IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 3, 28 pages.

https://doi.org/10.62056/andkmp-3y

Non-interactive Private Multivariate Function Evaluation using Homomorphic Table Lookup

Ruixiao Li[1] and Hayato Yamana[2]

[1] Waseda University, Dept. of CSCE, Tokyo, Japan
[2] Waseda University, Faculty of Science and Engineering, Tokyo, Japan

Abstract. To address security issues in cloud computing, fully homomorphic encryption (FHE) enables a third party to evaluate functions using ciphertexts that do not leak information to the cloud server. The remaining problems of FHE include high computational costs and limited arithmetic operations, only evaluating additions and multiplications. Arbitrary functions can be evaluated using a precomputed lookup table (LUT), which is one of the solutions for those problems. Previous studies proposed LUT-enabled computation methods 1) with bit-wise FHE and 2) with word-wise FHE. The performance of LUT-enabled computation with bit-wise FHE drops quickly when evaluating BigNum functions because of the complexity being $O(s \cdot 2^{d \cdot m})$, where m represents the number of inputs, d and s represent the bit lengths of the inputs and outputs, respectively. Thus, LUT-enabled computation with word-wise FHE, which handles a set of bits with one operation, has also been proposed; however, previous studies are limited in evaluating multivariate functions within two inputs and cannot speed up the evaluation when the domain size of the integer exceeds $2N$, where N is the number of elements packed into a single ciphertext. In this study, we propose a non-interactive model, in which no decryption is required, to evaluate arbitrary multivariate functions using homomorphic table lookup with word-wise FHE. The proposed LUT-enabled computation method 1) decreases the complexity to $O(2^{d \cdot m}/l)$, where l is the element size of FHE packing; 2) extends the input and output domain sizes to evaluate multivariate functions over two inputs; and 3) adopts a multidimensional table for enabling multithreading to reduce latency. The experimental results demonstrate that evaluating a 10-bit two-input function and a 5-bit three-input function takes approximately 90.5 and 105.5 s with 16-thread, respectively. Our proposed method achieves 3.2x and 23.1x speedup to evaluate two-bit and three-bit 3-input functions compared with naive LUT-enabled computation with bit-wise FHE.

Keywords: Function evaluation · secure computing · lookup table · fully homomorphic encryption

1 Introduction

Privacy-preserving systems facilitate the safeguarding of personal privacy while utilizing cloud computing applications. Common cloud privacy-preserving technologies include secure multiparty computing (SMPC), differential privacy (DP), and homomorphic encryption (HE). Since the pioneering work of Yao [Yao82], SMPC has been employed in various systems to protect sensitive data without revealing them, as demonstrated in previous studies [BPTG15, CDH$^+$19, GCH$^+$18, MRVW21, CMTB16, DCW13]. However, a shortcoming of SMPC is the significant communication cost associated with a multiparty

---

E-mail: liruixiao@yama.info.waseda.ac.jp (Ruixiao Li), yamana@yama.info.waseda.ac.jp (Hayato Yamana)

Received: 2024-07-07 Accepted: 2024-09-02

interactive model of massive data. Furthermore, SMPC is designed for specific types of protocols, and the implementation of general functions is challenging. DP conceals personally identifiable information by introducing noise into a dataset. DP has widespread applications in diverse systems, as indicated in references such as [ZC20, ZTG+19, CBK+20, LDL15]. Nonetheless, DP is difficult to balance between privacy and usability because a high level of privacy requires more noise, which may lead to unknown effects.

HE allows a cloud server to evaluate functions over encrypted data and resists quantum computing to provide a high security level. The challenges of HE include its high computational costs and limited operations that apply only to additions and multiplications. In 2009, Gentry [Gen09] introduced a fully holomorphic encryption (FHE) scheme based on ideal lattices to apply both homomorphic addition and multiplication without time limitations. Two encoding methods are adopted for different FHE schemes, bit-wise encoding adopted to such as GSW [GSW13], FHEW [DM15] and TFHE [CGGI20] schemes which encrypt data bit-by-bit, and word-wise encoding adopted to such as BGV [Bra12], BFV [BGV14, FV12], and CKKS [CKKS17] schemes which encrypt a vector of integers or complex numbers. Word-wise encoding allows for handling more data in one operation to improve efficiency compared with bit-wise encoding. However, word-wise FHE limits the types of operations in that it can only adapt to additions and multiplications.

To improve the efficiency of evaluating complicated functions such as logarithms or divisions with FHE, the existing studies adopt three main ideas: 1) polynomial approximation over FHE, 2) naive LUT method with bit-wise FHE, and 3) improved LUT method with word-wise FHE.

Xie et al. [XBF+14], Gilad et al. [GDL+16], Chou et al. [CBL+18], and Hesamifard et al. [HTG19] used the polynomial approximation (PA) to replace direct computation with polynomial evaluation over FHE. Polynomial approximation enables the approximation of arbitrary functions using a polynomial composed of only additions and multiplications. A shortcoming of the PA is that it guarantees accuracy within a specific input range of relatively smooth functions; otherwise, the accuracy drops rapidly. PA works well for activation functions used in neural networks. The polynomials with a higher degree improve the accuracy; however, a high degree requires an increased depth of multiplication level of FHE, which leads to a long latency and is not acceptable for data-driven applications.

The other solution is to use precomputed lookup tables (LUT) of the objective function with FHE. However, the latency of existing studies [DM15, CGGI20, CGH+18] using bit-wise FHE increases rapidly with the bit length. The computational complexity of the naive LUT method with bit-wise FHE is $O(s \cdot 2^{d \cdot m})$, where m is the number of inputs, d and s are the input and output bit lengths, respectively. Maeda et al. [MMN22] achieved a uni/bivariate function evaluation with LUT using a word-wise FHE with a complexity of $O(N)$ for a 2-input function evaluation, where N is the input domain size and can be further extended to $2N$. However, [MMN22] provided a proposal specialized for bivariates; the solution for multivariates with more than two is not apparent, and it cannot handle an integer larger than $2N$, where N falls within the number of elements of the FHE packing, that is, the size of the vector. Otherwise, the advantage of complexity is lost. In addition, [MMN22] did not extend the output domain size, which must be the same as the input domain size.

Li et al. [LY21, LY24] introduced an interactive model that employed a trusted party to communicate with the cloud to evaluate multi-input functions using word-wise FHE with LUT. Even if the trusted party cannot infer the function and input/output from a randomly selected LUT with redundant data points, the index distribution and output index are leaked to the trusted party. In this study, we retained most of the strengths of [LY21, LY24] and used a non-interactive model in which all computations are over ciphertexts. The non-interactive model does not require a trusted party. To address these problems, the contributions of this study are as follows:

Contributions:

1) We propose a private multivariate function evaluation protocol that uses word-wise FHE with LUTs. Our method allows for the evaluation of an arbitrary function $\overbrace{\mathbb{Z}_N \times ... \times \mathbb{Z}_N}^{m} \to \mathbb{Z}_{n \cdot N}$, where N is the input domain size even if that does not fall within the element size of FHE packing, m is the number of inputs, and n is any constant. We describe an original method of multidimensional table lookup construction and processing to adapt arbitrary multivariate function evaluation with word-wise FHE and reduce the execution time. Meanwhile, we show a series of experiment results to demonstrate the practicality of the proposed method.

2) We reduce the computational complexity from $O(s \cdot 2^{\sum_{i=1}^{m} d_i})$ using the bit-wise LUT method to $O(2^{\sum_{i=1}^{m} d_i}/l)$ by using the packing technique of word-wise FHE, where m is the number of inputs, l is the element size of FHE packing, d_i and s are the bit lengths of i-th input and output, respectively.

3) We propose a BigNum decomposition and table separation method to reduce the latency and extend the output domain size. Our proposed method allows the evaluation of large integers with a small plaintext space that can flexibly extend the output domain size. Multidimensional LUT construction enables multithreading to decrease runtime through parallelization.

Our proposed LUT method is adaptable to any function by employing accurate input and output tables for a given function and provides highly accurate results even for noncontiguous functions with a wider input range than polynomial approximation functions. Thus, our protocol can expand the use of FHE, which makes it possible to implement complex functions in real-world applications that have been difficult to adopt FHE; for example, the privacy-preserving anomaly detection systems in smart grids [LBDY22].

The rest of this paper is organized as follows. The existing related works are in Section 2. Section 3 introduces the preliminaries of this study. The details of the proposed non-interactive multivariable function evaluation method with FHE are presented in Section 4. We present complexity analysis and performance evaluation in Sections 5 and 6. Section 7 compares the proposed method to related studies. Finally, we conclude this study in Section 8.

2 Related Work

To address the challenge that FHE cannot evaluate complicated functions that are not composed of additions and multiplications, such as logarithms and divisions, the existing related study introduces three methods: 1) polynomial approximation over FHE, 2) a naive method of LUT that uses bit-wise FHE homomorphic table lookup, and 3) an improved homomorphic table lookup method that uses word-wise FHE to achieve lower latency.

In this section, we introduce the advantages and disadvantages of the previous studies.

2.1 Polynomial Approximation over FHE

Xie et al. [XBF$^+$14] first used the polynomial approximation (PA) to replace the direct computation of the activation function used in neural networks with a polynomial evaluation over FHE. Commonly used activation functions, such as the Swish and Tanh functions, cannot be evaluated with FHE directly. PA enables the execution of approximate arbitrary functions by using a polynomial [CT12] composed only of additions and multiplications.

Chabanne et al. [CdWM$^+$17] applied a polynomial approximation to the ReLU function with CKKS [CKKS17] using polynomials of degrees 2 through 6 on a light convolutional neural network (CNN). ReLU function is defined as $f(x) = max(0, x)$, which cannot be directly computed with FHE because of the comparison. Gilad-Bachrach et al. [GDL$^+$16] and Chou et al. [CBL+18] introduced CryptoNets, which use PA to compute activation functions in an inference over encrypted data using a neural network model. Lee et al. [LLNK22] proposed the PA of the Sign function and determined the optimal set of degrees for a minimax composite polynomial by considering the number of nonscalar multiplications and the depth consumption. This approach effectively reduces the function runtime by an average of 45 % with the PA-based FHE.

Hesamifard et al. [HTG19] designed approximate Sigmoid, ReLU, and Tanh functions with low-degree polynomials and trained CNNs with PA to improve accuracy. [HTG19] achieved 99.25 % accuracy when applied to the MNIST dataset, a commonly used handwritten digits dataset. Cheon et al. [CKP22] introduced domain extension polynomials (DEPs) to extend the range of inputs while maintaining the features of the original function in its original input range. An experiment [CKP22] with bit-wise FHE exploited the logistic function in the range $[-7683, 7683]$.

The challenge of PA is that it only guarantees accuracy within a specific input range of relatively smooth functions; otherwise, the accuracy decreases rapidly. The PA works well for activation functions used in neural networks. Polynomials with a higher degree improve accuracy; however, a high degree requires an increased depth of multiplication level of FHE, which leads to a long latency and is not acceptable for data-driven applications.

2.2 Homomorphic Table Lookup with Bit-Wise FHE

Crawford et al. [CGH+18] replaced the direct computations of complicated functions with homomorphic table lookups to improve the efficiency of bit-wise-encoding-based FHE. They [CGH+18] built a precomputed table containing the input T_{fin} and output T_{fout} data points of the objective function f, where $T_{fin} = x$ and $T_{fout} = f(x)$. Using the MUX gates, the combined input $ct(\vec{q})$ returns $ct(\vec{r}[i])$, where $0 \leq i < s$ and s is the bit length of the output.

$$ct(\vec{r}[i]) \leftarrow \sum_{j=1}^{2^{m\cdot d}} \left(\prod_{k=1}^{m\cdot d} \Big(ct(\vec{q}[k]) \oplus T_{fin}[j,k] \oplus ct(1) \Big) \otimes T_{fout}[j,i] \right) \tag{1}$$

where m is the number of inputs and d is the bit length of the input.

Carpov et al. [CIM19] and Chillotti et al. [CGGI20] improved the bootstrapping process in the bit-wise FHE scheme called TFHE, which is used to reduce the noise from multiplications in the ciphertext to decrease the latency. The experimental result of LUT shows [CIM19] requires approximately 1.57 s to assess an arbitrary 6-to-6-bit function. [CGGI20] evaluates an 8-to-8-bit function in 1.096 s and a 16-to-8-bit function in 2.192 s. Boura et al. [BGGJ20] and Lu et al. [jLHH+21] proposed a framework called PEGASUS, which enables switching back and forth between bit-wise and word-wise schemes such as FHEW [DM15] and CKKS [CKKS17]. PEGASUS allows the evaluation of arithmetic functions on word-wise FHE to enhance efficiency and enables the evaluation of complicated functions on bit-wise FHE with logic circuits.

However, all the aforementioned LUT studies are based on bit-wise encoding FHE. Because bit-wise encoding encodes and encrypts data bit-by-bit, the complexity of the naive LUT method is $O(s \cdot 2^{d\cdot m})$. This complexity grows exponentially with the input bit length, where d and s represent the bit lengths of the input and output, respectively, and m is the number of inputs, which is not suitable for evaluating BigNum integer functions.

2.3 Homomorphic Table Lookup with Word-Wise FHE

Okada et al. [OCHK18] proposed a linear depth algorithm for univariate and bivariate functions using word-wise FHE. They decomposed the two-input function into two single-input functions. LUT contains coefficients prepared by approximating the functions using polynomial interpolation. In their experiments, they compared their results to those of Chen et al. [CG15], Xu et al. [XCWF16], and Chen et al. [CFLW17]. The results show that they achieved a 2.45x faster execution than the fastest bit-wise algorithm [CFLW17] mentioned in their paper. Based on Okada et al.'s work [OCHK18], Maeda et al. [MMN22] improved the algorithm by adopting the Paterson-Stockmeyer method to decrease the complexity. They prepared all the coefficient LUT of polynomials $f_0(x), ..., f_{d_i}(x), ..., f_{N-1}(x)$ to compute an arbitrary bivariate function, where $0 \leq d_i < N$ and N is the input domain size. The LUT contains the coefficients $c_{i,d}$ of the polynomial $f_d(x) = f(x, d)$

$$f_d(x) = c_{0,d} + c_{1,d}x + c_{2,d}x^2 + ... + c_{N-1,d}x^{N-1} \mod t \tag{2}$$

where $c_{i,d}$ is precomputed using polynomial interpolation, and t is plaintext space. The results demonstrate that the proposed method evaluates 12-to-12-bit functions in 57.5 s.

Prior studies with word-wise encoding FHE schemes evaluated large integers with half of the required plaintext space as a bivariate function to reduce latency because a large plaintext space leads to a long latency. In [MMN22], the input domain size can be further extended from N to $2N$ to evaluate the functions $\mathbb{Z}_N \times \mathbb{Z}_N \to \mathbb{Z}_N$ to $\mathbb{Z}_{2N} \times \mathbb{Z}_{2N} \to \mathbb{Z}_{2N}$. However, [MMN22] specialized in uni/bivariate functions. The solution for multivariate functions, in which more than two variates are not apparent in [MMN22], and it cannot handle an integer larger than $\mathbb{Z}_{2N}$, where N falls within the number of packing elements. Otherwise, the advantage of complexity is lost. Additionally, [MMN22] did not increase the domain size of the output. Even if we evaluate a univariate function by decomposing the large input integer size of $\mathbb{Z}_{2N}$ into half and considering it a bivariate function, the output domain size is still $\mathbb{Z}_N$.

3 Preliminaries

3.1 Notation

Table 1 summarizes the notation used in this study. We used uppercase letters to represent matrices unless otherwise specified. The input and output data points are stored separately in LUTs T_{in} and T_{out}. For example, assuming a two-input function $f(x_0, x_1)$, where $0 \leq x_i < 2$ and $0 \leq i < 2$, $T_{in} = [0, 1]$ and $T_{out} = [f(0,0), f(0,1), f(1,0), f(1,1)]$. The construction of LUTs is described in Section 4.2. We denote the vectors in $\vec{\cdot}$ and multidimensional vectors in uppercase letters, i.e., $[\![\vec{a}]\!]$ and $[\![A[i]]\!]$ represent an encrypted vector $\vec{a}$ and the encrypted i-th row of matrix (two-dimensional vector) A, respectively. In addition, we denote an encrypted vector whose elements are all x as $[\![\vec{a}_{el=x}]\!]$.

3.2 SIMD Operation over Word-Wise FHE

Smart and Vercauteren [SV14] introduced a fully homomorphic element-wise single instruction multiple data (SIMD) operation based on the packing method of the polynomial-Chinese remainder theorem (polynomial-CRT). Using [SV14, BGH13], we pack l elements, each of which is called a slot (hereinafter referred to as slot), as a single plaintext or ciphertext. Slot-wise SIMD operations allow us to compute all the slots in parallel.

Let us pack and encrypt two vectors $\vec{a} = [(a_0, ..., a_{l-1})]$ and $\vec{b} = [(b_0, ..., b_{l-1})]$ into ciphertext $[\![\vec{a}]\!]$ and $[\![\vec{b}]\!]$. The slot-wise SIMD addition and multiplication operations are as follows:

$$\begin{aligned} [\![\vec{a}]\!] \oplus [\![\vec{b}]\!] &= [\![(a_0 + b_0, ..., a_{l-1} + b_{l-1})]\!] \\ [\![\vec{a}]\!] \otimes [\![\vec{b}]\!] &= [\![(a_0 \times b_0, ..., a_{l-1} \times b_{l-1})]\!] \end{aligned} \tag{3}$$

Table 1: Notations

Notation	Description
m	the number of inputs for an objective multi-input function
d_i, s	the bit-length of i-th input and output, respectively, $0 \leq i < m$
c_i	an input value for a given function, $0 \leq i < m$
r	an output value for a given function
R	the intermediate results (shown in Section 4.3 and 4.4)
t	plaintext space, which is a power of two plus one
l	the number of elements set by FHE, which is a power of two
l'	the number of used elements in an encrypted vector if $\lvert T_{in}\rvert \leq l$ (shown in Section 4.3 and 4.4, $l' = \lvert T_{in}\rvert = 2^d$)
T_{in}, T_{out}	the LUT of input and output data points
$\lvert T_{in}\rvert, \lvert T_{out}\rvert$	the number of input and output data points in LUT
k_{in}, k_{out}	the number of rows of T_{in}, T_{out}, each row can be packed as a single plaintext and $\lvert T_{in}\rvert = k_{in} \times l, \lvert T_{out}\rvert = k_{out} \times l$
$[\![\cdot]\!]$	a ciphertext
$[\cdot]$	a plaintext
$[\![\vec{a}]\!]$	an encrypted vector $\vec{a}$
$[\![A[i]]\!]$	an encrypted i-th row of matrix A
$[\![\vec{a}_{el=x}]\!]$	an encrypted vector whose all used elements are x
$Enc(\cdot)$	encryption operation
$Dec(\cdot)$	decryption operation
$\oplus, \ominus, \otimes$	homomorphic addition, subtraction and multiplication

3.3 Homomorphic Equality Comparison of Integers with Fermat's Little Theorem

FHE cannot directly compare integers because FHE cannot reveal the values during the processing. In this section, we introduce how to use the characteristics of FHE and Fermat's little theorem, a fundamental result in number theory, to compare integer equality.

The equality comparison $Eq(a, b)$ check whether the two integers a, b are equal is defined as follows:

$$Eq(a,b) = \begin{cases} 1, & a = b \\ 0, & otherwise \end{cases} \tag{4}$$

The FHE computations are modulus computations over ring $\mathcal{R}$. We set the plaintext modulus to t, which is prime, and all computations are mod by t. Using the modulus computation characteristics, we adopt Fermat's Little Theorem to implement the equality method.

Theorem 1 (Fermat's Little Theorem)**.** *Let t be a prime. For any integer a that is not divisible by t, we have*

$$a^{t-1} \equiv 1 (\mod t) \tag{5}$$

Based on Fermat's Little Theorem, we have the following integer equality method:

$$Eq(a,b) = 1 - (a-b)^{t-1} \tag{6}$$

4 Proposed Non-interactive Private Multivariate Function Evaluation

The remaining problems in previous studies include the following:

p-1) The existing study [OCHK18, MMN22] provided a proposal specialized for uni/bivariate functions.

p-2) The complexity advantage of the existing study [MMN22] requires input and output domain sizes no more than $2N$, where N falls within the number of slots of FHE packing.

p-3) The naive LUT method using bit-wise FHE has a high computational complexity that increases with the bit length. This complexity is given by $O(s \cdot 2^{\sum_{i=1}^{m} d_i})$, where m is the number of inputs, d_i and s are the bit lengths of i-th input and output, respectively.

To address these problems, we propose a new non-interactive model that adopts the following solutions:

s-1) We propose a new LUT processing protocol with word-wise FHE to enable an arbitrary function evaluation $\overbrace{\mathbb{Z}_N \times ... \times \mathbb{Z}_N}^{m} \rightarrow \mathbb{Z}_{n \cdot N}$, where N is the input domain size that does not fall within the slot size of FHE packing, m is the number of input, and n is any constant, which solves p-1) and p-2).

s-2) We reduce the computational complexity from $O(s \cdot 2^{\sum_{i=1}^{m} d_i})$ using the bit-wise LUT method to $O(2^{\sum_{i=1}^{m} d_i}/l)$ by using the packing technique of word-wise FHE, where m is the number of inputs, l is the slot size of FHE packing, d_i and s are bit-lengths of i-th input and output, respectively, which solves p-3).

s-3) We propose a BigNum decomposing and table separation method to reduce latency and extend the output domain size. Our proposal allows us to evaluate large integers with small plaintext space, which can flexibly extend the output domain size. The multidimensional LUT construction adapts the multithreading technique to decrease runtime by parallelization.

We present our system overview and initial table construction in Sections 4.1 and 4.2, respectively. Details of the proposed method for single- and multi-input functions are provided in Sections 4.3 and 4.4. We present the integer-decomposing and table separation method in Section 4.5.

4.1 System Overview

The proposed system is shown in Figure 1 and includes two parties: a user and a server. The user is honest; the server is semi-honest and follows the protocol but is curious about obtaining sensitive data. All computations on the server are performed over the ciphertext. Thus, neither the input nor the output is visible to the server. In the initialization phase, the user generates a set of keys and maintains a secret key (SK). The public key (PK) and evaluation keys, that is, the relinearlization key (RelinKey) and rotation key (RotKey),

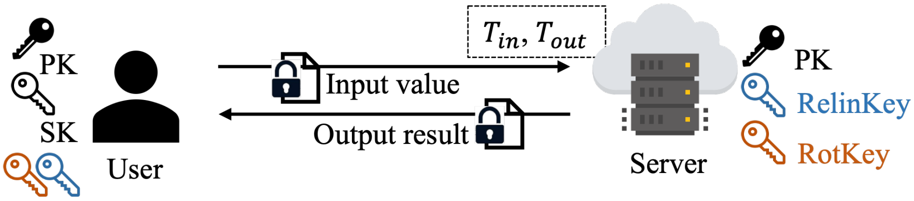


Figure 1: System overview

are shared with the server. The server maintains the LUTs of the objective function. We assume that the function owner is the server and that the LUTs are stored as plaintexts. Note that if the function owner is not the server, the LUTs are maintained by ciphertexts.

The user sends encrypted input values $\{[\![\vec{c}_{el=c_0}]\!], ..., [\![\vec{c}_{el=c_{m-1}}]\!]\}$ of the objective m-input function to the server, and the server returns the output result $[\![r]\!] = [\![f(c_0, ..., c_{m-1})]\!]$. The result may be used to perform further computations on the server if needed.

Our proposed system replaces the direct computation of a given function over ciphertexts with LUT processing. Figure 2 shows a flow chart of the processing. Note that we prepare the input data points of the pre-computed LUT in T_{in} and output data points in T_{out}. For the m-input function, each input uses the same input LUT, which is seen as one dimension. The output LUT is a m-dimensional hypercube that holds corresponding output data points to the input data points.

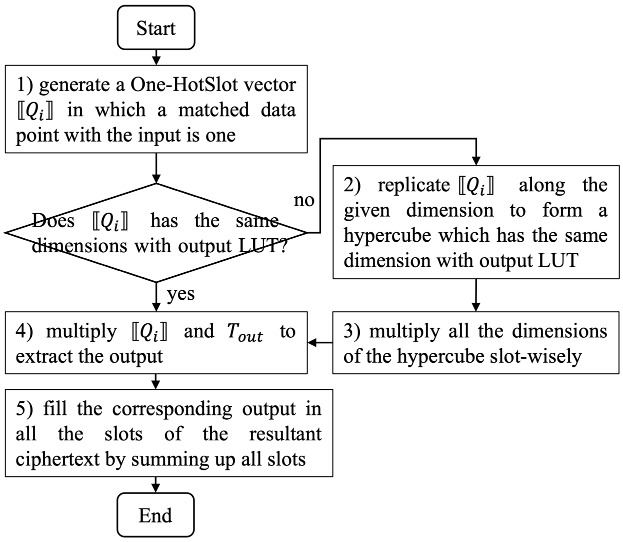


Figure 2: Flow chart of the LUT processing

The input x_i of the function $f(x_0, ..., x_{m-1})$ is encrypted as one ciphertext. Step 1) searches the matched data point in the input LUT T_{in} with each input x_i, resulting in having a One-HotSlot vector $[\![Q_i]\!]$ in which the matched slot is one and the other slots are zero, which is used for selecting output in the output LUT T_{out}. The matching computation adopts Fermat's Little Theorem, as shown below.

$$\texttt{One-HotSlot}(input) := allOneVector - (input - T_{in})^{t-1}, \tag{7}$$

where the $allOneVector$ is a vector whose all slots are one.

When we handle a BigNum input, it is decomposed into multiple vectors, followed by adopting Equation 7 for each decomposed input to match the data point with the T_{in}.

We skip Steps 2) and 3) for the one-input function because the number of dimensions of input and output LUTs is the same (details in Section 4.3). Step 4) extracts the output of the function f by extracting the matched data point in the output LUT T_{out} by multiplying the one-hot slot vector and the output LUT T_{out}. The resultant vector (one or more ciphertexts) has the result value of $f(x)$ in the i-th slot with zeros in other slots. Finally, Step 5) sums up all the slots (in all the ciphertexts if there exist plural ciphertexts), resulting in a single ciphertext with the result value of $f(x)$ in all the slots.

The number of dimensions of input and output LUTs is different when computing a m-input function $f(x_0, ..., x_{m-1})$ (details in Section 4.4). For each dimension, we apply Equation 7 to match each input with corresponding input LUT T_{in}, then Step 2) replicates it along the given dimension to form a hypercube. This yields a m dimensional hypercube where the i-th dimension has all 1 in the hyperplane x_i. Step 3) multiplies all the dimensions of the hypercube slot-wisely, resulting in a hypercube in which the position corresponding to $(x_0, ..., x_{m-1})$ is one and the others are zeros. Step 4) multiplies all the hypercubes and the T_{out} to have the result of $f(x_0, ..., x_{m-1})$ in one slot. Step 5) The result is a single ciphertext whose all slots are the output by summing up all slots.

The above method can handle multi-dimensional table lookups where the inputs and outputs are sized up to the plaintext modulus. Larger indexes can then be supported by considering each large index as multiple inputs smaller than t. Larger outputs can also be handled by preparing multiple output LUTs, where each T_{out} contains a part of the output. In this case, we need one more step to combine a set of outputs.

We show the details in the following sections.

4.2 Construction of Lookup Table

For the objective function f, we store the inputs and outputs in LUTs T_{in} and T_{out}, respectively, which we call input and output data points. The input table T_{in} contains all the possible d-bit input values, logically we have $T_{in} = [0, 1, ..., 2^{d-1}]$. The output table similarly contains all the corresponding output values, $T_{out} = [f(0), f(1), ..., f(2^{d-1})]$. In the simplest case with one input and where $l = 2^d$ and the output modulus is less than t, we present each of these tables using a single native plaintext element and fill each data point in one slot, with the i-th slot of T_{in} contains i and the i-th slot of T_{out} contains $f(i)$. If $l > 2^d$, then we still use one native plaintext for each of the tables, and fill the data points into equal-interval slots from the first slot, whereas unused slots are filled with zero. If $l < 2^d$, then we use multiple native plaintexts for each of the tables, and think of them as a two-dimensional array with the columns being the slots of a single native plaintext. Each output data point in T_{out} corresponds to an input data point (a set of input data points for multi-input functions) in T_{in}. The LUTs are constructed as multi-dimensional vectors when the number of data points exceeds the number of slots. The multi-dimensional vectors of LUTs allow us to adapt the multithreading technique to parallelize the computations among the rows.

4.2.1 Construction of Lookup Table for One-input Function

The input and output data points correspond individually for a d-bit one-input function $f(x)$. The number of data points is $|T_{in}| = |T_{out}| = 2^d$. T_{in}, T_{out} are 2-dimensional vectors with a column of length l; the row length is $k_{in} = k_{out} = 2^d/l$, which is an integer when $|T_{in}|(= |T_{out}|) \geq l$. This is because l is the slot size, which is a power of two in the FHE setting. The output data point corresponded to the input data point $T_{in}[indI_{row}, indI_{col}]$ is shown as $T_{out}[indO_{row}, indO_{col}]$, where $0 \leq indI_{row}(= indO_{row}) < k_{in}(= k_{out})$ and $0 \leq indI_{col}(= indO_{col}) < l$.

Note that when the number of input data points is less than the number of slots, $|T_{in}|(= |T_{out}|) = 2^d < l$, the data points are filled into equal-interval slots from the first slot. The interval is $l/|T_{in}|$, which is also an integer because l is a power of two in the FHE setting. Because even if the number of data points is smaller than l, FHE still needs to pack l data points into a single ciphertext. We fill the data points into equal-interval slots to reduce the complexity of preparing the final results described in Algorithm 2 (Section 4.3).

Two examples of a 4-bit one-input function LUTs are shown in Figure 3. The number of data points is $|T_{in}|(= |T_{out}|) = 16$. We show the situation when the number of slots is 4 or 32. When the number of slots $l = 4 < |T_{in}|$, all slots are filled with data points. When the number of slots $l = 32 > |T_{in}|$, the data points are filled into every two slots from the first slot, and the unused slots are filled with zeros.

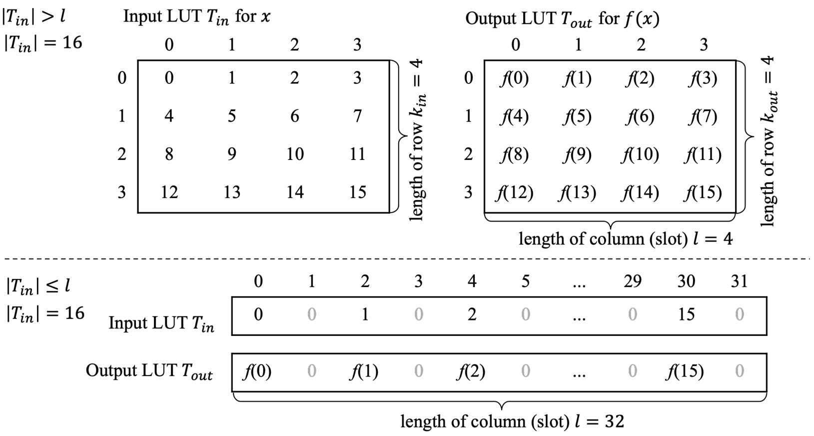


Figure 3: LUT construction for one-input function

4.2.2 Construction of Lookup Table for Multi-input Function

For a d-bit m-input function $f(x_0, ..., x_{m-1})$ where $m(> 1)$ is the number of inputs. We generate a pair of input and output LUTs T_{in} and T_{out} as multi-dimensional vectors. We prepare a single T_{in} shared by m inputs. Thus, the number of input data points is $|T_{in}| = 2^d$. The size of T_{in} is $(l \times k_{in})$, where $k_{in} = 2^d/l$ is an integer such that l is a power of two in the FHE setting. The number of corresponding output data points is $|T_{out}| = 2^{m \cdot d}$ and the size of T_{out} is $(l \times k_{out})$ where $k_{out} = 2^{m \cdot d}/l$ is also an integer. We consider the T_{in} to be a two-dimensional vector (matrix) if the number of data points exceeds the slot size. The output LUT is an m-dimensional hypercube table; each input corresponds to a hypercube dimension whose size is $|T_{in}|$. We show examples in Figure 4.

The output data point $T_{out}[indO_{row}, indO_{col}]$ corresponds to the set of m input data points $\{T_{in}[indI^0_{row}, indI^0_{col}],...,T_{in}[indI^{m-1}_{row}, indI^{m-1}_{col}]\}$. Here, we denote $ind_i = indI^i_{row} \times l + indI^i_{col}$ and switch the input indices to $\{ind_0, ..., ind_{m-1}\}$ for easier understanding, where $0 \le i < m$. The corresponding output index $[indO_{row}, indO_{col}]$ is computed using Equation 8.

We denote the indices corresponding to the output $T_{out}[indO_{row}, indO_{col}]$ for the set of input data points $\{T_{in}[indI^0_{row}, indI^0_{col}], ..., T_{in}[indI^{m-1}_{row}, indI^{m-1}_{col}]\}$, where we specify $ind_{out} = ind_{m-1} + \sum_{i=0}^{m-2}(ind_i \times 2^{d(m-1-i)})$ and calculate the values of $indO_{row}, indO_{col}$ as follows.

$$\begin{aligned} indO_{row} &= \lfloor ind_{out}/l \rfloor \\ indO_{col} &= ind_{out} \mod l \end{aligned} \tag{8}$$

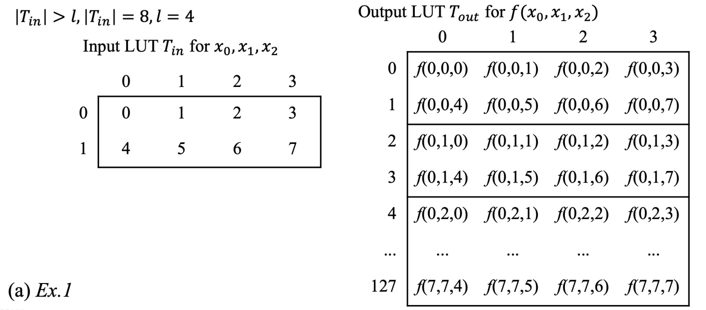
Similar to the one-input function, when the number of input data points is less than or equal to the number of slots satisfying $|T_{in}| \le l$, we fill the data points into equal-interval slots from the first slot, whereas unused slots are filled with zero. The interval is $l/|T_{in}|$, which is an integer because l is the power of two in the FHE setting and $|T_{in}| = 2^d$.

Figure 4 shows two examples of a three-input function: the first example is a 3-bit three-input function and the second one is a 2-bit three-input function. We set the number of slots to 4 and 8. The number of data points is $|T_{in}| = 2^3 = 8$ for 3-bit integers and $|T_{in}| = 2^2 = 4$ for 2-bit integers, respectively. All inputs use the same T_{in} because the input domain sizes are the same. We assume the inputs are $\{x_0, x_1, x_2\} = \{0, 1, 3\}$.

(Ex.1): When $|T_{in}| > l$, the set of index of input data points in T_{in} is $\{[0, 0], [0, 1], [0, 3]\}$ and we switch them to $\{0, 1, 3\}$ as one-dimensional representation. The corresponding output data point is $T_{out}[2, 3]$ whose index is computed using Equation 8 as $ind_{out} = 3 + 1 \times 2^{3 \cdot 1} + 0 \times 2^{3 \cdot 2} = 11$. The index of the row is $2 = \lfloor 11/4 \rfloor$ and that of the column is

$3 = 11 \mod 4$.

(Ex.2): When $|T_{in}| \leq l$, the set of index of input data points $\{0, 1, 3\}$ in T_{in} is switched to $\{0, 2, 6\}$ as shown in Figure 4 (b). The corresponding output data point for the input $\{0, 1, 3\}$ is $T_{out}[1, 6]$ whose index is computed using Equation 8 as $ind_{out} = 6 + 2 \times 2^{2 \cdot 1} + 0 \times 2^{2 \cdot 2} = 14$. The index of the row is $1 = \lfloor 14/8 \rfloor$, and that of the column is $6 = 14 \mod 8$.



$|T_{in}| > l, |T_{in}| = 8, l = 4$

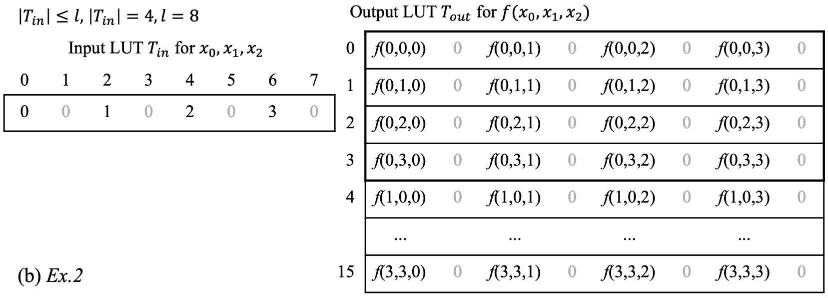
Input LUT T_{in} for x_0, x_1, x_2

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Output LUT T_{out} for $f(x_0, x_1, x_2)$

	0	1	2	3
0	f(0,0,0)	f(0,0,1)	f(0,0,2)	f(0,0,3)
1	f(0,0,4)	f(0,0,5)	f(0,0,6)	f(0,0,7)
2	f(0,1,0)	f(0,1,1)	f(0,1,2)	f(0,1,3)
3	f(0,1,4)	f(0,1,5)	f(0,1,6)	f(0,1,7)
4	f(0,2,0)	f(0,2,1)	f(0,2,2)	f(0,2,3)
...	...	...	...	...
127	f(7,7,4)	f(7,7,5)	f(7,7,6)	f(7,7,7)

(a) *Ex.1*



$|T_{in}| \leq l, |T_{in}| = 4, l = 8$

Input LUT T_{in} for x_0, x_1, x_2

0	1	2	3	4	5	6	7
0	0	1	0	2	0	3	0

Output LUT T_{out} for $f(x_0, x_1, x_2)$

0	f(0,0,0)	0	f(0,0,1)	0	f(0,0,2)	0	f(0,0,3)	0
1	f(0,1,0)	0	f(0,1,1)	0	f(0,1,2)	0	f(0,1,3)	0
2	f(0,2,0)	0	f(0,2,1)	0	f(0,2,2)	0	f(0,2,3)	0
3	f(0,3,0)	0	f(0,3,1)	0	f(0,3,2)	0	f(0,3,3)	0
4	f(1,0,0)	0	f(1,0,1)	0	f(1,0,2)	0	f(1,0,3)	0
	...		...		...		...	
15	f(3,3,0)	0	f(3,3,1)	0	f(3,3,2)	0	f(3,3,3)	0

(b) *Ex.2*

Figure 4: LUT construction examples for three-input function

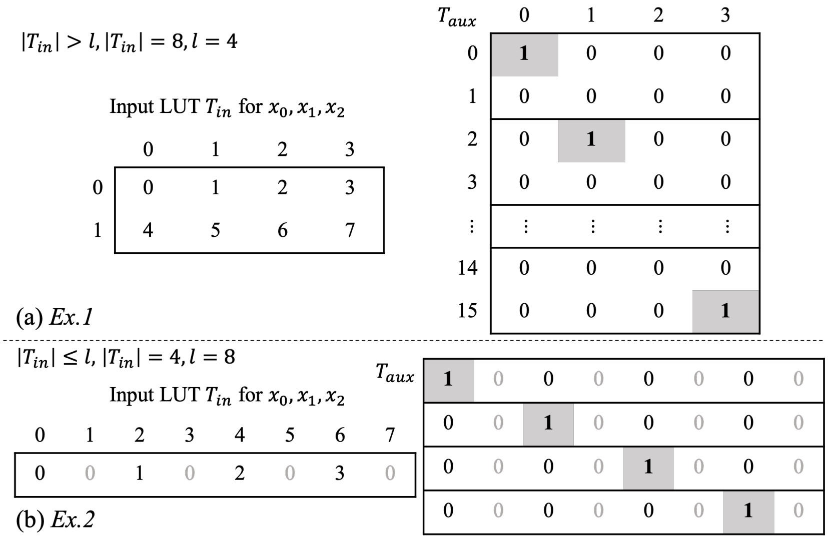
Auxiliary table T_{aux}: Besides the T_{in} and T_{out}, we generate an $|T_{in}|$-dimensional vector T_{aux}. We use T_{aux} to select the specific matched dimension of the hypercube table T_{out} when extracting the output because T_{in} corresponds to just one input, whereas T_{out} depends on all the inputs. For example, if we have 2-bit inputs for 2-input functions, the T_{in} is of size 4, but T_{out} is of size 16. To match the unique output $f(c_0, ..., c_{m-1})$ in the T_{out}, we need to generate a hypercube query whose size is the same as the output table size, in which only the matched slot is one, and the others are zero.

The reason why we construct T_{aux} is that the number of dimensions between T_{in} and T_{out} are different. We match each input with T_{in}, resulting in m intermediate results that are One-HotSlot vectors. Then, we replicate each intermediate result to expand the number of dimensions to generate a hypercube whose dimension is the same as T_{out}, followed by multiplying all hypercubes to extract the output. During the above steps, the matched slot of c_i in every dimension becomes 1. Thus, we prepare the auxiliary table T_{aux} so that only one matched slot in the i-th dimension is set to 1. We explain how to use the auxiliary table in Section 4.4 and show an example in Figures 9 and 10.

T_{aux} is a $|T_{in}|$-dimensional vector, where the i-th slot in the i-th dimension is 1, and other slots are all zero that can be considered as a combination of $|T_{in}|$ One-HotSlot vectors. Each One-HotSlot vector is one dimension of T_{aux} and has the same size as T_{in}.

When $|T_{in}| > l$, the slots of the index $T_{aux}[\lfloor (i + i \cdot |T_{in}|)/l \rfloor, (i + i \cdot |T_{in}|) \mod l]$ are one, and the others are zero, where $0 \leq i < |T_{in}|$. When $|T_{in}| \leq l$, the slots of the index $T_{aux}[i, i \cdot l/|T_{in}|]$ are one, and the others are zero, where $0 \leq i < |T_{in}|$.

Figure 5 presents two examples of T_{aux} for $|T_{in}| > l$ and $|T_{in}| \leq l$, respectively.



$|T_{in}| > l, |T_{in}| = 8, l = 4$

Input LUT T_{in} for x_0, x_1, x_2

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

T_{aux}	0	1	2	3
0	**1**	0	0	0
1	0	0	0	0
2	0	**1**	0	0
3	0	0	0	0
⋮	⋮	⋮	⋮	⋮
14	0	0	0	0
15	0	0	0	**1**

(a) *Ex.1*

$|T_{in}| \leq l, |T_{in}| = 4, l = 8$

Input LUT T_{in} for x_0, x_1, x_2

0	1	2	3	4	5	6	7
0	0	1	0	2	0	3	0

T_{aux}

1	0	0	0	0	0	0	0
0	0	**1**	0	0	0	0	0
0	0	0	0	**1**	0	0	0
0	0	0	0	0	0	**1**	0

(b) *Ex.2*

Figure 5: LUT construction examples for auxiliary tables

4.3 One-input functions evaluation

In this section, we first introduce the two algorithms used in our study, 1) `One-HotSlot` and 2) `PartialSum`, followed by a description of the homomorphic table lookup method for one-input functions.

Algorithm `One-HotSlot` [MMN22]: To find the data point in the input LUT T_{in} that matches input a, we use the homomorphic equality comparison of integers with Fermat's Little Theorem described in Section 3.3 to construct the algorithm `One-HotSlot` [MMN22]. `One-HotSlot` computes the matched slot between a ciphertext $[\![\vec{a}_{el=a}]\!] = [\![(a, ..., a)]\!]$ and a given plaintext $[\vec{b}]$ of the vector $\vec{b}$ whose slots are distinct integers. The output is an encrypted vector, where only the i-th slot is one if $b_i = a$ and the other slots are all zero. Based on our proposed table construction technique, if $|T_{in}| \leq l$, then the input value a fills every $l/|T_{in}|$ slots. For example, the number of input data points for a 1-bit single-input function is $|T_{in}| = 2$. We assume that the number of slots is $l = 8$. The encrypted input is $[\![\vec{a}]\!] = [\![(a, 0, 0, 0, a, 0, 0, 0)]\!]$, $T_{in} = [0, 0, 0, 0, 1, 0, 0, 0]$, and $T_{out} = [f(0), 0, 0, 0, f(1), 0, 0, 0]$.

We present `One-HotSlot` [MMN22] in Algorithm 1.

Algorithm `PartialSum` [KSW$^+$18]: `TotalSum` [HS14] algorithm is used to fill all slots by the sum of slots. Based on our proposed table construction, when $|T_{in}| \leq l$ we only use $|T_{in}| = l'$ slots in each row. The `PartialSum` [KSW$^+$18] algorithm fills the slots used by the sum of all slots, which can decrease the complexity from $O(\log l)$ to $O(\log l')$ compared with `TotalSum` [HS14]. Examples are presented in Figure 6.

`PartialSum` [KSW$^+$18] is shown in Algorithm 2.

Homomorphic table lookup method for the one-input functions
Because the input LUT T_{in} is a 2-dimensional vector. We pack each row as plaintext and adopt `One-HotSlot` for all rows to match the input $[\![\vec{c}_{el=c}]\!]$ in parallel. The result is denoted by $[\![R]\!]$, which is a one-hot slot vector. We multiply $[\![R]\!]$ by the output LUT T_{out} and sum all ciphertexts. The result $[\![r]\!]$ is a single ciphertext in which only the matched output remains; others are all zero. Finally, we use `PartialSum` to fill all slots used with the output.

We show an example of a 4-bit one-input function evaluation in Figure 7. We set the encrypted input $[\![\vec{c}_{el=5}]\!] = [\![(5, 5, 5, 5)]\!]$, and the number of slots is four. The one-input function evaluation algorithm is presented as Algorithm 3.

Algorithm 1: One-HotSlot($[\![\vec{a}_{el=a}]\!], [\vec{b}], ptOne, t$) [MMN22]

```
Input: $[\![\vec{a}_{el=a}]\!]$: a ciphertext of a vector whose all used elements are the input $a$;
       $[\vec{b}]$: a plaintext packed a vector; $ptOne$: a plaintext of a vector whose all
       used elements are one; $t$: plaintext modulus
Output: $c$: a ciphertext of a vector in which only the slot $a = b_i$ is 1 others are 0,
        where $b_i$ is one of the element in $\vec{b}$
$temp \leftarrow [\![\vec{a}_{el=a}]\!] \ominus [\vec{b}]$;
$temp \leftarrow temp^{t-1}$;                    ▷based on Fermat's Little Theorem
return $c \leftarrow ptOne \ominus temp$;
```

Algorithm 2: PartialSum(ct, l, l') [KSW+18]

```
Input: $ct$: a ciphertext; $l$: the number of slots; $l'$: the number of used slots in one
       vector
Output: $ct$: a ciphertext of vector that all used elements are the sum of elements
        in original $ct$
$rotN \leftarrow l/l'$;   ▷$rotN$ is an integer because both $l$ and $l'$ are a power of
  two.
$count \leftarrow \log_2 l'$;
for $i = 0$ to $count - 1$ do
    $temp \leftarrow$ Rot($ct, 2^i \cdot rotN$);        ▷right rotate $ct$ by $2^i \cdot rotN$ elements
    $ct \leftarrow ct \oplus temp$;
end
return $ct$;
```

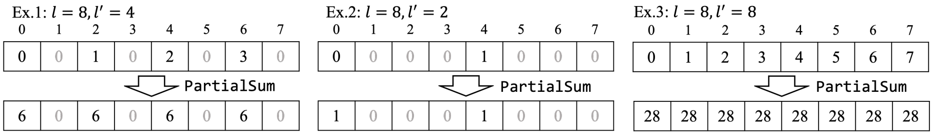


Figure 6: Examples of Algorithm `PartialSum`

Algorithm 3: One-input functions evaluation($[\![\vec{c}_{el=c}]\!], T_{in}, T_{out}, k_{in}(= k_{out}), l, l', ptOne, t$)

```
Input: $[\![\vec{c}_{el=c}]\!]$: the ciphertext of input; $T_{in}, T_{out}$: the LUTs whose each row
       packed as a plaintext; $k_{in}(= k_{out})$: the number of rows of LUTs; $l$: the
       number of slots; $l'$: the number of used slots in one vector; $ptOne$: a
       plaintext of a vector whose all used elements are one; $t$: plaintext modulus
Output: $r$: the ciphertext of the output result
$[\![r]\!] \leftarrow []$ ;                                          ▷a ciphertext
$[\![r]\!] \leftarrow \bigoplus_{i=0}^{k_{out}-1}(T_{out}[i] \otimes$ OneHotSlot($[\![\vec{c}_{el=c}]\!], T_{in}[i], ptOne, t$));
return $[\![r]\!] \leftarrow$PartialSum($[\![r]\!], l, l'$);
```

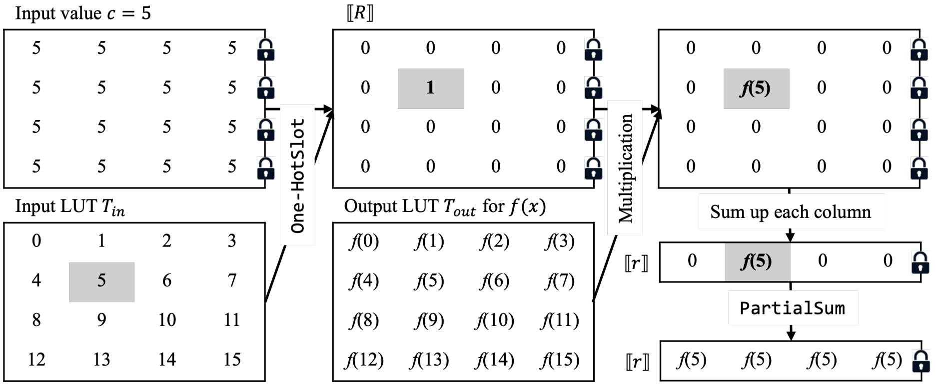


Figure 7: An example of 4-bit one-input function evaluation

4.4 Multi-input functions evaluation

In this section, we first introduce the two algorithms: 1) `VectorSum` and 2) `VectorExpand` used in our study, followed by a description of the homomorphic table lookup method for multi-input functions.

Algorithm `VectorSum`: We introduce `VectorSum` to fill all the used slots l' in a vector using the sum of the slots, as shown in Algorithm 4. An encrypted 2-dimensional vector is a set of ciphertexts because each row is encrypted as a ciphertext. This algorithm fills a specific dimension of the output LUT hypercube with 1s to extract the final output. Examples are shown in Figure 8 (a). White shaded slots are used.

Algorithm `VectorExpand`: `VectorExpand` is shown in Algorithm 5. `VectorExpand` replicates every k' rows (ciphertexts), which we define as one dimension of the vector, n times to combine as shown in Figure 8 (b). We use this algorithm to expand the dimensions of the intermediate result to be the same as T_{out} hypercube to extract the output.

Algorithm 4: VectorSum($[\![M]\!], l, l'$)

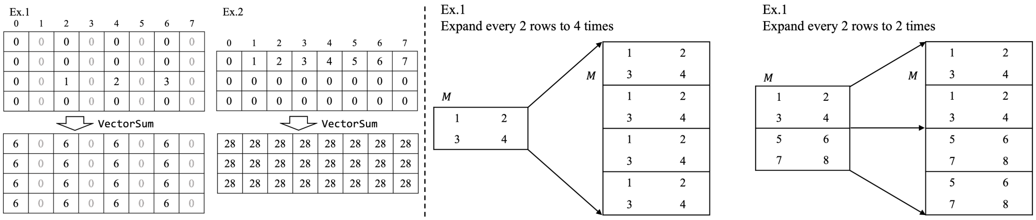
```
Input: $[\![M]\!]$: an encrypted vector; $l$: the number of slots; $l'$: the number of used slots in one vector
Output: $[\![M]\!]$: an encrypted vector that all used elements are the sum of elements in $M$
k ← the number of ciphertexts in $[\![M]\!]$;
$sum \leftarrow \bigoplus_{i=0}^{k-1}$ PartialSum($[\![M[i]]\!], l'$);
for $i = 0$ to $k - 1$ do
    $[\![M[i]]\!] \leftarrow sum$;
end
return $[\![M]\!]$;
```

Homomorphic table lookup method for the multi-input functions Similar to the one-input function, we determine the data points in the input LUT T_{in} that match each input $[\![\vec{c}_{el=c_i}]\!]$, using `One-HotSlot` to generate queries $[\![Q_i]\!]$, where $0 \leq i < m$. Next, we use $\{[\![Q_0]\!], ..., [\![Q_{m-1}]\!]\}$ and the pre-prepared T_{aux} to generate the same size hypercube as T_{out} to extract the output.

The algorithm of multi-input function evaluation is shown as Algorithm 6. The detailed steps are described below. Figures 9 – 10 show the evaluation processes for 2-bit three-input function $f(c_0, c_1, c_2)$, assuming $c_0 = 1, c_1 = 0$, and $c_2 = 2$. The result is presented in the $[\![r]\!]$. The lines below denote those in Algorithm 6:

Algorithm 5: VectorExpand($[\![M]\!], k', n$)

```
Input: $[\![M]\!]$: a vector of ciphertexts; $k'$: the number of a set of ciphertexts for
       expanding; $n$: the expand times
Output: $[\![M]\!]$: a vector of ciphertexts
1 $k \leftarrow$ the number of ciphertexts in $[\![M]\!]$;
2 $[\![temp]\!] \leftarrow []$;                                  ▷a vector of ciphertexts
3 for $i = 0$ to $k \times n - 1$ do
4     $[\![temp[i]]\!] \leftarrow [\![M[\lfloor i/n \rfloor \cdot k' + i \mod k']]\!]$;
5 end
6 return $[\![M]\!] \leftarrow [\![temp]\!]$;
```



0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	2	0	3	0
0	0	0	0	0	0	0	0

6	0	6	0	6	0	6	0
6	0	6	0	6	0	6	0
6	0	6	0	6	0	6	0
6	0	6	0	6	0	6	0

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

28	28	28	28	28	28	28	28
28	28	28	28	28	28	28	28
28	28	28	28	28	28	28	28

(a) Examples of `VectorSum`

(b) Examples of `VectorExpand`

Figure 8: Examples of `VectorSum` and `VectorExpand`

step 1) Generation of One-HotSlot vector $[\![Q_i]\!]$ (lines 2 to 6)
Step 1 generates an encrypted One-HotSlot vector $[\![Q_i]\!]$ for each input, where $0 \le i < m$. By applying `One-HotSlot` algorithm between each row of T_{in} and input $[\![\vec{c}_{el=c_i}]\!]$ to generate an encrypted One-HotSlot vector in which the matched data point in c_i with T_{in} is one and the other data points are zero.

step 2) Generation of $[\![R_i]\!]$ (`VectorSum` part in line 7 to 9)
Step 2 first expands $[\![Q_i]\!]$ to have the same size of T_{aux}. Then, multiply the expanded $[\![Q_i]\!]$ to the auxiliary table T_{aux}, where $0 \le i < m-2$. Then, we use `VectorSum` for each dimension to fill out all slots with the sum of slots, resulting in $[\![R_i]\!]$ whose matched dimension for i-th input are all 1s.

step 3) Further expansion of the expanded $[\![R_i]\!]$ generated in step 2 (`VectorExpand` part in line 7 to 9)
If the number of inputs is over two, we again expand each $[\![R_i]\!]$ to have the same size as the hypercube table T_{out}. We apply `VectorExpand` to every $k_{in} \cdot 2^{d \cdot i}$ rows of the generated expanded $[\![R_i]\!]$ in step 2, $2^{d(m-1)}$ times. This step is shown as step 3 in Figure 9 and 10.

step 4) Generation of $[\![R]\!]$ (line 10)
We slot-wise multiply all $\{[\![R_0]\!], ..., [\![R_{m-2}]\!]\}$ generated from steps 2 and 3 to obtain one hypercube $[\![R]\!]$ in which the matched data point with the input $\{c_0, ..., c_{m-2}\}$ is one, and the others are zero, where the size of $[\![R]\!]$ is the same as T_{out}.

step 5) Extraction of output (line 11)
We multiply the expanded $[\![Q_{m-1}]\!]$ and $[\![R]\!]$ to obtain a one-hot slot vector where only the matched data point is one and the other data points are zero, followed by multiplying with T_{out}. After that, we sum up each column for all rows to have the resultant ciphertext $[\![r]\!]$. Examples are shown in Figure 10.

step 6) Obtaining final result (line 12)
Finally, we use `PartialSum` [KSW+18] to fill all of the used slots with the resultant output.

```
Algorithm 6: $m$-input functions evaluation($\{[\![\vec{c}_{el=c_0}]\!], ..., [\![\vec{c}_{el=c_{m-1}}]\!]\}, T_{in}, T_{out}, T_{aux}, k_{in}, k_{out}, l, ptOne, t$)
  Input: $\{[\![\vec{c}_{el=c_0}]\!], ..., [\![\vec{c}_{el=c_{m-1}}]\!]\}$: the ciphertexts of inputs, where $m > 1$; $T_{in}, T_{out}$:
         the LUTs whose each row is packed as a plaintext; $T_{aux}$: a pre-prepared
         multidimensional vector whose each row is packed as a plaintext; $k_{out}$: the
         number of rows of $T_{out}$; $k_{in}$: the number of rows of $T_{in}$; $l$: the number of
         slots; $ptOne$: a plaintext of a vector whose all used elements are one; $t$:
         plaintext modulus
  Output: $r$: the ciphertext of the output result
 1 $[\![r]\!] \leftarrow []$ ;                                                ▷a ciphertext
 2 for $i = 0$ to $m - 1$ do
 3     for $j = 0$ to $k_{in} - 1$ do
 4         $[\![Q_i[j]]\!] \leftarrow$ One-HotSlot($[\![\vec{c}_{el=c_i}]\!], T_{in}[j], ptOne, t$);
 5     end
 6 end
 7 for $i = 0$ to $m - 2$ do
 8     $[\![R_i]\!] \leftarrow$ VectorSum(VectorExpand($[\![Q_i]\!] \otimes T_{aux}, k_{in} \cdot 2^{d \cdot i}, 2^{d(m-1)}$), $l, l'$);
 9 end
10 $[\![R]\!] \leftarrow \bigotimes [\![R_i]\!]$;
11 $[\![r]\!] \leftarrow \bigoplus_{i=0}^{k_{out}-1} [\![R[i]]\!] \otimes [\![Q_{m-1}[i \mod k_{in}]]\!] \otimes T_{out}[i]$;
12 return $[\![r]\!] \leftarrow$PartialSum($[\![r]\!], l, l'$);
```

4.5 Extension of Output Domain Size

We decompose the BigNum integers into small-bit integers as introduced in [LY21] to extend the domain sizes for both input and output. This method expands the output domain by adding a small latency.

Let the plaintext modulus be $2^w + 1 = t$ and $w < d$. We decompose the d-bit large integer a to w-bit small integers $\{a_0, ..., a_{n-1}\}$ as

$$a = a_0 + a_1 \times 2^w + ... + a_{n-1} \times 2^{(n-1)w} \tag{9}$$

, where $n = \lceil d/w \rceil$. We reconstruct the input LUT using d-bit integers and w-bit small integers. The output LUT with d-bit large integers are separated into subtables, each with w-bit small integers.

Consider a simple example that extends the 4-bit one-input function $f(x) = y$ to a 2-bit two-input function $f(x_0, x_1) = y$ to extend the output domain size. Note that the data points of the 4-bit one-input function are small to prepare lookup tables; however, we explain how to extend the domain size of the output lookup tables by decomposing the original output table into multiple output tables. For example, when the plaintext modulus $t = 5$, we can only store 2-bit integers that are smaller than 5. We evaluate a 4-bit one-input function $f(x) = y$ as a 2-bit two-input function using a single T_{in} holding 2^2 data points and two T_{out}s, each of which holds 2^4 data points, as shown in Figure 11. Each of the 4-bit output data points is decomposed into two 2-bit data points and stored in two LUTs with the same index. In the example in Figure 11, the input is $x = x_0 + 2^2 \times x_1 = 2 + 2 \times 2^2 = 10$ which is decomposed into $x_0 = 2$ and $x_1 = 2$.

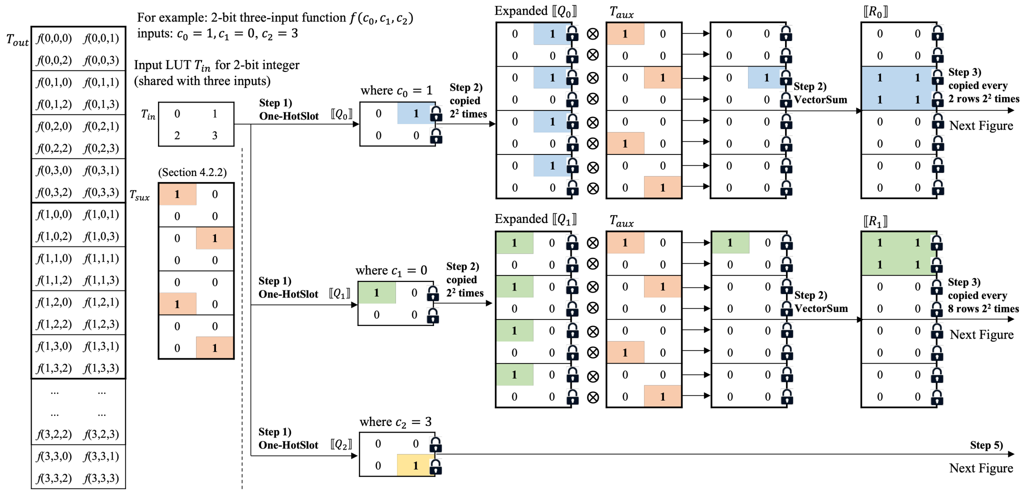


Figure 9: An example of LUT processing for 2-bit three-input function evaluation with $f(1, 0, 3)$ (from step 1 to step 3)

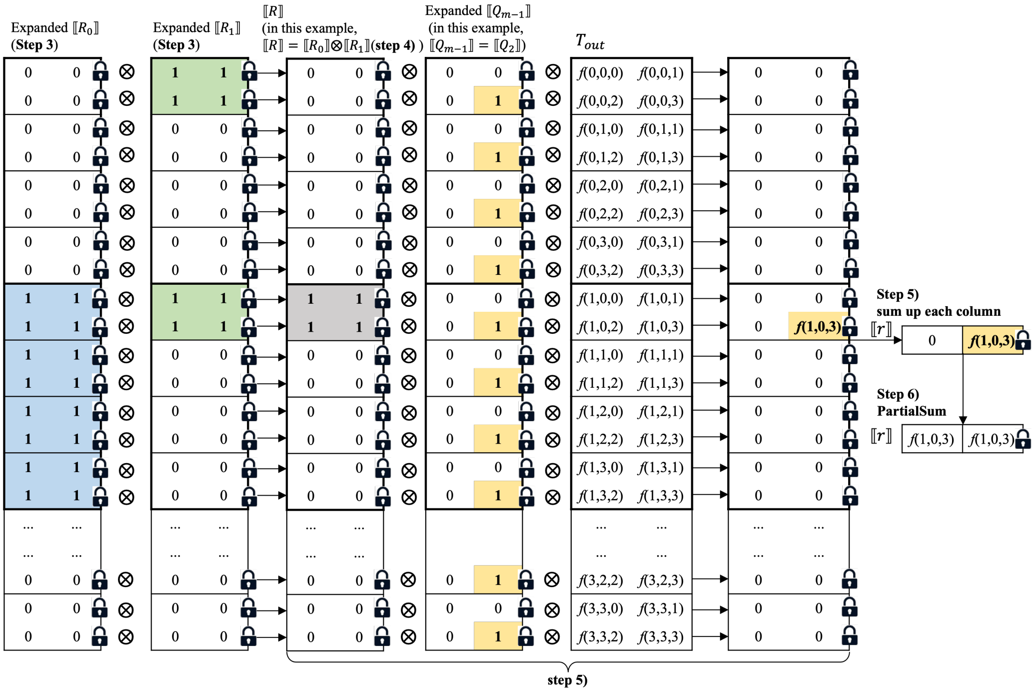
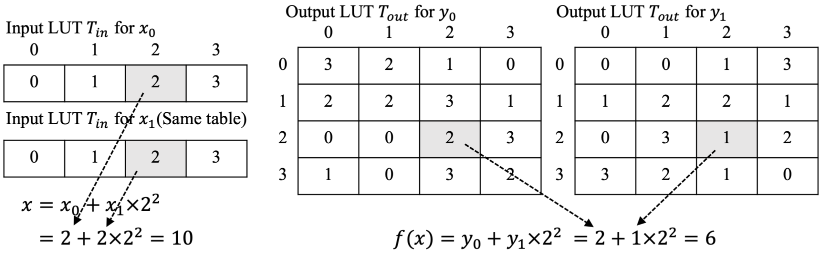


Figure 10: An example of LUT processing for 2-bit three-input function evaluation with $f(1, 0, 3)$ (from step 3 to step 6)

The output $f(10) = y_0 + y_1 \times 2^2 = 2 + 1 \times 2^2 = 6$ is decomposed to $y_0 = 2$ and $y_1 = 1$. Therefore, we can evaluate BigNum integers as a multi-input function with each input having a small plaintext space. As the indices of the corresponding decomposed outputs in each subtable are the same, we extract each decomposed output and recompose the output after decryption.



Input LUT T_{in} for x_0

0	1	2	3
0	1	2	3

Input LUT T_{in} for x_1(Same table)

0	1	2	3
0	1	2	3

Output LUT T_{out} for y_0

	0	1	2	3
0	3	2	1	0
1	2	2	3	1
2	0	0	2	3
3	1	0	3	2

Output LUT T_{out} for y_1

	0	1	2	3
0	0	0	1	3
1	1	2	2	1
2	0	3	1	2
3	3	2	1	0

Figure 11: An example of LUTs for BigNum function evaluation

5 Complexity Analysis

This section presents the computational complexities of the algorithms mentioned above and the total complexity of the entire processing. In the following calculation, we ignore the number of plaintext calculations because ciphertext computations require a runtime that is more than 100 times longer than that of plaintext calculations. Thus, the following complexities represent the order of ciphertext calculations:

The `One-HotSlot` algorithm requires two subtractions (additions) between ciphertext and plaintext ($ct + pt$) and $\log_2(t-1)$ times multiplications between ciphertext and ciphertext ($ct \times ct$), where t is the plaintext modulus. Ignoring plaintext calculation, the complexity is $O(log_2(t-1))$. The `PartialSum` algorithm requires $\log_2 l'$ times rotations and additions between ciphertext and ciphertext, where l' is the number of slots used. The number of slots used is $l' = 2^d$ for d-bit input if $2^d \leq l$, and $l' = l$ if $2^d > l$. Its complexity is $O(\log_2 l')$. The `VectorSum` algorithm requires k times `PartialSum` and a one-time addition of ciphertexts ($ct + ct$), where k is the number of ciphertexts in the encrypted matrix. Its complexity is $O(k \cdot \log_2 l')$. The `VectorExpand` algorithm requires no addition or multiplication over ciphertext. The complexity is $O(k \cdot n)$ where k denotes the number of ciphertexts in the encrypted matrix and n denotes the required expansion time. During the processing of d-bit m-input function evaluation, the complexity is $O(2^{d \cdot m}/l)$ (see Algorithm 6 in Section 4.4), and the multiplication depth is $\log_2(t-1) + m$, where m is the number of inputs. Table 2 summarizes the complexity of each algorithm, ignoring the computations over plaintexts in Table 2. Note that if the output LUT has one row, the complexity is $O(log_2(t-1))$ but not $O(2^{d \cdot m}/l)$.

Table 2: Algorithm complexity

Algorithm	Complexity
`One-HotSlot`	$O(log_2(t-1))$
`PartialSum`	$O(\log_2 l')$
`VectorSum`	$O(k \cdot \log_2 l')$
`VectorExpand`	$O(k \cdot n)$
Entire processing	$O(2^{d \cdot m}/l)$

6 Experiment Evaluation

In this section, we present the experimental evaluation. We implemented our proposed method for one-input and multi-input functions to confirm the runtime with different input and output bit lengths.

We implemented the proposed method using the OpenFHE library [1] v1.1.4 [BBB+22]. The source code is available from https://github.com/ruixiaoLee/FunctionEval-FHE-LUT. The machine with Ubuntu 20.04.6 LTS (Focal Fossa) OS has an Intel(R) Xeon(R) Gold 5220R 2.20GHz CPU (24 cores, 48 threads) and 60 GB memory. The compiler used is GUN 9.4.0 with CMake 3.16.3. The multithreading technique is adopted by using OpenMP 4.5.

Both BFV and BGV schemes can be adapted to the proposed method. In the experiment, we used the BFV scheme and set the plaintext modulus to $t = 2^{16} + 1 = 65537$, the security level to `HESd_128_classic`, and the others to the default parameters in the library. For the one-, two-, and three-input function evaluation, the multiplicative depth is 17, 18, and 19, respectively. The following results, i.e., the runtimes of the function evaluations, are the averages of five times runs.

6.1 Runtime and memory usage of d-bit one-input functions

Table 3 presents the evaluation runtimes and memory usage from receiving the input to extracting the output of one input function using one thread. Because the plaintext modulus is $t = 2^{16} + 1$, the maximum input domain is 16-bit. With the aforementioned parameters, the number of slots is $32,768$, which implies that if the input domain size is less than 16-bit, the LUTs have only one row, resulting in execution with one thread. The results show that the runtimes of different numbers of bits increase by less than one second until the domain size reaches 15 bits because all data points are handled by one ciphertext, that is, one row. However, when the domain size exceeds 15-bit, the number of rows increases to two, and the execution time increases by approximately 1.85 times for 16-bit input when using one thread. The memory usage increases linearly with the number of bits from 1 to 15; however, the increase is slight. Meanwhile, the runtime is almost the same because the number of rows in LUT stays the same, i.e., one when using 1 to 15 bits. The difference when varying the number of bits is the last step to fill all used slots with the sum of all slots, where the `PartialSum` algorithm is used. The smaller number of bits leads to fewer used slots, which needs a smaller number of rotations and additional operations in `PartialSum`. The maximum memory usage of 15-bit and 16-bit one-input functions increased but not as much as runtime because we used one thread, and both used all slots but differed by only one row in size.

Table 3: Evaluation results of runtime and memory usage for one-input functions

	# Bit							
	1	2	3	4	5	6	7	8
Time [s]	4.011	3.981	4.036	4.080	4.124	4.169	4.219	4.277
Mem. Usage [MB]	199	259	320	381	441	501	562	622
	# Bit							
	9	10	11	12	13	14	15	16
Time [s]	4.317	4.362	4.408	4.445	4.487	4.535	4.583	8.459
Mem. Usage [MB]	683	744	804	865	925	986	1,046	1,057

---

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

6.2 Runtime and memory usage of d-bit multi-input functions

Table 4 presents the runtime results for d-bit two-input function evaluation. Table 6 presents the runtime results of d-bit three-input function evaluation. The runtime is significantly affected by the size of the output LUT. Because we set the number of slots to be the same in all experiments, the runtime is highly affected by the number of rows k_{out}. The result using one thread shows we evaluate 10-bit two-input functions by our protocol within 748.9 s, and 12-bit two-input functions cost 3,446.1 s. Using 16 threads reduces the runtime to 90.5 and 404.7 s, respectively. Evaluating 5-bit (6-bit) three-input functions requires 895.9 s (3,963.8 s) with one thread. We reduced the runtime to 105.5 and 449.5 s, respectively, using 16 threads.

With the same number of threads, the runtime and memory usage exponentially increase with the number of bits, because the size of the data point in LUTs increases exponentially. In our experiments, the maximum memory usage is approximately 54,848 MB (53.6 GB) for 12-bit two-input functions using 16 threads.

Table 4: Evaluation results of runtime for two-input functions [s]

# Thread	# Bit				
	2	3	4	5	6
1	10.367	11.939	15.579	23.770	41.877
4	6.251	6.994	7.888	10.308	15.822
8	4.749	4.958	5.934	7.874	10.905
16	4.659	4.859	5.191	6.555	9.738
# Thread	# Bit				
	7	8	9	10	12
1	81.021	167.491	352.068	748.940	3446.110
4	26.639	50.502	102.534	215.154	970.995
8	18.441	37.706	69.817	153.349	683.537
16	16.108	24.871	51.034	90.503	404.749

Table 5: Evaluation results of memory usage for two-input functions [MB]

# Thread	# Bit				
	2	3	4	5	6
1	353	477	653	933	1,422
4	446	585	829	1,010	1,518
8	448	614	812	1,170	1,768
16	513	755	1,368	1,704	1,929
# Thread	# Bit				
	7	8	9	10	12
1	2,327	4,064	7,466	14,198	54,304
4	2,490	4,198	7,569	14,264	54,448
8	2,753	4,367	7,740	14,483	54,531
16	2,870	4,753	8,264	14,906	54,848

In addition, we list the LUT size for each experiment in Table 8 and the runtime of the primitive operations provided in OpenFHE v1.1.4 in Table 9.

The LUT size increases exponentially with the number of bits because the number of data points in input and output LUT is $|T_{in}| + |T_{out}| = 2^d + 2^{m \cdot d}$. However, the LUTs are all plaintexts, which benefits the LUT size by not being too large; thus, the maximum table size in our experiment is 557.1 MB for 12-bit two-input functions.

For the runtime of the primitive operations, we vary the multiplicative depth while keeping the other parameters the same as those in the other experiments. Each result shows the average runtime for 1,000-time evaluations in Table 9.

Table 6: Evaluation results of runtime for three-input functions [s]

# Thread	# Bit				
	2	3	4	5	6
1	22.235	55.007	206.645	895.922	3,963.782
4	11.841	20.108	62.758	252.663	1,114.618
8	8.271	15.328	45.273	160.304	701.309
16	7.407	11.898	28.931	105.469	449.488

Table 7: Evaluation results of memory usage for three-input functions [MB]

# Thread	# Bit				
	2	3	4	5	6
1	482	1,024	2,949	10,378	39,764
4	650	1,162	3,131	10,579	39,878
8	663	1,319	3,240	10,621	40,099
16	1,312	1,720	3,668	11,232	40,420

Table 8: LUT size of our work [MB]

One-input							
# Bit	10	11	12	13	14	15	16
Size	0.129	0.135	0.146	0.170	0.230	0.355	0.771
Two-input							
# Bit	2	4	6	8	9	10	12
Size	0.564	2.264	8.264	34.064	66.064	131.064	557.073
Three-input							
# Bit	2	3	4	5	6		
Size	1.413	4.663	18.164	67.164	261.164		

7 Comparison and Discussion

To confirm the performance of our proposed method, we compared the runtime with 1) the word-wise LUT method [OCHK18] [MMN22] and 2) the naive bit-wise LUT method [CGH$^+$18].

7.1 Comparsion with word-wise LUT work

We implemented Okada et al.'s method [OCHK18] and Maeda et al.'s method [MMN22] using the BFV scheme in the OpenFHE [BBB$^+$22] library. The parameters are set to be the same as those in our study, but the multiplication depth is set to 17, which is lower than ours. This is because the minimum number of multiplications required in [OCHK18] and [MMN22] are smaller than that required by the proposed method.

The one-input function evaluation method [MMN22] is nearly the same as ours but requires a one-dimensional LUT. In [OCHK18] and [MMN22], the input domain size is set to N, where $N \leq (t-1)/2$ and t is plaintext space. We set the same plaintext space

Table 9: Runtime of primitive operation [ms]

Depth	$ct \times ct$	$ct + ct$	$ct \times pt$	$ct + pt$
1	7.938	0.102	0.374	0.561
17	169.483	2.355	5.714	4.961
18	166.078	2.198	6.367	5.373
19	169.597	2.529	6.317	5.337

as that used in the experiment in [MMN22], which is $2^{16} + 1$, implying that the input domain size must be less than 15-bit, while our proposed method can reach 16-bit with a two-dimensional LUT. The runtime results for a single thread are presented in Table 3.

The runtimes of different input domain sizes of two-input function evaluation by using [OCHK18] and [MMN22] with a single thread are shown in Table 10. Note that we used a single thread for a fair comparison.

Okada et al.'s scheme [OCHK18] consumed 4.1 s for evaluating a 2-bit two-input function, while a 10-bit two-input function needs 24,653.8 s. The complexity of [OCHK18] is $O(N^2)$, where N is the domain size of both the input and the output. Even when the multiplication depth increases linearly, the latency increases rapidly as the number of bits increases. Our study achieved faster runtime results when the bit length $d > 4$. Maeda et al.'s scheme [MMN22] consumed 43.8 s to evaluate a 10-bit two-input function, whereas a 12-bit two-input function needs 103.0 s. The complexity of [MMN22] is $O(N)$, where N is the domain size of both the input and the output. The result of our study shows that evaluating a 10-bit or 12-bit two-input function needs 748.9 and 3,446.1 s, respectively.

The results in Table 10 show that the runtime of our method for a two-input function is worse than [MMN22] but better than [OCHK18] (when the bit length $d > 4$). However, our method allows us to evaluate an arbitrary function $\overbrace{\mathbb{Z}_N \times \ldots \times \mathbb{Z}_N}^{m} \to \mathbb{Z}_{n \cdot N}$ whose input is more than two and expands the output domain size with integer-decomposing and table separation method, where N is the input domain size, m is the number of inputs, and n is any constant.

Table 10: Runtime comparison with [OCHK18, MMN22] for two-input functions [s]

Method	# Bit					
	2	4	6	8	10	12
[OCHK18]	4.085	21.954	154.189	1,711.983	24,653.800	-
Ours	10.367	15.579	41.877	167.491	748.940	3,446.110
[MMN22]	6.250	8.509	12.876	21.381	43.757	102.971

7.2 Compare with naive bit-wise LUT implementation

Okada et al. [OCHK18] compared the function evaluation latency of their work with those of Chen et al. [CG15], Xu et al. [XCWF16], and Chen et al. [CFLW17], who used bit-wise FHE. The experimental results in [OCHK18] demonstrate that their work is faster than [CFLW17], which is the fastest bit-wise implementation mentioned in their study.

As a complement, we implement the naive LUT method [CGH+18] for m-input function with a bit-wise FHE scheme, FHEW/TFHE [MP21, CGGI20], using the OpenFHE library. We used the default parameters and `STD128` security level. As explained in Section 2.2, the number of multiplications in the naive bit-wise LUT is $(m \cdot d + 1)$ for a d-bit m-input function. Because we can combine m d-bit inputs into a single $(m \cdot d)$-bit input to evaluate using the bit-wise method. The required multiplication depth is $(\log_2(t-1) + m)$ in this work, where m is the number of inputs and t is the plaintext space satisfied $t > 2^d$. Thus, when the plaintext space satisfies $t > 2^{m(d-1)} + 1$, our required depth is larger than the naive bit-wise LUT.

Besides using the word-wise HE to encrypt each integer in this work, we plan to use word-wise HE to encrypt each bit of an integer in the future. The equality function in this work is shown as Equation 6, which requires depth to be $\log_2(t-1)$. We set a large plaintext space even for SmallNum in our experiments, which leads to a depth larger than d. However, if $\vec{x} = (x_1, ..., x_d)$ is the bit vector with binary expansion of x and similarly $\vec{a} = (a_1, ..., a_d)$ is the binary expansion of y, then the equality function can be computed as $\prod_{i=0}^{d}(a_i + x_i + 1)$. In this case, we can reduce the required depth to d.

Table 11 shows the runtime results of one-, two-, and three-input functions using the naive LUT method with bit-wise FHE for different bit-lengths of input and output. In this experiment, we set the same number of bits for both input and output. The results show that evaluating a 3-bit three-input function requires approximately 1,799.9 s. By using 16 threads, the runtime decreases to 275.3 s. While our proposed method requires 11.9 s, which is approximately 23 times faster.

Table 11: Runtime of naive LUT method with bit-wise FHE [s]

# of	One-input						
Thread	1-bit	2-bit	3-bit	4-bit	5-bit	6-bit	7-bit
1	1.936	5.214	13.743	35.073	86.256	205.347	478.276
4	1.121	2.635	6.015	14.406	34.599	81.697	189.501
8	1.122	2.591	4.663	10.829	25.605	60.088	138.932
16	1.121	2.595	4.664	9.171	21.448	50.109	116.223

# of	Two-input				Three-input		
Thread	1-bit	2-bit	3-bit	4-bit	1-bit	2-bit	3-bit
1	4.388	28.270	164.600	872.765	10.399	150.602	1,799.947
4	2.025	10.392	57.844	304.036	4.048	49.687	586.421
8	2.022	7.285	38.795	203.460	2.889	31.690	370.709
16	2.022	5.734	30.307	158.640	2.904	23.665	275.339

8 Conclusion

We propose a non-interactive privacy-preserving function evaluation model to evaluate functions with a pre-prepared auxiliary table and input and output data-point tables using the word-wise LUT method. The input and output domain sizes are extended to $\overbrace{\mathbb{Z}_N \times ... \times \mathbb{Z}_N}^{m} \to \mathbb{Z}_{n \cdot N}$, where N is the input domain size within the plaintext space, m is the number of inputs, and $n \in \mathbb{Z}^+$. To the best of our knowledge, our method is the first protocol that allows the evaluation of arbitrary multivariate functions using word-wise FHE. Our proposed LUT method is adaptable to any function with accurate input and output tables, delivering highly accurate results even for noncontiguous functions with a wider input range than polynomial approximation methods. Consequently, our protocol can enhance the application of FHE, enabling complicated functions in real-world scenarios where FHE has previously been challenging, such as privacy-preserving anomaly detection systems in smart grids [LBDY22]. The experimental results show that we evaluated a 15-bit one-input function within 4.6 s and a 16-bit one-input function within 8.5 s. The 10-bit two-input function requires 90.5 s, and the 5-bit three-input function requires 105.5 s with 16-thread. Compared to a naive implementation with bit-wise LUT, we decreased the latency by approximately 3.2 and 23.1 times for evaluating two-bit and three-bit 3-input functions using 16-thread, respectively.

References

[BBB+22] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022*, pages 53–63. ACM, 2022. doi:10.1145/3560827.3563379.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: combining ring-lwe-based fully homomorphic encryption schemes. *J. Math. Cryptol.*, 14(1):316–338, 2020. URL: https://doi.org/10.1515/jmc-2019-0026, doi:10.1515/JMC-2019-0026.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013: 16th International Conference on Theory and Practice of Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 1–13, Nara, Japan, February 26 – March 1, 2013. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-642-36362-7_1.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014. doi:10.1145/2633600.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society. doi:10.14722/ndss.2015.23241.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-642-32009-5_50.

[CBK+20] Mahawaga Arachchige Pathum Chamikara, Peter Bertók, Ibrahim Khalil, Dongxi Liu, and Seyit Camtepe. Privacy preserving face recognition utilizing differential privacy. *Comput. Secur.*, 97:101951, 2020. URL: https://doi.org/10.1016/j.cose.2020.101951, doi:10.1016/J.COSE.2020.101951.

[CBL+18] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *CoRR*, abs/1811.09953, 2018. URL: http://arxiv.org/abs/1811.09953, arXiv:1811.09953.

[CDH+19] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj S. Katti, Anderson C. A. Nascimento, Wing-Sea Poon, and Stacey Truex. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IEEE Trans. Dependable Secur. Comput.*, 16(2):217–230, 2019. doi:10.1109/TDSC.2017.2679189.

[CdWM+17] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. URL: https://eprint.iacr.org/2017/035.

[CFLW17] Jingwei Chen, Yong Feng, Yang Liu, and Wenyuan Wu. Faster binary arithmetic operations on encrypted integers. In *The 7th International Workshop on Computer Science and Engineering, Beijing, 25-27 June, 2017, Proceedings*, pages 956–960, 2017. doi:10.18178/wcse.2017.06.166.

[CG15] Yao Chen and Guang Gong. Integer arithmetic over ciphertext and homomorphic data aggregation. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 628–632. IEEE, 2015. doi:10.1109/CNS.2015.7346877.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

[CGH+18] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. Doing real work with FHE: the case of logistic regression. In Michael Brenner and Kurt Rohloff, editors, *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018, Toronto, ON, Canada, October 19, 2018*, pages 1–12. ACM, 2018. doi:10.1145/3267973.3267974.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 106–126, San Francisco, CA, USA, March 4–8, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-12612-4_6.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland. doi:10.1007/978-3-319-70694-8_15.

[CKP22] Jung Hee Cheon, Wootae Kim, and Jai Hyun Park. Efficient homomorphic evaluation on large intervals. *IEEE Trans. Inf. Forensics Secur.*, 17:2553–2568, 2022. doi:10.1109/TIFS.2022.3188145.

[CMTB16] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin R. B. Butler. Secure outsourced garbled circuit evaluation for mobile devices. *J. Comput. Secur.*, 24(2):137–180, 2016. doi:10.3233/JCS-150540.

[CT12] Michael A. Cohen and Can Ozan Tan. A polynomial approximation for arbitrary functions. *Appl. Math. Lett.*, 25(11):1947–1952, 2012. URL: https://doi.org/10.1016/j.aml.2012.03.007, doi:10.1016/J.AML.2012.03.007.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 789–800, Berlin, Germany, November 4–8, 2013. ACM Press. doi:10.1145/2508859.2516701.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, April 26–30, 2015. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-662-46800-5_24.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. URL: https://eprint.iacr.org/2012/144.

[GCH⁺18] Chong-zhi Gao, Qiong Cheng, Pei He, Willy Susilo, and Jin Li. Privacy-preserving naive bayes classifiers secure against the substitution-then-comparison attack. *Inf. Sci.*, 444:72–88, 2018. URL: https://doi.org/10.1016/j.ins.2018.02.058, doi:10.1016/J.INS.2018.02.058.

[GDL⁺16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.html.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press. doi:10.1145/1536414.1536440.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-642-40041-4_5.

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-662-44371-2_31.

[HTG19] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Deep neural networks classification over encrypted data. In Gail-Joon Ahn, Bhavani Thuraisingham, Murat Kantarcioglu, and Ram Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 97–108. ACM, 2019. doi:10.1145/3292006.3300044.

[jLHH+21] Wen jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy*, pages 1057–1073, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press. `doi:10.1109/SP40001.2021.00043`.

[KSW+18] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Med Inform*, 6(2):e19, Apr 2018. URL: `http://medinform.jmir.org/2018/2/e19/`, `doi:10.2196/medinform.8805`.

[LBDY22] Ruixiao Li, Shameek Bhattacharjee, Sajal K. Das, and Hayato Yamana. Look-up table based FHE system for privacy preserving anomaly detection in smart grids. In *2022 IEEE International Conference on Smart Computing, SMARTCOMP 2022, Helsinki, Finland, June 20-24, 2022*, pages 108–115. IEEE, 2022. `doi:10.1109/SMARTCOMP55677.2022.00030`.

[LDL15] Hongwei Li, Yuanshun Dai, and Xiaodong Lin. Efficient e-health data release with consistency guarantee under differential privacy. In *17th International Conference on E-health Networking, Application & Services, HealthCom 2015, Boston, MA, USA, October 14-17, 2015*, pages 602–608. IEEE, 2015. URL: `https://doi.org/10.1109/HealthCom.2015.7454576`, `doi:10.1109/HEALTHCOM.2015.7454576`.

[LLNK22] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Trans. Dependable Secur. Comput.*, 19(6):3711–3727, 2022. `doi:10.1109/TDSC.2021.3105111`.

[LY21] Ruixiao Li and Hayato Yamana. Fast and accurate function evaluation with LUT over integer-based fully homomorphic encryption. In Leonard Barolli, Isaac Woungang, and Tomoya Enokido, editors, *Advanced Information Networking and Applications - Proceedings of the 35th International Conference on Advanced Information Networking and Applications (AINA-2021), Toronto, ON, Canada, 12-14 May, 2021, Volume 2*, volume 226 of *Lecture Notes in Networks and Systems*, pages 620–633. Springer, 2021. `doi:10.1007/978-3-030-75075-6\_51`.

[LY24] Ruixiao Li and Hayato Yamana. Privacy preserving function evaluation using lookup tables with word-wise fhe. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E107-A(8):1–15, 2024. `doi:10.1587/transfun.2023EAP1114`.

[MMN22] Daisuke Maeda, Koki Morimura, and Takashi Nishide. Efficient homomorphic evaluation of arbitrary bivariate integer functions. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022*, pages 13–22. ACM, 2022. `doi:10.1145/3560827.3563378`.

[MP21] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*, pages 17–28. WAHC@ACM, 2021. `doi:10.1145/3474366.3486924`.

[MRVW21] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Díaz, editors, *FC 2021: 25th International Conference on Financial Cryptography and Data Security, Part I*, volume 12674 of *Lecture Notes in Computer Science*, pages 249–270, Virtual Event, March 1–5, 2021. Springer, Berlin, Heidelberg, Germany. doi:10.1007/978-3-662-64322-8_12.

[OCHK18] Hiroki Okada, Carlos Cid, Seira Hidano, and Shinsaku Kiyomoto. Linear depth integer-wise homomorphic division. In Olivier Blazy and Chan Yeob Yeun, editors, *Information Security Theory and Practice - 12th IFIP WG 11.2 International Conference, WISTP 2018, Brussels, Belgium, December 10-11, 2018, Revised Selected Papers*, volume 11469 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2018. doi:10.1007/978-3-030-20074-9_8.

[SV14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014. doi:10.1007/s10623-012-9720-4.

[XBF$^+$14] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin E. Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *CoRR*, abs/1412.6181, 2014. URL: http://arxiv.org/abs/1412.6181, arXiv:1412.6181.

[XCWF16] Chen Xu, Jingwei Chen, Wenyuan Wu, and Yong Feng. Homomorphically encrypted arithmetic operations over the integer ring. In Feng Bao, Liqun Chen, Robert H. Deng, and Guojun Wang, editors, *Information Security Practice and Experience - 12th International Conference, ISPEC 2016, Zhangjiajie, China, November 16-18, 2016, Proceedings*, volume 10060 of *Lecture Notes in Computer Science*, pages 167–181, 2016. doi:10.1007/978-3-319-49151-6_12.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press. doi:10.1109/SFCS.1982.38.

[ZC20] Xu Zheng and Zhipeng Cai. Privacy-preserved data sharing towards multiple parties in industrial iots. *IEEE J. Sel. Areas Commun.*, 38(5):968–979, 2020. doi:10.1109/JSAC.2020.2980802.

[ZTG$^+$19] Maede Zolanvari, Marcio Andrey Teixeira, Lav Gupta, Khaled M. Khan, and Raj Jain. Machine learning-based network vulnerability analysis of industrial internet of things. *IEEE Internet Things J.*, 6(4):6822–6834, 2019. doi:10.1109/JIOT.2019.2912022.