Tree-based Lookup Table on Batched Encrypted Queries using Homomorphic Encryption

Jung Hee Cheon, *Member, IEEE,* Hyeongmin Choe, Jai Hyun Park

Abstract—**Homomorphic encryption (HE) is in the spotlight as a solution for privacy-related issues in various real-world scenarios. However, the limited types of operations supported by each HE scheme have been a major drawback in applications. While HE schemes based on learning-with-error (LWE) problem provide efficient lookup table (LUT) evaluation in terms of latency, they have downsides in arithmetic operations and low throughput compared to HE schemes based on ring LWE (RLWE) problem. The use of HE on circuits containing LUT has been partly limited if they contain arithmetic operations or their computational width is large.**

In this paper, we propose homomorphic algorithms for batched queries on LUTs by using RLWE-based HE schemes. To look up encrypted LUTs of size n on encrypted queries, our algorithms use $O(\log n)$ homomorphic comparisons and $O(n)$ multiplications. For unencrypted LUTs, our algorithms use $O(\log n)$ comparisons, $O(\sqrt{n})$ ciphertext multiplications, and $O(n)$ scalar multiplications.

We provide a proof-of-concept implementation based on CKKS scheme (Asiacrypt 2017). The amortized running time for an encrypted (Resp. unencrypted) LUT of size 512 is 0.041 (Resp. 0.025) seconds. Our implementation reported roughly 2.4-6.0x higher throughput than the current implementation of LWE-based schemes, with more flexibility on the structure of the LUTs.

Index Terms—**Homomorphic Encryption, Homomorphic Operations, Lookup Table**

I. Introduction

Homomorphic encryption (HE) is a cryptographic encryption that supports homomorphic operations between ciphertexts without a decryption process. One of the interesting applications of HE is securing machine learning (ML) algorithms. In the scenario of machine-learning-as-a-service (MLaaS), the clients send private queries to the untrusted computing server and receive the results of ML algorithms. The clients can privatize their private queries by utilizing HE as following general framework: (a) the client sends homomorphically encrypted queries (and public key if needed), (b) the server computes homomorphic ML algorithms on the encrypted queries, and (c) the client receives and decrypts the encrypted result of the homomorphic computation to obtain the result.

The efficiency of the above framework highly depends on the feature of the homomorphic ML algorithm and underlying HE scheme. In particular, the appropriate choice of HE scheme and a decent interpretation of the target ML algorithm for the HE scheme is important. The mainstream fully HE (FHE) schemes rely on lattice problems and have ciphertexts of either learning-with-errors (LWE) [1], [2] or ring learning-with-errors (RLWE) format [3], [4], [5]. LWE-based schemes are usually preferable for non-arithmetic circuits with small computational widths, while RLWE-based schemes are mostly advantageous for arithmetic circuits with large computational widths. An inappropriate choice of (F)HE scheme can induce the overhead while compiling the ML algorithms with underlying HE operations.

However, some HE applications require the strengths of both HE formats; in particular, a circuit might consist of LWE-friendly operational types but with a large computational width (in which RLWE is preferable). As various ML algorithms with complicated non-arithmetic functions are devised, and ML evaluations are often being batched for enhancing efficiency, such requirements are becoming more in demand. As a specific example, a company might want to send clients' private ID numbers to the server of a government agent, asking for the result of ML algorithms on the information (e.g., height and credit information) corresponding to each ID number. This requires the server to look up a lookup table on a number of ID cards, which requires non-arithmetic operation and a large computational width.

The appropriate HE scheme was unclear for the mixed cases. Bleach [6] partly addressed these mixed cases by showing that the RLWE-based CKKS scheme provides higher throughput than the LWE-based TFHE scheme for binary operations. However, it was still unclear how to choose an appropriate HE format for the case of LWE-friendly operations other than binary operations when computational widths are large.

In this work, we introduce secure lookup table evaluation algorithms based on RLWE FHE schemes. With our algorithms, a client can utilize RLWE FHE scheme to delegate evaluations of circuits (e.g., ML models) that contain lookup table evaluations. Also, we point out that all discrete non-arithmetic functions can be represented as a lookup table. This implies that we can evaluate general non-arithmetic functions, which is generally difficult for RLWE FHE schemes. This is beneficial to securely evaluate ML algorithms, as many ML models require evaluating complicated functions such

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2020-0-00840, Development and Library Implementation of Fully Homomorphic Machine Learning Algorithms supporting Neural Network Learning over Encrypted Data). All authors contributed equally, and the authors are listed in alphabetical order. *(Corresponding author: Jai Hyun Park)*

J. H. Cheon is with Department of Mathematical Sciences, Seoul National University and CryptoLab, South Korea. (e-mail:jhcheon@snu.ac.kr)

H. Choe is with Department of Mathematical Sciences, Seoul National University. (e-mail:sixtail528@snu.ac.kr)

J. H. Park is with Department of Mathematical Sciences, Seoul National University. (e-mail: jhyunp@snu.ac.kr)

as activation functions or loss functions as in [7]. As our algorithm uses RLWE FHE schemes, it is evident that the target circuits are well-suited to contain (RLWE-friendly) arithmetic operations.

A. Contribution

Secure evaluation of public lookup table. We suggest RLWE-based algorithms for secure batch evaluation of public lookup tables. Our best algorithm for the public lookup table utilizes a baby-step / giant-step approach to reduce the number of homomorphic multiplications between ciphertexts, which is a costly operation for HE. To evaluate a public lookup table of size n, our algorithm requires $\log n$ homomorphic comparisons, $3\sqrt{n} - \log n + O(1)$ homomorphic multiplications between ciphertexts, and $2n + O(1)$ homomorphic multiplications between a plaintext and a ciphertext. We implemented a proof-of-concept implementation of our algorithms with the CKKS scheme. It takes 70 milliseconds per query for the batch evaluation of public real-valued LUT of size 8192.

Secure evaluation of private lookup table. We also introduce RLWE-based algorithms for secure batch evaluation of private sorted lookup tables.[1] Our best algorithm requires $\log n$ homomorphic comparison and $2n + O(1)$ homomorphic multiplications to evaluate encrypted lookup table of size n. In our proof-of-concept implementation, it takes 41 milliseconds per query for the batch evaluation of public real-valued LUT of size 512. To the best of our knowledge, this is the first attempt to address encrypted lookup tables. [2]

Choice of HE schemes. In this work, we show that for batch lookup table evaluation, which has been believed to be an LWE-friendly operation, the RLWE-based CKKS scheme is able to provide higher throughput than LWE-based solutions. We remark that our method with RLWE HE schemes reports 2.4-6.0x higher throughput than the current implementation of LWE-based solutions. This might imply that the choice of FHE scheme for each application should be mostly determined by the computational width rather than its operational type.

B. Technical Overview

1) Motivation and high-level description: The basic idea is to perform a binary search in encrypted states. Suppose we are given sorted and encrypted table keys, and an encrypted query among the table keys. We split the table keys into two half blocks. By comparing the query to the median of given table keys, we can determine which half block the query belongs. We keep searching the query on the half block it belongs, and once find the exact position of the query, we can get the corresponding table value.

However, since the table keys and queries are encrypted, it is not trivial to how to keep searching on appropriate blocks in encrypted states. We first suggest two big approaches: (1) incrementally generating *one-hot indicator* that indicates which subblock the query belongs to among equally cut subblocks, and (2) repeatedly halving the lookup table while keeping the query and target in it. Then, we mix two approaches, introducing a hybrid one.

First approach: One-hot vector. For the first approach, we consider the one-hot ciphertext vector, $(\underline{c}_1, \underline{c}_2, \cdots, \underline{c}_m)$ such that $c_i = 1$ if the query belong to i th subblock among equally cut m subblocks, and $c_i = 0$ otherwise. Then, we can compute the median value of the subblock query belongs to as follows.

$$\underline{x}' = \sum_{i=1}^{m} \underline{c}_i \cdot \text{ (Median of } i \text{ th subblock)}$$

By comparing the query to $\underline{x}'$, we can generate one-hot ciphertext vector $(\underline{c}'_1, \cdots, \underline{c}'_{2m})$ for the equally cut $2m$ subblocks. More precisely,

$$\underline{c}'_{2i} = (1 - \texttt{Compare}(\text{query}, \underline{x}')) \cdot \underline{c}_i$$
$$\underline{c}'_{2i+1} = \texttt{Compare}(\text{query}, \underline{x}') \cdot \underline{c}_i.$$

Figure 1 describes our first approach.

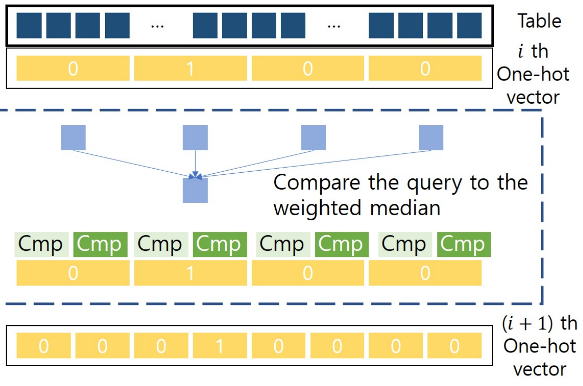
Table
i th One-hot vector
0
1
0
0
Compare the query to the weighted median
Cmp
0
1
0
0
$(i+1)$ th One-hot vector
0
0
0
1
0
0
0
0

Fig. 1. Visualization of the first approach.

This approach is similar to an existing work on the lookup table with CKKS [6]. However, [6] requires $O(n \log n)$ homomorphic multiplications for generating one-hot vector, while ours requires $O(n)$ homomorphic multiplications. If we apply the idea from [9] for efficiently computing the one-hot vectors to [6], the cost would be almost the same as our first approach. We will suggest two different approaches that show better performances than this first approach.

Second approach: Table folding. For the second approach, we halve the key table to perform a binary search in encrypted states. To be more precise, given a query and a key table that the query belongs to, we split the key table in half, and compute the halved key table containing the query by using the following equation. For given bit b and blocks X_1 and X_2,

$$(1-b)X_1 + bX_2 = \begin{cases} X_1 & \text{if } b = 0 \\ X_2 & \text{if } b = 1, \end{cases}$$

Once we let the bit b be the result of the homomorphic comparison between the query and the median value of current

[1]Even when given (encrypted) lookup table is not sorted, we can sort the table using [8]. We remark that once we sort a lookup table, a client can send multiple queries on the lookup table without additional sorting.

[2]We can modify existing works to partly tackle this problem; however, they still require more computation than our best algorithm.

table keys, then $(1-b)X_1 + bX_2$ is the half block the query belongs to. Figure 2 visualizes our second approach.

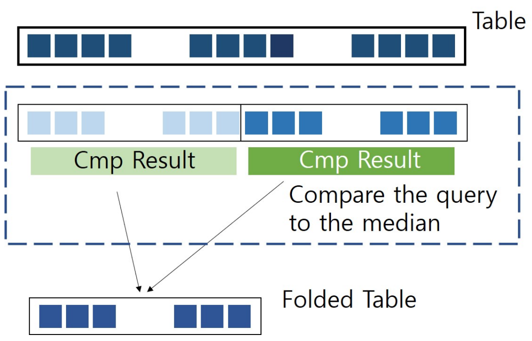

Fig. 2. Visualization of the second approach.

Our second approach requires less computation and memory than the first approach for both private and public lookup tables. We refer to Section III-A and III-B for the details.

Third approach: Hybrid. Finally, we combine two approaches to achieve an algorithm with a better performance. We observe that the first approach is faster in the earlier rounds while it is slower in the later rounds. On the other hand, the second approach is slower in the early rounds, but the iteration becomes faster as the rounds increase. Thus, our third approach is to combine the two approaches by starting with the first approach and ending with the second approach. Figure 3 illustrates the motivation of the third approach.

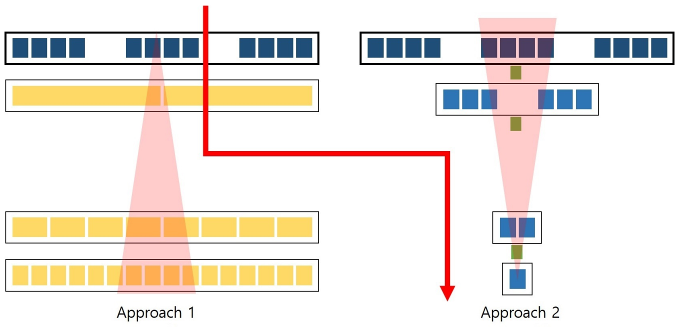

Fig. 3. Illustration of the motivation of the third approach.

In order to carefully connect the two approaches, we should perform supplementary operations. However, in some cases (e.g., when the table key values are public and unencrypted), we can perform the supplementary operations efficiently, and our third approach improves the efficiency. We refer to Section III-B for the details.

2) Adaptation to approximate computations: We adapt approximate FHE to apply our algorithms to applications with real-valued data. In particular, we consider CKKS scheme, which is a HE scheme that supports addition and multiplication between real-valued data, and it supports approximate computations. For example, the known homomorphic comparison methods for CKKS are approximate methods [10], [11], [12]. Real-valued computation, however, means that the computation is approximate upon certain precision bits. We stress that the cost of homomorphic comparison methods for approximate HE depends on its precision. Thus, it is important to analyze the precision of homomorphic comparison in algorithms.

In Section IV, we analyze the precision of homomorphic comparison methods that can be used for our algorithms to achieve correct results. In addition, we suggest a heuristic optimization to address bad lookup tables that require too intensive homomorphic comparisons.

C. Related Work

1) FHEW/TFHE based lookup table: In the recent approaches for FHEW and TFHE FHE schemes, the lookup table evaluation can be done during the bootstrapping procedures [1], [13], [14], which is often called *programmable bootstrapping* or *functional bootstrapping*. By using the whole (Resp. the half of) plaintext space $\mathbb{Z}_t$ as the keys for the lookup tables, the plaintext is replaced by the corresponding table value. The table keys are the integers from 0 to $t-1$ (Resp. $t/2-1$), thus uniform and public. As the plaintext space of the FHEW/TFHE scheme is a maximum of 8 bits, the lookup tables of larger sizes or larger precisions cannot be evaluated via programmable bootstrapping. We note that, even for such a small size of lookup tables with public keys, our third approach shows better throughput.

As the restriction on the table size and the precision is unavoidable for programmable bootstrapping of a single ciphertext, some tree-based approaches using multiple ciphertexts have recently been introduced in [15], [16]. Specifically in [16], with many input ciphertexts and with partly split tables, it first obtains the ciphertexts, each encrypting a bit of the query value for the lookup table evaluation. This can be viewed as a logarithmic number of comparisons between the query and the table keys, where the table keys are uniform and dense. After decomposing the bits, one of the split tables is chosen by using the encrypted bits. We note that this procedure is similar to our first approach but is evaluated in a very different way.

In both approaches, however, it is inefficient to deal with either sparse lookup tables (i.e., of which size is less than the bit-length of table keys) or private lookup tables. Hence, TFHE-based solutions are not satisfactory to our problem, the evaluation of lookup tables with (possibly) dense and (possibly) private real-valued table keys.

Also, we remark that TFHE/FHEW schemes might have a disadvantage in arithmetic operations compared to RLWE FHE schemes. This might be problematic if a client delegates the lookup table evaluation consecutive to arithmetic circuits. There have been several works on scheme switching [17], [18], [19] to address this issue, but they induce a computational overhead compared to solely RLWE FHE solutions.

2) BGV/BFV based lookup table: There have been several works on table lookup with integer HE schemes [20], [7] with bit-wise encodings. However, we stress that bit-wise encodings induce a computational overhead for (high-precision) arithmetic operations, as mentioned in [17]. In particular, [20] exploits bit-wise HE for the real arithmetic and takes 3.8 seconds to add two encrypted 8-bit integers. In contrast, CKKS scheme, which supports arithmetic operations on encrypted real numbers, takes less than a second to add 65536 pairs of real numbers in our implementation. This is a downside of bit-wise encodings for real-world applications with real-valued data.

Also, we point out that, as in TFHE/FHEW solutions, bit-wise HE solutions are bounded by lookup tables of which the size is equal to the bit-length of table keys. Once evaluating a lookup table smaller than the bit-length of its table keys, bit-wise HE schemes are not preferable.

3) CKKS based lookup table: Bleach [6] also suggested a homomorphic evaluation method of a (public) lookup table. For a given query, it first extracts the (encrypted) binary representation of the query. Then, it generates a one-hot vector using the bits and completes the evaluation of the lookup table by inner product.

Their approach is similar to our first approach, Algorithm 2 in Section III-B. However, it requires $n \log n$ homomorphic multiplications to generate the one-hot vector, in which n is the size of the lookup table. In contrast, ours uses $\Omega(n)$ homomorphic multiplications to generate the one-hot vector. Also, as in TFHE/FHEW and BGV/BFV solutions, we remark that Bleach considers only the dense lookup tables (i.e., the size of the lookup table equals the bit-length of each query), while our interest includes more general lookup tables, e.g., sparse lookup tables and encrypted lookup tables. Bleach also focuses on public lookup tables, while we address both private and public lookup tables in this work.

Recently, [9] suggests an advanced way to generate a one-hot map. Applying it to Bleach turns out to be a similar method to our first approach (Algorithm 2). Nevertheless, it still is bounded by dense and unencrypted lookup tables. Also, we remark that our third approach (Algorithm 3) shows a better performance for public lookup tables than Algorithm 2. For private lookup tables, the second approach (Algorithm 1) is better than the first.

A recent paper [21] introduced a tree-based approach for the range search problem. To solve the range search problem, it traverses a (encrypted) tree by using a copy-and-recurse technique similar to our second approach. Once applying the copy-and-recurse to the lookup table problem with optimizations, we can obtain a similar algorithm to Algorithm 1 for encrypted lookup tables. In contrast to this work, [21] focuses on the range search problem using encrypted trees, while our focus is on lookup tables, which might or might not be encrypted.

4) Private information retrieval: Private information retrieval (PIR) is a problem in which a user aims to retrieve an item from a server while hiding which item is retrieved. Our methods also give instant solutions for PIR. However, we stress that our setting is different from PIR. The distinction between our method and PIR is that (1) we search the lookup table by the key query rather than the index query, (2) we expect both the input and output of the algorithm to be HE ciphertexts with the same format, to easily utilize our LUT algorithm inside larger homomorphic circuits, and (3) our homomorphic algorithm is non-interactive; that is, our algorithm does not require any interaction except the initial query of the user and the final response of the server. Our problem and algorithms are more specific than general PIR and are more likely to be adopted for HE applications.

II. Preliminaries

A. Notations

Throughout this paper, we denote the encryption of a message m as $\underline{m}$. Also, for ease of discussion, we assume that we are given a lookup table consisting of $n = 2^k$ pairs of input and output values: $\{x(i), y(i) = f(x(i))\}_{i=0}^{n-1}$ where n is a power of 2. We call $x(i)$ as key data and $y(i)$ as value data of the lookup table.

For the bits $b_1, b_2, \cdots, b_k$, the binary representation of i, i.e., $i = \sum_{j=1}^{k} b_j 2^{k-1-j}$, we abuse the notation as:

$$x(i) = x(\vec{i}) = x(b_1, \cdots, b_k)$$
$$y(i) = y(\vec{i}) = y(b_1, \cdots, b_k).$$

A query is a ciphertext of an input value $\underline{x}_t = \underline{x}(\beta_1, \cdots, \beta_k)$ where $0 \le t < n$, and we aim to find its corresponding table value, $\underline{y}_t = \underline{y}(\beta_1, \cdots, \beta_k)$.

B. Homomorphic Encryption

We consider word-wise homomorphic encryption schemes that support arithmetic operations (i.e., addition and multiplication) in encrypted states. The plaintext space of word-wise HE is $\mathbb{C}^{N/2}$ (in CKKS scheme [5], [22]) or $\mathbb{Z}_q$ in which q is either a prime or a power of prime (in BGV and BFV schemes [3], [4]). A homomorphic encryption scheme commonly supports the following operations.

- **cAdd**$(\underline{\mathrm{m}}_1, \mathrm{m}_2)$: for given ciphertext $\underline{\mathrm{m}}_1$ of m_1 and a plaintext m_2, returns a ciphertext of $\mathrm{m}_1 + \mathrm{m}_2$.
- **Add**$(\underline{\mathrm{m}}_1, \underline{\mathrm{m}}_2)$: for given ciphertext $\underline{\mathrm{m}}_1$ and $\underline{\mathrm{m}}_2$ of m_1 and m_2, returns a ciphertext of $\mathrm{m}_1 + \mathrm{m}_2$.
- **cMult**$(\underline{\mathrm{m}}_1, \mathrm{m}_2)$: for given ciphertext $\underline{\mathrm{m}}_1$ of m_1 and a plaintext m_2, returns a ciphertext of $\mathrm{m}_1 \odot \mathrm{m}_2$, where $\odot$ is multiplication between two plaintexts. For example, in the case of CKKS, $\odot$ is a componentwise multiplication.
- **Mult**$(\underline{\mathrm{m}}_1, \underline{\mathrm{m}}_2)$: for given ciphertext $\underline{\mathrm{m}}_1$ and $\underline{\mathrm{m}}_2$ of m_1 and m_2, returns a ciphertext of $\mathrm{m}_1 \odot \mathrm{m}_2$.

Here, `cAdd` and `cMult` are abbreviations of constant-addition and constant-multiplication, which regard a plaintext as a constant.

C. Homomorphic Comparison

In this work, we exploit the homomorphic comparison method for two HE ciphertexts, i.e., `Compare`$(\cdot, \cdot)$ that satisfies the following.

$$\texttt{Compare}(\underline{m_1}, \underline{m_2}) = \begin{cases} \underline{0} & \text{if } m_1 < m_2 \\ \underline{0.5} & \text{if } m_1 = m_2 \\ \underline{1} & \text{if } m_1 > m_2 \end{cases}$$

There have been several works of homomorphic comparison for each category of schemes.

For CKKS scheme, [11] proposed an efficient comparison operation. The key idea of [11] is to construct a homomorphic comparison algorithm that uses less number of homomorphic multiplications. The homomorphic comparison alogorithm in [11] requires $\Theta(\log(1/\epsilon))+\Theta(\log\log(1/\alpha))$ computational complexity to obtain an approximate comparison result of $a, b \in [0,1]$ satisfying $|a-b| \geq \epsilon$ within α error. We mainly used [11] in Section IV. For BFV/BGV scheme, recently [23] suggested *less-than* function for homomorphic comparison operation on BFV/BGV ciphertexts.

D. Straw Man Solutions

1) Lagrange Interpolation: One of the most trivial approaches to implementing the evaluation of TLU with HE is to use polynomial interpolation, i.e., Lagrange interpolation. For given LUT, $\{(x_i, y_i)\}_{i=1}^n$, Lagrange interpolation is the polynomial $p(\cdot)$ of degree n such that $p(x_i) = y_i$ for each $i = 1, 2, \cdots, n$ as follows.

$$p(x) := \sum_{i=1}^{n} \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right) y_i = \sum_{i=0}^{n-1} c_i x^i$$

Since homomorphic multiplication is much slower than the other operations, it is prevalent to exploit the Paterson-Stockmeyer algorithm to evaluate higher degree polynomials on HE ciphertexts with less number of multiplications [24]. To utilize the Paterson-Stockmeyer algorithm, we should compute the coefficients c_i of the polynomial before the evaluation. However, the size of the coefficients of Lagrange interpolations becomes exponentially big when we consider the large LUTs, and it is infeasible to use the Paterson-Stockmeyer algorithm for the evaluation of large LUTs.

To stably evaluate the Lagrange interpolate, to the best of our knowledge, we should use the formula $p(x) = \sum_{i=1}^{n} \left(\prod_{j \neq i} (x - x_j)/(x_i - x_j) \right) y_i$. However, it requires a number of homomorphic divisions, which makes it infeasible to be evaluated by HE. Even for the unencrypted LUTs, it requires $O(n^2)$ homomorphic multiplications. This is impractical for LUTs of large sizes, e.g., $n > 2^8$.

2) Exhaustive Comparison: Another straw-man solution for evaluating LUTs using HE and a homomorphic comparison algorithm is to compare the given query to all table keys. This uses $\Omega(n)$ homomorphic comparisons to evaluate a LUT of size n, and $O(n)$ homomorphic multiplications to aggregate the comparison results. Since homomorphic comparison is time-consuming compared to homomorphic addition and multiplications, this solution is also impractical for LUTs of large sizes.

E. Threat Model

Throughout the paper, we mainly consider a client-server model. For the setting of a private lookup table case, a client sends a private query (key data) and a private lookup table to an untrusted computing server, and the server sends back the (encrypted) value data from the lookup table corresponding to the query. During the protocol, the server should not learn any information about the query and the table.

For the public lookup table case, the client sends a private query to the server, and the server responds with the (encrypted) value data that corresponds to the query. The server should not learn about the query.

Our security relies on the IND-CPA security of homomorphic encryption. Since the client sends the public key and the homomorphically encrypted query (and table) to the server without auxiliary information, the server cannot learn any information due to the security of relying HE scheme.

III. Proposed Algorithms

In this section, we suggest algorithms for the evaluation of the lookup table on encrypted queries. We assume that we are given a word-wise HE scheme that supports homomorphic addition and multiplication operations, and comparison. For example, CKKS scheme supports homomorphic addition and multiplication, and there have been several works on homomorphic comparison methods for CKKS ciphertexts [11], [10], [12]. BGV scheme, which is another example of word-wise HE scheme, also supports homomorphic addition and multiplication, and [23] provides a homomorphic comparison method for BGV.

We point out that `Mult` (homomorphic multiplication) is much slower than `Add` (homomorphic addition), and `Compare` (homomorphic comparison) is much slower than `Mult` (homomorphic multiplication). Also, we stress that `cMult` (homomorphic multiplication between a plaintext and a ciphertext) can be done much faster than `Mult` (multiplication between two ciphertexts). Taking these points into account, the number of comparisons should be minimized as the top priority, and the number of multiplication bewteen ciphertexts has the second priority. The cost for multiplication between a plaintext and a ciphertext and the cost for addition are relatively minor.

In this section, we suggest algorithms for the evaluation of a lookup table on encrypted queries using homomorphic addition, multiplication, and comparison operations.

A. Private Encrypted Lookup Tables

We begin with the case of a sorted and encrypted lookup table and an encrypted query. This means that the lookup table is encrypted by HE, so it is concealed from the computing server.

1) Algorithm 1: Folding table: The high-level idea of our first algorithm is a homomorphic binary search. For a given query, we compare it to the median of the table keys. If the query is smaller than the median, we keep searching on the first half of the table. Otherwise, we keep searching for the last half.

However, since we are dealing with an encrypted table and query, we cannot observe the result of the comparison between the query and the median table key. We exploit a linear combination to address this issue. Suppose we have encryption of the bit $\underline{\beta}$ that indicates whether the query is smaller than the median key, i.e., $\beta = 0$ if the query is smaller than the median value, 1 otherwise. If $\beta = 0$, we should keep searching on the first half of the table, and if $\beta = 1$, we should keep searching on the last half.

We utilize the fact that $\underline{m_1} + \underline{b} \times (\underline{m_2 - m_1})$ is $\underline{m_1}$ if $\beta = 0$, and $\underline{m_2}$ if $\beta = 1$. By using this, we can *fold the table* in half so that it keeps containing the query after folding it. More precisely, for each $i = 1, 2 \cdots, m-1$, we let

$$\tilde{\underline{x}}_i \;:=\; \underline{x}_i + \underline{\beta}(\underline{x}_{i+m} - \underline{x}_i) \;=\; \begin{cases} \underline{x}_i, & \text{if } x < x_m \\ \underline{x}_{i+m}, & \text{otherwise} \end{cases}, \quad (1)$$

$$\tilde{\underline{y}}_i \;:=\; \underline{y}_i + \underline{\beta}(\underline{y}_{i+m} - \underline{y}_i) \;=\; \begin{cases} \underline{y}_i, & \text{if } x < x_m \\ \underline{y}_{i+m}, & \text{otherwise} \end{cases}. \quad (2)$$

Then, it immediately comes out that

$$\{\tilde{\underline{x}}_1, \cdots, \tilde{\underline{x}}_m\} \;=\; \begin{cases} \{\underline{x}_1, \cdots, \underline{x}_m\}, & \text{if } x < x_m \\ \{\underline{x}_{m+1}, \cdots, \underline{x}_{2m}\}, & \text{otherwise} \end{cases},$$

$$\{\tilde{\underline{y}}_1, \cdots, \tilde{\underline{y}}_m\} \;=\; \begin{cases} \{\underline{y}_1, \cdots, \underline{y}_m\}, & \text{if } x < x_m \\ \{\underline{y}_{m+1}, \cdots, \underline{y}_{2m}\}, & \text{otherwise} \end{cases}.$$

When folding the table keys, we can halve table values as well. Finally, after $\log n$ times of folding, we can finally find the table values that correspond to the given query. Algorithm 1 presents our algorithm in detail.

Algorithm 1 Folding table

```
Input: $\{0 \le \underline{x}_0 < \cdots < \underline{x}_{n-1} \le 1\}$, $\{\underline{y}_0, \cdots, \underline{y}_{n-1}\} \in [0,1]^n$
Input: a query $\underline{x} = \underline{x}_t$ for some $0 \le t < n$
Output: An encrypted output $\underline{y}_t$.
 1: for $i \leftarrow 0$ to $n-1$ do
 2:     $\tilde{\underline{x}}_i \leftarrow \underline{x}_i$
 3:     $\tilde{\underline{y}}_i \leftarrow \underline{y}_i$
 4: end for
 5: $m \leftarrow n$
 6: while $m > 1$ do
 7:     $m \leftarrow m/2$
 8:     $\underline{\beta} \leftarrow \texttt{Compare}\left(\underline{x} + \underline{x}, \tilde{\underline{x}}_{m-1} + \tilde{\underline{x}}_m\right)$
 9:     for $0 \leftarrow 1$ to $m-1$ do
10:         $\tilde{\underline{x}}_i \leftarrow \tilde{\underline{x}}_i + \underline{\beta} \cdot (\tilde{\underline{x}}_{i+m} - \tilde{\underline{x}}_i)$
11:         $\tilde{\underline{y}}_i \leftarrow \tilde{\underline{y}}_i + \underline{\beta} \cdot (\tilde{\underline{y}}_{i+m} - \tilde{\underline{y}}_i)$
12:     end for
13: end while
14: return $\tilde{\underline{y}}_0$
```

Theorem 1. *Algorithm 1 is correct.*

Proof. For the while-loop in line $6-13$, we consider the loop invariant that

$$\tilde{x}_i = x_t,\ \tilde{y}_i = y_t$$

for some $0 \le i \le m-1$. Initially, $0 \le t \le n-1$, so the loop invariant trivially holds. Suppose the loop invariant holds at the beginning of a loop iteration. There exists $0 \le i < m$ such that $\tilde{x}_i = x_t$ and $\tilde{y}_i = y_t$. If $0 \le i < \frac{m}{2}$, then $\beta = 0$ and $x_t = \tilde{x}_i = \tilde{x}_i + \beta(\tilde{x}_{i+m/2} - \tilde{x}_i)$. Otherwise, i.e. $\frac{m}{2} \le i < m$, then $\beta = 1$, and $x_t = \tilde{x}_i = \tilde{x}_i + (1-\beta)(\tilde{x}_{i-m/2} - \tilde{x}_i) = \tilde{x}_{i-m/2} + \beta(\tilde{x}_i - \tilde{x}_{i-m/2})$. Similar result holds for $\tilde{y}_i$ and y_t. Hence, the loop invariant satisfies the maintenance. Finally, after the loop terminates, from the loop invariant, $\tilde{y}_0 = y_t$. $\square$

Cost analysis. For each iteration, Algorithm 1 uses 1 homomorphic comparison, and $2m$ homomorphic multiplications. Thus, for the entire algorithm, $\log n$ homomorphic comparison, and $n + n/2 + \cdots + 2 = 2n - 2$ homomorphic multiplications are needed.

2) Algorithm 2: One-hot indicator: Our second approach is similar to the first approach, Algorithm 1, but not to fold the table. Instead, we repeatedly subdivide the *one-hot indicator* that indicates which subblock the given query belongs among uniformly cut subblocks.

To be more precise, assume that we are given a query $\underline{x} = \underline{x}_t$ and a public table $0 \le x_0 \le \cdots \le x_{n-1} \le 1$ and $\{y_i\}_{i=0}^{n-1}$. In the first iteration, we compute $\underline{B}(0) = \texttt{Compare}(\underline{x}, x_{n/2})$ and $\underline{B}(1) = 1 - \underline{B}(0)$. Then, $(\underline{B}(0), \underline{B}(1))$ is a one-hot vector that indicates which half-block the query $\underline{x}_t$ belongs. In the second iteration, we compute $\underline{\beta} = \texttt{Compare}\left(\underline{x}, \underline{B}(0)x_{n/4} + \underline{B}(1)x_{3n/4}\right)$ and let

$$\begin{aligned} \underline{B}(0,0) &= \underline{\beta}\,\underline{B}(0), & \underline{B}(0,1) &= (1-\underline{\beta})\underline{B}(0), \\ \underline{B}(1,0) &= \underline{\beta}\,\underline{B}(1), & \underline{B}(1,1) &= (1-\underline{\beta})\underline{B}(1). \end{aligned}$$

Then, $(\underline{B}(0,0), \underline{B}(0,1), \underline{B}(1,0), \underline{B}(1,1))$ is the one-hot vector that indicates which quarter-block the query $\underline{x}_t$ belongs. By repeating this process, we can extend the one-hot indicator.

Generally, for ease of discussion, let the binary representation of the index of a given query t be $\{\beta_i\}_{i=0}^{k-1}$, i.e., $t = \sum_{i=1}^{k-1} \beta_i 2^{k-1-i}$. For each $r \ge 1$ and $(b_1, \cdots, b_r) \in \{0,1\}^r$, let

$$B(b_1, \cdots, b_r) = \begin{cases} 1 & \text{if } b_1 = \beta_1, \cdots, b_r = \beta_r \\ 0 & \text{otherwise} \end{cases}$$

be the one-hot indicator that indicates the $1/2^r$ subblock containing the query. In order to generate a one-hot indicator for $1/2^{r+1}$ subblock, it suffices to compare the query to the median value of the $1/2^r$ subblock it belongs. We note that we can compute the median value by the following linear combination.

$$\sum_{\vec{b} \in \{0,1\}^r} B(\vec{b})\,(\text{Media of } b \text{ th subblock})\,.$$

Thus, we can compute β_{r+1} as follows.

$$\beta_{r+1} = \texttt{Compare}\left(x,\ 0.5 \sum_{\vec{b} \in \{0,1\}^r} B(\vec{b}) \left(x(\vec{b}, 1, \vec{0})\right)\right).$$

Then, we also can compute $B(b_1, \cdots, b_r, b_{r+1})$ as:

$$\begin{aligned} B(b_1, \cdots, b_r, 1) &= B(b_1, \cdots, b_r) \cdot \beta_{r+1} \\ B(b_1, \cdots, b_r, 0) &= B(b_1, \cdots, b_r) \cdot (1 - \beta_{r+1}) \\ &= B(b_1, \cdots, b_r) - B(b_1, \cdots, b_r, 1), \end{aligned}$$

which is the one-hot indicator that indicates the $1/2^{r+1}$ subblock containing the query.

After $\log n$ iterations, we compute $B(b_1, \cdots, b_k)$ for each $(b_1, \cdots, b_k) \in \{0,1\}^k$, and we are able to yield the result, $y(\beta_1, \cdots, \beta_k)$ as follows.

$$y(\beta_1, \cdots, \beta_k) = \sum_{\vec{b} \in \{0,1\}^k} B(\vec{b}) \cdot y(\vec{b})$$

Algorithm 2 Expanding one-hot indicator

```
Input: $\{0 \le \underline{x}_0 < \cdots < \underline{x}_{n-1} \le 1\}$, $\{\underline{y}_0, \cdots, \underline{y}_{n-1}\} \in [0,1]^n$
Input: a query $\underline{x} = \underline{x}_t$ for some $0 \le t < n$
Output: An encrypted output $\underline{y}_t$.
 1: $\beta \leftarrow$ Compare($\underline{x} + \underline{x}, \underline{x}(0, \vec{1}) + \underline{x}(1, \vec{0})$)
 2: $(B(0), B(1)) \leftarrow (1 - \beta, \beta)$
 3: $m \leftarrow 1$
 4: while $m < \log n$ do
 5:     $\underline{x'} \leftarrow 0$
 6:     for $\vec{b} \leftarrow \{0,1\}^m$ do
 7:         $\underline{x'} \leftarrow \underline{x'} + \underline{B}(\vec{b}) \left(\underline{x}(\vec{b}, 1, \vec{0}) + \underline{x}(\vec{b}, 0, \vec{1})\right)$
 8:     end for
 9:     $\underline{\beta} \leftarrow$ Compare($\underline{x} + \underline{x}, \underline{x'}$)
10:     for $\vec{b} \leftarrow \{0,1\}^m$ do
11:         $\underline{B}(\vec{b}, 1) \leftarrow \underline{\beta} \cdot \underline{B}(\vec{b})$
12:         $\underline{B}(\vec{b}, 0) \leftarrow \underline{B}(\vec{b}) - \underline{B}(\vec{b}, 1)$
13:     end for
14:     $m \leftarrow m + 1$
15: end while
16: $\underline{y'} \leftarrow 0$
17: for $\vec{b} \leftarrow \{0,1\}^{\log n}$ do
18:     $\underline{y'} \leftarrow \underline{y'} + \underline{B}(\vec{b}) \cdot \underline{y}(\vec{b})$
19: end for
20: return $\underline{y'}$
```

Algorithm 2 describes our detailed algorithm.

Theorem 2. *Algorithm 2 is correct.*

Proof. Let the binary representation of t be $\sum_{i=1}^{k-1} \beta_i 2^{k-1-i}$. For the while-loop in line $4-15$, we consider the loop invariant that for each $\vec{b} \in \{0,1\}^m$, $B(\vec{b}) = 1$ if $b_i = \beta_i$ for all i, and $B(\vec{b}) = 0$ otherwise.

At the initial state of the loop, the loop invariant trivially holds. Suppose the loop invariant holds at the beginning of loop iteration. Then, x' is a weighted sum of the median values of each $1/2^m$ subblocks of lookup table keys. Thus, x' is the median value of the $1/2^m$ subblock of lookup table keys that the query belongs to. As a consequence, $\beta = \beta_{m+1}$, and $B(\beta_1, \cdots, \beta_m, \beta) = B(\beta_1, \cdots, \beta_m) = 1$ and $B(\beta_1, \cdots, \beta_m, 1 - \beta) = 0$. Also, $B(\vec{b}) = B(\vec{b}, 0) = B(\vec{b}, 1) = 0$ unless $\vec{b} = \vec{\beta}$. Hence, the loop invariant satisfies the maintenance.

Finally, after once the loop terminates, $B(\vec{b}) = 1$ if and only if $\vec{b} = \vec{\beta}$. Thus, $y' = y(\vec{\beta}) = y_t$ as desired. □

Cost analysis. For each iteration, Algorithm 2 uses 1 homomorphic comparison, 2^{m+1} additions and 2^{m+1} multiplications. Also, to compute y' from B, it uses other n additions and n multiplications. Thus, for the entire algorithm, $\log n$ homomorphic comparisons, $3n + O(1)$ additions and $3n + O(1)$ multiplications are used. By using the fact that $\sum_{\vec{(b)} \in \{0,1\}^r} B(\vec{b}) = 1$ for each $r = 1, 2 \cdots, k$, we can reduce the number of additions and multiplications to $3n - \log n + O(1)$.

TABLE I
REQUIRED NUMBER OF OPERATIONS TO PERFORM ALGORITHMS 1 AND 2 ON ENCRYPTED LOOKUP TABLES OF SIZE n.

	`Compare`	`Mult`
Exhaustive comparison	$O(n)$	$O(n)$
Algorithm 1	$\log n$	$2n + O(1)$
Algorithm 2	$\log n$	$3n - \log n + O(1)$

Remark 1. *Algorithm 1 is faster than Algorithm 2 in the most cases.*

Remark 2. *We assumed that the table keys were encrypted and sorted. We note that if the table keys are encrypted but not sorted, our algorithm should be preceded by homomorphic sorting algorithms such as [8].*

B. Public Unencrypted Lookup Tables

Now we consider the unencrypted lookup table with an encrypted query. This means that the lookup table is public, while the query is still concealed.

We note that in this case, we can exploit the fact that `cMult` (i.e., multiplication between plaintext and ciphertext) is much faster than `Mult` (i.e., homomorphic multiplication between two ciphertexts).

1) Algorithm 1: Folding table: The analogy of Algorithm 1 properly works for the unencrypted lookup-table $\{x_i\}_{i=0}^{n-1}$ and $\{y_i\}_{i=0}^{n-1}$. Note that $\tilde{x}$ and $\tilde{y}$'s are unencrypted in the first iteration. Thus, the algorithm uses $\log n$ number of `Compare`, $n + O(1)$ number of `cMult`, and $n + O(1)$ number of `Mult`.

2) Algorithm 2: One-hot indicator: The analogy of Algorithm 2 properly works for the unencrypted lookup tables. Note that computing x' and y' can be done with `cMult` instead of `Mult`. Thus, the algorithm uses $\log n$ number of `Compare`, $2n + O(1)$ number of `cMult`, and $n - \log n + O(1)$ number of `Mult`.

3) Algorithm 3: Hybrid: Algorithm 1 diligently folds the table and Algorithm 2 do not fold the table. We now mix and match both algorithms to get an algorithm with a better performance.

The key observation is that Algorithm 1 uses about $n/2, n/4, \cdots, 2, 1$ multiplications in each loop iteration, while Algorithm 2 uses about $1, 2, 4, \cdots, n/2$ multiplications in each loop iteration. This is due to the fact that *folding* in Algorithm 1 is costly at the beginning, but becomes faster as the table keeps being halved; on the other hand, *expanding the one-hot indicator* in Algorithm 2 is fast at the beginning, but becomes heavier as the size of the one-hot indicator grows.

We delay the folding; we begin with Algorithm 2, and end with Algorithm 1. In particular, we extend the one-hot indicator for $\log n/2$ times, fold the lookup table into a smaller table of size $\sqrt{n}$ by using the one-hot indicator, and run Algorithm 1 on it.

Algorithm description. The algorithm consists of baby-step, lazy folding, and giant-step:

1) *Baby-step*: For a given encrypted query, similar to Algorithm 2, we first find the one-hot indicator that indicates which $\sqrt{n}$ sub-block the query belongs among $\sqrt{n}$ number of $\sqrt{n}$ sub-blocks.
2) *Lazy folding*: Then, by inner product the one-hot indicator with the given table, we get a table of size $\sqrt{n}$ that contains the query. We shall call this *lazy folding*.
3) *Giant-step*: Finally, analogous to Algorithm 1, we are finally able to get the encrypted table value corresponding to the given encrypted key query.

Our detailed algorithm is presented in Algorithm 3.

Cost analysis. Algorithm 3 consists of three steps: (1) Baby-step: finding the one-hot indicator of size $\sqrt{n}$, (2) Lazy folding, and (3) Giant-step: repeating the table folding.

Finding the $\sqrt{n}$ subblock that the query belongs to (as an analogy of Algorithm 2) uses $\log n/2$ number of `Compare`, $(2\sqrt{n} + O(1))$ number of `cMult`, and $(\sqrt{n} - \log n + O(1))$ number of `Mult`, as it is the analogy of Algorithm 2.

Lazy folding, connecting the analogies of Algorithm 1 and 2, uses $2n$ homomorphic multiplications. However, since the table is unencrypted, it suffices to perform `cMult` instead of `Mult`. We stress out that `cMult` is much faster than `Mult`; thus, the cost for lazy folding is not very significant. Also, we remark that by utilizing the fact that $\sum_{j=0}^{\sqrt{n}-1} B(\vec{j}) = 1$, we can reduce the number of multiplications to $2n - 2\sqrt{n}$.

At last, we repeating the table folding as in Algorithm 1. Note that after lazy folding, all table keys and values are encrypted. It uses $\log n/2$ number of `Compare`, and $(2\sqrt{n} + O(1))$ number of `Mult`.

To put it all together, Algorithm 3 uses $\log n$ number of `Compare`, $(2n+O(1))$ number of `cMult`, and $(3\sqrt{n} - \log n + O(1))$ number of `Mult`.

We also point out that while Algorithm 1 and 2 require $\Omega(n)$ additional memory space, Algorithm 3 uses $O(\sqrt{n})$ additional memory space.

Algorithm 3 Baby-step / giant-step

```
Input: $\{0 \le \underline{x}_0 < \cdots < \underline{x}_{n-1} \le 1\}$, $\{\underline{y}_0, \cdots, \underline{y}_{n-1}\} \in [0,1]^n$
Input: a query $\underline{x} = \underline{x}_t$ for some $0 \le t < n$
Output: An encrypted output $\underline{y}_t$.
$\beta \leftarrow$ Compare$(\underline{x} + \underline{x}, \underline{x}(0, \vec{1}) + \underline{x}(1, \vec{0}))$
$(B(0), B(1)) \leftarrow (1 - \beta, \beta)$
for $m \leftarrow 1$ to $\log n/2 - 1$ do
    $\underline{x'} \leftarrow 0$
    for $\vec{b} \leftarrow \{0,1\}^m$ do
        $\underline{x'} \leftarrow \underline{x'} + \underline{B}(\vec{b}) \left(\underline{x}(\vec{b}, 1, \vec{0}) + \underline{x}(\vec{b}, 0, \vec{1})\right)$
    end for
    $\underline{\beta} \leftarrow$ Compare$(\underline{x} + \underline{x}, \underline{x'})$
    for $\vec{b} \leftarrow \{0,1\}^m$ do
        $\underline{B}(\vec{b}, 1) \leftarrow \underline{\beta} \cdot \underline{B}(\vec{b})$
        $\underline{B}(\vec{b}, 0) \leftarrow \underline{B}(\vec{b}) - \underline{B}(\vec{b}, 1)$
    end for
end for
for $i \leftarrow 0$ to $\sqrt{n} - 1$ do
    $\underline{\tilde{x}}_i, \underline{\tilde{y}}_i \leftarrow 0$
    for $j \leftarrow 0$ to $\sqrt{n} - 1$ do
        $\underline{\tilde{x}}_i \leftarrow \underline{\tilde{x}}_i + \underline{B}(j)x(j, i)$
        $\underline{\tilde{y}}_i \leftarrow \underline{\tilde{y}}_i + \underline{B}(j)y(j, i)$
    end for
end for
$m \leftarrow \sqrt{n}$
while $m > 1$ do
    $m \leftarrow m/2$
    $\underline{\beta} \leftarrow$ Compare$\left(\underline{x} + \underline{x}, \underline{\tilde{x}}_{m-1} + \underline{\tilde{x}}_m\right)$
    for $i \leftarrow 0$ to $m - 1$ do
        $\underline{\tilde{x}}_i \leftarrow \underline{\tilde{x}}_i + \underline{\beta} \cdot (\underline{\tilde{x}}_{i+m} - \underline{\tilde{x}}_i)$
        $\underline{\tilde{y}}_i \leftarrow \underline{\tilde{y}}_i + \underline{\beta} \cdot (\underline{\tilde{y}}_{i+m} - \underline{\tilde{y}}_i)$
    end for
end while
return $\underline{\tilde{y}}_0$
```

TABLE II
REQUIRED NUMBER OF OPERATIONS TO PERFORM ALGORITHMS 1, 2 AND 3 ON UNENCRYPTED LOOKUP TABLES OF SIZE n.

	`Compare`	`Mult`	`cMult`
Lagrange Interpolate	$O(1)$	$O(n^2)$	$O(n^2)$
Exhaustive Comparison	$O(n)$	$O(1)$	$O(n)$
Algorithm 1	$\log n$	$n + O(1)$	$n + O(1)$
Algorithm 2	$\log n$	$n - \log n + O(1)$	$2n + O(1)$
Algorithm 3	$\log n$	$3\sqrt{n} - \log n + O(1)$	$2n + O(1)$

Remark 3. *In this section, we imitated binary search; that is, we halves the table by using one homomorphic comparison operation. In a point of fact, we can fold the table to make a smaller table of size $1/(c+1)$, by comparing the query to c table keys. When we are able to perform homomorphic comparison in a SIMD manner, we can simultaneously compare c ciphertexts at once, and this generalization might reduce the practical runtime.*

IV. APPROXIMATE CASE: CKKS ADAPTATION

In this section, we suggest some practical adaptations and optimizations for the application of our algorithm to approximate HE, e.g., CKKS scheme.

We stress that we are mainly focusing on applications of HE to real-valued data for real-world applications. A real-valued computation means the approximate computation with some precision bits. As a point of real-valued computation, CKKS, for example, serves approximate computation rather than exact computations, and it has a great advantage in being applied to real-world applications.

With this manner of approximate computation, homomorphic comparison methods for CKKS, [11], output approximate results of the comparison. Since we exploit homomorphic comparison repeatedly, we should carefully manage the error accompanied by the homomorphic comparison methods. We remark that the cost for the homomorphic comparison method in [11] depends on the precision of input and output ciphertexts. The homomorphic comparison algoirhtm in [11] requires $\Theta(\log(1/\epsilon)) + \Theta(\log\log(1/\alpha))$ computational cost to obtain an approximate comparison result of $a, b \in [0,1]$ satisfying $|a - b| \ge \epsilon$ within α error.

In this section, we analyze the error accompanied by approximate homomorphic comparison, and suggest an optimization for it.

A. Approximate Homomorphic Comparison

We let $\texttt{Compare}(a, b; \epsilon, \alpha)$ be the approximate comparison that outputs the comparison of $a, b \in [0, 2]$ satisfying $|a - b| \geq \epsilon$ within α error. As mentioned in [11], the cost for $\texttt{Compare}(a, b; \epsilon, \alpha)$ is $\Theta(\log(1/\epsilon)) + \Theta(\log\log(1/\alpha))$. This means that we require more computations to achieve better precision.

Suppose we are given a lookup table $\{x_i, y_i\}_{i=1}^{n}$ such that $0 \leq x_1 < x_2 < \cdots < x_n \leq 1$, and $0 \leq y_i \leq 1$. Let Δ be the minimum difference of table keys, i.e., $\Delta := \min \|x_{i+1} - x_i\|$. Then, Δ determines the precision of table keys, and the cost for homomorphic comparison depends on Δ. In particular, suppose we want to evaluate the lookup table on the encrypted query as in Section III within error A. For ease of discussion, we further assume that $A < \Delta/2$.

Theorem 3. *In Algorithm 1, 2, and 3 with approximate comparison,* $\texttt{Compare}(\cdot, \cdot; \Delta/2, ((1 + 2A)^{1/\log n} - 1)/2))$, *gives correct result within the error A.*

Proof. Here, we prove the correctness of Algorithm 3. Let $\alpha := ((1 + 2A)^{1/\log n} - 1)/2$. Also, for given query $\underline{x_t}$, let the binary representation of t be $\{\beta_i\}_{i=1}^{k}$, i.e., $t = \sum_{i=1}^{k} \beta_i 2^{k-1-i}$.

We begin with the for-loop in line 3-13. We consider the loop invariant that for each $\vec{c} \in \{0, 1\}^{k-m}$,

$$\sum_{\vec{b} \in \{0,1\}^m} B(\vec{b}) x(\vec{b}, \vec{c}) = x(\beta_1, \cdots, \beta_m, \vec{c})$$

within the error $((2\alpha+1)^m - 1)/2$. Initially, the loop invariant trivially holds since $B(0) = 1 - \beta_1$ and $B(1) = \beta_1$ within error α. Suppose the loop invariant holds at the beginning of a loop iteration. We note that for each $\vec{b} \in \{0, 1\}^m$ and a bit e,

$$B(\vec{b}, e) = (1 - e)B(\vec{b}) + eB(\vec{b}).$$

Also, $x' = x(\beta_1, \cdots, \beta_m, 1, \vec{0}) + x(\beta_1, \cdots, \beta_m, 0, \vec{1})$ within error $(1 + 2\alpha)^m - 1 = (1 + 2A)^{m/\log n} - 1 < 2A < \Delta$, so $\beta = \beta_{m+1}$ within error α. For each $\vec{c} \in \{0, 1\}^{k-m-1}$,

$$\begin{aligned}
&\left| \left(\sum_{\vec{b} \in \{0,1\}^{m+1}} B(\vec{b}) x(\vec{b}, \vec{c}) \right) - x(\beta_1, \cdots, \beta_{m+1}, \vec{c}) \right| \\
&= \left| \sum_{\vec{b} \in \{0,1\}^m} ((1 - \beta) B(\vec{b}) x(\vec{b}, 0, \vec{c}) + \beta B(\vec{b}) x(\vec{b}, 1, \vec{c})) \right. \\
&\qquad \left. - x(\beta_1, \cdots, \beta_{m+1}, \vec{c}) \right| \\
&\leq \left| (1 - \beta) \left(\sum_{\vec{b} \in \{0,1\}^m} B(\vec{b}) x(\vec{b}, 0, \vec{c}) - x(\beta_1, \cdots, \beta_m, 0, \vec{c}) \right) \right| \\
&\quad + \left| \beta \left(\sum_{\vec{b} \in \{0,1\}^m} B(\vec{b}) x(\vec{b}, 1, \vec{c}) - x(\beta_1, \cdots, \beta_m, 1, \vec{c}) \right) \right| \\
&\quad + \left| (\beta_{m+1} - \beta) \left(x(\beta_1, \cdots, \beta_m, 0, \vec{c}) - x(\beta_1, \cdots, \beta_m, 1, \vec{c}) \right) \right| \\
&\leq (|1 - \beta| + |\beta|) \frac{(2\alpha + 1)^m - 1}{2} + \alpha = \frac{(2\alpha + 1)^{m+1} - 1}{2}.
\end{aligned}$$

Thus, the loop invariant satisfies the maintenance. After the loop terminates, we yield the fact that for each $i = 0, 1 \cdots, \sqrt{n}$,

$$\tilde{x}_i = x(\beta_1, \cdots, \beta_{k/2}, \vec{i})$$

in line 20. The same result holds for $\tilde{y}'$ as well.

Now, for the while-loop in line 22-29, we consider the loop invariant that for each $i = 0, 1, \cdots, m - 1$,

$$\tilde{x}'_i = x(\beta_1, \cdots, \beta_{k - \log m}, \vec{i})$$

and

$$\tilde{y}'_i = y(\beta_1, \cdots, \beta_{k - \log m}, \vec{i})$$

within error $((2\alpha + 1)^{\log n - \log m} - 1)/2$. At the initial state of the loop, the loop invariant trivially holds. Suppose the loop invariant holds at the beginning of a loop iteration; then $x' = x(\beta_1, \cdots, \beta_m, 1, \vec{0}) + x(\beta_1, \cdots, \beta_m, 0, \vec{1})$ within error $(1 + 2\alpha)^m - 1 = (1 + 2A)^{m/\log n} - 1 < 2A < \Delta$, so $\beta = \beta_{m+1}$ within error α. Thus, with the same argument above, this loop invariant also has a maintenance. Finally, once the loop terminates, $\tilde{y}_0$ is $y(\beta_1, \cdots, \beta_k) = y_t$ within error $((1 + 2\alpha)^{\log n} - 1)/2 = A$. ☐

We roughly approximate $((1 + 2A)^{1/\log n} - 1)/2 \approx A/\log n$ by using that α is small. Then, the computational cost of each homomorphic comparison in our algorithms is: $\Theta(\log(1/\Delta)) + \Theta(\log\log((\log n)/A))$. The overall computational cost for homomorphic comparison is $\Theta(\log n \log \frac{1}{\Delta})) + \Theta(\log n \log\log(\frac{\log n}{A}))$.

B. Reducing 1/Δ

As we describe above, the computational cost for the homomorphic comparison algorithm depends on Δ, the minimum difference between table keys. The homomorphic comparison might be potentially slow if the given lookup table is bad, i.e., it has an extremely small Δ. For the unencrypted bad lookup tables, we suggest a heuristic method to increase Δ with a few extra operations.

We first point out that for an uniformly distributed n table keys in $[0, 1]$, the expected value of Δ is $\frac{1}{n^2}$. Our key idea is to randomize the keys of a given bad table to yield reasonable Δ, i.e., $\Delta \approx \frac{1}{n^2}$. We seek for an efficiently evaluatable polynomial $p(\cdot)$ such that the new table keys $\{p(x_1), \cdots, p(x_n)\}$ has a reasonable $\Delta = \min\{p(x_i) - p(x_{i-1})\}$. Once we have such a polynomial, for a given encrypted query $(\underline{x_t})$, we first compute $p(\underline{x_t})$, and evaluate lookup table $\{p(x_1), \cdots, p(x_n)\}$ on the updated query, $p(\underline{x_t})$. to get the appropriate $\underline{y_t}$. We note that since $\{p(x_1), \cdots, p(x_n)\}$ has a reasonable Δ, the cost for homomorphic comparison operation might be reasonable.

The randomizing polynomial $p(\cdot)$ should be able to be evaluated with less number of multiplications and should have a great capability to diffuse the key values. We suggest utilizing Chebyshev polynomial basis, which is obtained from the recurrence relation:

$$\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x).
\end{aligned}$$

Since $T_{2k}(x) = 2T_k^2(x) - 1$ for each $k = 1, 2, \cdots$, we can evaluate Chebyshev polynomials of high degree with relatively less number of multiplications.

We randomly search for an appropriate randomizing polynomial as follows. For small $d \approx \log n$, we randomly pick $r_0, r_1, \cdots, r_d$ from $[-1, 1]$, and let

$$q(x) := \sum_{i=0}^{d} r_i T_{2^i}(x)$$
$$p(x) := \frac{q(x) - \min\{q(x_i)\}}{\max\{q(x_i)\} - \min\{q(x_i)\}}.$$

Note that $p(x_i) \in [0, 1]$ by definition. Heuristically, under the assumption that $p(\cdot)$ randomizes the table keys, $\{x_i\}_{i=0}^{n-1}$, into a uniformlly distributed keys in $[0, 1]$, the expected value of $\Delta(p)$ $(:= \min\{p(x_i) - p(x_j)\})$ is $1/n^2$. Thus, after several choices of r_i's, we heuristically find a good randomizing polynomial.

Algorithm 4 Searching a good randomizing polynomial

```
Input: $\{x_0, \cdots, x_{n-1}\} \in [0,1]^n$, a threshold $T$
Output: A good randomization $p(\cdot)$.
 1: for $d \leftarrow 0$ to $\log n$ do
 2:     ctr $\leftarrow 0$
 3:     for $i \leftarrow 0$ to $d$ do
 4:         $r_i \leftarrow_{\$} [-1, 1]$
 5:     end for
 6:     $q(x) := \sum_{i=0}^{d} r_i T_{2^i}(x)$
 7:     $m \leftarrow \min(q(x_j))$
 8:     $M \leftarrow \max(q(x_j) - m)$
 9:     $p(x) \leftarrow (q(x) - m)/M$
10:     Sort $\{p(x_j)\}_{j=0}^{n-1}$ and update the index.
11:     $\Delta \leftarrow \min(p(x_j) - p(x_{j-1}))$
12:     if $\Delta > 1/n^2$ then
13:         return $p(\cdot)$
14:     else
15:         ctr $\leftarrow$ ctr + 1
16:         if ctr $> T$ then
17:             break
18:         end if
19:     end if
20: end for
21: return Null
```

Even though Algorithm 4 uses $\Omega(n \log^2 n)$ operations, Algorithm 4 is performed on unencrypted data, so the time cost is negligible compared to the computations on encrypted data. Also, we can reuse the randomizing polynomial when we evaluate the same lookup table after.

Remark 4. *If we store the coefficients of the randomizing polynomial together with the encrypted lookup table, Algorithm 4 can be exploited for the case of an encrypted lookup table as well.*

Remark 5. *In our proof-of-concept implementation in Section V, we did not include Algrotihm 4 since we focused on the lookup tables with uniformly distributed key values.*

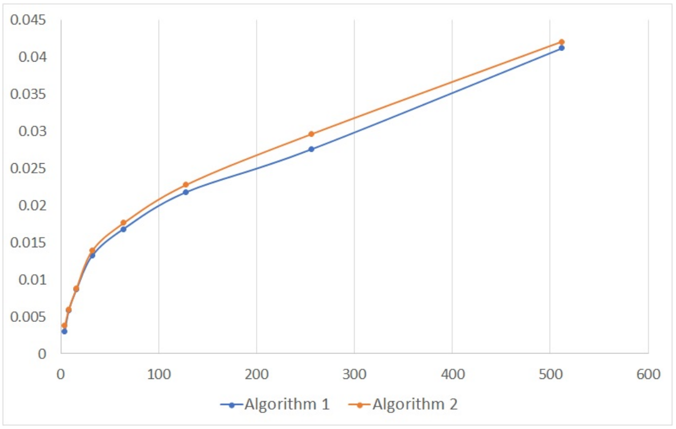

Fig. 4. The amortized running time of Algorithm 1 and 2 for various sizes of encrypted LUTs. Time is in seconds (sec).

V. Experiments

In this section, we implement our algorithms in Section III by using the CKKS scheme as we described in Section IV. We implemented Algorithm 1 and 2 for the encrypted lookup tables, and Algorithm 1, 2 and 3 for the unencrypted lookup tables.

Experimental Environment. All experiments were performed on an Intel Xeon Silver 4114 CPU at 2.20GHz processor. We used a single thread for the experiments.

Parameter. For the CKKS implementation, we used the quotient polynomial ring $\mathbb{Z}_Q[X]/(X^N+1)$ with the dimension $\log_2 N = 17$. The size of the maximum modulus, $\log_2 PQ$ is 2070. We sampled the ternary secret key with a hamming weight of 64. We note that our parameter is about 128-bit secure, i.e., $\approx 2^{128}$ operations are needed to recover plaintext with the current best attacks [25].

A. Private Encrypted Table

For the encrypted LUTs, we implement Algorithm 1 and 2 by using the CKKS scheme. We perform our algorithm for each LUT of size 2^k where $2 \le k \le 9$. We select the table keys uniformly on $[0, 1]$, and the table value randomly on $[0, 1]$.

We report the amortized running time in Table III, and present the graph in Figure 4. We remark that we fully used SIMD property of CKKS, so evaluate 65536 queries simultaneously. We also note that for the smaller size of LUTs, the cost from the homomorphic comparison is dominant, while for the larger size of LUTs, the cost from homomorphic multiplications is dominant.

B. Public Unencrypted Table

For the unencrypted LUTs, we implement Algorithm 1 and 3 by using CKKS scheme. We perform Algorithm 1 for each LUT of size 2^k where $2 \le k \le 12$, and Algorithm 3 for each LUT of size 2^k where $2 \le k \le 13$, We select the table keys uniformly on $[0, 1]$, and the table value randomly on $[0, 1]$.

TABLE III
THE AMORTIZED RUNNING TIME OF ALGORITHM 1 AND 2 FOR VARIOUS SIZES OF ENCRYPTED LUTS. TIME IS IN MILLISECONDS (MS).

	4	8	16	32	64	128	256	512
Algorithm 1	3.01	5.80	8.67	13.21	16.81	21.78	27.55	41.18
Algorithm 2	3.78	5.98	8.86	13.90	17.61	22.76	29.58	42.02

TABLE IV
THE AMORTIZED RUNNING TIME OF ALGORITHM 1 AND 3 FOR VARIOUS SIZES OF UNENCRYPTED LUTS. TIME IS IN MILLISECONDS (MS).

	4	8	16	32	64	128	256	512	1024	2048	4096	8192
Algorithm 1	2.98	5.72	8.75	12.95	15.93	20.06	24.65	31.74	43.72	71.18	113.54	-
Algorithm 3	2.97	5.71	8.40	12.66	15.25	18.46	21.46	25.17	29.70	43.54	51.28	70.26

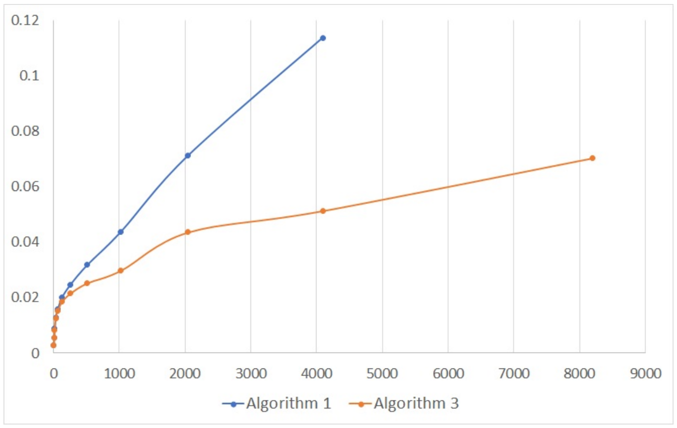

Fig. 5. The amortized running time of Algorithm 1 and 3 for various sizes of unencrypted LUTs. Time is in seconds (sec).

We report the amortized running time in Table IV, and present the graph in Figure 5. We remark that we fully used the SIMD property of CKKS, so evaluate 65536 queries simultaneously. We also note that for smaller LUTs, the cost from `Compare` is dominant, while for larger LUTs, the costs from `Mult` and `cMult` are dominant. The difference in running time between Algorithm 1 and 3 is substantial; thus, the efficiency of Algorithm 3 we suggested in Section III-B3 has been proved.

VI. CONCLUSION

In this work, we suggested HE-based algorithms for the evaluation of lookup tables on encrypted queries. For the encrypted lookup tables, we introduced two algorithms that use $O(\log n)$ homomorphic comparisons and $O(n)$ homomorphic multiplications. For the unencrypted lookup tables, we proposed a better algorithm that uses $O(\log n)$ homomorphic comparisons, $O(\sqrt{n})$ homomorphic multiplications between ciphertexts, and $O(n)$ homomorphic multiplications between a ciphertext and a plaintext.

REFERENCES

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," Journal of Cryptology, vol. 33, no. 1, pp. 34–91, 2020.

[2] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in Annual international conference on the theory and applications of cryptographic techniques. Springer, 2015, pp. 617–640.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," ACM Transactions on Computation Theory (TOCT), vol. 6, no. 3, pp. 1–36, 2014.

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, 2012.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in International Conference on the Theory and Application of Cryptology and Information Security. Springer, 2017, pp. 409–437.

[6] N. Drucker, G. Moshkowich, T. Pelleg, and H. Shaul, "BLEACH: cleaning errors in discrete computations over ckks," Journal of Cryptology, vol. 37, no. 1, p. 3, 2024.

[7] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," in 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2019, pp. 40–48.

[8] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, "Efficient sorting of homomorphic encrypted data with k-way sorting network," IEEE Transactions on Information Forensics and Security, vol. 16, pp. 4389–4404, 2021.

[9] E. Aharoni, N. Drucker, E. Kushnir, R. Masalha, and H. Shaul, "Generating one-hot maps under encryption," in International Symposium on Cyber Security, Cryptology, and Machine Learning. Springer, 2023, pp. 96–116.

[10] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers," in International Conference on the Theory and Application of Cryptology and Information Security. Springer, 2019, pp. 415–445.

[11] J. H. Cheon, D. Kim, and D. Kim, "Efficient homomorphic comparison methods with optimal complexity," in International Conference on the Theory and Application of Cryptology and Information Security. Springer, 2020, pp. 221–256.

[12] E. Lee, J.-W. Lee, Y.-S. Kim, and J.-S. No, "Optimization of homomorphic comparison algorithm on RNS-CKKS scheme," IEEE Access, vol. 10, pp. 26 163–26 176, 2022.

[13] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5. Springer, 2021, pp. 1–19.

[14] K. Kluczniak and L. Schild, "FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 501–537, 2023.

[15] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in TFHE," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 229–253, 2021.

[16] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Parameter optimization and larger precision for (T)FHE," Journal of Cryptology, vol. 36, no. 3, p. 28, 2023.

[17] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption," in 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021, pp. 1057–1073.

[18] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, “Chimera: Combining ring-lwe-based fully homomorphic encryption schemes,” Journal of Mathematical Cryptology, vol. 14, no. 1, pp. 316–338, 2020.

[19] Y. Bae, J. H. Cheon, J. Kim, J. H. Park, and D. Stehlé, “HERMES: Efficient ring packing using mlwe ciphertexts and application to transciphering,” in Annual International Cryptology Conference. Springer, 2023, pp. 37–69.

[20] J. L. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup, “Doing real work with fhe: the case of logistic regression,” in Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, 2018, pp. 1–12.

[21] E. Kushnir, G. Moshkowich, and H. Shaul, “Secure range-searching using copy-and-recurse,” Cryptology ePrint Archive, 2023.

[22] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for approximate homomorphic encryption,” in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2018, pp. 360–384.

[23] I. Iliashenko and V. Zucca, “Faster homomorphic comparison operations for bgv and bfv,” Proceedings on Privacy Enhancing Technologies, vol. 2021, no. 3, pp. 246–264, 2021.

[24] M. S. Paterson and L. J. Stockmeyer, “On the number of nonscalar multiplications necessary to evaluate polynomials,” SIAM Journal on Computing, vol. 2, no. 1, pp. 60–66, 1973.

[25] M. R. Albrecht. (2017) A Sage Module for estimating the concrete security of Learning with Errors instances. https://bitbucket.org/malb/lwe-estimator.

Jung Hee Cheon is a professor of mathematics and the director of IMDARC (the Center for Industrial Math) at Seoul National University. He received his Ph.D. degree in mathematics from KAIST and is working on computational number theory, cryptology, and information security. He served as a program committee member of Crypto/Eurocrypt and was a program co-chair of ANTS XI, Asiacrypt 2015-2016, MathCrypt 2018-2021, and PQCrypto 2021-2022. He is one of two invited speakers in Asiacrypt 2020. He is an associate editor of “Design, Codes and Cryptography”, “Journal of Communication Network,” and “Journal of Cryptology,” which is the flagship journal in cryptology. He is a co-inventor of braid cryptography and approximate homomorphic encryption HEaaN. He received the Best Paper Award in Asiacrypt 2008 and Eurocrypt 2015, was selected as Scientist of the Month by the Korean government in 2018, and won the POSCO Science Prize in 2019 and the PKC Test-of-Time award in 2021. He received the Korean Order of Service Merit (Green Stripes, 2022) and is a member of the Korean Academy of Science and Technology. He was appointed as a fellow of IACR in 2023.

Hyeongmin Choe received a B.S. degree from the Department of Mathematical Science, Seoul National University, in 2019, where he is currently pursuing a Ph.D. degree advised by Prof. Jung Hee Cheon. He is working on lattice-based cryptography, focusing on homomorphic encryption, public key encryption, and digital signatures.

Jai Hyun Park received a B.S. degree from the Department of Mathematical Science, Seoul National University, in 2020, where he is currently pursuing a Ph.D. degree advised by Prof. Jung Hee Cheon. His research interests include cryptography from theory to practice, focusing on homomorphic encryption and its applications.