we can say that FHE schemes can implement any efficient algorithm. There are numerous FHE schemes that follow this argument [3–12], including HELib and TFHE.

The purpose of this study is to realize representation of encrypted floating-point real number and their operations. Various machine learning algorithms use real numbers to manage their input, output, and intermediate values, as is the case with the recently introduced Deep Neural Network. There have been various attempts to realize these algorithms using FHE and to perform machine learning and classification even when receiving encrypted input values. However, due to the bitwise nature of the plaintexts in most FHE schemes as described above, the existing attempts have treated real numbers with fixed-point representation in most cases. Since using fixed-point numbers in this field causes the degradation of the accuracy and range of the expressed values, it also affects the algorithm performance. For this reason, floating-point representation is preferred in numerical applications [13].

Unfortunately, there have been only few attempts to represent floating-point numbers with FHE. A representative study is the HEAAN method [14]. In HEAAN, floating point numbers can be thought of because they have the advantage of keeping the same significant digits regardless of the size of the computed result. However, HEAAN has a few drawbacks. First, it does not provide comparison operations on given numbers. This is a big problem when implementing various applications using HEAAN. Second, HEAAN provides the division operation only within a limited range of realizations. That is, if "a" is the real number contained in a cipher text, then $|1-a| \leq 1/2$ should be held. Therefore, in order to use HEAAN for applications, it is necessary to scaling the values.

Apart from HEAAN, TFHE was proposed in 2017, which supports very fast bootstrapping [7]. Sadly, because it does not provide floating-point operations, a new method is needed to support the encrypted floating-point representation over TFHE and the operations that can work on top of the representation.

Therefore, we propose an encrypted floating-point representation on which it is provided efficient floating-point operations over the encrypted values by the above FHE schemes. In the proposed method, real numbers are expressed with sign, exponent, and mantissa similarly to the conventional IEEE 754 expression. However, it is designed to achieve the efficiency of the entire operations by utilizing the characteristics of each FHE method. Supporting the efficient floating-point operations over the proposed encrypted representation is not an easy task because we have to consider all possible values that ciphertexts of the proposed representation must have when we design the computation circuits to implement the operations.

In this paper, we compare the performance of two representations each of which is implemented on top of HEAAN and FHE, respectively. As a result of performance comparison, HEAAN has a computation speed of several tens to hundreds of times faster than that of TFHE-based implementation. In the comparison operation, TFHE has tens to hundreds of times higher performance than HEAAN. In the case of the equality checking operation, we could observe that TFHE case is about 5 times faster than HEAAN. The proposed method is expected to be used for the implementation of the algorithms in privacy-preserving machine learning. In particular, it tells you what type of FHE schemes to use depending on the type of operation being used. Based on this, we hope that it could be widely used in encrypted data processing and analysis applications.

This paper is organized as follows. Section 2 describes the preliminary to help understanding this study. Section 3 introduces the research motivation and related work. In Section 4, the proposed method is given, which is followed by the performance evaluation of implementation results in Section 5. Section 6 concludes the paper.

2. Preliminary

2.1. IEEE 754 Floating-Point Representation. The proposed representation is a variant of IEEE 754 standard floating-point representation to support the arithmetic and relational operations efficiently with the underlying FHE schemes. Therefore, to understand the proposed representation easily, we first briefly overview it. In IEEE 754, two types of floating-point representation are defined, single-precision and double-precision, depending on the number of bits assigned for a representation. In this work, we only focus on the single-precision representation as this is only applied for the proposed one.

Figure 1 shows the IEEE 754 single-precision representation. It is composed of three parts: 1-bit sign, 8-bit exponent, and 23-bit mantissa. The sign bits represent if the value is positive (0) or negative (1). The exponent bits represent the exponent which is in the range between −127 and 127. The "fraction" bits contain the significant bits of the value. In normal mode, Figure 1 represents a real number $(-1)^{(\text{sign})} * 1.(\text{FRACTION})\,(2) * 2^{(\text{EXPONENT}(2)-127)}$, where binary values represent unsigned positive integers.

The arithmetic and relational operations between the numbers expressed in this way can be carried out as follows. In the case of addition and subtraction, the exponents of two operands are made equal. The mantissas of them are adjusted following the changed exponents, respectively. After that, the operation is performed with the adjusted mantissas. In the case of multiplication and division, operations are performed on exponents and mantissas of two operands, respectively. After processing, adjustment is performed on the resultant exponent and mantissa in order to match the expression rules. Please refer to [15] for details.

2.2. Fully Homomorphic Encryption (FHE). FHE is a special cryptosystem that can process encrypted data without decrypting them. This is useful for data delegation or privacy-preserving data processing. Figure 2 shows the difference between FHE and a conventional encryption scheme when addition of two encrypted numbers is performed. With the ciphertexts by FHE, addition of their hidden plaintext values is possible without decryption.

An FHE scheme is defined with the following algorithms:

ParamGen $(1^{\lambda}, N, L) \longrightarrow param$: this generates a parameter for key generation. It takes the security parameter λ, the number of slots in a ciphertext N, and the possible multiplicative depth of a fresh ciphertext without bootstrapping L.

KeyGen $(param) \longrightarrow sk, pk, evk$: this is a key generation algorithm, which takes the parameter from the parameter generation algorithm and outputs sk, a secret key for decryption, pk, a public key for encryption, and evk, a set of the evaluation keys with which some computation over the ciphertexts can be done if the ciphertexts are generated with the matched pk.

Enc $(\vec{\mathrm{m}}, pk) \longrightarrow \mathrm{C}$: an encryption algorithm which takes a vector of plaintext message $\vec{\mathrm{m}}$ of dimension N and a public key and outputs a ciphertext that contains $\vec{\mathrm{m}}$.

Dec $(\mathrm{C}, sk) \longrightarrow \vec{\mathrm{m}}$ (or $\perp$): it takes a ciphertext c and a secret key sk, and outputs a plaintext $\vec{\mathrm{m}}$ if C is the result

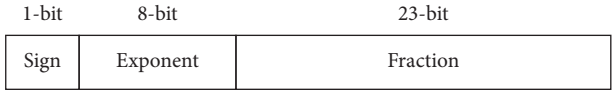

Figure 1: IEEE 754 single-precision floating-point number—bit representation.

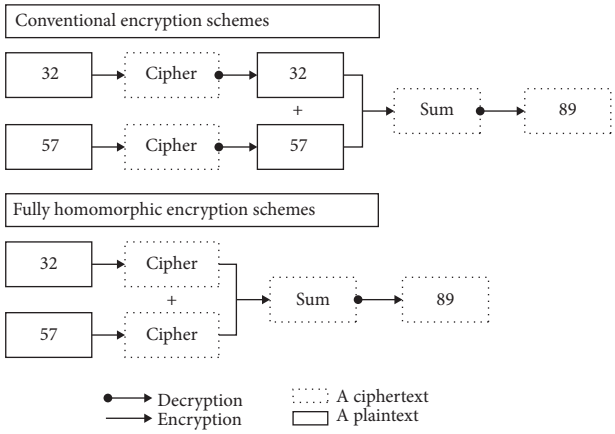

Figure 2: Adding two plaintexts in ciphertexts: comparison between conventional encryption and homomorphic encryption.

of Enc($\vec{m}$, *pk*) and *pk* is matched to *sk*. Otherwise, it outputs $\perp$.

In addition to the above algorithms, FHE provides computational algorithms between ciphertexts. However, they provide different functions for each FHE scheme. Therefore, they will be described when each FHE scheme is introduced. We introduce two recent FHE schemes, which are used to implement the proposed (encrypted) floating-point representation.

2.2.1. HEAAN. HEAAN's ciphertext can contain multiple plaintext bits in one slot, unlike the previous FHE methods [14, 16]. Let the corresponding bit length be the word size. HEAAN also supports various operations between the encrypted values of a word size in the ciphertext slots. Unfortunately, they are supported in an approximate fashion. That is, if an operation is performed between encrypted word-sized values, the bit-precision of the result is similar to that of an operand. In the case of multiplication, when the bit length of operands is n, the result's bit length should be $2n-1$ to represent the result correctly. However, in HEAAN, only the most significant n bits of the result are correct. At the expense of this bit accuracy, HEAAN supports very fast arithmetic operations on word-sized encrypted data. This fast computation speed allows HEAAN to be used in many fields such as machine learning and machine control [17, 18]. In addition, unlike TFHE, HEAAN has the advantage of providing a complete SIMD function. That is, there is an advantage that the value can be shifted between slots. The major disadvantage of HEAAN compared to TFHE is that it requires bootstrapping. That is, when performing multiplication operations more than a certain number of times consecutively, that is, when a multiplication circuit having a certain depth or more is performed, bootstrapping should be performed to do further multiplication while preserving the correct computation result. Otherwise, the ciphertext no longer has the correct plaintext.

HEAAN_Mult (C_0, C_1, *evk*) $\longrightarrow C_{mult}$: this algorithm multiplies the encrypted values stored in each slot of the same position in two ciphers C_0, C_1 with each other and stores the result in the same slot position of the resultant ciphertext C_{mult}.

HEAAN_Add/Sub (C_0, C_1, *evk*) $\longrightarrow C_{add}$ (C_{sub}): this algorithm performs slotwise addition (subtraction) with two ciphertexts C_0 and C_1 and outputs C_{add} (C_{sub}) which has the result of the computation.

HEAAN_Bootstrapping (C_0, *evk*) $\longrightarrow c_{new}$: this algorithm takes a ciphertext C_0 and returns a new ciphertext C_{new} with the same plaintext. We can execute a multiplicative circuit of a certain number of depths with Cnew, where the depth is defined by the parameters.

HEEAN_LeftShift (C_0, k, *evk*) $\longrightarrow C_{shifted}$: this algorithm generates a new ciphertext $C_{shifted}$ by moving the plaintexts in each slot of cipher C_0 to the left by k slots. It is a circular shift. Thus, we can shift the values in C_0 to the right by k slots by making the values shifted left by N-k slots, where N is the number of total slots.

HEAAN supports the following algorithms in addition to the common FHE algorithms described above.

2.2.2. TFHE. TFHE [9, 19] was created in 2017 using the GSW [20] technique. Each slot in the TFHE ciphertext contains a 1-bit plaintext. It supports bitwise AND and bitwise XOR operations on the encrypted plaintext bit in each slot. Based on those operations, it is possible to implement any arbitrary efficient algorithm which can work with encrypted inputs. In addition to the basic algorithms mentioned above, TFHE supports the following algorithms:

TFHE_AND (C_0, C_1, *evk*) $\longrightarrow c_{and}$: a bitwise AND operation is performed on the values of the slot at the same position in the two ciphertexts c_0 and c_1, and the result is stored in the slot of the same position in the resultant cipher C_{and}.

TFHE_XOR (C_0, C_1, *evk*) $\longrightarrow C_{xor}$: it executes the slotwise XOR operations on bit values of the same slot position in c_0 and c_1. The result is stored in the same slot position of the resultant ciphertext C_{xor}.

One advantage of THE is that it supports very fast bootstrapping. However, it needs bootstrapping at the end of every operation. Thus, we suppose that the bootstrapping is contained in the above bitwise operations.

2.2.3. An Overview for the Ciphertext Structures and Operations. To help the readers understand both FHE methods, we discuss the format of the ciphertext used and more details on them. Let us talk about HEAAN first. Figure 3 shows the structure of the HEAAN ciphertext and

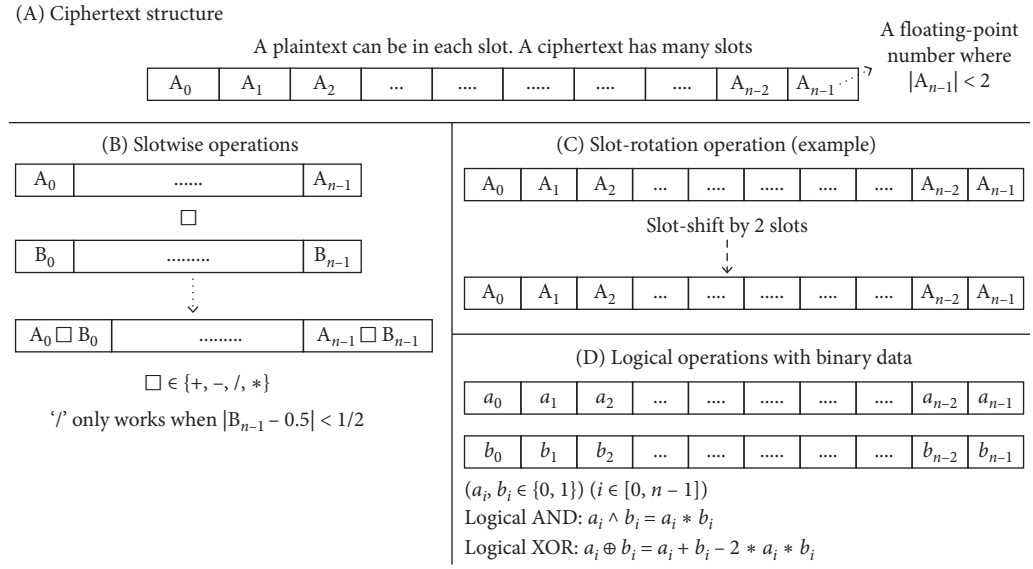

Figure 3: Ciphertext structure and operations in HEAAN.

the operations supported on it. In HEAAN, as shown in the figure below, a limited number of real numbers can be stored in one ciphertext. Arithmetic operations can be performed between those in the same slot as shown in Figure 3(B). In addition to this, as shown in Figure 3(C), it is possible to move the embedded plaintexts between slots. Finally, if there are two ciphertexts where only 0 or 1 are encrypted in each slot, then logical AND/XOR operations between the encrypted values are possible in the same slot position of both ciphertexts. These logical operations can be implemented by arithmetic subtraction, addition, and multiplication using the equations described in Figure 3(D). Unfortunately, there is a disadvantage that a slot can only store a fixed-point real-number plaintext. Also, a certain operation can work only when the stored plaintext value is within a limited range. TFHE has similar ciphertext structure as HEAAN, but each slot can contain only 0 or 1 (1 bit value). Also, slot-shift operation is not supported.

3. Related Work and System Model

3.1. Related Work. There have been some attempts to represent encrypted floating-point real numbers before this study. Arita and Nakasato [21] proposed the FHE4FX method that can perform addition and multiplication between encrypted floating-point numbers without decryption based on the FV FHE method [22]. It generates an encrypted floating-point representation by encrypting each sign, exponent, and mantissa part of a real number in separate ciphertexts. For the multiplication and addition operations between encrypted values, it first determines the type of FHE operations to be performed on the sign, exponent, and mantissa parts based on the comparison result of values of each part of the ciphertext. Unfortunately, there is some limitation in this approach that it is necessary to use the decryption key for computation and it does not support operations other than addition and multiplication.

Zhu et al. [23] proposed a method to represent floating-point numbers as Double List Tree format and provide the encryption function for them with Paillier cryptosystem [24]. Their work utilizes hashtag (#) to represent the location of the decimal points of encrypted numbers. However, in this method, the digits of the real number can be inferred through the position of the hashtag used for performing the operation of each digit. Furthermore, their arithmetic operations use the size comparison operation. However, they did not show how to implement the size comparison operation.

In the method of Jaschke and Armknecht [25], when an operation is performed on encrypted rational numbers, they are multiplied by an exponent of 2 to move the decimal place values to an integer position. After this, rounding is performed to remove the decimal point value, and the main operation is performed with only the integer parts of two operands. After that, the desired result is obtained by adjusting the digits. This process has the disadvantage that the precision of the operation is reduced because the calculation is performed after some bit values of input rational numbers are removed.

There are other methods such as those of Seiko and Nakasato [26] and Dowlin et al. [27]. However, they require a very large computation for floating-point operations. Also, as we mentioned in the previous section, HEAAN [14, 16] has some limitation on representing encrypted floating-point numbers. Also, there has been no approach to represent encrypted floating-point numbers on top of TFHE [9, 19].

In conclusion, there is no way to represent encrypted floating-point real numbers with similar level of accuracy and efficiency to the floating-point representation used in common computing environment, even if there are so many methods using existing FHE methods. In this paper, we aim to solve this problem and make it possible to represent encrypted floating-point numbers and to perform

computation with them efficiently. In other words, this research is the first study to propose and implement a size comparison operation and arithmetic operations over encrypted floating-point real-number representation efficiently using FHE.

3.2. System Model. The system model assumed in this study is as follows. The client translates the plaintext representation of the floating-point real numbers into the encrypted one using the proposed method and delivers it to the central server. The central server then performs an algorithm that can be performed on the encrypted input and returns the results back to the client. The client can decode the result to get the result. In this model, the client and server shares their public keys with each other.

The security model we assume is the general Honest-But-Curious (HBC) model. The central server performs predefined protocols, but it tries to obtain useful information of the clients while running the protocol. The clients are those who own their data, including both input and result of the protocol, and want it to be protected from the central server. Please check Figure 4 above for the detailed description of the system and security model.

4. Proposed Approach

In this section, we propose a new scheme to represent encrypted floating-point numbers using two methods, HEAAN and TFHE, respectively. Our proposal includes the methods to support the arithmetic and the relation operations over the encrypted numbers represented by the proposed methods. Before explaining each method, we first introduce the tools used to implement the proposed methods and then proceed to the description of the proposed one. We omit showing the evaluation key (*evk*) when we present the proposed methods because any calculation involving the proposed encrypted representation needs *evk*.

4.1. Building Blocks

4.1.1. Kogge–Stone Adder (KSA). The authors in [28] implemented the Kogge–Stone Adder [29], which performs addition on two input numbers, using FHE to have it work on encrypted input numbers. In [28], it is assumed that each slot of a ciphertext stores only one bit value, and all the slots are used to represent one number. Therefore, we can use [28] to implement the Kogge–Stone Adder over encrypted inputs using either TFHE or HEAAN, if we put only one-bit plaintext in each slot of ciphertext. Using this, it is possible to implement the adder using a circuit with $\log n$ multiplication depth when inputs are of n bits. Figure 5(A) shows the encrypted KSA (E-KSA) in [12]. Two ciphertexts are output of the adder where the first has the sum where all n slots are filled with outputs, whereas only one slot is used in the other ciphertext to represent the carry of the addition.

4.1.2. Comparator (COMP) Module. It compares the size of two ciphertexts and returns a ciphertext containing the result. For example, if there are two ciphers C_1 and C_2, and each represents a binary number using the bit values of all of its slots, this module returns a ciphertext of 1 when $C_1 > C_2$ and returns 0 otherwise (if the resultant ciphertext has many slots, it outputs the ciphertext containing 1 in every slot; this also holds in the case where the resultant ciphertext contains 0). It can be implemented as shown in Figure 5(B) using E-KSA module. The reason why this works is illustrated in Figure 5(C). For example, if C_1 contains an encryption of $1100_{(2)}$ (4 bits) and C_2 contains $1010_{(2)}$, we can see that a carry is generated as a result of adding the 1's complements of C_1 and C_2. Based on this, we can confirm that $C_1 > C_2$.

When using such a COMP circuit, if two ciphertexts C_1 and C_2 are input, we can make a circuit that outputs a ciphertext C_B containing the larger plaintext and another ciphertext C_S which contains the smaller plaintext among the plaintexts encrypted in C_1 and C_2, respectively. If we assume the result of executing the COMP module with input C_1 and C_2 as COMP (C_1, C_2), C_B can be derived by calculating $C_1 * \mathrm{COMP}(C_1, C_2) + C_2 * (\mathbf{1_c} - \mathrm{COMP}\ (C_1, C_2))$. Similarly, $C_S \longleftarrow C_1 * (\mathbf{1_c} - \mathrm{COMP}\ (C_1, C_2)) + C_2 * \mathrm{COMP}\ (C_1, C_2)$. $\mathbf{1_c}$ means an encoding of 1 where calculation is possible with other ciphertexts. If the ciphertext has many slots, $\mathbf{1_c}$ is an encoding of 1 in every slot. One caveat is that you cannot know whether C_B and C_S come from either C_1 or C_2, respectively, because the above formulas' calculation results depend on the results of COMP() circuit whose output is encrypted; thus, it cannot be known without decryption. We can define the above two circuits as MinMax() as follows:

$$C_b, C_s \longleftarrow \mathrm{MinMax}(C_1, C_2). \tag{1}$$

4.2. HEAAN-Based Method. The HEAAN-based method uses three ciphertexts to represent a single real number. The first ciphertext represents the sign of the real number, the second ciphertext stores the exponent value, and the last ciphertext stores the mantissa. The sign and exponent are expressed as bit values like IEEE 754 standard. However, when expressing a mantissa, the corresponding number is not stored as a multibit value. Instead, it is encrypted to a HEAAN ciphertext directly. One significant difference of this approach from IEEE 754 is that the value of the mantissa is assumed to be in the range [0, 1). This is because the HEAAN operation works better when the hidden plaintext in the ciphertext in [0, 1) than [1, 2).

Figure 6 shows the procedure on how a single floating-point number is encrypted using three ciphertexts. cresult [] is an array containing ciphertexts. It is assumed that the value assigned to the ciphertext cresult [] is encrypted and stored in the corresponding ciphertext. Figure 7 shows an example of converting a real number into the proposed encrypted representation.

4.2.1. Addition/Subtraction. The HEAAN-based addition method is implemented as shown in Figure 8. In this method, two operands are classified to a smaller number and

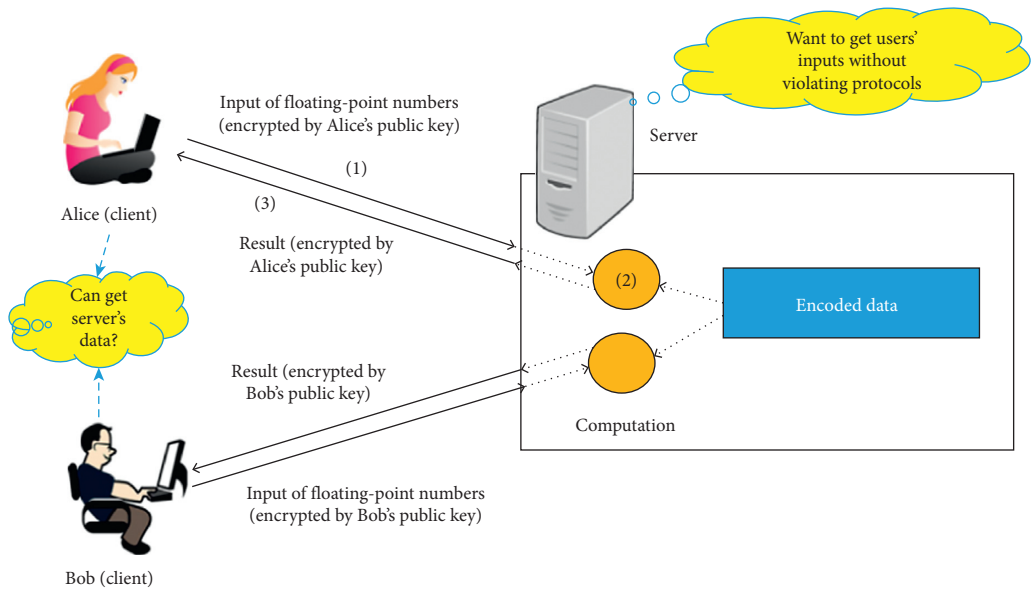

Figure 4: System and security model.

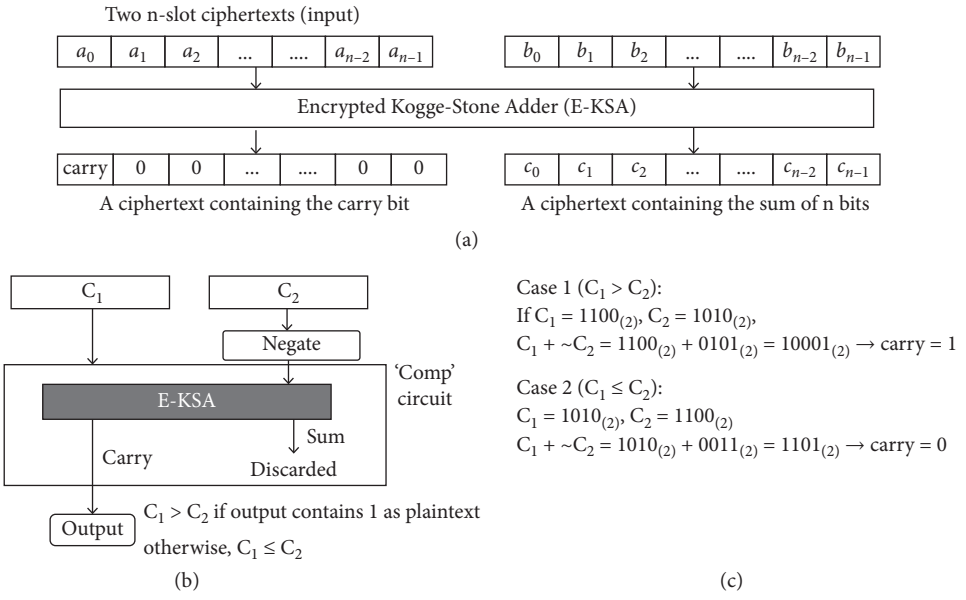

Figure 5: Building blocks: encrypted Kogge–Stone Adder and comparator circuit.

the other bigger number first. Then, the exponent value of the smaller number is increased to be matched to that of the bigger number. The increase of the value of the smaller number, which results from the increasing the exponent, is cancelled by dividing the mantissa part. After that, two mantissa parts are added to complete the addition.

The details of addition operation is as follows. We suppose two operands to be added as $C_1 = (C_1[\text{SIGN}], C_1[\text{EXPO}], C_1[\text{FRAC}])$, $C_2 = (C_2[\text{SIGN}], C_2[\text{EXPO}], C_2[\text{FRAC}])$. The function ReArrange (), which returns a larger value in c_B and a small value in c_S among the input C_1 and C_2, can be implemented as shown in Figure 9. C_1 and C_2 are compared based on their exponent values.

We suppose the result of Rearrange (C_1, C_2, EXPO) is C_S and C_B. We run the following addition code with C_S and C_B in Figure 10. The output of the code is $C_{out.}$ We discuss how the following code works. Because $C_S[\text{EXPO}] < C_B[\text{EXPO}]$ and $C_S[\text{FRAC}]$, $C_B[\text{EXPO}]$ in [0, 1), the sign of the addition result is $C_B[\text{SIGN}]$. Also, to perform calculation, the code increases $C_S[\text{EXPO}]$ to make it the same as $C_B[\text{EXPO}]$. To keep the original value of C_S after increasing $C_S[\text{EXPO}]$, the code decreases $C_S[\text{FRAC}]$ accordingly. After that, it returns the result of $C_B[\text{FRAC}] + C_S[\text{FRAC}]$ when both C_B and C_S have the same sign or $C_B[\text{FRAC}] - C_S[\text{FRAC}]$ if the signs are different. To keep the result's fraction is in [0, 1), the exponent is increased by 1 and the fraction is divided by 2.

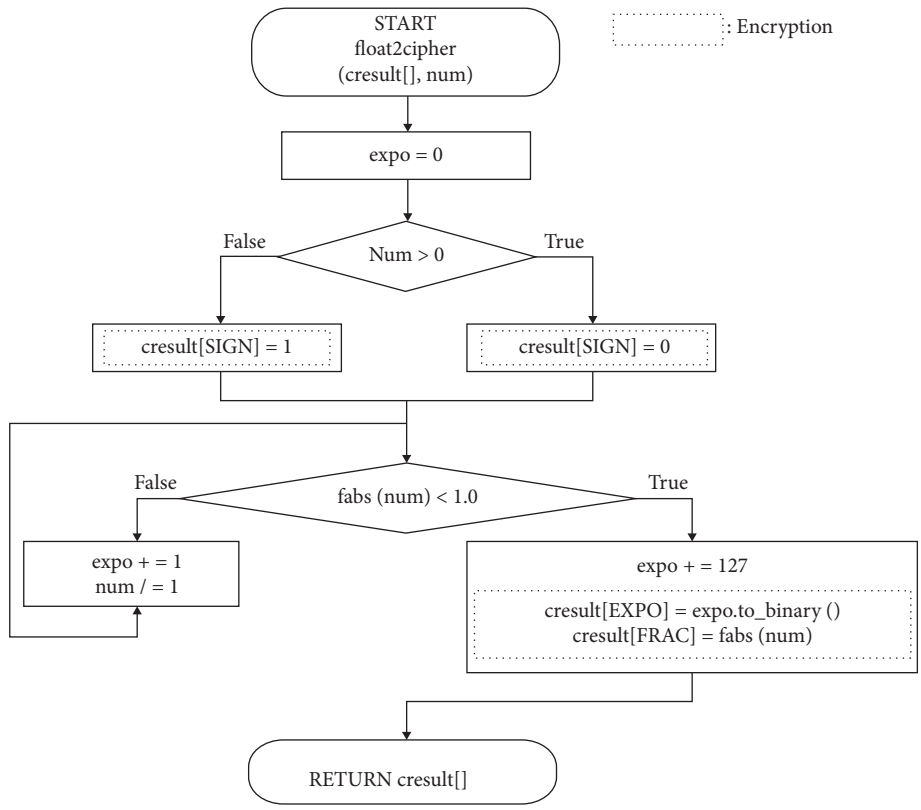

Figure 6: Procedure of generating a set of ciphertexts containing a single floating-point number.

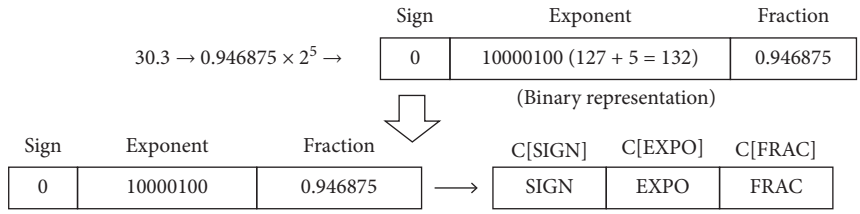

Figure 7: An example of representing 30.3 with the HEAAN-based proposed scheme. The exponent is converted to binary representation. Then, it is encrypted to C[expo]. Each encrypted bit is stored in each slot. On the other hand, the fraction is stored as it is in a HEAAN ciphertext.

Because the maximum binary string that C_B[EXPO] and C_S[EXPO] can have is $11111111_{(2)}$, C_{eqcc} is an encryption of 1 only once in the loop. In this case, $C_B[FRAC] + C_S[FRAC] * (1 - 2 * C_{sign})$ is given to C_{out}[-FRAC], where C_{sign} is the result of checking if C_B[SIGN] and C_S[SIGN] are the same: if so, it has the encryption of 0, and 1 otherwise. In Step 3, EQCC (C_B[EXPO], C_S[EXPO]) executes HEAAN_XNOR(C_B[EXPO], C_S[EXPO]) from inside to perform XNOR operation between the bit values in the same slot of C_B[EXPO] and C_S[EXPO], respectively. After that, we multiply all the bit values in each slot of XNOR result to create C_{comp}. C_{comp} becomes 1 if C_B[EXPO] and C_S[EXPO] have exactly the same bit value in all of their slots; otherwise, it becomes 0.

To perform subtraction, the sign of the number to be subtracted is inverted using HEAAN_XNOR, and then the addition operation is performed instead.

4.2.2. Multiplication. Multiplication is easier to implement than addition. First, the sign of the result value is equal to the value obtained by XORing the sign of two input numbers, and the result of the exponent part can be obtained by adding the exponents of the two inputs. E-KSA can be used to calculate it. Also, the mantissa can be obtained by multiplying the mantissas of two input numbers. In this case, the result of multiplying mantissa is still in [0, 1). Thus, we have nothing to do further. Figure 11 shows the process of multiplying two encrypted floating-point real numbers C_1 and C_2 in the proposed method.

4.2.3. Comparison. In the HEAAN-based approach, $COMP_{HEAAN}()$ function is used as a tool to perform the size comparison operation of the encrypted floating-point values. $COMP_{HEAAN}$ () takes two HEAAN ciphertexts and

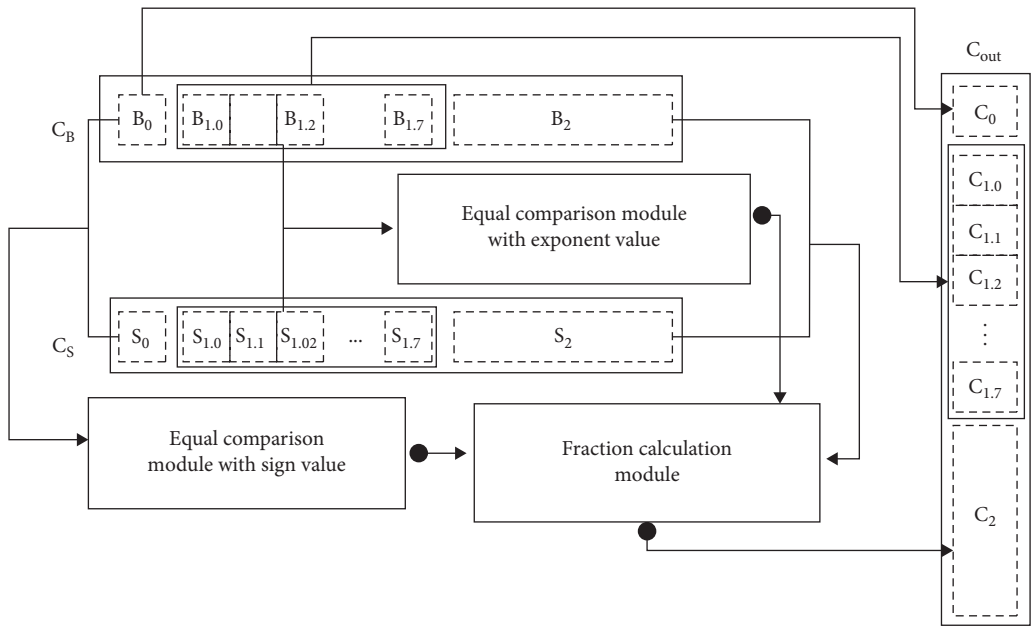

Figure 8: Overview of addition operation in the HEAAN-based method.

Input: $C_1 = (C_1[SIGN], C_1[EXPO], c_1[FRAC])$, $C_2 = (C_2[SIGN], C_2[EXPO], C_2[FRAC])$,
opt (either EXPO or FRAC)
1. if (opt == EXPO) $c_{COMP} \leftarrow COMP\ (C_1[EXPO], C_2[EXPO])$
   Else $C_{COMP} \leftarrow COMP_{HEEAN}\ (C_1[FRAC], C_2[FRAC])$ // $COMP_{HEEAN}$ is explained at 4.2.3.
2. $C_B[EXPO] = C_1[EXPO] * c_{COMP} + (1 - c_{COMP}) * C_2[EXPO]$
3. $C_B[EXPO] = C_2[EXPO] * c_{COMP} + (1 - c_{COMP}) * C_1[EXPO]$
4. $C_B[SIGN] = C_1[SIGN] * c_{COMP} + (1 - c_{COMP}) * C_2[SIGN]$
5. $C_S[SIGN] = C_2[SIGN] * c_{COMP} + (1 - c_{COMP}) * C_1[SIGN]$
6. $C_S[FRAC] = C_1[FRAC] * c_{COMP} + (1 - c_{COMP}) * C_2[FRAC]$
7. $C_S[FRAC] = C_2[FRAC] * c_{COMP} + (1 - c_{COMP}) * C_1[FRAC]$
8. Output C_B, $= (C_B[SIGN], C_B[EXPO], C_B[FRAC])$, $C_S = (C_S[SIGN], C_S[EXPO], C_S[FRAC])$, c_{COMP}

Figure 9: Rearrange() function.

Input: $C_B[SIGN]$, $C_S[SIGN]$, $C_B[EXPO]$, $C_S[EXPO]$, $C_B[FRAC]$, $C_S[FRAC]$,
1. $i \leftarrow 1$, $C_{out}[FRAC] \leftarrow 0$
2. $C_{sign} \leftarrow HEAAN\_XOR\ (C_S[SIGN], C_B[SIGN])$
3. $C_{eqcc} \leftarrow EQCC\ (C_B[EXPO], C_S[EXPO])$ // EQCC: Equality comparison circuit.
4. $C_{out}[FRAC] \leftarrow C_{out}[FRAC] + C_{eqcc} * (C_S[FRAC] + C_S[FRAC] * (1 - 2C_{sign}))$
5. $C_S[FRAC] \leftarrow C_S[FRAC]/2$, $C_S[EXPO] \leftarrow C_S[EXPO] + 1$
6. $i \leftarrow i + 1$
7. if $i < 256$ goto step 3
8. Output $C_{out} = (C_B[SIGN], C_S[EXPO], C_{out}[FRAC])$

Figure 10: Addition method in HEAAN-based approach.

the computation key *evk* of the ciphertext and performs the following operation [30]:

$$\mathrm{COMP}_{\mathrm{HEAAN}}\left(C_{1\,(\mathrm{HEAAN})}, C_{2\,(\mathrm{HEAAN})}, \mathrm{ev}k\right) \longrightarrow 1 \text{ if } \cdot \left(C_{1\,(\mathrm{HEAAN})} > C_{2\,(\mathrm{HEAAN})}\right) \text{else } 0. \tag{2}$$

Using the above circuit, we can compare the size of two encrypted floating-point numbers $C_1 = (C_1[SIGN], C_1[EXPO], C_1[FRAC])$ and $C_2 = (C_2[SIGN], C_2[EXPO], C_2[FRAC])$ using the code written in Figure 12. If the code outputs an encryption of 1, it says C_1's absolute value is greater than C_2's and it returns an encryption of 0 otherwise. Step 2 in Figure 12 is described in Figure 13 in detail.

A circuit for determining if two encrypted representations are equal can be implemented similar to addition/subtraction circuit: it can be easily obtained by calculating

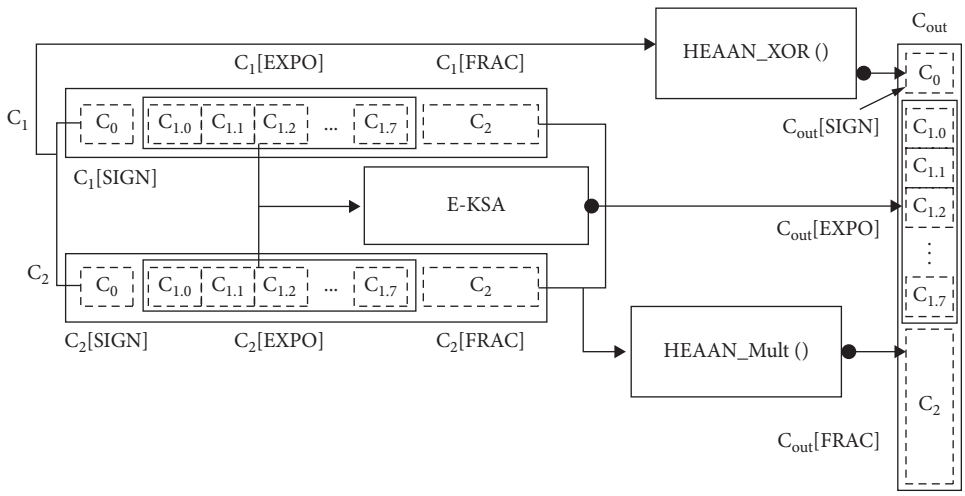

Figure 11: Multiplication with HEAAN-based representation.

Input: C_1[EXPO], C_2[EXPO], C_1[FRAC], C_2[FRAC],

1. C_B, C_S, $c_{comp} \leftarrow$ Rearrange (C_1, C_2, FRAC)
2. Adjust C_S to make C_S[FRAC] < C_B[FRAC] < 2 ∗ C_S[FRAC]
3. $c_{eqcc} \leftarrow$ EQCC (C_B[EXPO], C_S[EXPO]) // EQCC: Equality comparison circuit.
4. $c_{comp2} \leftarrow$ COMP(C_B[EXPO], C_S[EXPO])
5. $c_{comp\_HEAAN} \leftarrow COMP_{HEAAN}$ (C_B[FRAC], C_S[FRAC])
6. $c_{out} \leftarrow c_{eqcc} * (c_{comp\_HEAAN}) + (1 - c_{eqcc}) * c_{comp2}$
7. Output $(c_{out} * C_{comp}) + (1 - c_{out}) * (1 - c_{comp})$

Figure 12: Absolute value comparison circuit with HEAAN-based representation.

Input: C_B[EXPO], C_S[EXPO], C_B[FRAC], C_S[FRAC],

2.1. $i \leftarrow 1$, $c_{out} \leftarrow 0$, $c_{prev} \leftarrow 0$, $C_1 \leftarrow C_S$[FRAC], C_S[FRAC] $\leftarrow 0$, $C_{expo} \leftarrow C_S$[EXPO], C_S[EXPO] $\leftarrow 0$

2.2. $C_2 \leftarrow C_1 * 2$

2.3. $c_{curr} \leftarrow COMP_{HEAAN}$ (C_B[FRAC], C_2)

2.4. C_S[FRAC] $\leftarrow$ Cs[FRAC] + $(1 - c_{prev}) * c_{curr} * C_S$[FRAC],

2.5. C_S[EXPO] $\leftarrow C_S$[EXPO] + $(1 - c_{prev}) * c_{curr} * C_{expo}$

2.5. $i \leftarrow i + 1$, $C_1 \leftarrow C_2$, $C_{expo} \leftarrow C_{expo} - 1$

2.6. if $i < 256$ goto step 3

2.7. Output C_S[FRAC], C_S[EXPO]

Figure 13: Details in Step 2 of the absolute value comparison circuit.

whether the FRAC part and the SIGN part of both floating-point numbers to be compared are the same using $COMP_{HEAAN}$() after equalizing their EXPO parts.

4.2.4. Division. The division operation is similar to the multiplication operation except that the exponent part of the divisor is subtracted from the exponent of the dividend, and the inverse of the mantissa of the divisor is calculated using the inverse function and then it is multiplied by the mantissa of the dividend. In this case, if we do nothing further, the mantissa of the resultant value may be greater than 1. To deal with this, before performing the multiplication operations in the fraction part, the mantissa of the dividend is corrected so that it is smaller than the mantissa of the divisor. However, it is greater if the mantissa of the dividend is doubled. The process can be described as follows.

Step 1 in Figure 14 can be specified in detail as follows in Figure 15.

4.3. TFHE-Based Method. The TFHE-based expression follows the standard IEEE 754 exactly. For example, to represent 30.3 with the TFHE-based representation, we first translate this as $1.11100100110011001100110 \times 2^{3}{}_{(2)}$. Then we add 127 to the exponent to represent the exponent bits. The fraction part stores all the fraction bits except the first "1" at the left of the decimal point. In this approach, sign, exponents, and mantissa are encrypted by separate ciphertexts. The number of ciphertexts in C[SIGN] is 1, C[EXPO] is 8,

Input: C_1[SIGN], C_2[SIGN], C_1[EXPO], C_2[EXPO], C_1[FRAC], C_2[FRAC] (C_1: Dividend, C_2: Divisor)
1. Adjust C_1 to make C_1[FRAC] < C_2[FRAC] < 2 ∗ C_1[FRAC]
2. $C_{expo} \leftarrow$ HEAAN_XOR (C_2[EXPO], 1_c) + 1 // 2′s complement of C_2[EXPO]
3. Cout[EXPO] ← E-KSA (C_1[EXPO], C_{expo}) // C_{out}[EXPO] = C_1[EXPO] − C_2[EXPO]
4. $C_{inv} \leftarrow 1/C_2$[FRAC]
5. C_{out}[FRAC] ← C_1[FRAC] ∗ C_{inv}
6. C_{out}[SIGN] ← HEAAN_XOR (C_1[SIGN], C_2[SIGN])
7. Output C_{out} = (C_{out}[SIGN], C_{out}[EXPO], C_{out}[SIGN])

Figure 14: Division operation in HEAAN-based representation.

Input: C_1[EXPO], C_2[EXPO], C_1[FRAC], C_2[FRAC],
1.1. $i \leftarrow 1$, $c_{out} \leftarrow 0$, $c_{prev} \leftarrow 0$, $C_A \leftarrow C_1$[FRAC], C_1[FRAC] ← 0, $C_{expo} \leftarrow C_1$[EXPO], C_1[EXPO] ← 0
1.2. $C_{2A} \leftarrow C_A * 2$
1.3. $c_{curr} \leftarrow COMP_{HEAAN}$ (C_2[FRAC], C_2)
1.4. C_1[FRAC] ← C_1[FRAC] + (1 − c_{prev}) ∗ c_{curr} ∗ C_1[FRAC],
1.5. C_1[EXPO] ← C_1[EXPO] + (1 − c_{prev}) ∗ c_{curr} ∗ C_{expo}
1.5. $i \leftarrow i + 1$, $C_A \leftarrow C_{2A}$, $C_{expo} \leftarrow C_{expo} - 1$
1.6. if i < 256 goto step 3
1.7. Output C_1[FRAC], C_1[EXPO]

Figure 15: Details of Step 1 of the division operation.

and C[FRAC] is 23 when the encrypted floating-point number is represented as C = (C [SIGN], C [EXPO], C [FRAC]). Also, for ease of expression, TFHE_XOR () operation is written as ⊕, and TFHE_AND () operation is indicated by. Figure 16 explains the procedure of making a TFHE-based encrypted floating-point number representation with an example of 30.3.

To explain the methods clearly, we are going to use the following notations. An array of ciphertexts C[] = $c_0||c_1||\ldots||c_{n-1}$ can be described as C[0] = c_0, C[1] = c_1,. . . , $C_{[n-1]} = c_{n-1}$, where c_i ($i \in [0, n-1]$) refers to an encryption of a single bit value. If we describe C[] = $(c_{n-1}\, c_{n-2} \ldots c_2 c_1 c_0)_{(2)}$, as a binary integer representation, then C[0] = c_0, C[1] = c_1,. . . , $C_{[n-1]} = c_{n-1}$, too. If there are two array of ciphertexs A[] and C[], where both are represented as binary integers and they are added using E-KSA circuit, then the result C_R[] is $2^{n-1}(c_{n-1} + a_{n-1}) + 2^{n-2}(c_{n-2} + a_{n-2}) + \ldots + 2^1(c_1 + a_1) + c_{0+}a_0$. Also, you have to keep in mind that $c \oplus (c_0||c_1||\ldots||c_{n-1}) = (c \oplus c_0||c \oplus c_1\ldots||c \oplus c_{n-1})$ and $(c_{a0}||c_{a1}||\ldots||c_{an}) \oplus (c_{b0}||c_{b1}||\ldots||c_{bn}) = (c_{a0} \oplus c_{b0}||c_{a1} \oplus c_{b1}||\ldots||c_{an} \oplus c_{bn})$. The concatenation of two ciphertext arrays A[] and B[] can be described as A[]||B[], and it generates a new array of ciphertexts. Finally, C[] = C[] >> 1 means C[i + 1] = C[i] for all i = 0, . . ., len(C[])−2 and C[0] = 0.

4.3.1. Addition/Subtraction. The addition algorithm is constructed in a manner as similar as the HEAAN-based method. However, in this TFHE based method, arrays of ciphertexts are used to represent the components of the representation and each ciphertext has one bit value, which complicates the process as shown in Figure 17. We use E-KSA as building block. The input of E-KSA, however, is arrays of ciphertexts instead of ciphertexts of multiple slots. Our implementation of E-KSA considered that. Figure 18 shows an implementation of Equality Comparison Circuit (TFHE_EQCC()) for TFHE representation, which is also a

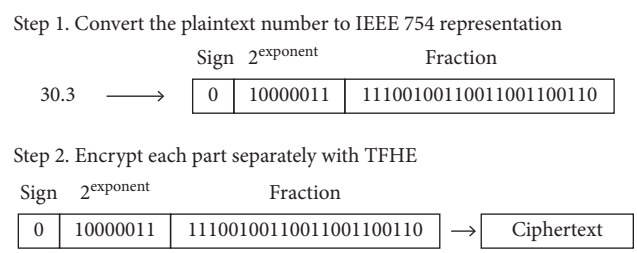

Figure 16: Creating a ciphertext of a floating-point number with TFHE.

building block for addition operation. We also introduce the negate function (Negate()) which flips the bits in the input set of ciphertexts in Figure 19.

Figure 20 shows the procedure of addition operation with TFHE-based representation. For subtraction, the addition operation can be used after modifying the sign of the value to be subtracted.

4.3.2. Multiplication. The multiplication operation proceeds as follows. First, a basic operation is performed in which an XOR operation is performed on sign portions and the addition is performed on exponent portions. Then, the result of the multiplication operation of the fraction parts is applied. When adding the exponent parts, 127 is subtracted from the addition result to get the correct result. This is because the exponent part is basically expressed by adding 127 so that a negative number can be expressed with unsigned bit-strings. The overall multiplication process is shown in Figure 21 below.

Among the three operations in Figure 21, (1) and (2) are easy to implement, so we focus on (3). To perform (3), the numbers actually represented by C_1[FRAC] and C_2[FRAC] must be restored. The bits in C_1[FRAC] and C_2[FRAC] mean the values to the right of the decimal point in the two numbers' fraction parts, and the leading "1" to the left of the

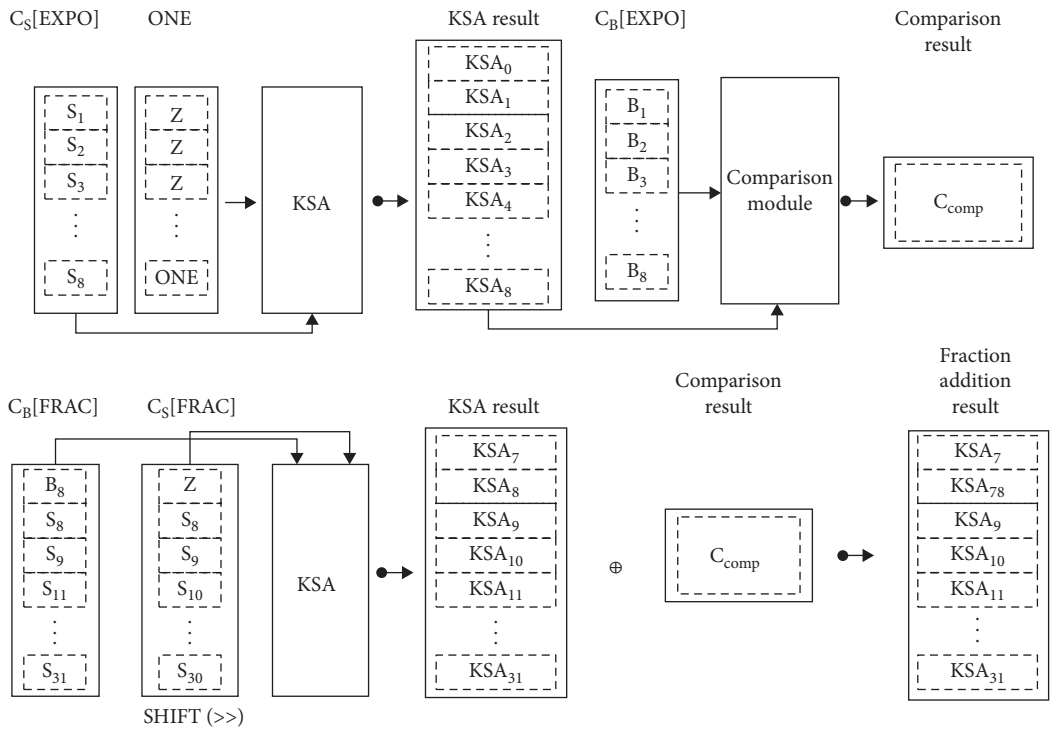

Figure 17: Overview of addition operation in TFHE-based method.

Input: $C_A[]$, $C_B[]$ // Arrays of encrypted bits, assumes length of $C_A[]$ and $C_B[]$ are same.
1. $c_{out} \leftarrow 1$
2. For $i = 0$ to len ($C_A[]$) – 1
    2.1 $c \leftarrow C_A[i] \oplus C_B[i]$
    2.2 $c \leftarrow c \oplus 1$ // negate c
    2.3 $c_{out} \leftarrow c_{out} \cdot c$
3. Output c_{out}

Figure 18: Equality comparison circuit for TFHE-based representation (TFHE_EQCC).

Input: C[] // Arrays of encrypted bits, assumes length of $C_A[]$ and $C_B[]$ are same.
1. $c_{out} \leftarrow 1$
2. For $i = 0$ to len (C[]) – 1
    2.1 $C[i] \leftarrow C[i] \oplus 1$
3.C[] ← E-KSA (C[], 1) // 1 should be contained an array of ciphertext C'[] where the first is 1
    // and the others set to zero.
4. Output C[]

Figure 19: Negate() function used in TFHE-based representation.

decimal point is omitted as specified by the standard. Therefore, the multiplication calculation should be performed after recovering the "1". Therefore, the multiplication is performed by attaching an encoding of "1" to the most significant digit, which is just left to the decimal point. In this case, the multiplication result cannot exceed 4, so the result is a 46-bit value with two bits to the left of the decimal point and the remaining 44 bits to the right.

After calculating $C_m[]$, the final step is to check whether the value in the ciphertext $C_m[0]$, which represents the most significant bit of the multiplication result and 1 bit left away from the decimal point, is 1. If $C_m[0]$ is 1, the fraction part of the multiplication result is $C_m[1] \sim C_m[22]$, and the exponent value should be incremented by one. If $C_m[0]$ is 0, that means $C_m[1]$ is 1, so the exponent does not increase and the fraction part of the result is $C_m[2] \sim C_m[24]$. The method that reflects this is shown in Figure 22 below.

4.3.3. Division. The division operation is similar to the multiplication operation. That is, to calculate the exponent part, the exponent value of the denominator is subtracted from the numerator, and division operation is performed between the fraction parts of them. Since the exponent

Input: $C_B = (C_B[SIGN], C_B[EXPO], C_B[FRAC])$, $C_S = (C_S[SIGN], C_S[EXPO], C_S[FRAC])$

1. $c_{comp} \leftarrow COMP\ (C_B[EXPO]||C_B[FRAC], C_S[EXPO]||C_S[FRAC])$ // absolute value comparison
2. $c_{sign} \leftarrow 1 \oplus C_B[SIGN] \oplus C_S[SIGN]$
3. $C_{out}[SIGN] \leftarrow (c_{sign} \cdot C_B[SIGN]) \oplus ((1 \oplus c_{sign}) \cdot (c_{comp} \cdot C_B[SIGN] \oplus (1 \oplus c_{comp}) \cdot C_S[SIGN]))$
4. $C_{B\_temp}[] \leftarrow 0||1||C_B[FRAC][0]||C_B[FRAC][1]||\ldots||C_B[FRAC][22]$, //01.(FRAC_B)
   $C_{S\_temp}[] \leftarrow 0||1||C_S[FRAC][0]||C_S[FRAC][1]||\ldots||C_S[FRAC][22]$ //01.(FRAC_S)
5. $c_{eqcc} \leftarrow TFHE\_EQCC\ (C_B[EXPO], C_S[EXPO])$
6. // Dealing with the case where $c_{sign} = 1$
   - 6.1 $C_{add}[] \leftarrow E\text{-}KSA(C_{B\_temp}[], C_{S\_temp}[])$
   - 6.2 $C_{0\_e1}[] \leftarrow E\text{-}KSA(C_B[EXPO], C_{add}[0])$ // if $C_{add}[0]$ is 1, exponent is increased
   - 6.3 $C_u[] \leftarrow (C'_{add}[24]C'_{add}[23]\ldots C'_{add}[3]C'_{add}[2])_{(2)}$
     where $C'_{add}[i] \leftarrow C_{add}[i] \cdot (1 \oplus C_{add}[i-1]) \cdot (1 \oplus C_{add}[i-2]) \cdot \ldots \cdot (1 \oplus C_{add}[0])$
   - 6.4 $C'_u[] \leftarrow Negate\ (C_u[])$, $C'_u[] = E\text{-}KSA(C'_u[], 1)$
   - 6.5 $C_{0\_e1}[] \leftarrow E\text{-}KSA(C_{0\_e1}[], C'_u[])$
   - 6.6 $C_{0\_f1}[] \leftarrow C_{add}[0] \cdot (C_{add}[1]||C_{add}[2]||\ldots||C_{add}[23])$
   - 6.7 For $i = 1$ to 24
     - 6.7.1 $C_{0\_f1}[] \leftarrow C_{0\_f1}[] \oplus C'_{add}[i] \cdot (C_{add}[i+1]||\ldots||C_{add}[24]||0||\ldots||0)$// # of 0 is $(i-1)$
7. // Dealing with the case where $c_{sign} = 0$
   - 7.1 $C_{S\_temp}[] \leftarrow Negate\ (C_{S\_temp}[])$
   - 7.2 The rest of steps are the same as step 6 beginning from 6.1. However, C_{0_e2} is used in place of C_{0_e1} and C_{0_f2} is used in place of C_{0_f1}
8. $C_{out}[EXPO] += c_{eqcc} \cdot ((c_{sign} \cdot C_{0\_e1}) \oplus (1 \oplus c_{sign}) \cdot C_{0\_e2})$
9. $C_{out}[FRAC] += c_{eqcc} \cdot (c_{sign} \cdot C_{0\_f1} \oplus (1 \oplus c_{sign}) \cdot C_{0\_f2})$
10. $C_S[EXPO] \leftarrow E\text{-}KSA(C_S[EXPO], 1)$, $C_{S\_temp} \leftarrow C_{S\_temp} >> 1$
11. $i \leftarrow i + 1$
12. if $(i < 128)$ goto Step 5.
13. Output $C_{out} = (C_{out}[SIGN], C_{out}[EXPO], C_{out}[FRAC]$

Figure 20: The addition algorithm for the TFHE-based representation. It takes C_B, C_S and outputs C_{out}.

$C_{out}[SIGN] \leftarrow C_1[SIGN] \oplus C_2[SIGN]$ ----· (1)
$C_{out}[EXPO] \leftarrow C_1[EXPO] + C_2[EXPO] - 127$ ----(2)
$C_{out}[FRAC] \leftarrow C_1[FRAC] * C_2[FRAC]$ ----(3)

Figure 21: Multiplication in TFHE-based representation.

$$C_{out}[EXPO] = E\text{-}KSA(C_{out}[EXPO], C_m[0])$$
$$C_{out}[FRAC] = C_m[0] \cdot (C_m[1]||\ldots||C_m[23]) \oplus (1 \oplus C_m[0]) \cdot (C_m[2]||\ldots||C_m[24])$$

Figure 22: Details of calculating the exponent and fraction in multiplication in TFHE-based representation.

should be added to 127 $(=2^{((\text{length of exponent bits})-1)} - 1)$ to represent the negative exponent, 127 should be added to the resultant exponent after the subtraction. An overview of division operation to calculate the resultant exponent and fraction is shown in Figures 23 and 24, respectively.

After calculating the exponent and the fraction, we need to correct them because the range of the result of calculating the fraction part is (0.5, 2). Thus, if the fraction part's result is less than 1, 1 must be subtracted from the exponent and the bit position of the resultant fraction must be shifted to left by 1 bit. The sign is determined by XORing the signs of the numerator and denominator. Figure 25 shows the operations to be performed for division. $C_{out}[EXPO]$ and $C_{out}[FRAC]$ are the result values, $C_1 = (C_1[SIGN], C_1[EXPO], C_1[FRAC])$ is the numerator, and $C_2 = (C_2[SIGN], C_2[EXPO], C_2[FRAC])$ is the denominator.

As with multiplication, the most important and difficult process is (3). We use the nonrestoring division [31] method, which uses the divide-and-conquer to implement (3). This is known to be more efficient than the well-known methods such as those of Newton-Rapth [32] and Burnikel and Ziegler [33]. Figure 26 shows the result of implementing the nonrestoring division algorithm [15] to make it work on ciphertext input. This algorithm assumes that the highest-

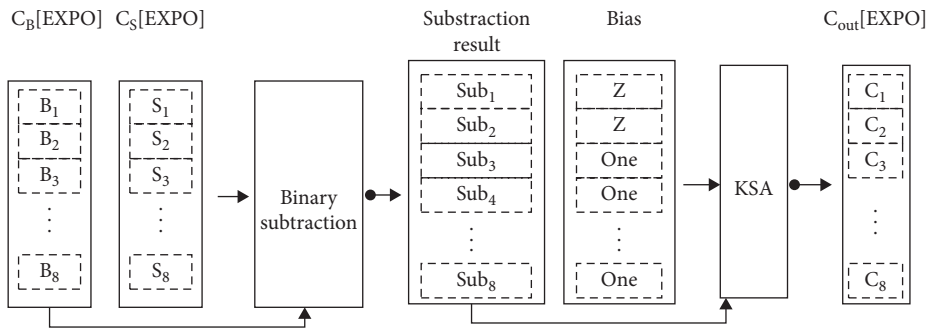

Figure 23: Exponent calculation module in the division operation of TFHE-based method.

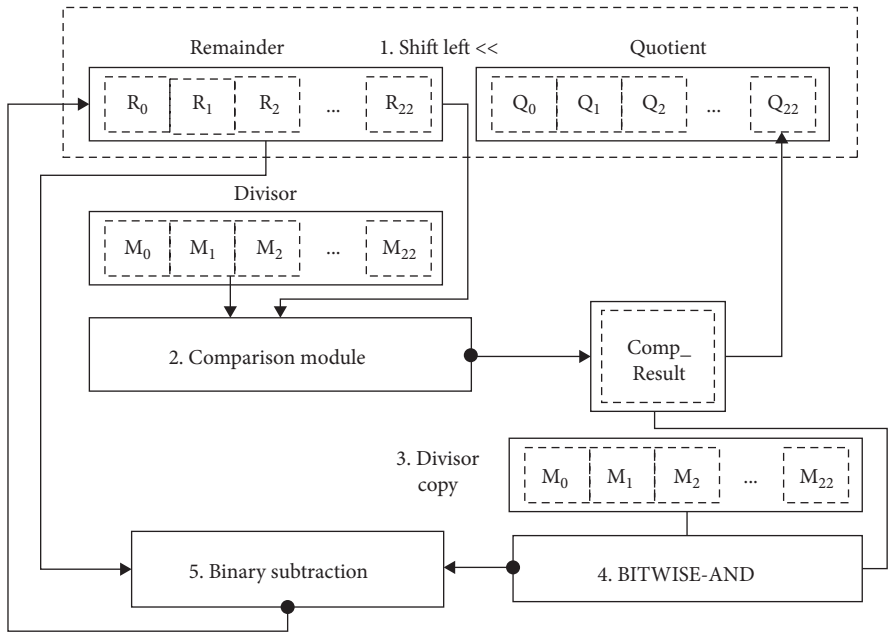

Figure 24: Fraction calculation module in the division operation of TFHE-based method.

$$C_{out}[SIGN] \leftarrow C_1[SIGN] \oplus C_2[SIGN] \quad \text{------ (1)}$$
$$C_{out}[EXPO] \leftarrow C_1[EXPO] - C_2[EXPO] + 127 \quad \text{------ (2)}$$
$$C_{out}[FRAC] \leftarrow C_1[FRAC]/C_2[FRAC] \quad \text{------ (3)}$$

Figure 25: An overview of the division operation in TFHE-based representation.

order bits are stored in index 0 of each input array, and the lower-order bits are stored as the index becomes larger. The bit length of both dividend and divisor is 24 bits each.

We deal with how to calculate (3) specifically. Since the mantissa is represented by omitting 1 immediately to the left of the decimal point which is the largest digit, in order to perform division correctly, the omitted bits should be restored before starting the division operation. To do this, the size of the ciphertext arrays indicating both the dividend and divisor is increased by one each, and we place an encoding of 1 to the highest position (i.e., the zeroth index). Then, division is performed through R_DIV (). The result is a 25-bit quotient. In this case, if the bit value of the highest position is 1, the next 23 bits are stored in the resultant fraction. Otherwise, the value of the second digit is 1 because the division quotient range is (0.5, 2). Therefore, the subsequent 23 bits are stored in the fraction part. If so, the value of exponent must be decreased by one. Figure 27 shows an example of the case where the left part of the decimal point becomes zero in the fraction part of the division result. The steps to be performed is shown in the figure. The details of (3) is shown in Figure 28.

4.3.4. Comparison and Equality Check. The comparison operation can be done using the COMP() circuit described in Figure 5(B), using the bitwise concatenation results of all of the sign, exponent, and fraction ciphertext arrays in both values to be compared as inputs. IEEE 754 standard exactly uses the same approach to compare the floating-point numbers. The process of comparing two numbers can be expressed as shown in Figure 29.

Input: $C_1[]$ (Dividend, 24bit), $C_2[]$ (Divisor, 24bit)
1. Initialize Q[]: Q[0] = 0, Q[1] = C1[0] ,…, Q[23] = C1[23] (len (Q[]) = 24 bit)
2. $i \leftarrow 0$
3. Initialize R[]: R[0] = 0, …, R[24] = 0
4. Q[] = Q[] << 1 (Q[0] ← Q[1], Q[1] ← Q[2], …, Q[23] ← Q[24], Q[24] ← 0)
5. R[] = R[] << 1
6. $c_{comp} \leftarrow$ COMP(R[], $C_2[]$)
7. Q[24] ← c_{comp}
8. $C_{sub}[] \leftarrow c_{comp}(C_2[0]||C_2[1]||\ldots||C_2[23])$
9. $C_{sub}[] \leftarrow C_{sub}[0] \oplus 1||C_{sub}[1] \oplus 1||\ldots||C_{sub}[23] \oplus 1$
10. $C_{sub}[] \leftarrow$ E-KSA($C_{sub}[]$, 1)
11. R[] ← E-KSA(R[], $C_{sub}[]$)
12. $i \leftarrow i + 1$, if $i < 24$ then goto step 4
13. Return Q[]

Figure 26: Restoring division algorithm (R_DIV()).

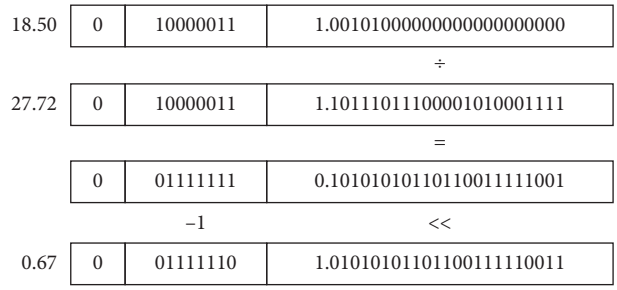

Figure 27: An example of normalizing the fraction part in the division result.

Input: C_1[FRAC], C_2[FRAC], C_{out}[EXPO]
Output: C_{out}[FRAC], (Updated) C_{out}[EXPO],
1. $C_{1\_temp}[] \leftarrow 1||C_1$[FRAC], $C_{2\_temp}[] \leftarrow 1||C_2$[FRAC]
2. Q[] ← R_DIV($C_{1\_temp}[]$, $C_{2\_temp}[]$)
3. C_{out}[FRAC] ← $Q[0]\cdot(Q[1]||Q[2]||\ldots||Q[23]) \oplus (1 \oplus Q[0])\cdot(Q[2]||Q[3]\ldots||Q[24])$
4. $C_{sub}[] \leftarrow 1||1\ldots1||Q[0]$ (len ($C_{sub}[]$) = 8)
5. $C_{sub}[] \leftarrow$ E-KSA($C_{sub}[]$, 1) // $C_{sub}[] = -Q[0]$
6. C_{out}[EXPO] ← E-KSA(C_{out}[EXPO], $C_{sub}[]$)

Figure 28: Procedure for calculating C_{out}[FRAC] with updating C_{out}[EXPO] in division.

Input: $C_1 = (C_1$[SIGN], C_1[EXPO], C_1[FRAC]), $C_2 = (C_2$[SIGN], C_2[EXPO], C_2[FRAC])
Output: c_{out} (a ciphertext that contains 1 if $C_1 > C_2$ or 0 otherwise)
1. $C_{1\_concat}[] \leftarrow C_1$[SIGN]||$C_1$[EXPO]||$C_1$[FRAC], $C_{2\_concat} \leftarrow C_2$[SIGN]||$C_2$[EXPO]||$C_2$[FRAC]
2. $c_{out} \leftarrow$ COMP (C_{1_concat}, C_{2_concat})
3. Output c_{out}

Figure 29: Comparison operation on TFHE-based representation.

The equality can be easily checked as they are represented using bits. Two values are treated the same only if they have the same bits.

4.4. Discussion on the Differences between the Implementation of Two Methods. The differences between the implementation of the two methods can be summarized as follows. First, in the operation with exponent values, the HEAAN-based method can reduce the number of operations by using the SIMD operation, but the TFHE-based method must perform operations in bit units, which requires large number of TFHE operations.

Second, the HEAAN-based implementation can implement the operations for dealing with the mantissa part with a small number of HEAAN operations using arithmetic operation provided by HEAAN, but in TFHE-based method, the operation must be processed for each bit of mantissa, which also requires a large number of TFHE operations.

In the above two cases, the HEAAN-based method seems to be dominant, but HEAAN requires bootstrapping when it needs complex arithmetic with their floating point representations: especially when it needs calculation of more than a predetermined multiplication depth. In this case, HEAAN needs a lot of computation resources. Thus, it will be much slower than the TFHE-based method.

5. Performance Evaluation

In this section, we compare the performance of the two encrypted floating-point values' representations proposed in this paper. The first subsection deals with the performance on various exponent bits. Then, we discuss the effect of the length of mantissa bits in the next subsection. Before showing the details of experiments, we provide the complexity analysis result of the operations in terms of the required multiplication operation on each FHE in Table 1. Because of the SIMD operation supported, HEAAN requires less number of multiplication operations. However, some operations which require heavy multiplication depth such as division need bootstrapping operation in HEAAN, which causes heavy delay.

5.1. Experiment on Various Exponent Bits. We measured the execution time of the operations on both methods on various exponent bits. The environment used is as shown in Table 2. The performance of the E-KSA(), COMP(), and MinMax() modules used to implement the operations in each representation is compared in Figure 30. Since HEAAN uses them only to deal with the exponent bits, we measure the execution time of the modules in processing only exponent bits in the TFHE case, for fair comparison. In order to show the performance change according to the bit length of the exponent, we measure the execution time with the exponent bit lengths of 2, 4, and 8 cases, respectively. The results are shown in the figure below. As shown in the figure, when bitwise operation is implemented in HEEAN, it takes more time than TFHE because it requires to perform interslot calculation. However, in the case of TFHE, the longer the bit length, the longer the execution time, but HEAAN is not.

Next, we compare the performance of arithmetic operations. Figure 31 shows the comparison of execution time of arithmetic operations. We can say the HEAAN-based method is superior to the TFHE-based one in terms of the speed of the arithmetic operations because the operations that should be done between the fraction parts of both operands can be executed more efficiently than the TFHE. One peculiar point is that the addition/subtraction operation is the slowest. This is due to the nature of the FHE operation. In the case of addition, the addition operation of the fraction part must be performed after the exponent has been adjusted. However, in the case of operations on data encrypted with FHE, since the exponent parts are also encrypted, we cannot make both encrypted operands have the same exponents. Therefore, the actual operation is performed by performing the operation of the fraction part for all possible exponent values and selecting the result only when the exponent values of the two operands match. Therefore, since the number of combinations of all possible exponents of both operands is large, the addition/subtraction operation takes the longest.

Figure 32 compares the performance of size comparison and equivalence operations. Since the comparison operation takes too long when the exponent is 8 bits, the experiment is performed only on 2-bit and 4-bit exponents. In this case, the HEAAN-based method consumes too many multiply operation depths for comparison operations, resulting in poor performance due to the large number of bootstrapping. Finally, we compare the execution time of the equality check operation. We can confirm that TFHE is five times faster because of the inefficiency of the equality check operation between HEAAN ciphertexts.

Table 1: Computational complexity analysis on the operations in terms of the number of FHE multiplication operations required.

Operations	TFHE	HEAAN
Addition	$O(2^e(m+e)\log(m+e))$	$O(2^e\log(e))$
Multiplication	$O(m^2 + e\log(e))$	$O(2^e e)$
Division	$O(m^2 + me + e\log(e))$	$O(m\log(e)) + 2$ bootstraps
Comparison	$O((m+e)\log(m+e))$	$O(\log(e))$

m: bit length of mantissa, e: bit length of exponent.

Table 2: Experiment environment for the experiments in Section 5.1

	TFHE	HEAAN
Parameters	Lambda = 128	log N = 15, log p = 23, log q = 29, log Q = 620, log T = 2
Experiment environment	Intel i7-6700, 3.40 GHz, 16.0 GB RAM, Ubuntu 14.04LTS	

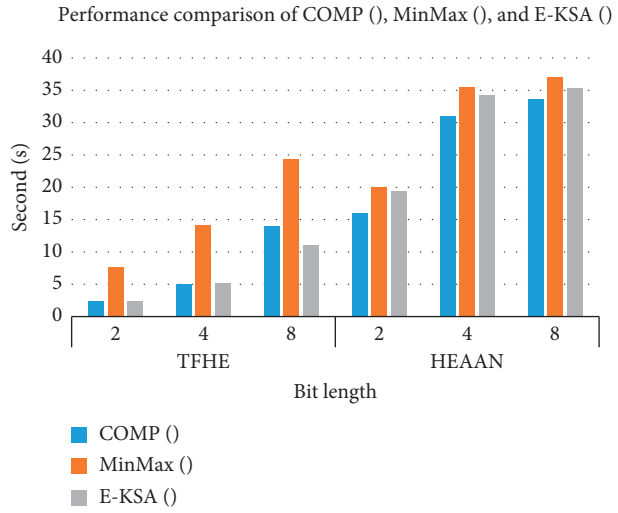

Figure 30: Performance comparison of building blocks.

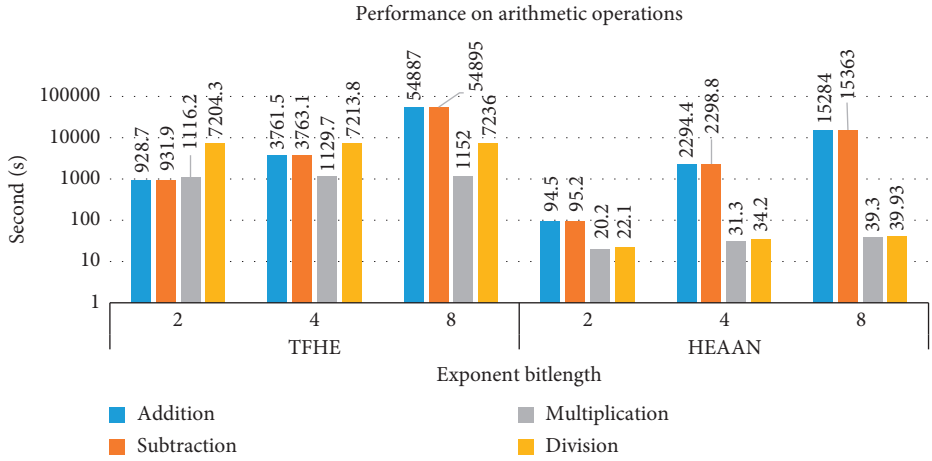

Figure 31: Performance on arithmetic operations.

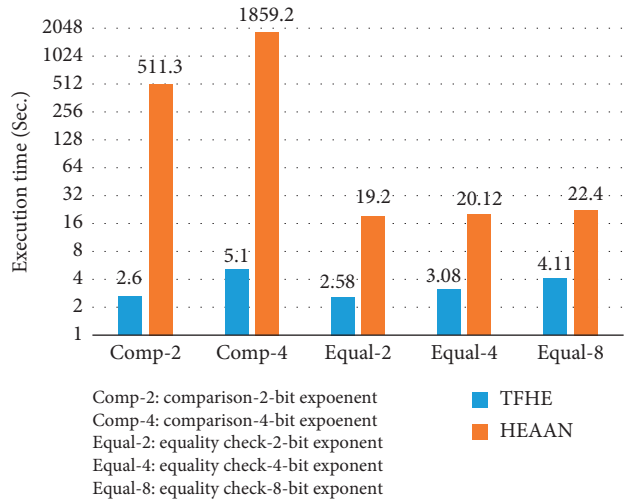

Figure 32: Performance comparisons on comparison and equality-check operations on various exponent bits.

5.2. Experiments on Various Mantissa Bits. Experiments with various mantissa bit lengths are presented in this subsection. The experimental environment is shown in Table 3. According to the developers of the libraries, both libraries offer 128-bit security in this setting. The experimental results are shown in Table 4 below. In order to provide mantissas of various bit lengths in the HEAAN-based method, we adjusted the "quantize-bit" parameter value in HEAAN. In each slot of the HEAAN ciphertext, there is a plaintext value and the noise which is attached to the plaintext value during encryption or arithmetic operations. The quantize-bit defines the bit length of those combined values. Since the bit length of noise added during encryption is at most five with a very high probability, which is around $(1 - 3 * 10^{-7})$ (this refers to Pr $[\mu\text{-}5\sigma<X<\mu+5\sigma]$, where X is a random variable of standard normal distribution with parameter (μ, σ); the σ value used in HEAAN is 3.2), we set the quantize bit as the bit length of the mantissa bit plus five. The quantize bit values used are shown as p-bit in Table 4.

Table 3: Experiment environment for the experiments in Section 5.2.

	TFHE	HEAAN
Parameters	Lambda = 128	log N = 15, log p = (Vary. See Table 4), log Q = 1446, log T = 2
Experiment environment	Intel i9-9900K, 3.6 GHz, 64.0 GB RAM, Ubuntu 18.04 LTS	

From Table 4, we can see that the comparison operation is much faster in the TFHE-based method than in the HEAAN-based implementation. However, in the multiplication operation, HEAAN-based method is better than TFHE in terms of the required time. Regarding the other operations, HEAAN seems better. However, as the exponent bit length is longer, the gap of the performance will be reduced because more multiplication depth is necessary, so bootstrapping may be needed in the HEAAN-based implementation. Thus, it is estimated that TFHE-based method might be better if the exponent bit is above a certain threshold.

5.3. Experiment on Precision in the HEAAN-Based Method. To verify the precision of the HEAAN-based method, we conducted the following experiment. After selecting two random float-type values, they were multiplied by each other, and then another random float-type number was selected and multiplied by the previous result of multiplication. This process was repeated 10 times. We performed this process using only plaintext and performed the same process with the encrypted values generated with HEAAN-based method. We decrypt the result of multiplication in each step and compare it with the result of multiplication with the plaintexts. To measure the error of calculation, we used the following formula: |(multiplication result with plaintexts)−(multiplication result with HEAAN ciphertexts)| |(multiplication result with plaintexts)|. It means the relative error of calculation. The measurement results are shown in Table 5 below. Since 2^{-23} is around 1.192E-7, we can say the error generated by the multiplication on two

Table 4: Experiment results on various mantissa bit lengths when the bit length of exponent is fixed to four (in seconds).

Fraction bit length	TFHE				HEAAN				Log p (p-bit)
	ADD	MULT	DIV	COMP	ADD	MULT	DIV	COMP	
23	489.613	162.195	686.513	5.70388	185.735	10.735	417.938	171.706	28
33	1351.28	446.062	2275.67	12.0943	166.174	10.783	716.644	152.721	38
43	1167.55	571.274	2950.23	12.6297	583.492	10.659	1066.72	531.607	48
52	1207.52	686.463	3559.19	13.0962	1110.84	10.449	2162.76	941.9	57

Table 5: Relative errors between the multiplication result with ciphertexts and that with plaintexts.

Multiplication round	1	2	3	4	5	6	7	8	9	10
Relative error	$1.21E-7$	$6.92E-8$	$7.91E-8$	$9.04E-8$	$1.03E-7$	0	$6.75E-8$	$1.54E-7$	0	0

HEAAN-based encrypted floating-point representations is small enough compared to the errors generated by the multiplication on float-type values.

6. Conclusion

In this paper, we have proposed a new encrypted floating-point number representation method that supports full arithmetic and comparison operations and is able to represent a real number whose range is very similar to that supported by the IEEE 754 standard. Our representation has been implemented on two recent FHE schemes, HEAAN and TFHE, to demonstrate the feasibility of the proposed method. We hope the proposed representation will be applied to the privacy-preserving applications where the floating point numbers are used in common, such as the machine learning with private data.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2019R1A2C4069769).

References

[1] S. Chhertri, N. Rashid, S. Faezi et al., "Security trends and advances in manufacturing systems in the era of industry 4.0," in *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, CA, USA, November 2017.

[2] J. Campos, P. Sharma, E. Jantunen, D. Baglee, and L. Fumagalli, "The challenges of cybersecurity frameworks to protect data required for the development of advanced maintenance," *Procedia CIRP*, vol. 47, p. 227, 2016.

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing—STOC'09*, pp. 169–178, Bethesda, MA, USA, May–June 2009.

[4] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–Eurocrypt 2015*, pp. 617–640, Springer, Berlin, Germany, 2015.

[5] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Eurocrypt'15*, pp. 641–670, Springer, Heidelberg, Germany, 2015.

[6] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.

[7] K. Lauter, M. Naehrig, and V. Vaikuntanathan, "Can homomorphic encryption be practical?," in *Proceedings of the 3rd ACM workshop on Cloud Computing Security Workshop—CCSW'11*, pp. 113–124, Chicago, IL, USA, October 2011.

[8] M. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Eurocrypt 2010, LNCS*, vol. 6110, pp. 24–42, Springer, Heidelberg, Germany, 2010.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Fully Homomorphic Encryption: Bootstrapping in Less than 0.1 Seconds," in *Asiacrypt 2016*, pp. 3–33, Springer, Heidelberg, Germany, 2016.

[10] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library—SEAL v2.1," in *Financial Cryptograph and Data Security*, Springer, Heidelberg, Germany, 2017.

[11] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," *Lecture Notes in Computer Science*, vol. 12, no. 7417, pp. 850–867, 2012.

[12] S. Halevi and V. Shoup, "Algorithms in HElib," in *Crypto'14*, vol. 8616, Springer, Heidelberg, Germany, 2014.

[13] C. Z. Janikow and Z. Michalewicz, "An experimental comparison of binary and floating point representations in genetic algorithms," in *Proceedings of the ICGA'91*, pp. 31–36, San Diego, Calif, USA, July 1991, http://www.cs.umsl.edu/~janikow/publications/1991/GAbin/text.pdf.

[14] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'17)*, pp. 409–437, Hong Kong, China, December 2017.

[15] G. David, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1992.

[16] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in

Proceedings of the 2018 Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'18), pp. 306–384, Aarhus, Denmark, May 2018.

[17] J. H. Cheon, K. Han, S.-M. Hong et al., "Toward a secure drone system: flying with real-time homomorphic authenticated encryption," *IEEE Access*, vol. 6, pp. 24325–24339, 2018.

[18] K. Han, S. Hong, J. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the 31st Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-19)*, Anaheim, CA, USA, July 2019, https://aaai.org/Conferences/AAAI-19/iaai-19/.

[19] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *Advances in Cryptology—ASIACRYPT 2017*, vol. 10624, pp. 377–408, Springer, Heidelberg, Germany, 2017.

[20] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO'13*, vol. 8042, pp. 75–92, Springer, Heidelberg, Germany, 2013.

[21] S. Arita and S. Nakasato, "Fully homomorphic encryption for point numbers," in *Proceedings of the 12th International Conference, Information Security Practice and Experience (ISPEC) 2016*, vol. 10143, pp. 253–270, Zhangjiajie, China, November 2016.

[22] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012, https://eprint.iacr.org/2012/144.

[23] T. Zhu, X. Zou, and J. Pan, "Query with SUM aggregate function on encrypted floating-point numbers in cloud," *Journal of Information Processing Systems (JIPS)*, vol. 13, no. 3, pp. 573–589, 2017.

[24] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 1999*, pp. 223–238, Prague, Czech Republic, May 1999.

[25] A. Jaschke and F. Armknecht, "Accelerating homomorphic computations on rational numbers," in *Proceedings of the Applied Cryptography and Network Security (ACNS) 14th International Conference*, pp. 19–22, Guildford, UK, June 2016.

[26] S. Arita and S. Nakasato, "Fully homomorphic encryption for point numbers," in *Proceedings of the International Conference on Information Security and Cryptology*, pp. 253–270, Springer, Seoul, Korea, December 2016.

[27] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Manual for using homomorphic encryption for bioinformatics," *Proceedings of the IEEE*, vol. 105, no. 3, 2017.

[28] K. Seo, P. Kim, and Y. Lee, "Implementation and performance enhancement of arithmetic adder for fully homomorphic encrypted data," *Journal of the Korea Institute of Information Security & Cryptography*, vol. 27, no. 3, pp. 413–426, 2017.

[29] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.

[30] J. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," in *IACR Eprint Archive*, Springer, Heidelberg, Germany, 2019, https://eprint.iacr.org/2019/417.

[31] S. Kaur, M. Singh, and R. Agarwal, "Vhdl implementation of non-restoring division algorithm using high speed adder/subtractor," *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, vol. 2, pp. 3317–3324, 2013, https://www.researchgate.net/publication/250612202_VHDL_Implementation_of_Non_Restoring_Division_Algorithm_Using_High_Speed_AdderSubtractor.

[32] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, no. 8, pp. 702–706, 1970.

[33] C. Burnikel and J. Ziegler, "Fast recursive division," Research Report Max-Planck-Institut fuer Informatik Research Report MPI-I-98-1-022, Max–Planck–Institut für Informatik, Saarbrücken, Germany, 1998.