PAPER

Privacy Preserving Function Evaluation Using Lookup Tables with Word-Wise FHE

Ruixiao LI[†a)], *Nonmember and* **Hayato YAMANA**[††b)], *Fellow*

SUMMARY Homomorphic encryption (HE) is a promising approach for privacy-preserving applications, enabling a third party to assess functions on encrypted data. However, problems persist in implementing privacy-preserving applications through HE, including 1) long function evaluation latency and 2) limited HE primitives only allowing us to perform additions and multiplications. A homomorphic lookup-table (LUT) method has emerged to solve the above problems and enhance function evaluation efficiency. By leveraging homomorphic LUTs, intricate operations can be substituted. Previously proposed LUTs use bit-wise HE, such as TFHE, to evaluate single-input functions. However, the latency increases with the bit-length of the function's input(s) and output. Additionally, an efficient implementation of multi-input functions remains an open question. This paper proposes a novel LUT-based privacy-preserving function evaluation method to handle multi-input functions while reducing the latency by adopting word-wise HE. Our optimization strategy adjusts table sizes to minimize the latency while preserving function output accuracy, especially for common machine-learning functions. Through our experimental evaluation utilizing the BFV scheme of the Microsoft SEAL library, we confirmed the runtime of arbitrary functions whose LUTs consist of all input-output combinations represented by given input bits: 1) single-input 12-bit functions in 0.14 s, 2) single-input 18-bit functions in 2.53 s, 3) two-input 6-bit functions in 0.17 s, and 4) three-input 4-bit functions in 0.20 s, employing four threads. Besides, we confirmed that our proposed table size optimization strategy worked well, achieving 1.2 times speed up with the same absolute error of order of magnitude of -4 ($a \times 10^{-4}$ where $1/\sqrt{10} \leq a < \sqrt{10}$) for Swish and 1.9 times speed up for ReLU while decreasing the absolute error from order -2 to -4 compared to the baseline, i.e., polynomial approximation.

key words: *function evaluation, privacy preserving, lookup table, fully homomorphic encryption*

1. Introduction

Cloud computing is widely used to provide numerous services to users; however, massive amounts of sensitive data stored in cloud servers cause data security issues [1]. We usually use encryption schemes to protect data; however, traditional encryption schemes, such as AES and DES, cannot enable computation over encrypted data. The encrypted data needs to be decrypted before the computations, which results in revealing the data to the cloud server or third parties.

The other solutions include secure multi-party computation (SMPC), differential privacy (DP), and homomorphic encryption (HE).

SMPC can be used in many cloud computing applications, such as machine learning [2], private set operations [3], [4], and secure genomic sequence comparison [5] models. SMPC is based on the secret sharing scheme and focuses on efficiency. The challenge for SMPC is the data transmission cost for large data applications. DP is another privacy-preserving scheme used in various applications [6]–[10]. However, privacy budget optimization and compatibility of DP among different applications remain to be solved [11]. Besides, DP cannot be used in applications that require exact computation owing to the noise added by the DP technique.

Alternatively, homomorphic encryption (HE) allows a third party to evaluate functions over ciphertext. Fully homomorphic encryption (FHE) proposed by Gentry [12], enables an arbitrary number of additions and multiplications over ciphertext. Thus, FHE can protect data during the computation to prevent data breaches to the server that computes the data, such as the cloud servers. In FHE, two types of encoding schemes exist, i.e., bit-wise and word-wise. Bit-wise encoding encrypts the data bit-by-bit to evaluate functions by constructing arbitrary circuits like logic circuits. Word-wise encoding encrypts a set of bits, i.e., a word, to improve the calculation efficiency but is limited to the functions combined by additions and multiplications.

The challenges of function evaluation with FHE are 1) high computational cost, which leads to long latency, and 2) lack of supporting functions that cannot be decomposed into additions and multiplications. For example, activation functions like ReLU and Swish, used in machine learning, cannot be implemented as they are because ReLU requires a branch operation, and Swish needs division. Prior researches have employed polynomial approximation or lookup tables (LUTs) to implement such functions. However, the polynomial approximation introduces significant calculation errors while LUTs face long latency.

Xie et al. [13] first proposed a polynomial approximation technique for deterministic functions using additions and multiplications only. Complex functions such as the ReLU and Swish functions are evaluated through polynomial approximation via FHE in [14]–[16]. However, precise results are constrained to a predetermined range, as accuracy significantly diminishes beyond the range. Although higher polynomial degrees can enhance accuracy, they necessitate a larger multiplication level denoted as L resulting in long computation latency.

Additionally, the works by Crawford et al. [17], Chillotti

Manuscript received September 17, 2023.
Manuscript revised October 19, 2023.
Manuscript publicized November 16, 2023.

[†]Graduate School of Fundamental Science and Engineering, Waseda University, Tokyo, 169-8555 Japan.
[††]Faculty of Science and Engineering, Waseda University, Tokyo, 169-8555 Japan.
a) E-mail: liruixiao@yama.info.waseda.ac.jp (Corresponding author)
b) E-mail: yamana@yama.info.waseda.ac.jp

DOI: 10.1587/transfun.2023EAP1114

et al. [18], Carpov et al. [19], Micciancio et al. [20] and Liu et al. [21] substitute complex functions with homomorphic table lookup. Their approach employs bit-wise FHE for LUT processing, incurring a computation cost of $O(s \cdot 2^d)$, where d represents the input bit-length and s represents the output bit-length of the function.

The straightforward implementation of multi-input functions with bit-wise LUT necessitates $O(s \cdot 2^{\sum_{i=1}^{m} d_i})$, where m is the number of inputs, each with a bit length of d. This is because we treat the inputs as a concatenation of multiple inputs. Thus, the latency of bit-wise LUTs grows exponentially with the number of input bits. Thus, bit-wise LUTs do not suit the functions with large input bits.

In 2021, Lu et al. [22] introduced PEGASUS, which seamlessly transitions between bit-wise and word-wise ciphertext schemes without decryption. This approach enables the evaluation of arithmetic functions with word-wise FHE while implementing LUT with bit-wise FHE to assess complex functions, which retains fast evaluations with word-wise FHE. However, their work did not show any multi-input functions. Another study by Maeda et al. [23] presented a LUT method for uni/bivariate functions utilizing word-wise FHE. Nonetheless, their method cannot accommodate multi-input functions with more than two inputs. Thus, how to construct multi-input functions over two inputs has remained an open question.

In summary, how to implement arbitrary functions with FHE efficiently is an open question. Specifically, 1) polynomial approximation introduces large error, 2) bit-wise LUT suffers from long latency, which increases exponentially to the input bits, and 3) word-wise LUT cannot handle multi-input functions over two inputs. To respond to the above problems, we propose a novel privacy preserving function evaluation method using LUTs with word-wise FHE. Figure 1 illustrates our adopted three-party model, consisting of a user, a computation server, and a trusted authority. The user and the computation server are semi-honest, i.e., honestly follow the protocol but are curious to obtain sensitive data, while the trusted authority is honest. The following are our contributions.

1) We propose a privacy-preserving function evaluation method using word-wise FHE to enable the evaluation of arbitrary multiple-input functions that can handle over two inputs. Furthermore, we devise the structure of LUTs to support multi-threaded processing, shortening the latency.

2) We employ slot-wise operations [24], i.e., SIMD, to parallelize FHE. The computational complexity is then reduced from $O(s \cdot 2^{\sum_{i=1}^{m} d_i})$ with naive bit-wise FHE to $O(\lceil 2^{\sum_{i=1}^{m} d_i}/l \rceil)$ with our proposed word-wise FHE adopting slots, where d and s represent the bit-length of input and the number of output data points, respectively; l denotes the number of slots in one ciphertext; m denotes the number of inputs.

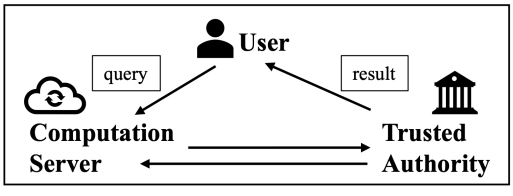

Fig. 1 Our model.

3) Our proposed nearby matching enables us to select the closest entry in a LUT to the input value(s) for further complexity reduction, which allows the number of data points in LUTs to be freely determined.

The experimental evaluation compares our proposed method with the widely employed polynomial approximation technique [14] and naive LUT implementation using bit-wise FHE. The experimental result shows 1.2 times speed up with the same absolute error of order 10^{-4} for Swish and 1.9 times speed up for ReLU while decreasing the absolute error from order 10^{-2} to 10^{-4} compared to polynomial approximation. Furthermore, our runtime was 12.8 times faster than the approach utilizing bit-wise FHE for a 3-bit single-input function.

This paper is the extended version of our previous papers [25], [26]. This paper provides detailed experimental evaluations compared to previous polynomial approximation methods to clarify the advantage of our method.

The rest of this paper is organized as follows. Section 2 gives necessary preliminaries, followed by related work in Sect. 3. Section 4 proposes our privacy preserving function evaluation method. Section 5 shows the complexity analysis of our work. Then, experimental evaluation and discussions are provided in Sects. 6 and 7. Finally, Sect. 8 concludes this paper.

2. Preliminaries

2.1 Definitions of Symbols

The notations used in this paper are shown in Table 1. The LUTs containing input and output entries of a given function are denoted as T_{in} and T_{out}, respectively. For example, assuming an single-input function $f(x) = |x|$, where x is an Integer satisfying $-2 \le x \le 2$; then we have $T_{in} = [-2, -1, 0, 1, 2]$ and $T_{out} = [2, 1, 0, 1, 2]$, where $f(T_{in}(i)) = T_{out}(i)$ and i shows the index of LUTs. We denote vectors in

Table 1 Definitions of symbols.

Symbols	Definitions
l	the number of slots (explained in Section 2.2)
m	the number of inputs for multi-input function
$\lvert T_{in} \rvert, \lvert T_{out} \rvert$	the number of data points in LUT T_{in}, T_{out}
d, s	the bit-length of input and output, respectively
$[\cdot]$	a vector of plaintexts
$\boxplus, \boxdot$	homomorphic addition and multiplication
$Enc(\cdot)$	encryption operation
$Dec(\cdot)$	decryption operation
$ct(\cdot)$	a ciphertext of a vector (l slots)
c	an input value for a given function
r	an output value for a given function
R	the intermediate results (explained in Section 4.5)
N_q	the number of PIR queries (explained in Section 2.3)
Q	the reconstructed PIR queries (explained in Section 4.5)

bold lowercase letters and matrices in uppercase letters, i.e., $ct(\boldsymbol{a})$, and $ct(A(i))$ represent an encrypted vector $\boldsymbol{a}$, and an encrypted i-th row of matrix A, respectively.

2.2 SIMD Operation over Fully Homomorphic Encryption

Element-wise single instruction multiple data (SIMD) operations with FHE were introduced by N. P. Smart and F. Vercauterenin [27] in 2014. A vector that holds l elements is encrypted as a single ciphertext by the word-wise FHE, where each element is called a slot [24]. An element is an Integer that is handled on modulo computation within a given modulus. For example, assuming two vectors $\boldsymbol{x} := [x_0, x_1, \ldots, x_{l-1}]$ and $\boldsymbol{y} := [y_0, y_1, \ldots, y_{l-1}]$ are encrypted by $Enc(\boldsymbol{x})$ and $Enc(\boldsymbol{y})$. The SIMD operation enables element-wise addition and multiplication as follows:

$$\begin{aligned} Dec(Enc(\boldsymbol{x}) \boxplus Enc(\boldsymbol{y})) &:= [x_0 + y_0, \ldots, x_{l-1} + y_{l-1}] \\ Dec(Enc(\boldsymbol{x}) \boxdot Enc(\boldsymbol{y})) &:= [x_0 \times y_0, \ldots, x_{l-1} \times y_{l-1}] \end{aligned} \tag{1}$$

2.3 Private Information Retrieval over FHE

Private information retrieval (PIR) was first introduced by Chor et al. [28]. PIR hides users' access information from a database when they search for data from the database. The main idea is as follows. Given a private database $\boldsymbol{D}$, where $\boldsymbol{D} := [d_0, \ldots, d_{l-1}]$. To hide the access information from the database, the user makes a vector $\boldsymbol{q} := [0, \ldots, 1, \ldots, 0]$, where only the index of the target data element is one and the other elements are zero. Then, the user sends the ciphertext of PIR query $ct(\boldsymbol{q})$ to the database. After receiving $ct(\boldsymbol{q})$, the database computes the encrypted inner product $\boldsymbol{D} \cdot ct(\boldsymbol{q})$, followed by sending back the inner product, i.e., the result, to the user. It is important to note that the result of the calculation between a plaintext and a ciphertext is a ciphertext. Since all of the operations in the database are over ciphertexts, we can hide the user's access information from the database.

3. Related Work

Related efforts aimed at implementing functions with FHE can be categorized into two main approaches: 1) polynomial approximation-based methods and 2) lookup table-based methods.

3.1 Polynomial Approximation-Based Methods

Polynomial approximation with FHE was initially introduced by Xie et al. in 2014 [13]. However, the approximation method encounters errors, particularly when operating with inputs beyond the predefined range. Additionally, achieving more accurate outputs demands higher polynomial degrees, resulting in long latency due to the necessity of a larger multiplication level.

Chabanne et al. [14] applied polynomial approximation to the ReLU function, utilizing degrees ranging from 2 to 6. This approach yielded accuracy ranging from 97.55% (2 degrees) to 97.91% (6 degrees) when applied to their neural network. E. Lee et al. [15] and J. Lee et al. [16] proposed utilizing a composition of minimax approximated polynomials with low degrees for functions such as Sign [15], ReLU [16], and max-pooling [16]. E. Lee et al.'s algorithm determined the optimal set of degrees for the minimax composite polynomial by considering the number of non-scalar multiplications and depth consumption. This approach effectively reduced function runtime by an average of 45% [15]. While E. Lee et al. achieved low degrees of polynomial approximation, the range of applicability for the approximation method remains limited.

3.2 Lookup Table-Based Methods

Crawford et al. [17], Chillotti et al. [18], Carpov et al. [19], Micciancio et al. [20], and Liu et al. [21] employed homomorphic table lookup to implement complex operations, utilizing bit-wise FHE. The naive bit-wise LUT incurs a computation cost of $O(s \cdot 2^d)$, where d represents the input bit-length and s signifies the output bit-length of a given function.

Boura et al. [29] and Lu et al. [22] introduced a technique that enables seamless transitions between polynomial and non-polynomial functions on encrypted data. This approach allows the evaluation of arithmetic functions using word-wise encoding FHE to enhance efficiency, as seen with schemes like CKKS [30], while assessing LUTs through bit-wise encoding FHE for functions that cannot be decomposed into additions and multiplications, as exemplified by FHEW [31]. Maeda et al. [23] introduced a LUT method for uni/bivariate functions using word-wise FHE. However, Maeda et al. and Lu et al. [22], [23] did not address the solution for multi-input functions with inputs exceeding two.

3.3 Summary

In summary, the problems to be solved are that 1) polynomial approximation introduces inherent errors, making it unable to adapt to exact calculations; 2) bit-wise LUT methods suffer from long latencies that increase exponentially with the bit-length of the function's input(s) and output; and 3) current word-wise LUT methods cannot handle multi-input functions with more than two inputs.

4. Privacy Preserving Function Evaluation Using LUTs with Word-Wise FHE

This section proposes a privacy-preserving function evaluation method using LUTs with word-wise FHE to evaluate arbitrary multiple-input functions with sufficient accuracy. To tackle the remaining problems shown in Sect. 3.3, we adopt the following strategies: 1) preparing enough LUT entries to ensure the accuracy of the function (described in

Table 2 Comparison between naive bit-wise LUT implementation and ours.

	FHE scheme	Multi-input function	LUT size	Server setting	Computational complexity
Naive	bit-wise	yes	fixed	single-server (semi-honest)	$O(s \cdot 2^{\sum_{i=1}^{m} d_i})$
Our	word-wise	yes	tunable	multi-server (semi-honest & honest)	$O(2^{\sum_{i=1}^{m} d_i}/l)$

d is input bit-length, s is output bit-length and m is the number of inputs for a given function, l is the number of slots

Sect. 4.2), 2) adopting word-wise FHE (BFV scheme [34]) to shorten the latency of the LUT processing (described in Sect. 4.3), and 3) proposing a new technique to handle multi-input functions over two inputs (described in Sect. 4.3 to Sect. 4.5).

Table 2 compares previous LUT implementations with ours, where the semi-honest server follows the protocol but is curious about other parties' data. We set the plaintext space larger than the bit-length of inputs and outputs. Using packing technique [24], we consolidate and encrypt the number of l integers within a single ciphertext, where l is the number of slots, which enables a parallel computation in a SIMD manner. The complexity of single-input functions using the LUT is $O(2^{\sum_{i=1}^{m} d_i}/l)$ when utilizing all data points of LUTs ($|T_{in}| = |T_{out}| = 2^{\sum_{i=1}^{m} d_i}$). A drawback of our approach is the communication between the computation server and the trusted authority to match the function's input data and the data points in LUTs. Although the input data for the function and matched result remain concealed from the trusted authority by adopting PIR [28], the data point distribution in input LUTs and the index of matched data point (neither the input nor output value) will be revealed to the trusted authority. Further security analysis is provided in Sect. 4.2.3.

4.1 System Overview

The proposed method involves three parties as shown in Fig. 2: 1) a user responsible for sending input value(s) of a given function, 2) a computational server (CS) which can be a cloud server tasked with function evaluation, and 3) a trusted authority (TA) responsible for managing the FHE key pairs and operating as a decryption server without knowing raw input and output data. The user and the CS are assumed to be semi-honest, while the TA is assumed to be honest; the three parties are deemed not to collude.

During the initialization phase, the TA generates a set of FHE keys, including the public key (PK) used for encryption, the secret key (SK) used for decryption, the relinearization key (RK) used for reducing ciphertext size, and the Galois key (GK) used for slot rotation within a ciphertext. The TA retains the SK while sharing the RK and GK with the CS, and distributing the PK to the other two parties.

The pre-computed LUTs for a given function are encrypted and stored within the CS, where an LUT provider also retains the PK. Both the input and output LUTs contain

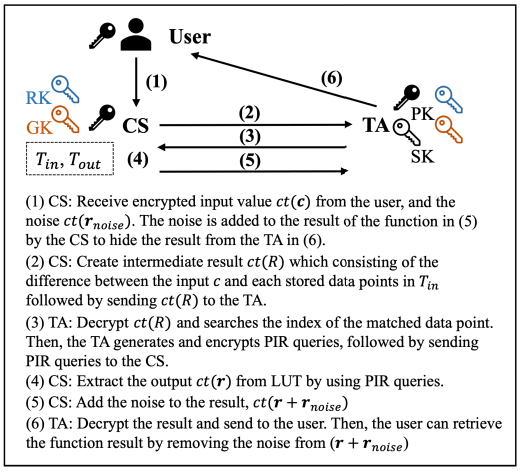

Fig. 2 The overview of our proposed system.

the input values and their corresponding outputs that satisfy the relationship $f(T_{in}(i)) = T_{out}(i)$, where i denotes the LUT index. The encrypted LUTs can be provided by users, the CS, or a third party, but not by the TA as the TA holds the secret key. The processing steps are shown in Fig. 2.

In the following, we explain 1) the preparation scheme of LUT data points, 2) the construction of LUTs, 3) the table separation scheme, and 4) the detailed LUT processing steps.

4.2 Preparation of LUT Data Points

We prepare a given function's data points, i.e., input values, before constructing the LUTs. First, we select the data points for the LUTs. Then, we convert all decimal points representing input values to integers because we adopt the BFV scheme [34] handing integers only. Finally, we add several redundant data points to the LUTs to prevent the TA from obtaining matched-index-related knowledge, such as statistics. The following subsections describe the three steps in detail.

4.2.1 Data Point Selection for LUTs

The techniques for LUT allocation, specifically data point selection techniques, have been researched extensively [32], [33] on the context that improves energy efficiency in computer systems by enabling approximate computation with LUTs. Tian et al. [32] and Raha et al. [33] proposed input-aware approximation techniques that assign more weight to frequently appearing inputs to reduce output errors while decreasing the number of LUT data points. Tian et al. [32] proposed a maximum error threshold to guarantee the precision of the function outputs. They then selected a predefined number of data points from the frequently appeared input data points, which can be adopted to our LUTs.

However, the data point selection technique is not the primary focus of this paper. Therefore, we employ the following two naive data point selection techniques in this work: 1) selecting data points of equal distance within the required

range (referred to as equidistant selection); 2) selecting data points from frequently appearing input data (referred to as input data-aware selection). For example, when using the equidistant selection method, we choose data points like $[0, 3, 6, 9, \ldots, 3n]$ with a gap of 3 between each consecutive data point. On the other hand, when applying the data-aware selection with five data points, we pick $[0, 6, 9, 12, 18]$ based on the top centroids 0, 6, 9, 12, and 18 that are frequently observed in the input data.

Note that the minimum and maximum data points within the plaintext space must be chosen as the boundaries. Without these boundary data points, we cannot accurately determine the nearest data point to the input, as described later in Step 3 of Sect. 4.5.

4.2.2 Decimal Points to Integers

We adopt the BFV scheme [34] in Microsoft SEAL [35] as word-wise FHE, which exclusively operates on integers. Thus, decimal points must be transformed into integers through scaling. We denote the scaling parameter as $\mathfrak{p}$. A decimal point value a is scaled to an integer using the formula $Round(\mathfrak{p} \times a)$. The highest achievable precision is determined by ensuring that the resulting integer remains. It is important to note that a larger plaintext space can handle numbers with greater precision. Our process involves initially rounding decimal points to the desired precision for scaling, followed by encryption.

For instance, given vector $\boldsymbol{x} = [2.345, 2.347, 2.521, 2.538, 3.124]$, after retaining 4 decimal places of precision, T_{in} becomes $T_{in} = [2345, 2347, 2521, 2538, 3124]$. When maintaining 3 decimal places of precision, T_{in} becomes $T_{in} = [235, 252, 254, 312]$, as both 2.345 and 2.347 are represented by the same integer after removing the last digit. With 2 decimal places of precision, T_{in} is reduced to $T_{in} = [23, 25, 31]$, resulting in lower precision but shorter runtime due to the decreased size of T_{in}.

4.2.3 Concealment of Matched Index from TA

Although the TA decrypts the intermediate result res sent from the CS, as shown in Fig. 2, the TA never knows the function itself, its input(s), or the function's outputs. This is because the res consists of a set of random-noise-added differences between the function's input values and the data points in T_{in} (as described in Step 5 of Sect. 4.5). However, the TA may know: 1) the index distribution of data points in LUTs and 2) the index of matched data point (neither the input nor output value). Although the intermediate result R is randomized with the noise, the TA will be able to infer the data point distribution of input LUTs by comparing the decrypted values in the R because the same noise value is added; however, the TA cannot know the values in LUTs. Another concern is that the TA will be able to know the statistics on how often a particular index of LUTs is selected.

Therefore, we introduce "redundant data points" to make it hard for the TA to infer the statistics of the selected index of LUTs. The CS maintains multiple versions of the LUT, each with distinct additional redundant data points. A distinct LUT is randomly selected for every input to provide LUT processing, making estimating the statistics hard. The more different versions of LUTs are prepared, the harder it is to infer the statistics of the selected index. In the following sections, we explain the concept of incorporating redundant data points into T_{in} with examples.

As described in Sect. 4.2.1, the data points in T_{in} are freely chosen to minimize output errors while simultaneously reducing the number of data points to accelerate LUT processing. For instance, techniques like equidistant or input data-aware selection might be employed. In simpler terms, these data selection methods can add redundant data points alongside the predefined data points. The inclusion of redundant data points alters the corresponding indices. Inspired by this concept, we propose to add random redundant data points into T_{in}.

We assume that the LUT provider adds redundant data points after establishing the predefined data points. Let us denote the number of data points as $|T_{in}|$ before adding redundant data points as num', and after their addition as num. We introduce $(num - num')$ redundant data points into T_{in}, each representing a random data point within the plaintext space but not previously included in T_{in}. Note that the corresponding T_{out} must also be updated under the new T_{in}.

For example, let the data points be $[-64, 0, 10, 20, 64]$, the number of the slot be 4, and the plaintext modulus be 129. Since the plaintext modulus is 129, we can encrypt the integer ranging from -64 to 64. Then, we have $T_{in}(0) = [-64, 0, 10, 20]$, $T_{in}(1) = [64, empty, empty, empty]$. In this example, we can add three redundant random values to T_{in} at most if we do not add other ciphertexts. The randomly added redundant data points must satisfy the range of plaintext modulus and all the data points in T_{in} must be in order. Assuming to add three redundant data points [−20, 30, 52], the new T_{in} becomes $T_{in}(0) = [-64, \mathit{-20}, 0, 10]$, and $T_{in}(1) = [20, \mathit{30}, \mathit{52}, 64]$, where italic data points are added. If we do not have any empty slots, we can add a new ciphertext. For example, let the predefined data points be [−64, 0, 10, 20, 30, 42, 55, 64] that is $T_{in}(0) = [-64, 0, 10, 20]$ and $T_{in}(1) = [30, 42, 55, 64]$. When adding four redundant data points $[-30, -20, -10, 33]$ to T_{in}, the new T_{in} becomes $T_{in}(0) = [-64, \mathit{-30}, \mathit{-20}, \mathit{-10}]$, $T_{in}(1) = [0, 10, 20, 30]$, and $T_{in}(2) = [\mathit{33}, 42, 55, 64]$. Note that increasing the number of ciphertexts, i.e., the number of rows of T_{in} or T_{out}, results in longer processing time; therefore, it is not recommended.

The drawback of this approach is that the party responsible for introducing redundant data points gains insight into both the function and the predefined data points. Therefore, this role should be fulfilled by the LUT provider, who supplies the LUTs to the CS. In this scenario, the LUT provider prepares a collection of LUTs that differ by incorporating redundant data points. Although this method increases security strength by generating many LUT variations, an exhaustive number of LUT preparations is unfeasible due to extensive storage requirements. For example, in our experi-

ment, each ciphertexts is approximately 262 KB and the LUT size in KB is $(\lceil 2^{m \cdot d}/l \rceil + \lceil 2^d/l \rceil) \times 262$ for d-bit m-input functions. We packed 2^{12} data points into one ciphertext. Examples of storage requirement are shown below.

- 32 MB corresponds with 2^{18} data points for an 18-bit single-input function, that 32 B $\approx (2^{18}/2^{12} + 2^{18}/2^{12}) \times$ 262 KB.
- 1.1 GB corresponds with 2^{24} data points for a 12-bit two-input function or 8-bit three-input function, that 1.1 GB $\approx (2^{12 \cdot 2}/2^{12} + 2^{12}/2^{12}) \times 262$ KB.

4.3 Construction of LUT

This subsection explains the construction method of LUTs. We first explain the single-input function case followed by the multi-input function case.

4.3.1 LUTs for Single-Input Functions

We construct input LUT T_{in} and output LUT T_{out} of a given single-input function $f(x)$. The T_{in} stores input data points for $f(x)$, while T_{out} stores corresponding outputs. All the data points in T_{in} are sorted in order.

In the following, $\boldsymbol{x}$ represents a vector that contains the prepared input data points for $f(\boldsymbol{x})$. In the example illustrated in Fig. 3, we define $\boldsymbol{x} := [-4, -3, -2, -1, 0, 1, 2, 4]$, where the plaintext space ranges from -4 to 4. The vector $f(\boldsymbol{x})$ represents the corresponding output data points for $\boldsymbol{x}$, denoted as $f(\boldsymbol{x}) = [f(-4), f(-3), f(-2), f(-1), f(0), f(1), f(2), f(4)]$. The straightforward construction of T_{in} and T_{out} is to meet the conditions $T_{in}(i) = \boldsymbol{x}(i)$ and $T_{out}(i) = f(\boldsymbol{x}(i))$ for all indices i that satisfy $0 \le i \le 7$.

As explained in Sect. 2.2, we accelerate LUT processing, which involves 1) searching for the matched element $\boldsymbol{x}(i)$ of the input in T_{in} and 2) extracting the corresponding output from T_{out}, by SIMD operations. In order to implement SIMD operations, we transform the LUTs into a two-dimensional format, as detailed below:

$$\begin{aligned} T_{in}(indIn_{row}, indIn_{col}) &= T_{in}(i) \\ T_{out}(indOut_{row}, indOut_{col}) &= T_{out}(i), \end{aligned} \tag{2}$$

where $indIn_{row} = indOut_{row} = \lfloor i/l \rfloor$ and $indIn_{col} = indOut_{col} = i \mod l$.

Referring to the example in Fig. 3, every row in T_{in} and T_{out} represents a ciphertext whose slot size is l. This structure facilitates column-wise addition and multiplication by slot-wise computation [24]. Additionally, we can leverage the multi-threading technique for row-wise parallelization.

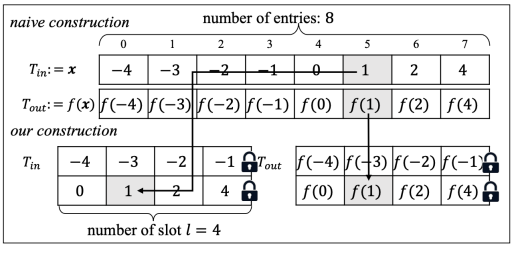

Fig. 3 An example of constructing LUTs for single-input function.

4.3.2 LUTs for Multi-Input Functions

In the case of a multi-input function, we consider m input vectors denoted as $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{m-1}$, along with a single output vector whose each element corresponds to $f(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{m-1})$. Regarding the LUTs, we prepare m-input LUTs denoted as T_{in}^j, where $0 \le j < m$. Additionally, there is one output LUT referred to as T_{out}, with a size of $|T_{out}| = \prod_{j=0}^{m-1} |T_{in}^j|$.

The index of corresponding output for the m-dimensional inputs $\boldsymbol{x}_0(i_0), \ldots, \boldsymbol{x}_{m-1}(i_{m-1})$ is $f(\boldsymbol{x}_0(i_0), \ldots, \boldsymbol{x}_{m-1}(i_{m-1}))$ whose index in T_{out} is $(indOut_{row}, indOut_{col})$, where $indOut_{row} = \lfloor ind_{out}/l \rfloor$ and $indOut_{col} = ind_{out} \mod l$, and

$$ind_{out} = \sum_{j=0}^{m-2} \left(\prod_{z=j+1}^{m-2} \left(|T_{in}^z| \right) \times i_j \right) + i_{m-1} \tag{3}$$

The LUTs construction example of a 6-bit 2-input function is shown in Fig. 4 and Fig. 5. In this example, the number of data points $|T_{in}| = 64$, and the number of slots $l = 8$, $T_{in}^0 := \boldsymbol{x}_0$ and $T_{in}^1 := \boldsymbol{x}_1$. $T_{in}^0(i_0)$ and $T_{in}^1(i_1)$ are transferred into two dimensional $T_{in}^0(indIn_{row}^0, indIn_{col}^0)$ and $T_{in}^1(indIn_{row}^1, indIn_{col}^1)$, respectively. When $i_0 = 30$ and $i_1 = 41$, we have $T_{in}^0(indIn_{row}^0, indIn_{col}^0) = (\lfloor 30/8 \rfloor, 30 \mod 8) = (3, 6)$ and, $T_{in}^1(indIn_{row}^1, indIn_{col}^1) = (\lfloor 41/8 \rfloor, 41 \mod 8) = (5, 1)$.

The output LUT holds $|T_{in}^0| \times |T_{in}^1| = 64 \times 64 = 4096$ corresponding data points. When $i_0 = 30$ and $i_1 = 41$, the index

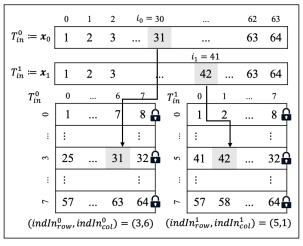

Fig. 4 An example of multi-input function's T_{in}.

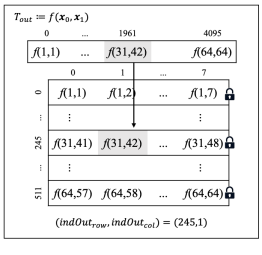

Fig. 5 An example of multi-input function's T_{out}.

of corresponding output is computed by Eq. (3) as $ind_{out} = 64 \times 30 + 41 = 1961$ and $T_{out}(indOut_{row}, indOut_{col}) = (\lfloor 1961/8 \rfloor, 1961 \mod 8) = (245, 1)$.

4.4 Table Separation

In order to shorten the runtime of LUT processing, we present our table separation technique [25]. In the BFV scheme [34] in Microsoft/SEAL [35], three primary parameters impact the runtime of FHE computations: the degree of polynomial modulus, the coefficient modulus, and the plaintext modulus (plaintext space). When a large plaintext space is needed to implement LUTs, meaning a large plaintext modulus, the consumption of the noise budget for each operation increases significantly, necessitating a greater noise budget for the initial ciphertext. Although it is possible to allocate more noise budget to the initially encrypted ciphertext using a larger coefficient modulus, this also considerably extends the runtime. Note that the polynomial modulus sets a limit on the maximum coefficient modulus, and a larger polynomial modulus supports more slots but leads to an increase in runtime.

Hence, reducing the plaintext space is essential to shorten the runtime of LUT processing. The subsequent table separation technique reduces the LUT's table size, contributing to the reduction of the plaintext space.

Let us assume we set the plaintext space as w-bit and decompose a u-bit large integer a to w-bit small integers $a_0, a_1, \ldots, a_{t-1}$, where $t = \lceil u/w \rceil$, shown in Eq. (4).

$$a = a_0 + a_1 \times 2^w + \ldots + a_{t-1} \times 2^{(t-1)w} \tag{4}$$

Each small integer is stored in the same index of the corresponding sub-table. For example, we can decompose the 6-bit integer 45 to three 2-bit integers such that $a_0 = 1, a_1 = 3, a_2 = 2$, i.e., $45 = 1 + 3 \times 2^2 + 2 \times 2^4$, and store in three sub-tables, i.e., L^{a_0}, L^{a_1}, and L^{a_2} separately.

We can apply the aforementioned technique to both the input and output LUTs; nevertheless, a limitation exists for the input LUT. The table separation technique can solely be applied to the input LUT only when full data points, specifically 2^u data points, are supplied.

4.5 LUT Processing with FHE

This subsection details the primary steps of our proposed LUT processing using FHE, facilitating the evaluation of arbitrary multi-input functions with both exact and nearby matching, where nearby matching returns the closest data point to the input while exact matching returns the matched data point.

The CS retains the encrypted LUTs and performs FHE-based LUT processing by communicating with the TA. The process consists of six key steps, enumerated as follows. (Each step is indicated by the corresponding number in parentheses in Fig. 2.)

Step 1: For a single-input function, the user sends both a ciphertext of the input $ct(\boldsymbol{c})$ and a ciphertext of the artificial noise $ct(\boldsymbol{r}_{noise})$ to the CS. Here, $ct(\boldsymbol{c}) = Enc([c, \ldots, c])$, where every element is filled with the same input value c for the function and $|\boldsymbol{c}| = l$, i.e., slot size of the ciphertext. The vector $ct(\boldsymbol{r}_{noise})$ is utilized to conceal the function's output from the TA in Step 6, where $|\boldsymbol{r}_{noise}| = l$ and every element is filled with different random value.

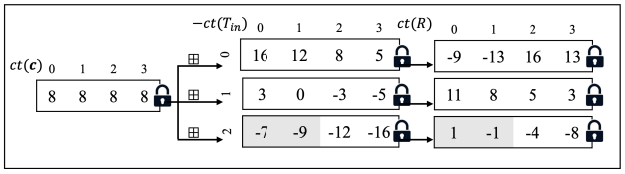

Fig. 6 Example of preparing intermediate result.

For an m-input function, the user sends m-fold inputs, denoted as $ct(\boldsymbol{c}_0), \ldots, ct(\boldsymbol{c}_{m-1})$, along with artificial noise $\boldsymbol{r}_{noise}$ to the CS, where $c_j (0 \le j < m-1)$ represents the j-th encrypted input vector of the m-input function, and $ct(\boldsymbol{c}_j) \leftarrow Enc([c_j, \ldots, c_j])$, with $|\boldsymbol{c}_j| = l$.

Step 2: The CS generates the intermediate result $ct(R^j)$ for input $ct(\boldsymbol{c}_j)$, and subsequently sends it to the TA. The intermediate result $ct(R^j)$ comprises the differences between each data point in T^j_{in} and the function's input $ct(\boldsymbol{c}_j)$. As shown in Fig. 6, the intermediate result $ct(R^j)$ is composed of the same number of ciphertexts as $ct(T^j_{in})$. Here, $ct(R^j(g))$ represents a single ciphertext consisting of l-fold data, where $0 \le g < \lceil |T^j_{in}|/l \rceil$.

$$ct(R^j(g)) = ct(\boldsymbol{c}_j) \boxplus -ct(T^j_{in}(g)) \tag{5}$$

The evaluation of Eq. (5) can be parallelized with g, i.e., ciphertext by ciphertext, by adopting multi-threading.

In the example shown in Fig. 6, we consider the inputs c as 8, the slot size l as 4, and $|T_{in}| = 12$. The encrypted input $ct(\boldsymbol{c})$ is a ciphertext that encrypts a vector with all elements set to the input value 8. Utilizing multi-threading, we simultaneously calculate the intermediate result $ct(R(g))$ for $0 \le g < 3$, running in parallel with g.

Step 3: The TA decrypts the received intermediate result $ct(R)$ and then identifies the index of the corresponding data point within $ct(T_{in})$. Note that for multi-input functions, this step is repeated for each input. Due to the TA's lack of knowledge about the input and the data points stored in $ct(T_{in})$, even after decrypting $ct(R)$, the matched data point is not revealed to the TA. The TA is only informed of the index of the matched data point. Furthermore, as elucidated in Sect. 4.2.3, since T_{in} incorporates distinct data points through the inclusion of redundant data points in every LUT processing, the TA cannot deduce the data points even upon observing a sequence of decrypted $ct(R)$ results.

When the function's input exactly matches one of the data points in T_{in}, the corresponding data point in R has a value of 0 by evaluating Eq. (5). If the TA is unable to identify a 0 value in R, the TA outputs the index with the smallest absolute value in R. This process is called a nearby search, which involves identifying the nearest data point. Notably, if there are two data points with the smallest

absolute values in R, one of them can be selected, as these two data points are equidistant from the function input value. For instance, in Fig. 6, the shaded data points $T_{in}(2,0)$ and $T_{in}(2,1)$ represent the smallest absolute values, both of which can be selected.

As elucidated in Sect. 4.2.1, T_{in} must contain the maximum and minimum data points of the plaintext space. Let us explain with an example. When assuming $T_{in} = [-2,-1,1,2]$ and input is -2 with the plaintext space ranging from -2 to 2, the resultant R is $[0,-1,2,1]$ after executing Eq. (5) with modulus calculations. As a result, the matched index is 0, because $R(0) = 0$. However, if the minimum data point is absent, and the data points in T_{in} is $[-1,1,2]$, we have R as $[-1,2,1]$. In this case, the two smallest absolute values exist at indices 0 and 2. However, the resultant index 2 is incorrect because $T_{in}(2) = 2$, which considerably deviates from the input value of -2. The rationale behind $R(2) = 1$ lies in the modulus calculation of polynomials. Consequently, including the maximum and minimum data points in T_{in} is essential to avert the side effect of modulus calculations that might hinder the identification of the nearest data point(s). The proof is provided below.

Theorem: Having both $-p$ and p data points is a necessary and sufficient condition for the smallest absolute value of the R to be the closest data point(s) to any of the inputs, where $[-p,p]$ is the plaintext space.

proof: Let c be an input value for a given function, d be a data point in LUT, where $c,d \in [-p,p]$. The distance between d and c is $|c-d| \le 2p$ because of $-2p \le c-d \le 2p$.

Sufficient condition: When $-p \le c-d \le p$, the $|c-d|$ is within the plaintext space, where $|\cdot|$ shows absolute value. Thus, the closest data point(s) to c is the data point(s) with the smallest $|c-d|$. On the other hand, when $p < c-d$ or $c-d < -p$, the $|c-d|$ does not show the real distance because of the modulus calculations, which are classified into the following two patterns:

1): When $c-d > p$, $|c-d|$ becomes $|c-d-(2p+1)|$ after modulus calculation. $\because d \in [-p,p], c-d > p \therefore$ the minimum $|c-d-(2p+1)|$ appears iff $d = -p$. Also, $\because c \le p$, $\therefore |c-p-1| = p-c+1$. Meanwhile, $|c-d| = |c-p| = p-c$ iff $d = p$. Thus, the distance between the data point p and c is smaller than that between $-p$ and c, and as a result, the closest data point never be the border that needs modulus calculation.

2): When $c-d < -p$, $|c-d|$ becomes $|c-d+(2p+1)|$ after modulus calculation. $\because d \in [-p,p], c-d < -p \therefore$ the minimum $|c-d+(2p+1)|$ appears iff $d = p$. Also, $\because -p \le c$, $\therefore |c+p+1| = c+p+1$. Meanwhile, $|c-d| = |c+p| = c+p$ iff $d = -p$. Thus, the distance between the data point $-p$ and c is smaller than that between p and c, and as a result, the closest data point never be the border that needs modulus calculation.

From 1) and 2), if we have both borders' data points, the closest data point(s) to any inputs is the data point which has the smallest absolute value in R. □

Necessary condition: The absolute value in R reflects the real distance between the data points in LUT with the input c iff it is calculated w/o modulus calculation. Thus, the closest data point must be the one that has a minimum distance w/o modulus calculation. Both border data points are needed because this can guarantee the existence of the distance w/o modulus calculation having a smaller distance than that with modulus calculation. Therefore, both borders' data points are needed to search for the closest data points to the input from the R. □

Algorithm 1: Searching for the matched data point to generate PIR queries

Input: m: the number of inputs, l: the number of slots,
R^j : m-fold decrypted intermediate result $0 \le j < m$,
k_{in}^j: the number of rows of T_{in}^j, $k_{in}^j = \lceil |T_{in}^j|/l \rceil$,
$N_q (= \lceil \log_l |T_{out}| \rceil)$: the number of PIR queries
Output: $ct(Q) := [ct(Q(0)), ..., ct(Q(N_q-1))]$: PIR queries
for $j = 0$ *to* $m-1$ **do**
  Find the smallest $abs(R^j(g,h))$ for $0 \le g \le k_{in}^j - 1$, $0 \le h \le l-1$;
  $(indIn_{row}^j, indIn_{col}^j) \leftarrow$ the index of $R^j(g,h)$;
end
Compute $indOut_{col}, ind_{out}$ by Eq. (6);
$Q \leftarrow [][]$; ▷create an empty matrix
Generate PIR query $Q(0)$ of length l in which only the $indOut_{col}$-th element is 1 and other elements are 0;
for $i = 1$ *to* $N_q - 1$ **do**
  $indQ_i \leftarrow \lfloor ind_{out}/l^i \rfloor \mod l$;
  $Q(i) \leftarrow LeftRotate(Q(i-1), indQ_i)$;
  ▷left rotate $Q(i-1)$ by $indQ_i$ elements
  $ct(Q(i)) \leftarrow Enc(Q(i))$;
end
return $ct(Q) := [ct(Q(0)), ..., ct(Q(N_q-1))]$

The Algorithm 1 shows how to search for the matched index in T_{in} and construct PIR queries. In Algorithm 1, the matched index for the j-th input of the m-input function is denoted as $indIn_{row}^j$ and $indIn_{col}^j$, where $0 \le j < m$. The index of the output LUT T_{out} that corresponds to the function's input, specified as $indOut_{row}$ and $indOut_{col}$, is determined through the following calculation.

$$\begin{aligned} indOut_{row} &= \lfloor ind_{out}/l \rfloor \\ indOut_{col} &= ind_{out} \mod l, \end{aligned} \tag{6}$$

where $ind_{out} = \sum_{j=0}^{m-2}(\prod_{z=j+1}^{m-2}(|T_{in}^z|) \times (indIn_{row}^j \times l + indIn_{col}^j)) + (indIn_{row}^{m-1} \times l + indIn_{col}^{m-1})$. Once the matched index in T_{out} is computed, the TA proceeds to generate PIR queries and send them back to the CS, effectively concealing the matched index in T_{out} from the CS. The number of PIR queries denoted as N_q is calculated by the formula $N_q = \lceil \log_l |T_{out}| \rceil$. Subsequently, the TA employs Algorithm 1 to produce the encrypted PIR queries, expressed as $ct(Q) := [ct(Q(0)), \ldots, ct(Q(N_q-1))]$.

Figure 7 shows an example of generating PIR queries for a single-input function. In this example, the absolute value of 1 is the smallest, whose index $(indOut_{row}, indOut_{col})$ is $(2,0)$ or $(2,1)$. As these two data points have the same distance from the function input value, we select $(2,0)$ this

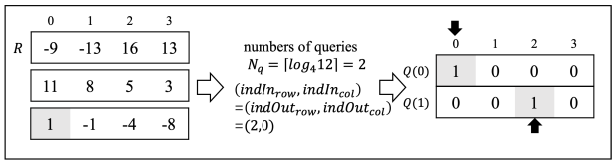

Fig. 7 Example of generating PIR queries.

Algorithm 2: Reconstructing the PIR queries

Input: $ct(Q)$: PIR queries, N_q: the number of PIR queries, k_{out}: the number of rows of T_{out}, where $k_{out} = \lceil |T_{out}|/l \rceil$
Output: $\boldsymbol{q}_{last}$: a $ct(T_{out})$-style ciphertext vector $\boldsymbol{q}_{last}$ whose element indicated by $ct(Q)$ is one and others are zero

```
q_last ← [];        ▷create an empty ciphertext vector
n ← N_q − 1;        ▷# of the remained PIR queries
q' ← ct(Q(n));      ▷the target PIR query
Function ReconstQ(n, q', q_last, k_out, ct(Q)):
    if n = 0 then
        append q' to q_last until |q_last| = k_out;
        return q_last;
    else
        temp ← q';      ▷save q' for recursive call
        for i = 0 to l − 1 do
            using rotation operation to right rotate q' by i
              slots and multiple to ct(Q(n − 1));
            q_last ← ReconsQ(n, q', q_last, k_out, ct(Q));
            q' ← temp;                  ▷restore q'
        end
    end
End Function
```

Algorithm 3: Extracting the output from LUT

Input: $ct(T_{out})$: encrypted output LUT, k_{out}: the number of rows of T_{out}, where $k_{out} = \lceil |T_{out}|/l \rceil$, $\boldsymbol{q}_{last}$: reconstructed T_{out}-style PIR queries
Output: $ct(\boldsymbol{r})$: a ciphertext of the output value

```
r' ← [];        ▷create an empty ciphertexts vector
for i = 0 to k_out − 1 do
    r'.append(q_last(i) ⊡ ct(T_out(i)));
end
ct(r) ← Σ_{i=0}^{|r'|−1} r'(i); ▷store the output into a single
  ciphertext
ct(r) ← totalSums(ct(r)); ▷fill in every element of
  ct(r) with the function result (note, totalSums
  calculates the summation of all elements
  followed by filling it in every element)
return ct(r);
```

time. The number of PIR queries is calculated by $N_q = \lceil \log_4 12 \rceil = 2$, i.e., $Q(0)$ and $Q(1)$. The $Q(0)$ is a vector in which the $indOut_{col}$-th element is 1 and others are 0. The $Q(1)$ is a rotated vector that left rotate $indQ_1$ elements of $Q(0)$, where $indQ_1 = \lfloor (2 \times 4 + 0)/4 \rfloor \mod 4 = 2$.

Step 4: The CS receives the PIR queries from the TA and creates a $ct(T_{out})$-style ciphertexts vector $\boldsymbol{q}_{last}$ in whose element indicated by $ct(Q)$ is one and others are zero by Algorithm 2. Then, the CS proceeds to extract the matched data point in T_{out} as $ct(\boldsymbol{r})$ by Algorithm 3.

Step 5: The CS adds the noise $ct(\boldsymbol{r}_{noise})$ to the function's output $ct(\boldsymbol{r})$ to create $ct(\boldsymbol{r} + \boldsymbol{r}_{noise})$. Subsequently, the CS sends $ct(\boldsymbol{r} + \boldsymbol{r}_{noise})$ to the TA.

Table 3 Computational complexity.

	Steps	Complexity	
CS	1) Traverse T_{in}	$O(m \cdot 2^d/l)$	$O(2^{m \cdot d}/l)$
	2) Extraction from T_{out}	$O(2^{m \cdot d}/l)$	
	3) Reconstructing PIR queries	$O(2^{m \cdot d}/l)$	
	4) Adding noise	$O(1)$	
TA	1) Decrypting $ct(R)$	$O(m \cdot 2^d/l)$	ciphertext: $O(m \cdot 2^d/l)$
	2) Encrypting $ct(Q)$	$O(N_q)$	
	3) Traverse R	$O(2^d \cdot m)$	plaintext: $O(2^d \cdot m)$
	4) Generate Q	$O(N_q \cdot l)$	

Step 6: The TA decrypts the received $ct(\boldsymbol{r} + \boldsymbol{r}_{noise})$ and then forwards $\boldsymbol{r} + \boldsymbol{r}_{noise}$ to the user. In the final step, the user obtains the function's result by subtracting $\boldsymbol{r} + \boldsymbol{r}_{noise}$ from the received data.

5. Complexity Analysis

This section shows the computational complexity of our proposed LUT processing. We assume to use the LUTs with fully occupied data points satisfying $|T_{out}| = 2^{m \cdot d}$ for a d-bit m-input function. We pack and encrypt l data points into a single ciphertext, then the number of PIR queries is $N_q = \lceil \log_l 2^{m \cdot d} \rceil$ (see Step 3 of Sect. 4.5).

First, the computational complexity in the CS is calculated as follows:

1) Searching for the matched input in T_{in} needs $O(m \cdot 2^d/l)$ (see Step 2 of Sect. 4.5).

2) Extracting the matched data point from T_{out} needs $O(2^{m \cdot d}/l)$ (Algorithm 3).

3) Algorithm 2 needs $O(2^{m \cdot d}/l)$ for reconstructing the PIR queries by calling the function `ReconstQ` $(N_q - 1)$ times, each of which needs $\lceil 2^{m \cdot d}/l \rceil + \log_2 l + \sum_{i=1}^{N_q - 2} l^i$ times rotations and $\lceil 2^{m \cdot d}/l \rceil \cdot 2 + \sum_{i=1}^{N_q - 2} l^i$ times multiplications.

4) Adding the noise to the output needs $O(1)$ because of one time addition (see Step 5 of Sect. 4.5).

Since $O(2^{m \cdot d}/l)$ is the largest term and has the highest magnitude among all operations, the computational complexity over ciphertext is defined as $O(2^{m \cdot d}/l)$.

Second, the computational complexity in the TA is calculated as follows (see Step 3 of Sect. 4.5 and Algorithm 1):

1) Decrypting $ct(R)$ needs $O(m \cdot 2^d/l)$.

2) Encrypting the PIR queries $ct(Q)$ needs $O(N_q)$,

3) Searching for the matched index over the matrix R with plaintext requires $O(2^d \cdot m)$ because of liner search.

4) Generating PIR queries matrix Q needs $O(N_q \cdot l)$.

Since $m \cdot 2^d/l > N_q$ and $2^d \cdot m > N_q \cdot l$, the computational complexity over ciphertext and plaintext are defined as $O(m \cdot 2^d/l)$ and $O(2^d \cdot m)$, respectively.

The computational complexity is summarized in Table 3.

6. Experimental Evaluation

This section evaluates our proposed method by conducting the following four experiments. (The source code is available at: https://github.com/ruixiaoLee/lut$_s$eal$_s$imulation.)

Table 4 The parameters of FHE (BFV) setting for the proposed method.

Degree of polynomial modulus	8,192
Coefficient modulus	160 (50+30+30+50) bits
Plaintext modulus	786,433
Level	2

1) The first experiment measures the error of Swish and ReLU functions, which are common activation functions used in machine learning, as the number of LUT data points varies. We employ two simple data point selection techniques, equidistant- and population-based, to demonstrate the highly accurate outcomes of the LUT-based approach.

2) The second experiment measures the maximum overhead of d-bit single-input functions with LUTs consisting of 2^d data points. This experiment aims to confirm the upper bounds of latency, memory usage, and communication costs associated with our proposed method. Two specific fixed values are denoted: d as the bit-length of input data points, and s as the bit-length of output data points. Throughout all the experiments, we maintain the condition $d = s$ and use d to represent the bit-length of both input and output.

3) The third experiment measures the maximum overhead of d-bit input and output of two and three-input functions with our proposed method by varying the number of bits of data points $2^{m \cdot d}$ to handle, where m is the number of inputs, confirming the adaptability of our proposed method for multi-input functions.

4) The fourth experiment compares our proposed method and the polynomial approximation method [14] using the CKKS scheme in the Microsoft SEAL library. This comparison evaluates the accuracy and runtime when implementing Swish and ReLU functions. Furthermore, we compare our proposed method and the naive bit-wise LUT method, implemented using the OpenFHE library due to the unavailability of bit-wise encoding FHE in the Microsoft SEAL library.

Our proposed methods and the baselines were implemented and evaluated on the computer explained below. We used one machine to emulate all the parties, i.e., a user, the CS, and the TA. Our machine was equipped with four Intel(R) Xeon(R) E7-8880 v3 @ 2.30GHz CPUs with 3 TB main memory, with CentOS Linux 7 (Core) x86-64 OS running. We used gcc 9.3.1, CMake 3.14.3, and OpenMP 3.1, the OpenMP was used for multi-thread operations. The proposed method was implemented using Microsoft/SEAL[†] library v4.0.0. The parameters used in experiments 2) and 3) are shown in Table 4. It is important to note that we only use half of the slots in our experiments.

6.1 Error Evaluation with Different Numbers of LUT Data Points

We evaluate the error of Swish and ReLU functions by varying the numbers of LUT data points to verify the extent to which function outputs exhibit minimal deviations from the actual calculation results. The data points are selected using two techniques: 1) selecting data points of equal distance within the required range (referred to as equidistant selection) and 2) selecting data points from frequently encountered input data (referred to as input data-aware selection).

[†] https://github.com/microsoft/SEAL

Table 5 The average error with different numbers of LUT data points.

Selection Method	#	Swish		ReLU	
		Err.[%]	Abs. Err.	Err.[%]	Abs. Err.
Equidistant Selection	2^6	55.13	2.70e-2	28.52	2.62e-2
	2^8	12.15	6.59e-3	6.35	6.41e-3
	2^{10}	2.69	1.64e-3	1.39	1.59e-3
	2^{12}	6.15e-1	4.11e-4	3.22e-01	4.00e-4
	2^{14}	1.35e-1	1.03e-4	6.71e-02	9.98e-5
Input data-aware Selection	2^6	10.12	1.08e-2	4.52	1.05e-2
	2^8	2.45	2.73e-3	1.05	2.58e-3
	2^{10}	8.33e-1	6.83e-4	4.09e-1	6.52e-4
	2^{12}	1.85e-1	1.76e-4	8.82e-2	1.67e-4
	2^{14}	4.48e-2	5.47e-5	2.24e-2	5.13e-5

#: the number of data points in LUT

We prepared a historical input dataset that follows a standard normal distribution, with a mean of zero, comprising 100,000 data. Out of these, 80% of the items were allocated as a training dataset (80,000 data) for fine-tuning the data points of input data-aware selection. The remaining 20% were designated as a test dataset (20,000 data) for calculating the error. For our evaluation, we established the data point range from -6.5536 to 6.5535, assuming that the LUT without selection consists of $2^{17} = 131,072$ data points spanning from $-65,536$ to $65,535$, with a scale parameter $\mathfrak{p} = 10,000$.

The average absolute error and error percentage are calculated by Eq. (7).

$$
\begin{aligned}
Avg.Abs.Err. &= avg(\sum_{i=1}^{N} abs(r^i_{lut} - r^i_{real})) \\
Avg.Per.Err. &= avg(\sum_{i=1}^{N} abs(\frac{r^i_{lut} - r^i_{real}}{r^i_{real}}) \times 100),
\end{aligned}
\tag{7}
$$

where N is the number of test data, r_{lut} is the output value using LUT, and r_{real} is the actual calculation result.

Table 5 shows the result. We confirm that the calculation error decreases as the number of data points increases. Besides, the LUT with input data-aware selection has better accuracy than the LUT with equidistant selection.

6.2 Runtime and Communication Cost Evaluation for Single-Input Function

We evaluate the overhead of d-bit single-input functions, where the LUTs contain a total of 2^d data points. In this experiment, we investigate the time consumption, memory usage, and communication cost of our proposed FHE-based method. With the FHE parameters shown in Table 4, the size of a single ciphertext is approximately 262 KB. We measure the overhead by varying the bit-length of both the input and

Table 6 The runtime of single-input function in each step.

bit-length d (# of cts*)	Steps	Runtime [s]		
		4-thread	2-thread	1-thread
18-bit (64)	Step1	1.67e-2		
	Step2	1.14e-2	1.77e-2	2.30e-2
	Step3	6.91e-2	1.20e-1	1.38e-1
	Step4-5	1.04	1.80	2.96
	Step6	5.63e-3		
	SUM of Step2-5	1.13	1.94	3.12
16-bit (16)	Step1	1.67e-2		
	Step2	3.40e-3	5.16e-3	5.68e-3
	Step3	3.06e-2	4.37e-2	4.61e-2
	Step4-5	3.63e-1	5.86e-1	8.23e-1
	Step6	5.63e-3		
	SUM of Step2-5	3.97e-1	6.35e-1	8.75e-1
14-bit (4)	Step1	1.67e-2		
	Step2	1.43e-3	1.44e-3	8.88e-4
	Step3	2.09e-2	2.36e-2	2.51e-2
	Step4-5	1.77e-1	2.32e-1	2.73e-1
	Step6	5.63e-3		
	SUM of Step2-5	2.00e-1	2.57e-1	2.99e-1
12-bit (1)	Step1	1.67e-2		
	Step2	3.68e-4	3.41e-4	2.62e-4
	Step3	1.10e-2	1.09e-2	1.10e-2
	Step4-5	8.69e-2	8.71e-2	8.69e-2
	Step6	5.63e-3		
	SUM of Step2-5	9.83e-2	9.84e-2	9.81e-2

* the number of ciphertexts in T_{in}, which is the same as T_{out}

Table 7 The main memory consumption in each step.

bit-length d (# of cts*)	Steps	Memory usage [MB]		
		4-thread	2-thread	1-thread
16-bit (64)	Step1	33.54		
	Step2	299.04	177.54	122.03
	Step3	267.02	134.63	71.44
	Step4-5	309.87	184.67	131.64
	Step6	31.25		
14-bit (16)	Step1	33.54		
	Step2	279.54	150.53	84.53
	Step3	240.48	109.03	43.34
	Step4-5	283.36	158.54	97.34
	Step6	31.25		
14-bit (4)	Step1	33.54		
	Step2	272.54	140.54	74.78
	Step3	236.34	103.58	37.82
	Step4-5	279.41	148.54	86.23
	Step6	31.25		
12-bit (1)	Step1	33.54		
	Step2	80.54	76.54	74.53
	Step3	39.10	35.09	33.08
	Step4-5	88.73	84.72	82.72
	Step6	31.25		

* the number of ciphertexts in T_{in}, which is same as T_{out}

Table 8 The communication cost between the CS and the TA.

bit-length d	18-bit	16-bit	14-bit	12-bit
# of cts*	64+2	16+2	4+2	1+1
CS to TA	17 MB	4.1 MB	1.1 MB	262 KB
TA to CS	520 KB			262 KB
Est. time (under 100 Mbps)	1.400 s	0.369 s	0.129 s	0.041 s

*: the number of ciphertexts of (intermediate results + PIR queries)

output data points to 18-bit, 16-bit, 14-bit, and 12-bit, using one, two, and four threads for evaluation. Since we maintain the same parameters in this experiment, the number of slots remains a constant value. The number of ciphertexts in T_{in} and T_{out} is shown in Table 6 and Table 7.

Table 6 shows the runtime for each step described in Sect. 4.5, varying the length of input bits. The runtime from steps 2 to 5 in Table 6 shows the latency of the function evaluation after receiving the input value $ct(\boldsymbol{c})$ to outputting the result $ct(\boldsymbol{r} + \boldsymbol{r}_{noise})$. Table 7 shows the memory consumption in each step, and Table 8 shows the transferred data size between the CS and the TA from Step 2 to 4 for handling the intermediate result and PIR query. Table 8 shows the simulated data transferring time between the CS and the TA under an ideal 100 Mbps net-connection environment.

As shown in Tables 6 and 8, we can evaluate the 18-bit function within 4.52 s (3.12+1.40) by one thread and 2.53 s (1.13+1.40) by four threads. The runtime increases with LUT size (the number of ciphertexts in LUT). The multi-thread implementation reduced the runtime by one-third by four threads but increased memory consumption by 2.4 times. The largest memory consumption in this experiment was 309.87 MB when evaluating an 18-bit function using four threads.

6.3 Runtime and Communication Cost Evaluation for Multi-Input Function

We evaluate two and three-input functions using our proposed method, varying the number of bits for both input and output data points. This process helps to confirm the adaptability of our proposed method across various functions.

We employed the same FHE parameters as presented in Table 4. Table 9 shows the overall runtime from Step 2 to 5 under different thread counts with the time for communication. In this experiment, we evaluated two-input functions ranging from 4 to 12 bits and three-input functions ranging from 4 to 8 bits.

The runtime increases rapidly as the number of inputs for the function increases, primarily because the size of the output LUT expands exponentially with the number of inputs. In this experiment, the number of data points of output LUT is $2^{m \cdot d}$, where the number of data points of input LUT is 2^d in case preparing all data points that can be expressed in d-bit; m shows the number of inputs for the function. The result shows the worst runtime when evaluating d-bit functions because we prepare all data points in the LUT. For instance, for 12-bit two-input functions and 8-bit three-input functions, the maximum number of output data points is $2^{24} = 16,777,216$.

As demonstrated in Table 9, we can evaluate an arbitrary 12-bit two-input function or 8-bit three-input function within 208.5 s using one thread, and within 17 s using 16 threads, which is approximately 12 times faster. Evaluating 6- and 4-bit two-input functions, or 4-bit three-input functions, required 0.2 s. Note that the runtimes of 6- or 4-bit two-input functions and 4-bit three-input functions were not impacted by an increase in the number of threads, as each input LUT and output LUT both consist of a single ciphertext,

Table 9 The runtime of m-input functions.

m	bit-length d (# of cts*)	Runtime[s] 16-thread	4-thread	1-thread	Est. comm. cost (under 100Mbps)
2	12 (4, 096)	16.77	56.66	206.83	0.081 s
	10 (256)	1.22	3.59	12.10	
	8 (16)	0.21	0.39	0.84	
	6 (1)	0.11			0.061 s
	4 (1)				
3	8 (4, 096)	16.84	57.79	208.31	0.102 s
	6 (64)	0.42	1.07	3.03	
	4 (1)	0.11			0.082 s

* the number of ciphertexts in T_{out}, every T_{in} in this experiment fits to one ciphertext

which prevents parallelization.

6.4 Comparison with Related Work

The fourth experiment compares the accuracy and runtime of our proposed LUT-based method with the polynomial approximation methods [14] when implementing Swish and ReLU functions. Additionally, we provide a brief comparison of the runtime between our proposed LUT-based method and the naive bit-wise LUT method.

1) Error and Runtime Comparison with the Polynomial Approximation Method

We implemented the polynomial approximation and our proposed methods using the CKKS and the BFV schemes in the Microsoft SEAL library, respectively. For the polynomial approximation, we use the polyfit[†] function from NumPy in Python and show the results in Table 10. The approximation range used was chosen as $[-3, 3]$. We omitted terms with coefficients less than 10^{-5}. We used the same test dataset and LUTs in Sect. 6.1. The average absolute error is defined by Eq. (7). The results and the parameters are shown in Table 11.

As shown in Table 11, the runtime of our proposed method is the same when the number of data points is smaller than 2^{12} because one row, i.e., one ciphertext, can store up to 2^{12} data points so that one row is enough to implement. The absolute error on average of the output value for the Swish (ReLU) function is $4.81 \times 10^{-4}(2.77 \times 10^{-2})$ when using the degree-8 polynomial approximation, which requires 115 (185) ms. Meanwhile, our proposed LUT method with input data-aware selection can achieve $3.28 \times 10^{-4}(2.92 \times 10^{-4})$ computation error on average with approximately 95 (95) ms including communication time. Thus, we confirmed that our proposed method achieves higher accuracy with a shorter runtime compared to the polynomial approximation method.

Note that the polynomial approximation method suits batch-style function evaluations because the polynomial approximation method can evaluate the number of l inputs simultaneously using the SIMD technique. On the contrary, our proposed method already adopted the SIMD technique for handling LUT-based function evaluation; thus, our method cannot fit batch-style function evaluations.

[†] https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html

Table 10 Polynomial approximation result.

Function	Degree	Polynomial Approximation
ReLU	2	$0.1563x^2 + 0.5003x + 0.2812$
	4	$-0.01519x^4 - 0.0001013x^3 + 0.2734x^2 + 0.5004x + 0.1758$
	6	$0.003017x^6 + 0.00003482x^5 - 0.05222x^4 - 0.002848x^3 + 0.3845x^2 + 0.5004x + 0.1282$
	8	$-0.0007634x^8 - 0.00001257x^7 + 0.01584x^6 + 0.0001584x^5 - 0.1188x^4 - 0.0005484x^3 + 0.4935x^2 + 0.5004x + 0.1009$
Swish	2	$0.1576x^2 + 0.5002x + 0.06811$
	4	$-0.00826x^4 - 0.00002634x^3 + 0.2213x^2 + 0.5001x + 0.01076$
	6	$0.0005721x^6 - 0.01528x^4 - 0.00002186x^3 + 0.2424x^2 + 0.5x + 0.001729$
	8	$-0.00004065x^8 + 0.001255x^6 - 0.01883x^4 - 0.00001078x^3 + 0.2482x^2 + 0.5x + 0.0002781$

2) Runtime Comparison with the Bit-Wise LUT Method

In this section, we compare the runtime of the bit-wise LUT method with that of our proposed word-wise LUT method. First, the runtimes of the bit-wise LUT method proposed in previous papers are reviewed, and then the results of the actual implementation are presented.

Carpov et al. [19] showed that the latency of 6-to-6-bit functions was under 1.6 s. Chillotti et al. [18] evaluated 8-to-8-bit and 16-to-8-bit functions spent approximately 1.1 s and 2.2 s, respectively. Lu et al. [22] reported 0.97 s for 7-bit functions. Since these results are evaluated in different environments, they are not directly comparable; however, compared to our results in Tables 6 and 8, our method can evaluate a 12-bit function in 0.14 s by one thread, which shows the superiority of our method.

To confirm our superiority in detail, we implement the naive bit-wise LUT method as a reference. Since the Microsoft SEAL library does not support bit-wise encoding FHE, we adopt OpenFHE[††] library v1.0.3. The security level is set as `STD128` and other parameters are left as the default. We also set the bit-length of input and output to be the same, represented as $d(= s)$.

To evaluate m-input d-to-s-bit function, we combine the inputs as a single input whose bit-length is $m \cdot d$. The encrypted vector of input bits is $\boldsymbol{q} = [ct(q_0), \ldots, ct(q_{m \cdot d-1})]$. The T_{in} and T_{out} are matrices, each of whose elements are encrypted bits, where the number of data points is $2^{m \cdot d}$, bit-length of input and output are $m \cdot d$ and s, respectively. The encrypted vector of result $\boldsymbol{r} = [ct(r_0), \ldots, ct(r_{s-1})]$ is computed as follows.

$$ct(r_i) = \sum_{j=1}^{2^{m \cdot d}} \left(\prod_{k=1}^{m \cdot d} \left(ct(q_k) \boxdot T_{in}(j,k) \boxdot ct(1) \right) \boxdot T_{out}(j,i) \right), \tag{8}$$

where $0 \leq i \leq s$.

The results with 4-thread are shown in Table 12. Our

[††] https://github.com/openfheorg/openfhe-development

Table 11 Runtime comparison between polynomial approximation and our proposed method.

Method	Parameters				Conditions	Swish		ReLU	
	Degree of poly. modulus	Scale or pt modulus	Coefficient modulus [bit]	Initial level		runtime	Abs. Err.	runtime	Abs. Err.
Polynomial approximation	16,384	30 bits	220 (50+30+30 +30+30+50)	4	degree-8	115 ms	4.81e-4	185 ms	2.77e-2
			190 (50+30+30 +30+50)	3	degree-6	74 ms	1.21e-3	89 ms	3.70e-2
					degree-4	60 ms	6.96e-3	60 ms	5.66e-2
			160 (50+30+30+50)	2	degree-2	22 ms	4.38e-2	22 ms	1.06e-1
Proposed LUT-based method	8,192	786,433	130 (50+30+50)	1	EDS # $=2^{12}$	95 ms	7.91e-4	95 ms	7.50e-4
					IDAS # $=2^{12}$		3.28e-4		2.92e-4
					EDS # $=2^{8}$		1.33e-2		1.31e-2
					IDAS # $=2^{8}$		5.47e-3		5.15e-3

pt: plaintext, EDS: Equidistant data selection, IDAS: Input data-aware selection, #: number of data points

Table 12 Runtime comparison in seconds [s] of m-input functions between naive bit-wise LUT and our proposed method (using 4-thread).

m	bit-length d	Naive bit-wise method	Proposed method
1	3	33.954	1.39e-1
	2	11.636	
	1	4.295	
2	3	377.912	0.169
	2	61.580	
	1	8.032	
3	3	3723.015	0.193
	2	305.476	
	1	20.817	

proposed method showed the same runtime regardless of the input bit-length because the proposed method uses the same number of ciphertexts even if the bit-length changes. As shown in Table 2, if 2^d does not exceed slot size, the proposed method's runtime stays the same. On the contrary, the naive bit-wise LUT increases its runtime with the increase of the bit-length because the runtime is proportional to $O(s \cdot 2^{m \cdot d})$ as shown in Table 2.

When the number of inputs increases, our proposed method's runtime increases as the output LUT size increases (when the N_q remains unchanged). As for the naive bit-wise implementation, the runtime increases with the same reason when the bit-length increases.

For the 1-bit one-, two-, and three-input functions, our method decreases the runtime to approximately 30.9, 47.5, and 107.9 times compared to the naive bit-wise LUT implementation. Note that we used different FHE libraries to implement the two methods; thus, we cannot compare the runtime directly. However, we could confirm how the runtime increases in each method.

7. Discussion

Our proposed method needs the TA to search for the matched data point while hiding input and output information from the TA. However, this introduces additional communication overhead between the CS and the TA. The overhead, as shown in Table 8, is not significant; however, it is important to note that the TA must remain online. Besides, as we discussed in Sect. 4.2.3, the preparation of multiple versions of LUTs makes it hard for the TA to infer the statistics of the selected index of LUTs. However, even if we prepare many versions, 1) the index distribution of data points in LUTs and 2) the index of matched data point (neither the input nor output value) is leaked to the TA, limiting our proposed method.

To circumvent the need for a TA, an alternative approach is to employ a trusted execution environment (TEE) like Intel SGX [37] and AMD SEV [38]. For instance, Xiao et al. [39] and Takeshita et al. [40] employed the TEE of Intel SGX to manage FHE key pairs, combining plaintext execution in the TEE with ciphertext execution using FHE in the rich execution environment (REE). Yakupoglu et al. [41] developed three secure multi-party computation (SMPC) protocols leveraging the TEE of Intel SGX and advanced encryption standards (AES). Although TEEs have limited memory sizes, implementing the TA's execution within a TEE could be feasible since the decryption of intermediate data and the subsequent search for the minimal absolute value to generate PIR queries do not necessitate large memory resources.

8. Conclusion

This paper introduces a privacy-preserving function evaluation method using Look-Up Tables (LUTs) with word-wise-based Fully Homomorphic Encryption (FHE). Our proposed function evaluation approach offers flexibility in balancing accuracy and runtime by adjusting the LUT size to suit specific application requirements. Experimental results demonstrated that we were able to evaluate an 18-bit (12-bit) single-input function within 2.53 (0.14) s and a 6-bit two-input function within 0.17 s using the proposed method, inclusive of communication time. These evaluations were conducted using four threads. Additionally, our method produced more accurate outputs with shorter runtimes than the polynomial approximation method. By incorporating existing LUT data point selection techniques, it is possible to enhance accuracy further using smaller LUT sizes, contributing to reduced runtimes. Our future work involves exploring the combination of FHE and Trusted Execution Environments (TEEs) to further shorten the runtime.

References

[1] D. Zissis and D. Lekkas, "Addressing cloud computing security issues," Future Generation Computer Systems, vol.28, no.3, pp.583–

592, 2012.
[2] C.Z. Gao, Q. Cheng, P. He, W. Susilo, and J. Li, "Privacy-preserving Naive Bayes classifiers secure against the substitution-then-comparison attack," Information Sciences, vol.444, pp.72–88, 2018.
[3] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: An efficient and scalable protocol," Proc. 2013 ACM SIGSAC Conf. on Comput. & Commun. Security, pp.789–800, 2013.
[4] P. Rindal and M. Rosulek, "Improved private set intersection against malicious adversaries," Advances in Cryptology–EUROCRYPT'17, LNCS, vol.10210, pp.235–259, 2017.
[5] M. Aziz, D. Alhadidi, and N. Mohammed, "Secure approximation of edit distance on genomic data," BMC Med Genomics, vol.10, no.2, no.41, 2017. https://doi.org/10.1186/s12920-017-0279-9
[6] Y. Li, D. Yang, and X. Hu, "A differential privacy-based privacy-preserving data publishing algorithm for transit smart card data," Trans. Research Part C: Emerging Technologies, vol.115, no.102634, pp.1–21, 2020.
[7] J. Wei, Y. Lin, X. Yao, J. Zhang, and X. Liu, "Differential privacy-based genetic matching in personalized medicine," IEEE Trans. Emerg. Topics Comput., vol.9, no.3, pp.1109–1125, 2021.
[8] Y. Wang, Z. Huang, S. Mitra, and G.E. Dullerud, "Differential privacy in linear distributed control systems: Entropy minimizing mechanisms and performance tradeoffs," IEEE Trans. Control. Netw. Syst., vol.4, no.1, pp.118–130, 2017.
[9] Y. Lu, X. Huang, Y. Dai, S. Maharjan, and Y. Zhang, "Blockchain and federated learning for privacy-preserved data sharing in industrial IoT," IEEE Trans. Ind. Informat., vol.16, no.6, pp.4177–4186, 2019.
[10] J. Wei, Y. Lin, X. Yao, and V.K.A. Sandor, "Differential privacy-based trajectory community recommendation in social network," J. Parallel and Distrib. Comput., vol.133, pp.136–148, 2019.
[11] B. Jiang, J. Li, G. Yue, and H. Song, "Differential privacy for industrial internet of things: Opportunities, applications, and challenges," IEEE Internet Things J., vol.8, no.13, pp.10430–10451, 2021.
[12] C. Gentry, "Fully homomorphic encryption using ideal lattices," Proc. 41st ann. ACM Symp. on Theory of Comput., pp.169–178, May 2009.
[13] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. Lauter, and M. Naehrig, "Crypto-nets: Neural networks over encrypted data," arXiv preprint arXiv:1412.6181, 2014.
[14] H. Chabanne, A. De Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network," Cryptology ePrint Archive, 2017/035, 2017.
[15] E. Lee, J.W. Lee, J.S. No, and Y.S. Kim, "Minimax approximation of sign function by composite polynomial for homomorphic comparison," IEEE Trans. Dependable and Secure Comput., vol.19, no.6, pp.3711–3727, 2021.
[16] J. Lee, E. Lee, J.W. Lee, Y. Kim, Y.S. Kim, and J.S. No, "Precise approximation of convolutional neural networks for homomorphically encrypted data," arXiv preprint arXiv:2105.10879, 2021.
[17] J.L.H. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup, "Doing real work with FHE: The case of logistic regression," Proc. 6th W. on Encrypted Computing & Applied Homomorphic Cryptography, pp.1–12, 2018.
[18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachéne, "TFHE: Fast fully homomorphic encryption over the torus," J. Cryptol., vol.33, no.1, pp.34–91, 2020.
[19] S. Carpov, M. Izabachéne, and V. Mollimard, "New techniques for multi-value input homomorphic evaluation and applications," Topics in Cryptology–CT-RSA, LNCS, vol.11405, pp.106–126, 2019.
[20] D. Micciancio and Y. Polyakov, "Bootstrapping in FHEW-like cryptosystems," Proc. 9th on W. on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'21), pp.17–28, 2021.
[21] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping," Cryptology ePrint Archive, Paper 2021/1337, 2021.
[22] W.J. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption," Proc. 2021 IEEE Symp. on Security and Privacy (S&P), pp.1057–1073, 2021.
[23] D. Maeda, K. Morimura, S. Narisada, K. Fukushima, and T. Nishide, "Efficient homomorphic evaluation of arbitrary uni/bivariate integer functions and their applications," Cryptology ePrint Archive, Paper 2023/366, 2023.
[24] Z. Brakerski, C. Gentry, and S. Halevi, "Packed ciphertexts in LWE-based homomorphic encryption," Public-Key Cryptography – PKC 2013, LNCS, vol.7778, pp.1–13, 2013.
[25] R. Li, and H. Yamana, "Fast and accurate function evaluation with LUT over integer-based fully homomorphic encryption," Advanced Information Networking and Applications: Proc. 35th Int'l Conf. on Advanced Information Networking and Applications, AINA-2021, LNNS, vol.226, pp.620–633, 2021.
[26] R. Li, S. Bhattacharjee, S.K. Das, and H. Yamana, "Look-up table based FHE system for privacy preserving anomaly detection in smart grids," Proc. 2022 IEEE Int'l Conf. on Smart Computing (SMARTCOMP), pp.108–115, 2022.
[27] N.P. Smart and F. Vercauteren, "Fully homomorphic simd operations," J. Designs, Codes and Cryptography, vol.71, no.1, pp.57–81, 2014.
[28] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," J. ACM (JACM), vol.45, no.6, pp.965–981, 1998.
[29] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "CHIMERA: Combining ring-LWE-based fully homomorphic encryption schemes," J. Math. Cryptol., vol.14, no.1, pp.316–338, 2020.
[30] J.H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Advances in Cryptology–ASIACRYPT, LNCS, vol.10624, pp.409–437, 2017.
[31] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," Advances in Cryptology–EUROCRYPT, LNCS, vol.9056, pp.617–640, 2015.
[32] Y. Tian, Q. Zhang, T. Wang, and Q. Xu, "Lookup table allocation for approximate computing with memory under quality constraints," Proc. 2018 Design, Automation & Test in Europe Conf. & Exhibition (DATE), pp.153–158, 2018.
[33] A. Raha and V. Raghunathan, "qLUT: Input-aware quantized table lookup for energy-efficient approximate accelerators," ACM Trans. Embed. Comput. Syst. (TECS), vol.16, no.5s, pp.1–23, 2017.
[34] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, 2012/144, 2012.
[35] Microsoft Research, Redmond, WA., Microsoft/SEAL (release 4.0), https://github.com/Microsoft/SEAL, March 2022.
[36] L. Jiang and L. Ju, "FHEBench: Benchmarking fully homomorphic encryption schemes," arXiv preprint arXiv:2203.00728, 2022.
[37] F. McKeen, I. Alexandrovich, A. Berenzon, C.V. Rozas, H. Shafi, V. Shanbhogue, and U.R. Savagaonkar, "Innovative instructions and software model for isolated execution," Proc. 2nd Int'l W. on Hardware and Architectural Support for Security, vol.10, no.1, 2013.
[38] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper, 2016.
[39] H. Xiao, Q. Zhang, Q. Pei, and W. Shi, "Privacy-preserving neural network inference framework via homomorphic encryption and SGX," Proc. 2021 IEEE 41st Int'l Conf. on Distrib. Comput. Syst. (ICDCS), pp.751–761, 2021.
[40] J. Takeshita, C. McKechney, J. Pajak, A. Papadimitriou, R. Karl, and T. Jung, "GPS: Integration of graphene, PALISADE, and SGX for large-scale aggregations of distributed data," Cryptology ePrint Archive, no.2021/1155, pp.1–21, 2021.
[41] C. Yakupoglu and K. Rohloff, "PREFHE, PREFHE-AES and PREFHE-SGX: Secure multiparty computation protocols from fully homomorphic encryption and proxy reencryption with AES and intel SGX," Proc. Security and Privacy in Commun. Netw., LNICST, vol.462, pp.819–837, 2023.

Ruixiao Li is a doctor course student at Waseda University whose interest is in secure computing and cloud computing.

Hayato Yamana has been a Professor with the Faculty of Science and Engineering, Waseda University, since 2005. He is a member of ACM, IEEE, IPSJ, DBSJ and IEICE. He is a fellow of IPSJ and IEICE. He served on the Board of Governors of the IEEE Computer Society from 2018 to 2020.