Function Evaluation with Fully Homomorphic Encryption using Table Lookup

Ruixiao LI †, Yu ISHIMAKI ‡, Hayato YAMANA*
†School of Fundamental Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555, Japan.
‡Graduate School of Fundamental Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
*Faculty of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
Email: {liruixiao, yuishi, yamana}@yama.info.waseda.ac.jp

Abstract: Fully homomorphic encryption (FHE) allows a third party to evaluate arithmetic circuits over encrypted data without decryption. However, FHE typically only supports addition and multiplication over encrypted data. Thus, functions that cannot be directly represented using additions or multiplications (such as reciprocals and logarithms) are not compatible with FHE. To overcome this limitation, Crawford et al. (WAHC 2018) proposed a method of evaluating such functions using a lookup table (LUT). While this approach provides the ability to evaluate any functions using FHE, two problems arise. First, it uses bitwise encoding, i.e., an integer is decomposed into a set of bits, and then each bit is encrypted individually, which typically results in a large overhead in terms of both computational and storage costs. Second, the size of the LUT can be huge, depending on the function to be evaluated, as all possible pairs of input and output values of the function must be included. This necessitates an expensive computation for LUT evaluation with FHE. In this study, we propose a more efficient function evaluation protocol with FHE than the method of Crawford et al. Our protocol allows the use of integer encoding, i.e., a single integer is encrypted as a single ciphertext, rather than using bitwise encoding. In addition, our protocol guarantees that no information on the underlying plaintext between two parties can be leaked, via a combination of permutation operations and private information retrieval. Our experimental results in a multi-threaded environment show that the runtime of our protocol is approximately 51 s when the size of the LUT is 448,000. Thus, we surmise that our protocol is more practical.
Keyword: Function Evaluation, Lookup Table, Fully Homomorphic Encryption, Cloud Computing, Private Information Retrieval

1. INTRODUCTION

Nowadays, cloud computing services and big data techniques are proliferating rapidly. With the widespread adoption of big data technology, data security issues have received increasing attention. The existence of information uploaded by large numbers of users, stored in cloud servers, increases data security risks [1][2]. Several mature traditional encryption schemes are available, such as AES and DES. However, once data has been encrypted, operations cannot be performed on it without decryption. In 2009, a fully homomorphic encryption (FHE) scheme was introduced by Gentry [3], which allows a third party to perform function evaluations over encrypted data without decryption. However, the limitations of FHE, such as 1) only allowing the evaluation of functions composed of additions and multiplications and 2) its large computational cost and memory usage, make it difficult to adapt to big data.

The previous study [4] by Crawford et al. replaces function evaluation with a lookup table (LUT) protocol for FHE. This enables the evaluation of some complex functions using FHE, such as reciprocals and logarithms. However, two problems remain in their approach: 1) they employ bitwise encoding, which is not scalable; and 2) the LUT must include all possible input values, meaning that it can be huge, increasing the evaluation time. In this study, our protocol adopts integer encoding rather than bitwise encoding. We generate two LUT matrices, where each row in an LUT matrix includes only some of values, to speed up the evaluation. This approach is implemented with a two-party protocol, consisting of a server and a decryptor, where the latter owns a decryption key to evaluate complex functions such as reciprocals and logarithms. Although decryption is required on the encrypted data, our protocol guarantees that no information on the underlying plaintext can be leaked, by combining permutation operations with private information retrieval (PIR) [5]. The query generation method is based on our previous work [6]. In addition, a similar procedure is performed in every row in an LUT matrix, so that multi-thread operations can be adopted for all rows to further reduce the evaluation time.

This remainder of this paper is organized as follows. In Section 2, we describe FHE, BGV cryptosystems, single instruction multiple data (SIMD)-style operations, and PIR from a single database. In Section 3, related work is discussed. In Section 4, our proposed model and overall protocol for two parties is described. In Sections 5 and 6, a security analysis, experimental results, and an evaluation are presented. Finally, in Section 7 we conclude the paper.

2. PRELIMINARIES

2.1 FULLY HOMOMORPHIC ENCRYPTION

FHE is an encryption scheme that allows a third party to evaluate arbitrary functions using modular arithmetic (mod p) over encrypted data without decryption.

When $p = 2$, a number is encoded into a binary string. This is introduced in, e.g., the GSW scheme (Gentry et al. [7]), which can encrypt each bit as ciphertext. Using logic-circuit AND and XOR operations, any function can be

evaluated. However, this approach involves a large ciphertext:plaintext ratio.

When $p > 2$, a number is encoded into an integer. This FHE operation is introduced in, e.g., the BGV scheme (Brakerski et al. [8]), which can encrypt a large integer as a single ciphertext. Thus, the integer circuit has a better ciphertext:plaintext ratio, which means that more data can be processed with one operation. Using polynomial approximations, a function consisting only of additions and multiplications can be evaluated. However, this approach has some restrictions in terms of functionality, in that functions that cannot be directly represented using additions and multiplications (such as reciprocals and logarithms) are not compatible with FHE. Thus, it is ideal to apply an integer circuit for the evaluation of any function.

A. BGV Cryptosystem

In this section, we describe the following four main algorithms in the BGV scheme.

1. Setup: $params \leftarrow \text{FHE.Setup}\ (1^{\lambda},1^{L})$

Input the security parameter λ and number of levels L, output a set of parameters $params$.

2. Key Generation:

$(pk, sk) \leftarrow \text{FHE.SecretKeyGen}(params)$

Input the set of parameters, generate a pair of keys: public key pk and secret key sk.

3. Encryption: $c \leftarrow \text{FHE.Enc}(pk, m)$

Input the public key, encrypt a message m as a ciphertext c.

4. Decryption: $m \leftarrow \text{FHE.Dec}(sk, c)$

Input the secret key, decrypt a ciphertext c as a message m.

B. SIMD-Style Operations

In [9], Smart and Vercauteren introduced a type of ciphertext packing technique based on the Chinese reminder theorem (CRT), which can support SIMD-style operations. In their work [9], a CRT-represented ciphertext generated from l plaintexts can be considered as a vector consisting of l slots, each of which contains one plaintext. The SIMD-style operations over the CRT-represented ciphertext are performed slot-wise in parallel.

For example, we denote two vectors of length l by $\boldsymbol{x} = [x_1, \dots, x_l]$ and $\boldsymbol{y} = [y_1, \dots, y_l]$. Using the packing method [9], we can pack all the elements of a vector into a single ciphertext. Then, the slot-wise addition and multiplication operations over a packed ciphertext can be respectively expressed as follows:

$$Dec(Enc(\boldsymbol{x}) \boxplus Enc(\boldsymbol{y})) = [(x_1 + y_1), \dots, (x_l + y_l)]$$
$$Dec(Enc(\boldsymbol{x}) \boxdot Enc(\boldsymbol{y})) = [(x_1 \times y_1), \dots, (x_l \times y_l)]$$

2.3 Single Database PIR from FHE

The PIR protocol [5] allows a user to retrieve a record from a database server without letting the server learn which element is selected by the user.

Aguilar et al. [10] introduced a PIR scheme with a query compression technique called XPIR. Here, a PIR query in this work includes n ciphertexts, where n is the number of elements in the database. To reduce the size of a query and increase the efficiency, Angel et al. [11] proposed SealPIR, which reduces the number of queries, i.e., ciphertexts, to one, a via new query encoding method.

The basic idea of PIR with homomorphic encryption is to assume a vector $\boldsymbol{v}$ of length n, where n is the number of elements in the database. A user has the index t that they would like to access. The user creates a binary vector $\boldsymbol{q}$ of length n as a query. The element whose index is t in the query is 1, and other elements are 0. Using the packing method in [9], the user encrypts $\boldsymbol{q}$ into a ciphertext and then sends it to the database. The database responds to the user with the result $Enc(\boldsymbol{v_R}) = Enc(\boldsymbol{v}) \boxdot Enc(\boldsymbol{q})$. The result $\boldsymbol{v_R}$ only contains the element with the index t in the database. After decryption, the user can retrieve v_t from the database as desired. (Fig. 2-3)

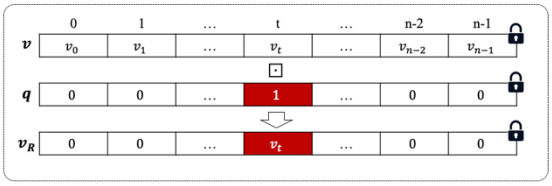

Fig. 2-3 An example of PIR from FHE

3. Related Work

It is difficult to evaluate certain functions with FHE, such as reciprocals and logarithms. Crawford et al. [4] proposed computing "complicated functions" using table lookup. When a function is difficult to evaluate, one can look up a table that includes all possible input values and corresponding output values, which have already been calculated. Using this solution, they implemented a low-precision approximation method for complex functions. A function f to be computed is pre-computed in a table T_f, such that $T_f[x] = f(x)$ for every x in some range. Then, given the encryptions of the bits of x, homomorphic table lookup is performed to obtain the bits of the value $T_f[x]$.

Two problems remain in this approach. First, bitwise encoding is employed, which is not scalable. Second, an LUT must include all possible input values, meaning that it may be huge, increasing the evaluation time. Fig. 3-1 presents an example of this approach.

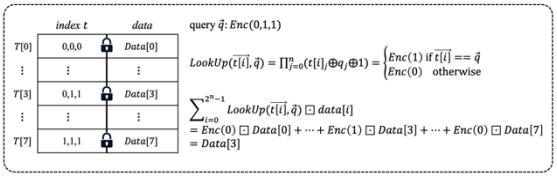

Fig. 3-1 Example of bitwise table lookup

4. Function Evaluation with FHE using Table Lookup

There are two fundamental factors that appear to limit the application of FHE to cloud computing services: 1) FHE with integer encoding can only evaluate functions composed of additions and multiplications, and 2) it requires a large computational cost and memory usage. To solve the first problem, we adopt LUTs to evaluate functions such as reciprocals and logarithms. To reduce the computational cost, we construct an LUT matrix that can be adopted for multi-thread operations.

4.1 Proposed Model

Our proposed LUT evaluation method relies on a two-party protocol, which consists of a computation server (CS) and a decryptor. Here, we suppose that a complex function f is evaluated at the CS, based on an encryption of x denoted by $\text{Enc}(x)$. Namely, the CS aims to evaluate

$\mathrm{Enc}(f(x))$, which is difficult to evaluate using an arithmetic circuit over FHE. For this propose, the decryptor is required to transform x into a PIR query. The input x and output $f(x)$ are private information in this protocol, which cannot be known by the CS or decryptor. In the initialization phase, the decryptor generates a pair of keys: a public key and secret key. The decryptor sends the public key to the CS, and owns the secret key. $\mathrm{Enc}(x)$ is the input of the CS, and $\mathrm{Enc}(f(x))$ is an output that can be used for other evaluations in the CS. For example, in a recommender system [12] [13], the cosine distance needs to be computed, which requires a square root operation. This is difficult to evaluate over FHE, because FHE only supports addition and multiplication. The input of the square root operation is a computational result. Fig. 4-1 presents an example. The function is $f(x) = \sqrt{x}$. The input of the square root operation is a computational result $Enc(x) = Enc(x_0) \boxplus \dots \boxplus Enc(x_i) \boxplus \dots \boxplus Enc(x_n)$. Here, $Enc(x_i)$ comes from user i, and $Enc(x)$ includes sensitive information for n users.

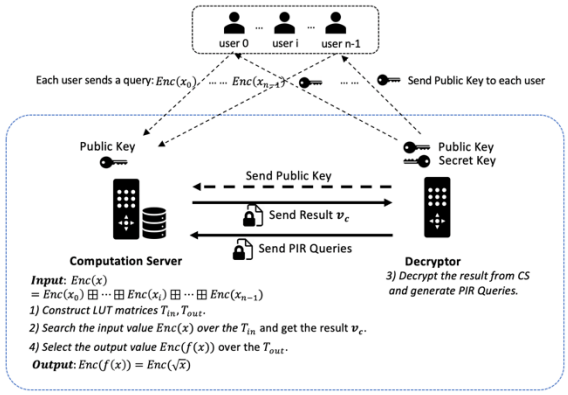

Fig. 4-1 AN EXAMPLE OF the PROPOSED MODEL

4.2 PROPOSED PROTOCOL

The overview of our protocol is as follows. 1) Construct lookup table: The CS generates two new permuted LUT matrices, based on a permutation vector generated by the CS. This is performed for each encrypted input in our protocol, and the two LUT matrices T_{in}, T_{out} are known and kept by the CS. The LUT matrix T_{in} holds all the input values of the function f, and its output is in LUT matrix T_{out}. 2) Search input value: The CS searches the input over the LUT matrix T_{in}. The result after this step also consists of ciphertext, which cannot be known by the CS. To determine the index of the input in T_{in} and generate the PIR query, the CS sends the result to the decryptor. 3) Generate PIR query: Using one decryption, the decryptor can obtain the index of the input in T_{in}, which is the same as the output index in T_{out}. Then, the decryptor generates the encrypted PIR query and sends it to the CS. 4) Select output value: The CS selects the output from T_{out} with the PIR query. Fig. 4-2 illustrates the overall protocol.

A. Construct Lookup Table

The LUT must include all input values of the function f and the corresponding output values. The original LUT can be known by the CS and the decryptor. We denote the input and output integer values in original LUT, respectively, by two vectors $\boldsymbol{v_{in}}, \boldsymbol{v_{out}} \in \mathbb{Z}^n$ of length n. The CS creates two new permuted LUT matrices T_{in}, T_{out} based on a permutation vector $\boldsymbol{v}$ generated by the CS.

FHE allows us to encrypt a vector of integers of length l as a single ciphertext. The vector $\boldsymbol{v_{in}} \in \mathbb{Z}^n$ holds all input values from the original LUT, where n is the number of elements. The vector $\boldsymbol{v_{out}} \in \mathbb{Z}^n$ holds all output values from the original LUT. The CS constructs two new LUT matrices $T_{in}, T_{out} \in \mathbb{Z}^{k \times l}$, where k is the number of rows and l is the number of columns (which is the same as the number of slots). We define $k = \lceil n/l \rceil \in \mathbb{Z}$, $|T_{in}| = k \times l \geq |\boldsymbol{v_{in}}| = n$ and $|T_{out}| = k \times l \geq |\boldsymbol{v_{out}}| = n$. Let $\boldsymbol{v}$ be a permutation vector that holds various random values within the integer range $[0, n-1]$.

Fig. 4-3 shows an example of lookup table construction. The input and output integer values in the original LUT are stored as two vectors $\boldsymbol{v_{in}}, \boldsymbol{v_{out}}$ of length 12. The new LUT matrices are T_{in}, T_{out}, with the number of slots $l = 4$ and number of rows $k = \lceil n/l \rceil = \lceil 12/4 \rceil = 3$. $\boldsymbol{v}$ is a permutation vector, which holds various random values in the integer range $[0,11]$. For ease of understanding, we add a permuted LUT vector in Fig. 4-3, which is divided into three rows in an LUT matrix.

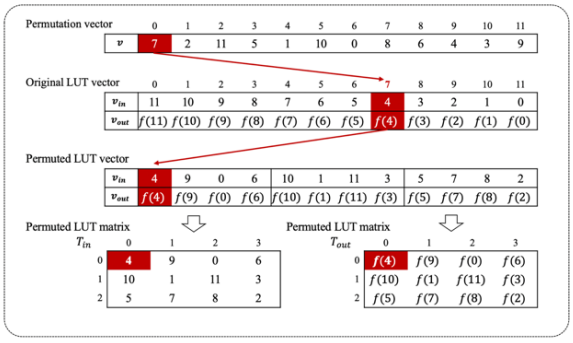

Fig. 4-3 EXAMPLE OF LOOKUP TABLE CONSTRUCTION

We represent how to construct the LUT matrices T_{in}, T_{out} from the original LUT vectors $\boldsymbol{v_{in}}, \boldsymbol{v_{out}}$ in *Algorithm 1*.

Algorithm 1: *Construct lookup table*

Input. vectors $\boldsymbol{v_{in}}, \boldsymbol{v_{out}} \in \mathbb{Z}^n$; a permutation vector $\boldsymbol{v} \in \mathbb{Z}^n$.
Output. LUT matrices $T_{in}, T_{out} \in \mathbb{Z}^{k \times l}$.

function LUTConstruction $(\boldsymbol{v_{in}}, \boldsymbol{v_{out}}, \boldsymbol{v})$
$s \leftarrow 0$
Fill all the elements in T_{in} and T_{out} by zeros
for $i = 0$ **to** $k - 1$ **do**
  for $j = 0$ **to** $l - 1$ **do**
    $index \leftarrow \boldsymbol{v}[s]$
    $T_{in}[i][j] \leftarrow \boldsymbol{v_{in}}[index]$
    $T_{out}[i][j] \leftarrow \boldsymbol{v_{out}}[index]$
    $s \leftarrow s + 1$
  end for
end for
return T_{in}, T_{out}
end function

B. Search Input Value

The CS owns the LUT matrices T_{in}, T_{out}. We denote the ciphertext of a vector with elements x as c. In addition, we denote slot-wise addition and multiplication over FHE by $\boxplus$ and $\boxdot$, respectively. For the i-th row in the matrix T_{in}, by executing the operation $c \boxplus (-T_{in}[i])$ we obtain the result as a ciphertext $\boldsymbol{v_c}[i]$. If the input value x matches the value at the index (t_{row}, t_{col}) in T_{in}, the $\boldsymbol{v_c}[t_{row}]$ encrypts a vector whose element at t_{col} is zero. Because the result $\boldsymbol{v_c}$, which will be sent to decryptor, is multiplied by the vector $\boldsymbol{r_i}$ that contains all uniformly random numbers in $\mathbb{Z}^+ \backslash \{0\}$ of length l, we can hide the true input

in the LUT matrix from the decryptor. We represent how to search the input over T_{in} in *Algorithm 2*.

Algorithm 2: *Search input value*

Input. LUT matrix $T_{in} \in \mathbb{Z}^{k \times l}$; input ciphertext c
Output. a vector of ciphertexts $\boldsymbol{v_c}$ of length k
1: **function** InputSearch (T_{in}, c)
2: **for** $i = 0$ **to** $k - 1$ **do**
3: Sample a vector $\boldsymbol{r_i}$ of uniformly random numbers
4: $\boldsymbol{v_c}[i] \leftarrow c \boxplus (-T_{in}[i])$ ▷$T_{in}[i]$: i-th row in T_{in}
5: $\boldsymbol{v_c}[i] \leftarrow v_c[i] \boxdot \boldsymbol{r_i}$ ▷$\boldsymbol{r_i}$: a random numbers vector
6: **end for**
7: **return** $\boldsymbol{v_c}$
8: **end function**

C. Generate PIR Query

The decryptor receives the result, which is a vector of ciphertexts $\boldsymbol{v_c}$ of length k. First, the decryptor decrypts $\boldsymbol{v_c}$ as a matrix V_{res}. Then, the decryptor finds the index of zero in V_{res}. Because this protocol accepts a single input value, which only matches one element of the LUT matrix T_{in}. Thus, after decryption there is also only one zero in V_{res}. We denote the index of the zero by (t_{row}, t_{col}). We know that the index (t_{row}, t_{col}) represents the input and output index in the permuted LUT matrices T_{in}, T_{out}. Here, we hide the index t_{col} from the CS using a PIR query. To hide the index t_{row} from the CS, we employ the query generation method described in [6], which can reduce the number of PIR queries. The number of PIR queries is the same as the number of dimensions $d = \lceil \log_l(l \times k) \rceil$ in a d-dimensional hypercube representation. In this study, we implemented both two-dimensional and two-dimensional cases, which means that we can support n up to l^3.

In the two-dimensional case, we generate two (to match the number of dimensions) queries $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$. The PIR query $\boldsymbol{q_0}$ is a vector whose t_{col}-th element is 1, with other elements 0. The query $\boldsymbol{q_1}$ is a permuted vector that left-rotates $t_1 = t_{row}$ elements from $\boldsymbol{q_0}$. For example, if $\boldsymbol{q_0} = \{0,0,1,0\}$ and $t_1 = 2$, then after left-rotating two elements from $\boldsymbol{q_0}$, we have that $\boldsymbol{q_1} = \{1,0,0,0\}$. If $t_1 = 3$, then after left-rotating three elements from $\boldsymbol{q_0}$ we have that $\boldsymbol{q_1} = \{0,0,0,1\}$. In the three-dimensional case, we generate three queries $\boldsymbol{q_0}, \boldsymbol{q_1}$, and $\boldsymbol{q_2}$. The PIR query $\boldsymbol{q_0}$ is a vector whose t_{col}-th element is 1, with other elements 0. The index $t_1 = t_{row} \bmod l$, and $t_2 = \lfloor t_{row}/l^2 \rfloor \bmod l$. The query $\boldsymbol{q_1}$ is a permuted vector that left-rotates t_1 elements from $\boldsymbol{q_0}$. The query $\boldsymbol{q_2}$ is a permuted vector that left-rotates t_2 elements from $\boldsymbol{q_1}$.

The decryptor encrypts the queries, and then sends them to the CS. We represent how to generate PIR queries for the two- and three-dimensional cases in *Algorithm 3* and *Algorithm 4*, respectively.

Algorithm 3: *Generate PIR query for two-dimensional case*

Input. a vector of ciphertexts $\boldsymbol{v_c}$ of length k
Output. two PIR queries $Enc(\boldsymbol{q_0})$ and $Enc(\boldsymbol{q_1})$
1: **function** QueryGeneration ($\boldsymbol{v_c}$)
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $V_{res}[i] \leftarrow Dec(\boldsymbol{v_c}[i])$ ▷a matrix V_{res}
4: **end for**
5: Fill all the element in $\boldsymbol{q_0}$ by zeros
6: **for** $i = 0$ **to** $k - 1$ **do**
7: **for** $j = 0$ **to** $l - 1$ **do**
8: **if** $V_{res}[i][j] == 0$ **then**
9: $ind \leftarrow i$
10: $\boldsymbol{q_0}[j] \leftarrow 1$
11: **end if**
12: **end for**
13: **end for**
14: $t_1 = ind$
15: $\boldsymbol{q_1} = \boldsymbol{q_0} \ll t_1$ ▷left-rotate by t_1-elements
16: **return** $Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1})$
17: **end function**

Algorithm 4: *Generate PIR query for three-dimensional case*

Input. a vector of ciphertexts $\boldsymbol{v_c}$ of length k
Output. three PIR queries $Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2})$.
1: **function** QueryGeneration ($\boldsymbol{v_c}$)
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $V_{res}[i] \leftarrow Dec(\boldsymbol{v_c}[i])$ ▷a matrix V_{res}
4: **end for**
5: Fill all the element in $\boldsymbol{q_0}$ by zeros
6: **for** $i = 0$ **to** $k - 1$ **do**
7: **for** $j = 0$ **to** $l - 1$ **do**
8: **if** $V_{res}[i][j] == 0$ **then**
9: $ind \leftarrow i$
10: $\boldsymbol{q_0}[j] \leftarrow 1$
11: **end if**
12: **end for**
13: **end for**
14: $t_1 \leftarrow \lfloor ind/l \rfloor \bmod l$
15: $t_2 \leftarrow \lfloor ind/l^2 \rfloor \bmod l$
16: $\boldsymbol{q_1} = \boldsymbol{q_0} \ll t_1$ ▷left-rotate by t_1-elements
17: $\boldsymbol{q_2} = \boldsymbol{q_1} \ll t_2$ ▷left-rotate by t_2-elements
18: **return** $Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2})$
19: **end function**

D. Select Output Value

The CS receives encrypted PIR queries from the decryptor, and then selects the output using the queries. We define a function $Perm(Ctxt, num)$, where $Ctxt$ is a ciphertext of a vector txt of length l, and num is an integer $num \in [0, l-1]$. The function $Perm(Ctxt, num)$ right-rotates num elements from the encrypted vector txt. For example, $Dec(Ctxt) = \{0,1,2\}$ and $Dec\big(Perm(Ctxt, 2)\big) = \{1,2,0\}$.

In the two-dimensional case, we first permute the query $Enc(\boldsymbol{q_1})$ by the function $Perm$. Second, for the i-th row of LUT T_{out}, we construct $\boldsymbol{q'}$ for the output selection using the operation $Enc(\boldsymbol{q_0}) \boxdot Perm(Enc(\boldsymbol{q_1}), i)$. Then, we denote the result of $q' \boxdot T_{out}[i]$ as a vector v_o of ciphertext of length k. By summing all ciphertexts in v_o, we obtain a ciphertext c', where only one slot contains the output $f(x)$, and remainder of the slots are 0.

In the three-dimensional case, for each $j \in [0, l-1] \cap \mathbb{Z}$ we first reconstruct a vector of ciphertext $\boldsymbol{v_q}$ of length l using the operation $Enc(\boldsymbol{q_1}) \boxdot Perm(Enc(\boldsymbol{q_2}), j)$. Second, for the i-th row of the LUT T_{out}, we construct q' for the output selection using the operation $Enc(\boldsymbol{q_0}) \boxdot Perm(\boldsymbol{v_q}[x], y)$. Then, we denote the result of $q' \boxdot T_{out}[i]$ as a vector $\boldsymbol{v_o}$ of ciphertext of length k. By summing all ciphertexts in $\boldsymbol{v_o}$, we obtain a ciphertext c', where only one slot contains the desired output $f(x)$, and the remainder of the slots are 0.

We represent how to select the output value over T_{output} in *Algorithm 5* and *Algorithm 6*.

Algorithm 5: *Output value selection for two-dimensional case*

Input. LUT matrix $T_{out} \in \mathbb{Z}^{k \times l}$; two PIR queries $Enc(\boldsymbol{q_0})$ and $Enc(\boldsymbol{q_1})$

Output. a ciphertext c'
1: **function** OutputSelection ($T_{out}, Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1})$)
2: $c' \leftarrow Enc(\boldsymbol{v_z})$ ▷an all-zero vector $\boldsymbol{v_z}$
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $q' \leftarrow Enc(\boldsymbol{q_0}) \boxdot Perm(Enc(\boldsymbol{q_1}), i)$
4: $\boldsymbol{v_o}[i] \leftarrow q' \boxdot (T_{out}[i])$ ▷a ciphertext q'
5: $c' \leftarrow c' \boxplus \boldsymbol{v_o}[i]$ ▷a vector of ciphertext $\boldsymbol{v_o}$
8: **end for**
9: **return** c'
10: **end function**

Algorithm 6: *Output value selection for three-dimensional case*

Input. LUT matrix $T_{out} \in \mathbb{Z}^{k \times l}$; three PIR query $Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2})$
Output. a ciphertext c'
1: **function** OutputSelection ($T_{out}, Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2})$)
2: $c' \leftarrow Enc(\boldsymbol{v_z})$ ▷an all-zero vector $\boldsymbol{v_z}$
3: **for** $j = 0$ **to** $l - 1$ **do**
4: $\boldsymbol{v_q}[j] \leftarrow Enc(q_1) \boxdot Perm(Enc(q_2), j)$
5: **end for**
6: **for** $i = 0$ **to** $k - 1$ **do**
7: $x = i/l$
8: $y = i\%l$
9: $q' \leftarrow Enc(\boldsymbol{q_0}) \boxdot Perm(\boldsymbol{v_q}[x], y)$
10: $\boldsymbol{v_o}[i] \leftarrow \boldsymbol{q'} \boxdot (T_{out}[i])$ ▷a ciphertext q'
11: $c' \leftarrow c' \boxplus \boldsymbol{v_o}[i]$ ▷a vector of ciphertext $\boldsymbol{v_o}$
12: **end for**
13: **return** c'
14: **end function**

Input: vectors $\boldsymbol{v_{in}}, \boldsymbol{v_{out}} \in \mathbb{Z}^n$; ciphertext c of input x.
Output: ciphertext c' of output $f(x)$
1. **Decryptor Setup**
[Create FHE parameters] The decryptor generates a pair of keys: a public key pk and a secret key sk.
Decryptor: $params \leftarrow FHE.Setup\ (1^{\lambda}, 1^{L}, b)$;
$(pk, sk) \leftarrow FHE.SecretKeyGen(params)$
[Send the public key] The decryptor sends the public key to CS.
$Decryptor \rightarrow CS: pk$
2. **Construct Lookup Table**
[Generate a permutation vector] CS generates a permutation vector $\boldsymbol{v} \in \mathbb{Z}^n$
[Construct LUT matrices] CS constructs LUT matrices $T_{in}, T_{out} \in \mathbb{Z}^{k \times l}$
$T_{in}, T_{out} \leftarrow LUTConstruction\ (\boldsymbol{v_{in}}, \boldsymbol{v_{out}}, \boldsymbol{v})$
(Algorithm 1)
3. **CS Searches Input Value over LUT matrix**
[Search input value over the LUT matrix] CS searches the input ciphertext c over the LUT matrix T_{in} and get the result $\boldsymbol{v_c}$
$\boldsymbol{v_c} \leftarrow InputSearch\ (T_{in}, c)$
(Algorithm 2)
[Send result to the decryptor] CS sends the result to the decryptor
4. **The Decryptor Generate PIR Query**
[Decrypt the result from the CS] The decryptor decrypts the result $\boldsymbol{v_c}$ to a matrix V_{res}
[Generate PIR queries] The decryptor performs two PIR queries in the two-dimensional case; three PIR queries in the three-dimensional case.
[Encrypt the PIR queries] The decryptor decrypts the PIR queries.
$Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}) \leftarrow$ *QueryGeneration* (v_c)
(2-dimensional case Algorithm 3)
$Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2}) \leftarrow$ *QueryGeneration* (v_c)
(3-dimensional case Algorithm 4)
[Send the PIR queries to the CS] Send the PIR queries to the CS
5. **The CS Selects Output Value from LUT matrix**
[Select output value from the LUT matrix] CS selects the output ciphertext c' from the LUT matrix T_{out} by the PIR queries from decryptor.
$c' \leftarrow OutputSelection\ (T_{out}, Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}))$
(two-dimensional case Algorithm 5)
$c' \leftarrow OutputSelection\ (T_{out}, Enc(\boldsymbol{q_0}), Enc(\boldsymbol{q_1}), Enc(\boldsymbol{q_2}))$
(three-dimensional case Algorithm 6)

Fig. 4-2 FUNCTION EVALUATION WITH FHE USING LUT PROTOCOL

5. SECURITY ANALYSIS

Our protocol is implemented using a two-party protocol, consisting of a CS and decryptor. Below, we demonstrate how the input value x and output value $f(x)$ are not revealed to the CS and decryptor. The decryptor is a semi-honest party, which owns the secret key but must follow the exact prespecified protocol, and cannot change its inputs or outputs.

We guarantee that no information on the underlying plaintext can leak to anyone in the four main algorithms in our protocol: *construct LUT matrix, search input value, generate PIR query,* and *select output value*. The original LUT can be known and uploaded by anyone, which means that the original LUT is public.

In the algorithm ***construct LUT matrix***, the CS uses the original LUT to construct two permuted LUT matrices T_{in}, T_{out}, and saves these in the CS. In the algorithm ***search input value***, the input c and output v_c of the algorithm are both ciphertext. Thus, x and $f(x)$ cannot be known by the CS. In the algorithm ***generate PIR query***, the decryptor receives the result v_c from the CS, and decrypts it to V_{res}. The decryptor knows the index (t_{row}, t_{col}), i.e., $V_{res}[t_{row}][t_{col}] = 0$, which is also the index of the value in T_{out} that we want to extract. Then, the decryptor generates the PIR queries. Because 1) the original LUT is permuted by a random permutation and the permuted LUT matrices T_{in}, T_{out} are not sent to decryptor, and 2) all the elements in V_{res} contain uniformly random values, the decryptor cannot know x and $f(x)$. In the algorithm ***select output value***, the input PIR queries and output c' are both ciphertexts. Thus, x and $f(x)$ cannot be known by the CS.

Therefore, neither CS nor the decryptor can know the input value x and output value $f(x)$.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the proposed protocol to confirm its efficiency, by implementing it with HElib*, which is based on the BGV scheme. The implemented protocol considers the two-dimensional and two-dimensional cases, which means that the protocol supports up to l^3 elements in the LUT matrix, whose number of columns is l (the number of slots). In the evaluation, one machine was prepared to operate as both the CS and decryptor, i.e., both modules run on the same machine. The communication between the CS and decryptor, such as ciphertext results and encrypted PIR queries, was handled by writing to files.

HElib*: https://github.com/shaih/HElib

The experimental evaluation consists of two evaluations: Experiment-1 and Experiment-2. Experiment-1 confirms the effectiveness of our proposed protocol in comparison with the method of Crawford et al.. Experiment-2 evaluates the feasibility of our proposed protocol, by measuring both the runtime and communication cost.

Experiment-1

Experiment-1 compares the proposed method with the related method of Crawford et al. [4]. The utilized FHE parameters are shown in Table 6-1, which are as consistent as possible with those of [4]. We set the plaintext space as $2^{15} = 32{,}768$, which means that in our protocol the input and output integers of the function are in a 15-bit range size. The number of elements in this experiment is 27,000 ($k = 15$). We employed the environment detailed in Table 6-2 for Experiment-1. We set a larger plaintext space and table size on the machine, whose CPU and memory are weaker to those used in [4], to perform the experiment. To implement a multi-thread environment, we utilize NTL/BasicThreadPool.h from the NTL* library, and we test the runtime for different numbers of threads.

Experiment-2

The purpose of this experiment is to measure the runtime and communication cost for our proposed protocol. We perform this experiment with different numbers of elements in the LUT matrix. Using the same FHE parameters as shown in Table 6-1, we vary the number of elements by varying the number of rows k in the LUT matrix. When $k \leq 224$, the number of dimensions is two, and when $224 < k \leq 224^2$ the number of dimensions is three. We employed the environment detailed in Table 6-2. To implement a multi-thread environment, we employ NTL/BasicThreadPool.h from the NTL library, and set the number of threads to 36, which means that we utilize two CPUs (each with 18 cores) in this experiment.

6.1 Runtime Results

We show the runtimes for the s*earch input value, generate PIR query,* and *select output value* algorithms in **Experiment-1**, along with the total time as the average over five trials, in Table 6-3. The runtime for **Experiment-2** is shown in Table 6-4.

Crawford et al. employed an Intel Xeon E5-2698 v3 (which is a Haswell processor), with two sockets and 16 cores per socket, at 2.30 GHz. The main memory size was 250 GB. They set $m = 2^{15} - 1 = 32767$, with the plaintext space $2^{11} = 2048$, so that 1800 slots were available. Each plaintext slot held a degree-15 extension ($L = 15$).

The result from Experiment-1 shows that we can look up 27,000 15-bit integers the LUT matrix in approximately 23 s, excluding the communication time. By using eight threads, the runtime can be reduced to 6 s excluding the communication time. The evaluation time is independent of the function. Our result shows that our proposed protocol using integer encoding is more practical than the method in [4].

The result from Experimente-2 shows that we can look up 448,000 19-bit integers in the LUT in approximately 20 min for one thread, excluding the communication time. By using 36 threads, the runtime can be reduced to approximately 51 s excluding the communication time. Our results show that the construction of the LUT matrices supports multi-thread execution in every row, which makes it possible to further reduce the runtime.

6.2 Communication Cost

In our protocol, the CS communicates with the decryptor. We measured the transferred data size for each input in our protocol, and calculated the transmission time. The communication cost in **Experiment-1** is shown in Table 6-5, and that in **Experiment-2** is shown in Table 6-6.

The result for Experiment-1 (Table 6-5) shows that when the transmission speed is 100 Mbps, we can look up 27,000 15-bit integers the LUT matrix in approximately 30 s using one thread, including approximately 7 s of communication time. Even our protocol involves communication between the CS and decryptor, the overall runtime of our protocol is approximately 120 times faster than that of Crawford et al. [4] running on one thread, and approximately 74 times faster running on eight threads.

The result from Experiment-2 (Table 6-6) shows that in our protocol, the communication cost is $O(k \cdot s)$ for the CS to send the result to the decryptor, and $O(d \cdot s)$ for the decryptor to send the PIR query to the CS, where s is the size of a single ciphertext, k is the number of rows in the LUT matrix, and d is the number of dimensions. As the number of elements in the LUT matrix increases, the transferred data size from the CS to the decryptor increases linearly, and the transferred data size from the decryptor to the CS increases in stages.

NTL*: https://www.shoup.net/ntl/

Table 6-1 FHE Parameters of HElib in the Experiment

Experiment	m	l	Security	Plaintext space	L
1	32,767	1,800	300	32,768 (2^15)	15
2	11,441	224	142	524,288 (2^19)	10

Table 6-2 Experimental Environment in the Experiment

Experiment	OS	CPU	Memory	# of CPUs (sockets)
1	Ubuntu 18.04.1	Intel(R) Core (TM) i7-8700 @3.2 GHz	15.4 GB	one (each with six cores)
2	CentOS 7.3.1611	Intel Xeon CPU E7-8880 v3 @ 2.3 GHz (Turbo Boost: 3.1 GHz)	3 TB	four (each with 18 cores)

Table 6-3 RUNTIME IN EXPERIMENT-1

	Our protocol				Crawford et al. [4]			
# of threads	1	2	4	8	1	2	4	8
(a) Search Input Value [s]	0.63	0.37	0.21	0.17	60 min	35 min	20 min	16 min
(b) Generate PIR Query [s]	14.90	8.33	4.28	3.32				
(c) Select Output Value [s]	7.72	4.71	2.65	2.20				
(a)+(b)+(c) Total [s]	23.25	13.42	7.14	5.69				

Table 6-4 RUNTIME IN EXPERIMENT-2

# of elements	44,800				448,000			
# of dimensions	2				3			
# of threads	1	4	12	36	1	4	12	36
(a) Search Input Value [s]	4.73	1.50	0.61	0.44	47.30	13.78	4.98	2.28
(b) Generate PIR Query [s]	53.55	15.02	5.44	2.47	534.18	149.98	51.57	17.98
(c) Select Output Value [s]	55.01	16.56	6.52	5.03	598.09	179.22	69.86	30.80
(a)+(b)+(c) Total [s]	113.29	33.08	12.57	5.90	1179.58	342.97	126.55	51.07

Table 6-5 COMMUNICATION COST IN EXPERIMENT-1

Data	**# of elements**	**Measured transferred data size [MB]**	**Calculated transmission time [ms]**		
			100 Mbps	**1 Gbps**	**10 Gbps**
CS to Decryptor	27,000	82	6406.25	640.6	64.06
Decryptor to CS		11	859.38	85.93	8.60

Table 6-6 COMMUNICATION COST IN EXPERIMENT-2

Data	**# of elements**	**# of dimensions**	**Measured transferred data size [MB]**	**Calculated transmission time**		
				100 Mbps	**1 Gbps**	**10 Gbps**
CS to Decryptor	11,200	2	66	5.16 s	0.52 s	0.05 s
	22,400		132	10.31 s	1.03 s	0.10 s
	33,600		197	15.39 s	1.54 s	0.15 s
	44,800		263	20.55 s	2.06 s	0.21 s
	112,000	3	656	51.25 s	5.13 s	0.51 s
	224,000		1,331	103.98 s	10.40 s	1.04 s
	336,000		2,048	160.00 s	16.00 s	1.60 s
	448,000		2,662	207.97 s	20.80 s	2.08 s
Decryptor to CS	11,200	2	2.7	210.9 ms	21.10 ms	2.11 ms
	22,400					
	33,600					
	44,800					
	50,176					
	50,177	3	4.0	312.50 ms	31.25 ms	3.13 ms
	112,000					
	224,000					
	336,000					
	448,000					

7. CONCLUSION

In this paper, we proposed an LUT protocol for evaluating any single-integer input functions. We proposed a new protocol to resolve the problems with the existing method in [4], i.e., 1) bitwise encoding is employed, which is not scalable, and 2) the LUT must include all possible input values, meaning that it may be huge, increasing the evaluation time.

Our protocol adopts integer encoding, which is more efficient than bitwise encoding. The experimental results show that we can look up 27,000 15-bit integers in the LUT matrix in approximately 30 s using one thread, including approximately 7 s of communication time. The evaluation time is independent of the function.

The construction of the LUT matrices supports multi-thread execution in every row, which makes it possible to further reduce the runtime. We can look up 448,000 19-bit integers in the LUT in approximately 51 s excluding the communication time using 36 threads.

Using the query generation method based on our previous work [6], we can reduce the communication cost. The communication cost is $O(k \cdot s)$ for the CS to send the result to the decryptor, and $O(d \cdot s)$ for the decryptor to send the PIR query to the CS, where s is the size of a single ciphertext, k is the number of rows in the LUT matrix, and d is the number of dimensions. As the number of elements in the LUT matrix increases, the transferred data size from the CS to the decryptor increases linearly, and the transferred data size from the decryptor to the CS increases in stages.

Our experimental result shows that we can achieve a shorter runtime using a weaker machine than in [4] to look up more integers, which demonstrates that our protocol is more practical than that in [4].

Currently, our proposed protocol can evaluate any functions with one input value. Our future work will include extending our protocol to handle multi-input values.

ACKNOWLEDGEMENT

This work was supported by JST CREST Grant Number JPMJCR1503, Japan and Japan-US Network Opportunity 2 by the Commissioned Research of National Institute of Information and Communications Technology (NICT), JAPAN.

REFERENCES

[1] M. D. Ryan.: Cloud computing security: The scientific challenge, and a survey of solutions. In Journal of Systems and Software, vol 86, issue 9, pp.2263-2268. (2013)

[2] S. M. Shariati, Abouzarjomehri and M. H. Ahmadzadegan.: Challenges and security issues in cloud computing from two perspectives: Data security and privacy protection. In Proc. of KBEI 2015, pp.1078-1082. (2015)

[3] C. Gentry.: Fully Homomorphic Encryption Using Ideal Lattices. In Proc. of STOC 2009, pp.169-178. (2009)

[4] J. L. H. Crawford, C. Gentry, S. Halevi, D. Platt and V. Shoup: Doing Real Work with FHE: The Case of Logistic Regression. In Proc. of WAHC 2018, pp. 1-12. (2018)

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan.: Private information retrieval. In Journal of the ACM, vol 45, issue 6, pp. 965-981. (1998)

[6] Y. Ishimaki, H. Imabayashi and H.Yamana.: Private Substring Search on Homomorphically Encrypted Data. In Proc. of SMARTCOMP 2017, pp.1-6. (2017)

[7] Z. Brakerski and V. Vaikuntanathan.: Lattice-based FHE as secure as PKE. In Proc. of ITCS 2014, pp. 1-12. (2014)

[8] Z. Brakerski, C. Gentry and V. Vaikuntanathan: (Leveled) Fully Homomorphic Encryption without Bootstrapping. In Proc. of ITCS 2012, pp.309-325. (2012)

[9] N. P. Smart and F. Vercauteren.: Fully homomorphic SIMD operations. In Journal of Designs, Codes and Cryptography, vol 71, issue 1, pp. 57-81. (2014)

[10] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M. Killijian.: XPIR: Private information retrieval for everyone. In Journal of Proceedings on Privacy Enhancing Technologies, vol 2016, issue 2, pp. 155-174. (2015)

[11] S. Angel, H. Chen, K. Laine, S. Setty.: PIR with Compressed Queries and Amortized Query Processing. In Proc. of S&P 2018, pp.962-979. (2017)

[12] S. Badsha, X. Yi, I. Khalil and E. Bertino.: Privacy Preserving User-based Recommender System. In Proc. of ICDCS 2017, pp. 1074-1083. (2017)

[13] Q. Tang and H. Wang.: Privacy-preserving Hybrid Recommender System. In Proc. of CSS 2017, pp. 59-66. (2017)