

# GraSS: Graph-based Similarity Search on Encrypted Query

Duhyeong Kim<sup>†</sup>, Yujin Nam<sup>‡</sup>, Wen Wang<sup>†</sup>, Huijing Gong<sup>†</sup>, Ishwar Bhati<sup>†</sup>, Rosario Cammarota<sup>†</sup>,  
Tajana S. Rosing<sup>‡</sup>, Mariano Tepper<sup>†</sup> and Theodore L. Willke<sup>†</sup>

<sup>†</sup>Intel Labs

<sup>‡</sup>University of California San Diego

**Abstract**—Similarity search, i.e., retrieving vectors in a database that are similar to a query, is the backbone of many applications. Especially, graph-based methods show state-of-the-art performance. For sensitive applications, it is critical to ensure the privacy of the query and the dataset. In this work, we introduce GraSS, a secure protocol between client (query owner) and server (dataset owner) for graph-based similarity search based on fully homomorphic encryption (FHE). Both the client-input privacy against the server and the server-input privacy against the client are achievable based on underlying security assumptions on FHE.

We first propose an FHE-friendly graph structure with a novel index encoding method that makes our protocol highly scalable in terms of data size, reducing the computational complexity of neighborhood retrieval process from  $O(n^2)$  to  $\tilde{O}(n)$  for the total number of nodes  $n$ . We also propose several core FHE algorithms to perform graph operations under the new graph structure. Finally, we introduce GraSS, an end-to-end solution of secure graph-based similarity search based on FHE. To the best of our knowledge, it is the first FHE-based solution for secure graph-based database search.

We implemented GraSS with an open-source FHE library and estimated the performance on a million-scale dataset. GraSS identifies (approximate) top-16 in about 83 hours achieving search accuracy of 0.918, making it over  $28\times$  faster than the previous best-known FHE-based solution.

## 1. Introduction

Similarity search is a foundational technique for a wide range of applications including image generation [1], natural language processing [2], and recommendation systems [3]. In Retrieval-Augmented Generation (RAG) [4]–[6], similarity search extends the outstanding capabilities of Generative AI models to obtain factually accurate, up-to-date, and verifiable results. A client issues a query vector to the database to retrieve items whose vectors are similar to the query. When searching through millions to billions of vectors, exact search becomes impractical and approximate methods are used in virtually all practical deployments. Among those, graph methods dominate due to their improved performance [7]. However, in applications involving sensitive user data (e.g., medical records, genomics, or targeted advertising), standard similarity search can create significant privacy risks, exposing sensitive information in the query and database data.

To address the privacy concerns, our work proposes a robust solution for a secure similarity search protocol based on fully homomorphic encryption (FHE) [8]–[15] that allows end-to-end computation over encrypted data without decryption in the intermediate stages. In general, FHE offers *non-interactive* secure inference service between a client and a server, which works as follows: 1. The client sends an encrypted query to the server, 2. the server performs an FHE algorithm corresponding to the inference service, 3. the server sends back the encrypted inference result to the client, and 4. the client obtains the cleartext inference result through decryption. We follow exactly the same protocol for graph-based similarity search where the query owner is the client and the graph owner is the server.

**Target Privacy.** In client-server model, there are two types of input privacy: The server who holds the graph data should learn nothing about the query, and the client, the query owner, should learn nothing about the graph besides the similarity search result. Client-input privacy against the server directly comes from indistinguishability under chosen plaintext attack (IND-CPA) security [16] of FHE. Server-input privacy against the client, while not directly guaranteed by IND-CPA, can be achieved by well-known circuit-privacy techniques such as noise flooding [17], [18] or ciphertext sanitization [19]–[22].

### 1.1. Our Work

In this work, we propose GraSS, a secure protocol for graph-based similarity search based on FHE, which is highly scalable in terms of the dataset size. To the best of our knowledge, this work is the first to explore the use of FHE in privacy-preserving database search based on graph.

**FHE-friendly Graph Data Structure.** At a high level, the main bottleneck of designing an efficient FHE algorithm for graph-based methods is *iterative* neighborhood information retrieval. For a single-round retrieval process, we can simply use conventional private information retrieval (PIR) [23], [24] to mask each column of the database  $DB \in \mathbb{R}^{n \times d}$  with the one-hot-encoding vector  $\mathbf{e}_x \in \{0, 1\}^n$  of an index  $x \in [0, n)$ , which retrieves the  $i$ -th row of  $DB$  in encrypted form. However, graph-based methods require to perform the retrieval process iteratively, which means a part of the retrieved (neighbor index) information should be transformed into the one-hot-encoding vector form for the next retrieval process.

The most naive way is to encode the index of all neighborhoods of every node into the one-hot-encoding vector form. However, the node index part of the graph database would be an  $n \times mn$  matrix, where  $m$  is the maximal graph degree, which results in  $O(n^2)$  computational complexity for neighborhood retrieval process. Such complexity is infeasible for a million-scale dataset.

In this paper, we first define an FHE-friendly structure of graph data which enables neighborhood information retrieval with  $\tilde{O}(n)$  complexity (Section 3.1). We also introduce several core FHE algorithms according to the new graph data structure, whose usage is not restricted to similarity search but also applicable to any other graph-based applications (Section 3.2). Finally, we apply the core FHE building blocks to design and implement the end-to-end secure solution GraSS for graph-based similarity search (Section 4).

**Design Rationale on FHE Algorithms.** Most of prior work [25]–[27] on FHE-based similarity search focused on homomorphic computation of global top- $k$  over the entire database and utilized bit-wise FHE [13], [14], which support homomorphic logical gate operations and look-up table operations as basic functionalities, since the majority of the global top- $k$  computation is comparison operation which is less efficient to compute with word-wise FHE [9]–[11], [15]. However, in general, bit-wise FHE is not as efficient as word-wise FHE for highly-parallelizable computations. Hence, using bit-wise FHE alone would not be an optimal design solution for super large-scale data. Refer to Section 2.2 for more details on bit/word-wise FHE.

For GraSS, we leverage the strengths of both worlds: We use word-wise FHE for highly-parallelizable polynomial operations and bit-wise FHE for poorly-parallelizable logical (comparison) operations, which can be homomorphically transformed to each other by scheme-switching [28], [29]. Note that graph-based similarity search requires fewer comparison operations (e.g., top- $k$ ) and more polynomial operations (e.g., distance computation, neighborhood retrieval) compared to non-graph-based similarity search algorithms. Hence, our design rationale fits perfectly with graph-based similarity search.

**Performance.** We implemented our end-to-end FHE solution GraSS for graph-based similarity search with CKKS (word-wise FHE) [15] and FHEW (bit-wise FHE) [13], [30] based on OpenFHE [31], which is an open-source FHE library that supports scheme-switching between CKKS and FHEW. Compared to the prior state-of-the-art based on FHE, we achieve over  $4.13\times$  and  $28.72\times$  speedups for 100k and 1 million-scale datasets, respectively.

## 1.2. Related Work

Various privacy-preserving techniques have been explored in the context of similarity search. Note that in some work, similarity search is addressed as nearest neighbor search, or as part of k-Nearest Neighbors (kNN) classifiers. Since our focus is on search algorithms rather than classifiers,

we include works on kNN classifiers but omit discussions on subsequent steps, such as majority voting mechanisms.

**FHE-based Similarity Search.** There are several works on FHE-based solution for similarity search following the client-server model [25]–[27], [32]. The most recent work [27] proposed a novel top- $k$  sorting network with only  $O(n \log k)$  comparison operations for  $k \ll \sqrt{n}$ . Their implementation based on bit-wise FHE takes about 642 seconds to find exact top-5 elements among the  $n = 1000$  data points with single-thread CPU. However, the implementation is limited to low-precision comparison operation (e.g., 6-bit in [27]), and its scalability in terms of the bit precision and the data size is non-trivial. In general, we need larger precision to obtain top- $k$  results with larger datasets. For example, while the input precision of comparison operations in [27] was 6-bit, our experiments for 1M dataset required about 14-bit precision for each comparison operation.

To obtain the same level of precision with ours, previous work based on bit-wise FHE need to either 1. increase FHE parameters accordingly or 2. encrypt a number into multiple ciphertexts after parsing into small digits. The former method will result in at least  $2^{14-6} = 256\times$  computational overhead<sup>1</sup> from the result of [27]. The latter method requires multi-precision computation for every operation, which implies another huge computational overhead. On the other hand, GraSS fully utilizes the SIMD packing property and large plaintext space of word-wise FHE in the protocol except in the argmin and local top- $k$  (for  $\leq 100$  inputs) steps, working more efficiently than the previous work for million-scale data. Refer to Section 5.4 for more details on the comparison with our work.

**MPC-based Similarity Search** Multi-Party Computation (MPC) is another approach to build a secure protocol that reduces computational demands but introduces significant communication overhead through the use of oblivious transfer and garbled circuits. The work of [18] utilized multiple cryptographic primitives including additive homomorphic encryption, oblivious RAM, and garbled circuits. The computational cost of overall protocol is very low: 400 seconds to run the protocol over million-to-billion-scale data. Contrary to FHE; however, MPC-based protocols are inherently interactive with large number of interaction rounds, and hence the client should stay online until the end of the protocols with large amount of communication cost, which might not be appropriate in some real-world use-cases.

**Other Graph Applications based on FHE.** There exist a few works on FHE-based graph operation/application (besides database search), such as Bellman-Ford algorithm solving the shortest-path problem [34] and graph convolutional network Inference [35]. These works commonly represent the data structure of a graph as its adjacent matrix of the size  $n \times n$ . The adjacent-matrix-based operation fundamentally accompanies  $O(n^2)$  computational

1. We follow the methodology of [33] to estimate the computational cost with respect to target precision.

complexity, and it is infeasible to run the algorithms over million-scale data in encrypted state.

It is worth noting that  $n \times n$  matrix-based operations have been a common approach for FHE-based similarity search, which similarly face scalability challenges [25], [26]. In contrast, the graph-based method we emphasize in this work conducts search in a graph-structured database using a non-exhaustive approach, offering improved scalability.

## 2. Preliminaries

### 2.1. Notations

Let  $G = (\mathcal{V}, \mathcal{E})$  be a regular graph (i.e., the same degree for each node) for the set of nodes  $\mathcal{V} := \{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}\} \subset \mathbb{R}^d$  and the set of edges  $\mathcal{E} := \{(\mathbf{v}_i, \mathbf{v}_{s_{i,j}})\}_{0 \leq i < n, 0 \leq j < m} \subseteq \mathcal{V} \times \mathcal{V}$ . We identify the index of each node  $\mathbf{v}_i$  as  $i \in [0, n)$ ; hence the node  $i$  has the neighborhoods with index  $s_{i,j}$  for  $0 \leq j < m$ . Here,  $m (< n)$  is the number of neighborhoods of each node (i.e., the degree of the graph). Let  $\ell$  be the smallest power-of-two integer that is larger than or equal to  $\lceil \log n \rceil$ , the maximal bit length of each node index.

Let  $R := \mathbb{Z}[X]/(X^N + 1)$  be the ring of integer polynomials modulo  $(X^N + 1)$  for power-of-two  $N$ , and let  $R_q := R/qR = \mathbb{Z}_q[X]/(X^N + 1)$  for any integer  $q > 1$ . We use the polynomial ring  $R$  of the ring dimension  $N$  for word-wise FHE. We denote the SIMD batch size and the plaintext modulus of word-wise FHE by  $b$  and  $t$ , respectively. For bit-wise LWE, we denote the LWE dimension and plaintext modulus by  $N'$  and  $t'$ , respectively. More details about word/bit-wise FHE will be explained in the next subsection.

### 2.2. Homomorphic Encryption

A number of FHE schemes [9]–[15], [36] have been suggested following Gentry’s blueprint [8], and the followings are regarded as the state-of-the-art FHE schemes with the best performance nowadays: BGV [9], B/FV [10], [11], CKKS [15] and FHEW/TFHE [13], [14].

**2.2.1. Word-wise FHE for Batch Polynomial Arithmetic.** The word-wise FHE schemes BGV, B/FV and CKKS commonly allow the batch encryption of multiple word-size plaintexts into a single ciphertext and support homomorphic addition and multiplication as basic operations.

Let  $\mathcal{M}^b$  be the message vector space which varies per scheme, e.g.,  $\mathcal{M} = \mathbb{Z}_t$  for a plaintext modulus  $t$  and  $1 < b \leq N$  for BGV and B/FV, and  $\mathcal{M} = \mathbb{R}$  and  $b = N/2$  for CKKS. The first step of encryption is to encode the message vector  $\mathbf{z} \in \mathcal{M}^b$  into an integer polynomial  $m(X) := \text{Ecd}(\mathbf{z}) \in R$ . We call  $m(X)$  the plaintext polynomial corresponding to the message vector  $\mathbf{z}$ . Then, the next step is to encrypt the plaintext  $m(X)$  with a secret key  $\text{sk}$  (with public key  $\text{pk}$  in public-key encryption setting) into a ciphertext  $\text{ct} := \text{Enc}(m(X)) \in R_Q^2$  for some large modulus  $Q$ , which is set depending on the depth of the target computation. The decryption procedure works in reverse: Decrypt a ciphertext into an underlying

plaintext polynomial and then decode the plaintext into a message vector in  $\mathcal{M}^b$ . For simplicity, we will call the whole procedure from ciphertext to message vector as the decryption  $\text{Dec}$ .

Let  $\text{pt}_0 = \text{Ecd}(x_0, \dots, x_{b-1})$ ,  $\text{pt}_1 = \text{Ecd}(y_0, \dots, y_{b-1})$  and  $\text{ct}_i = \text{Enc}(\text{pt}_i)$  for  $i = 0, 1$ . Then, the following homomorphism properties commonly hold for word-wise FHE:

$$\begin{aligned} \text{Dec}(\text{ADD}(\text{ct}_1, \text{ct}_2)) &= (x_0 + y_0, \dots, x_{b-1} + y_{b-1}), \\ \text{Dec}(\text{MULT}(\text{ct}_1, \text{ct}_2)) &= (x_0 \cdot y_0, \dots, x_{b-1} \cdot y_{b-1}), \\ \text{Dec}(\text{ROTATE}(\text{ct}_1; k)) &= (x_k, \dots, x_{b-1}, x_0, \dots, x_{k-1}). \end{aligned}$$

ADD, MULT and ROTATE indicate homomorphic addition, multiplication and (left-)rotation algorithms of ciphertexts, respectively. Note that the equality in each homomorphic property need to be replaced by the approximate equality in CKKS. The ciphertext-ciphertext (ct-ct) multiplication and homomorphic rotation algorithms require key-switching keys as an additional input, which are also known as relinearization key and rotation key. We omit these key inputs in MULT and ROTATE notations for convenience.

As described above, word-wise FHE schemes support parallel addition and multiplication homomorphically in a single instruction, multiple data (SIMD) manner. The SIMD property makes word-wise FHE schemes quite efficient for highly-parallelizable polynomial arithmetic in terms of amortized computational cost. Refer to Appendix A for a note on non-polynomial operations and additional FHE contexts including ciphertext level and bootstrapping in word-wise FHE.

**2.2.2. Bit-wise FHE for Logical Operations.** Bit-wise FHE is another approach to construct an efficient FHE scheme that supports logical operations (e.g., gate operations, comparison, etc.) and digital lookup table (LUT) as basic operations, contrary to word-wise FHE in the previous subsection which support polynomial arithmetics. FHEW [13], [30] and TFHE [37] are included in bit-wise schemes. These schemes use relatively small FHE parameters compared to word-wise FHE: bit-wise FHE encrypts a single message  $z \in \mathbb{Z}_{t_{LWE}}$  for relatively small plaintext modulus  $t_{LWE}$  into an LWE ciphertext of the form  $(\vec{a}, b) \in \mathbb{Z}_q^{N_{LWE}} \times \mathbb{Z}_q$ . Here, the (LWE) dimension  $N_{LWE}$  is usually between  $2^9$  and  $2^{11}$  and the ciphertext modulus is as small as  $q \approx 2^{11}$  in practice [38].

The main idea of bit-wise FHE is to perform bootstrapping for every basic homomorphic operation. Hence, when a user is given bit-wise FHE APIs, the user does not need to care about the ciphertext noise growth in the FHE program design. Due to such small parameter setting and the algorithmic differences, each basic operation of bit-wise FHE takes much lower latency than word-wise FHE, e.g., only few milliseconds per gate bootstrapping in FHEW/TFHE. However, usually bit-wise FHE schemes do not support batch computation in contrast to word-wise FHE, and hence bit-wise FHE is preferred to word-wise FHE when the target computation is poorly parallelizable and/or is composed of logical operations.

In this paper, we make the best use of the homomorphic sign evaluation algorithm `SIGN` recently introduced in [33] to compare the encrypted inputs with relatively high precision (e.g., 14-bit) in graph-based similarity search. This algorithm utilizes an intermediate-size LWE ciphertext with larger ciphertext/plaintext modulus ( $q_{large}, t_{large}$ ) compared to the original FHEW/TFHE, which enables our work to achieve much higher-precision sign evaluation than previous work.

### 2.3. Scheme-Switching Technique

The baseline strategy of our FHE algorithm design (GraSS) is to utilize word-wise FHE for highly parallelizable arithmetic operations (add, mult) and bit-wise FHE for poorly parallelizable logical operations. Therefore, we need a conversion technique between word-wise FHE and bit-wise FHE for an end-to-end FHE solution, which is called scheme-switching. The idea of scheme-switching between different FHE schemes was first introduced in [28], and its efficiency was significantly improved by a follow-up study [29]. Note that OpenFHE [31], one of a few open-source FHE libraries that are under maintenance, supports the scheme-switching between the word-wise FHE scheme CKKS and the bit-wise FHE scheme FHEW.

### 2.4. Graph-based Similarity Search

Graph-based methods are currently the best performing techniques for million-to-billion-scale similarity search [7]. Instead of performing global top- $k$  search over the entire data, Graph-based similarity search performs *local* top- $k$  search iteratively, which dramatically reduces the total number of comparison operations.

Let  $G = (\mathcal{V}, \mathcal{E})$  be the regular graph for the set of nodes  $\mathcal{V} := \{0, 1, \dots, n-1\}$  and the set of edges  $\mathcal{E} := \{(i, s_{i,j})\}_{0 \leq i < n, 0 \leq j < m} \subseteq \mathcal{V} \times \mathcal{V}$ , where  $s_{i,0} := i$  (self-node). Each node  $i \in \mathcal{V}$  corresponds to a feature vector  $\mathbf{v}_i \in \mathbb{R}^d$  for some  $d > 0$ . We encode the given graph  $G$  as Table 1, where  $\mathbf{idx}_i$  denotes an encoding of each node  $i \in V$ . A fundamental graph-based similarity search algorithm is as following:

---

**Algorithm 1** Greedy Graph Search [7, Algorithm 1]

---

**Require:** graph  $G = (\mathcal{V}, \mathcal{E})$ , query  $q$ , # of output  $k \in \mathbb{N}$ , search window size  $w \geq k$ , initial candidates  $\mathcal{S} \subset \mathcal{V}$ , similarity function `sim`

**Ensure:**  $k$  approximate nearest neighbors to  $q$  in  $G$

- 1:  $Q \leftarrow \mathcal{S}, E \leftarrow \emptyset$
  - 2: **while**  $Q - E \neq \emptyset$  **do**
  - 3:    $x \leftarrow$  The closest node to  $q$  in  $Q - E$  w.r.t. `sim`
  - 4:    $E \leftarrow E \cup \{x\}$
  - 5:    $Q' \leftarrow Q \cup N(x)$
  - 6:    $Q \leftarrow$  The  $w$  closest nodes to  $q$  in  $Q'$  w.r.t. `sim`
  - 7: **end while**
  - 8: **return** The  $k$  nearest nodes to  $q$  in  $Q$  w.r.t. `sim`
- 

Here,  $Q$  is a set of nodes which is updated to get closer to the input query  $q$  as the algorithm continues.  $E$  is a set of explored nodes (i.e., marked as the closest node to  $q$  in  $Q - E$  in line 3) and is important for the correctness of

the algorithm as it helps the algorithm to escape the local minimum. Note that the self-node  $x$  is also contained in the set of neighborhoods  $N(x)$ . The similarity function `sim` is usually defined as 2-norm distance  $\text{sim}(i, j) = \|\mathbf{v}_i - \mathbf{v}_j\|_2$  over  $\mathbb{R}^d$ .

## 3. FHE Framework for Graph Search

In this section, we introduce a novel approach that encodes a given graph dataset into its FHE-friendly form, as well as a set of core FHE algorithms that are key to performing graph operations homomorphically.

### 3.1. FHE-friendly Data Structure of Graph

We first encode a graph  $G = (\mathcal{V}, \mathcal{E})$  with small degree  $m = o(n)$  into a matrix/table form (See Table 1), where  $\mathbf{idx}_i$  denotes a proper encoding of the node  $i \in V$ .

**How to define the index encoding  $\mathbf{idx}_{s_{i,j}}$ ?** A common operation required for graph-based methods is to *retrieve the neighborhood information*  $\{(\mathbf{v}_{s_{x_i,j}}, \mathbf{idx}_{s_{x_i,j}})\}_{0 \leq j < m}$  of the current node  $x_i \in [0, n)$  (of  $i$ -th iteration), which is equivalent to  $x_i$ -th row extraction of the graph table, and one of the neighborhoods  $s_{x_i,j}$  is usually selected as the next node  $x_{i+1} = s_{x_i,j}$ . For the row extraction, we generally need the *one-hot encoding vector*  $\mathbf{e}_{x_i} \in \{0, 1\}^n$  of the node  $x_i$  and then mask every column of the table with  $\mathbf{e}_{x_i}$ . Then, all the rows except the  $x_i$ -th row is zeroized, and hence the summation of all rows after masking results in the  $x_i$ -th row. Namely, if we have  $\text{Enc}(\mathbf{e}_{x_i})$  for the node  $x_i$ , then we can simply extract its neighborhood information as  $\text{Enc}(\mathbf{v}_{s_{x_i,j}})$  and  $\text{Enc}(\mathbf{idx}_{s_{x_i,j}})$  for  $0 \leq j < m$ .

In graph-based methods, such neighborhood retrieval process needs to be done iteratively. That is, we need the encryption of the one-hot encoding vector of the next node  $\text{Enc}(\mathbf{e}_{x_{i+1}})$  for the next neighborhood retrieval process in  $(i+1)$ -th iteration. Hence, the most naive way to do this would be to set the encoding of each neighbor  $s_{i,j}$  as its one-hot encoding vector (i.e.,  $\mathbf{idx}_{s_{i,j}} = \mathbf{e}_{s_{i,j}}$ ) such that  $\text{Enc}(\mathbf{e}_{x_{i+1}})$  can be directly obtained as  $\text{Enc}(\mathbf{idx}_{s_{x_i,j}}) = \text{Enc}(\mathbf{e}_{x_{i+1}})$  for some  $0 \leq j < m$ .

However, this naive method is not feasible in practice due to the substantially large computational cost  $O(mn^2)$  of the neighborhood retrieval process, for the number of neighbor nodes  $m$  and the total number of nodes  $n$ . To be precise, if we set  $\mathbf{idx}_{s_{i,j}} = \mathbf{e}_{s_{i,j}}$ , then the graph table for the neighborhood information becomes an  $n \times mn$  matrix since  $\mathbf{idx}_{s_{i,j}} = \mathbf{e}_{s_{i,j}} \in \{0, 1\}^n$  for  $0 \leq i < n$  and  $0 \leq j < m$ . Hence, masking the graph table with a one-hot encoding vector requires  $mn^2$  scalar multiplications. Since we target million-scale  $n$ , the quadratic computational cost is considered unacceptable.

Instead, in GraSS, we set the index encoding  $\mathbf{idx}_{s_{i,j}}$  of  $s_{i,j}$  as its *binary representation* instead of its one-hot encoding, i.e.,

$$s_{i,j} = \sum_{u=0}^{\lceil \log n \rceil - 1} 2^u \cdot \text{id}x_{s_{i,j},u}$$

for  $\mathbf{idx}_{s_{i,j}} := (\text{id}x_{s_{i,j},0}, \dots, \text{id}x_{s_{i,j},\lceil \log n \rceil - 1}) \in \{0, 1\}^{\lceil \log n \rceil}$ . The main advantage of exploiting binary

TABLE 1: Node Vectors and Binary Index Table of Neighborhoods

Index	Neighborhood Information			
0	$(\mathbf{v}_{s_{0,0}}, \mathbf{idx}_{s_{0,0}})$	$(\mathbf{v}_{s_{0,1}}, \mathbf{idx}_{s_{0,1}})$	$\dots$	$(\mathbf{v}_{s_{0,m-1}}, \mathbf{idx}_{s_{0,m-1}})$
1	$(\mathbf{v}_{s_{1,0}}, \mathbf{idx}_{s_{1,0}})$	$(\mathbf{v}_{s_{1,1}}, \mathbf{idx}_{s_{1,1}})$	$\dots$	$(\mathbf{v}_{s_{1,m-1}}, \mathbf{idx}_{s_{1,m-1}})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	$(\mathbf{v}_{s_{n-1,0}}, \mathbf{idx}_{s_{n-1,0}})$	$(\mathbf{v}_{s_{n-1,1}}, \mathbf{idx}_{s_{n-1,1}})$	$\dots$	$(\mathbf{v}_{s_{n-1,m-1}}, \mathbf{idx}_{s_{n-1,m-1}})$

representation instead of one-hot encoding is that the computational cost of masking the graph table is reduced from  $O(n^2)$  to  $O(n \log n)$ . The remaining task is to convert the binary representation into the one-hot encoding vector homomorphically. Binary-to-one-hot conversion is explained in detail in Section 3.2.1 where we show that the conversion also requires  $O(n \log n)$  scalar multiplications which is similar to the masking process.

### 3.2. Core FHE Building Blocks

**3.2.1. Binary-to-One-hot Encoding Conversion.** We first explain the FHE algorithm of the conversion from the given binary representation  $\mathbf{idx}_x \in \{0, 1\}^{\lceil \log n \rceil}$  of the node  $x \in [0, n)$  into its one-hot encoding vector  $\mathbf{e}_x \in \{0, 1\}^n$  in an encrypted state. The conversion process is highly parallelizable, and hence we use word-wise FHE with plaintext batching property rather than bit-wise FHE. The batch size of word-wise FHE is denoted by  $b$  (Refer to Section 2.2.1).

The one-hot encoding vector  $\mathbf{e}_x$  can be interpreted as  $(x == i?)_{0 \leq i < n}$ , where  $i$ -th entry checks the equality of  $x$  and  $i$ . Our core observation is that the equality check  $(x == i?)$  for each  $0 \leq i < n$  can be re-expressed with the binary representation of  $s$  and  $i$  as  $\prod_{j=0}^{\lceil \log n \rceil - 1} (idx_{x,j} == idx_{i,j}?)$ . Hence, the one-hot en-

---

**Algorithm 2** Homomorphic Binary to One-Hot Conversion (BintoOneHot)

---

**Require:**  $\text{ct}_j = \text{Enc}(idx_{x,j}, idx_{x,j}, \dots, idx_{x,j})$  and  $\text{pt}_{j,u} := \text{Ecd}(\mathbf{b}_{j,u})$  for  $0 \leq j < \lceil \log n \rceil$  and  $0 \leq u < \lceil n/b \rceil$ .

**Ensure:**  $\{\text{ct}_{out,u}\}_{0 \leq u < \lceil n/b \rceil}$  (Encryption of  $\mathbf{e}_s$ )

```

1: for  $u \leftarrow 0$  to  $\lceil n/b \rceil$  do
2:   for  $j \leftarrow 0$  to  $\lceil \log n \rceil - 1$  do
3:      $\text{ct}_{j,u} \leftarrow \text{MULT}(\text{ct}_j, 2 \cdot \text{pt}_{j,u} - 1)$ 
4:      $\text{ct}_{j,u} \leftarrow \text{ADD}(\text{ct}_{j,u}, 1 - \text{pt}_{j,u})$ 
5:   end for
6:   for  $r \leftarrow \lceil \log \lceil \log n \rceil \rceil - 1$  to 0 do
7:     for  $j \leftarrow 0$  to  $2^r - 1$  do
8:       if  $j + 2^r < \lceil \log n \rceil$  then
9:          $\text{ct}_{j,u} \leftarrow \text{MULT}(\text{ct}_{j,u}, \text{ct}_{j+2^r,u})$ 
10:      end if
11:    end for
12:  end for
13:   $\text{ct}_{out,u} \leftarrow \text{ct}_{0,u}$ 
14: end for
15: return  $\text{ct}_{out,u}$ 

```

---

coding vector of the node  $s$  can be expressed as follows:

$$\begin{aligned}
 & (x == i?)_{0 \leq i < n} \\
 &= \prod_{j=0}^{\lceil \log n \rceil - 1} (idx_{x,j} == idx_{i,j}?)_{0 \leq i < n} \\
 &= \prod_{j=0}^{\lceil \log n \rceil - 1} (idx_{x,j} \cdot (2idx_{i,j} - 1) + (1 - idx_{i,j}))_{0 \leq i < n}
 \end{aligned}$$

, where  $\prod_{j=0}^{\lceil \log n \rceil - 1}$  is the Hadamard (entry-wise) product of the vectors of length  $n$ . Note that  $\mathbf{b}_j := (idx_{i,j})_{0 \leq i < n}$  is pre-computable vector for all  $0 \leq j < \lceil \log n \rceil$ .

Now assume that each entry of the binary representation  $\mathbf{idx}_x \in \{0, 1\}^{\lceil \log n \rceil}$  of the node  $x$  is encrypted separately, i.e.,  $\text{ct}_j := \text{Enc}(idx_{x,j}, idx_{x,j}, \dots, idx_{x,j})$  for  $0 \leq j < \lceil \log n \rceil$ , and each vector  $\mathbf{b}_j$  is parsed into  $\lceil n/b \rceil$  vectors  $\mathbf{b}_{j,u}$  of length  $b$  and then encoded as plaintext polynomials  $\{\text{pt}_{j,u}\}_{0 \leq u < \lceil n/b \rceil}$ . Then, the FHE algorithm for the conversion from binary representation to one-hot encoding is described as Algorithm 2.

**Complexity Analysis.** We use binary tree to multiply  $(idx_{x,j} \cdot (2idx_{i,j} - 1))_{0 \leq i < n} + (1 - idx_{i,j})_{0 \leq i < n}$  for  $0 \leq j < \lceil \log n \rceil$ . Hence, BintoOneHot requires  $\frac{n \lceil \log n \rceil}{b}$  ct-pt mults and  $\frac{n \lceil \log n \rceil}{b}$  ct-ct mults, and consumes  $\lceil \log \lceil \log n \rceil \rceil + 1$  levels.

**3.2.2. Batch Homomorphic Blind Rotation.** After masking each column of Table 1 with the encrypted one-hot encoding vector  $\text{Enc}(\mathbf{e}_{x_i})$  at  $i$ -th iteration, we obtain the binary index vectors  $\{\mathbf{idx}_{s_{x_i,j}}\}_{0 \leq j < m}$  where each entry  $idx_{s_{x_i,j},u}$  for  $0 \leq u < \lceil \log n \rceil$  is encrypted at  $[x_i]_b$ -th slot as  $\text{Enc}(0, \dots, 0, idx_{s_{x_i,j},u}, 0, \dots, 0)$  for  $[x_i]_b := x_i \pmod{b}$ . Since the slot position  $[x_i]_b$  is private, we are not able to directly compare  $\mathbf{idx}_{s_{x_i,j}}$  encrypted at  $[x_i]_b$ -th slot with another binary index  $\mathbf{idx}_{s_{x_{i'},j'}}$  encrypted at  $[x_{i'}]_b$ -th slot which is obtained in  $i'$ -th iteration for  $i' \neq i$ .

A very well-known solution is to broadcast the value to every slot through  $\log b$  homomorphic rotations and additions, i.e.,  $\text{Enc}(0, \dots, 0, idx_{s_{x_i,j},u}, 0, \dots, 0) \mapsto \text{Enc}(idx_{s_{x_i,j},u}, idx_{s_{x_i,j},u}, \dots, idx_{s_{x_i,j},u})$ , so that the slot position of  $idx_{s_{x_i,j},u}$  is not private anymore. However, since we want to broadcast  $idx_{s_{x_i,j},u}$  to every slot for all  $0 \leq u < \lceil \log n \rceil$  and  $0 \leq j < m$ , then the total number of rotations  $\log b \cdot \lceil \log n \rceil \cdot m$  ( $> 10,000$  in practice) to broadcast all the binary index vectors  $\{\mathbf{idx}_{s_{x_i,j}}\}_{1 \leq i \leq m}$  is too large.

2. For example, when  $n = 8$ , we pre-compute:  $\mathbf{b}_0 = (0, 1, 0, 1, 0, 1, 0, 1)$ ,  $\mathbf{b}_1 = (0, 0, 1, 1, 0, 0, 1, 1)$  and  $\mathbf{b}_2 = (0, 0, 0, 0, 1, 1, 1, 1)$ .

---

**Algorithm 3** Batch Homomorphic Blind Rotation (BlindRotate)

---

**Require:**  $\text{ct}_i := \text{Enc}(0, \dots, 0, z_i, 0, \dots, 0)$  ( $z_i$  at  $y$ -th slot) for  $0 \leq i < \lceil \log n \rceil$ ,  $\text{ct}_{idx_{y,u}} := \text{Enc}(idx_{y,u}, \dots, idx_{y,u})$  for  $0 \leq u < \log \ell$ .

**Ensure:**  $\text{ct}_{out} = \text{Enc}(\mathbf{z})$  for  $\mathbf{z} := (z_0, \dots, z_{\ell-1}, z_0, \dots, z_{\ell-1}, \dots, z_0, \dots, z_{\ell-1})$  where  $z_i := 0$  for  $\lceil \log n \rceil \leq i < \ell$ .

```
1: for  $u \leftarrow \log \ell - 1$  to  $0$  do
2:   for  $i \leftarrow 0$  to  $2^u - 1$  do
3:     if  $i + 2^u < \lceil \log n \rceil$  then
4:        $\text{ct}_{i+2^u} \leftarrow \text{ROTATE}(\text{ct}_{i+2^u}; -2^u)$ 
5:        $\text{ct}_i \leftarrow \text{ADD}(\text{ct}_i, \text{ct}_{i+2^u})$ 
6:     end if
7:   end for
8: end for
9:  $\text{ct}_{pack} \leftarrow \text{ct}_0$ 
10: for  $u \leftarrow \log \ell$  to  $\log b - 1$  do
11:    $\text{ct}_{temp} \leftarrow \text{ROTATE}(\text{ct}_{pack}; 2^u)$ 
12:    $\text{ct}_{pack} \leftarrow \text{ADD}(\text{ct}_{pack}, \text{ct}_{temp})$ 
13: end for
14:  $\text{ct}_{bc} \leftarrow \text{ct}_{pack}$ 
15: for  $u \leftarrow 0$  to  $\log \ell - 1$  do
16:    $\text{ct}_{temp1} \leftarrow \text{ROTATE}(\text{ct}_{bc}; 2^u)$ 
17:    $\text{ct}_{temp1} \leftarrow \text{MULT}(\text{ct}_{temp1}, \text{ct}_{idx_{y,u}})$ 
18:    $\text{ct}_{temp2} \leftarrow \text{SUB}(1, \text{ct}_{idx_{y,u}})$ 
19:    $\text{ct}_{bc} \leftarrow \text{MULT}(\text{ct}_{bc}, \text{ct}_{temp2})$ 
20:    $\text{ct}_{bc} \leftarrow \text{ADD}(\text{ct}_{bc}, \text{ct}_{temp1})$ 
21: end for
22: return  $\text{ct}_{out} \leftarrow \text{ct}_{bc}$ 
```

---

We propose a new FHE algorithm called *homomorphic blind rotation*, which performs the homomorphic rotation when the rotation index is encrypted. Instead of broadcasting  $idx_{s_x, j, u}$  at  $[x]_b$ -th slot to every slot, we left-rotate for  $[x]_b$  slots to put  $idx_{s_x, j, u}$  at the 0-th (first) slot. The main advantage of homomorphic blind rotation is that we can pack multiple encryptions of  $idx_{s_x, j, u}$ 's into a single ciphertext and perform the blind rotation simultaneously. As a result, the number of homomorphic rotations does not include the product of three different terms ( $\log b$ ,  $\lceil \log n \rceil$ , and  $m$ ) contrary to the naive broadcasting solution.

Assume that we are given the ciphertexts  $\text{ct}_i := \text{Enc}(0, \dots, 0, z_i, 0, \dots, 0)$  for  $0 \leq i < \lceil \log n \rceil$  where  $z_i$  is positioned at  $y$ -th slot for some private integer  $y \in [0, b-1]$ , and the information of  $y$  is given as entry-wise encryption of the binary representation  $\text{id}\mathbf{x}_y$  of  $y$ , i.e.,  $\text{ct}_{idx_{y,u}} := \text{Enc}(idx_{y,u}, \dots, idx_{y,u})$  for  $0 \leq u < \lceil \log n \rceil$ . Denoting by  $\text{rot}_y(\cdot)$  the right-rotation of a vector for  $y$  slots, each ciphertext  $\text{ct}_i$  can be re-expressed as  $\text{Enc}(\text{rot}_y(z_i, 0, 0, \dots, 0))$ .

We pack  $\text{ct}_i$ 's into  $\text{ct}_{pack} := \text{Enc}(\text{rot}_y(z_0, \dots, z_{\ell-1}, 0, \dots, 0))$  as the first step where  $z_i := 0$  for  $\lceil \log n \rceil \leq i < \ell$ , with  $(\lceil \log n \rceil - 1)$  rotations and additions. The next step is to broadcast the vector  $(z_0, z_1, \dots, z_{\ell-1})$  to the other slots. We can obtain  $\text{ct}_{bc} := \text{Enc}(\text{rot}_y(z_0, \dots, z_{\ell-1}, \dots, z_0, \dots, z_{\ell-1}))$  from  $\text{ct}_{pack}$  through  $(\log b - \log \ell)$  rotations and additions. Then,  $\text{ct}_{bc} = \text{Enc}(\text{rot}_{[y]_\ell}(z_0, \dots, z_{\ell-1}, \dots, z_0, \dots, z_{\ell-1}))$  for  $[y]_\ell := y \pmod{\ell}$ , since the vector  $(z_0, \dots, z_{\ell-1}, \dots, z_0, \dots, z_{\ell-1})$  is a repetition of  $(z_0, \dots, z_{\ell-1})$ . Hence, we only need  $\text{ct}_{idx_{y,u}}$  for  $u < \log \ell$ .

Now we repeat the following process for  $0 \leq u < \log \ell$  to

left-rotate  $\text{ct}_{bc}$  for  $[y]_\ell$  slots: If  $idx_{y,u} = 1$  then left-rotate for  $2^u$  slots, and otherwise do nothing. Denote by  $\mathbf{z} := (z_0, \dots, z_{\ell-1}, \dots, z_0, \dots, z_{\ell-1})$ , the process can be expressed as

- 1) Initialize  $s = [y]_\ell$ .
- 2) Repeat the following steps for  $0 \leq u < \log \ell$ :
  - $\text{rot}_{s-2^u \cdot idx_{y,u}}(\mathbf{z}) \leftarrow (1 - idx_{y,u}) \cdot \text{rot}_s(\mathbf{z}) + idx_{y,u} \cdot \text{rot}_{s-2^u}(\mathbf{z})$ .
  - $s \leftarrow s - 2^u \cdot idx_{y,u}$ .

Since  $[y]_\ell = \sum_{u=0}^{\log \ell - 1} 2^u \cdot idx_{y,u}$ , homomorphic computation of this process results in  $\text{Enc}(\text{rot}_0(\mathbf{z})) = \text{Enc}(\mathbf{z})$ .

**Complexity Analysis.** BlindRotate requires  $\lceil \log n \rceil + \log b$  rotations and  $2 \log \ell$  ct-ct mults, and consumes  $\log \ell$  levels.

**3.2.3. Homomorphic Set Intersection with Binary Indices.** Set intersection is one of the common operations used in many of graph-based methods. In similarity search (Algorithm 1), we need the index set intersection operation in two places: 1. To check if each node in  $Q$  has already been explored in previous iterations (line 3), and 2. To check if each node in  $Q$  has a duplicate in the neighborhood set  $N(x)$  (line 5).

We provide an FHE algorithm of set intersection given the ciphertexts of binary indices of each element. The algorithm takes the encryption of binary indices of each set as input, and then output the encryption of the vector which indicates if each node of one set is contained in the other set or not.

---

**Algorithm 4** Homomorphic Set Intersection (SetInt)

---

**Require:**  $\text{ct}_A = \text{Enc}(\text{idx}_{a_0}, \dots, \text{idx}_{a_0}, \dots, \text{idx}_{a_{|A|-1}}, \dots, \text{idx}_{a_{|A|-1}}), \text{ct}_B = \text{Enc}(\text{idx}_{b_0}, \dots, \text{idx}_{b_{|B|-1}}, \dots, \text{idx}_{b_0}, \dots, \text{idx}_{b_{|B|-1}})$ .

**Ensure:**  $\text{ct}_{out} = \text{Enc}((b_0 \in A?), \dots, (b_{|B|-1} \in A?), \dots)$

- 1:  $\text{ct}_{sub} \leftarrow \text{SUB}(\text{ct}_A, \text{ct}_B)$
- 2:  $\text{ct}_{sq} \leftarrow \text{MULT}(\text{ct}_{sub}, \text{ct}_{sub})$
- 3:  $\text{ct}_{out} \leftarrow \text{SUB}(1, \text{ct}_{sq})$
- 4: **for**  $r \leftarrow \log \ell - 1$  to 0 **do**
- 5:    $\text{ct}_{temp} \leftarrow \text{ROTATE}(\text{ct}_{out}; 2^r)$
- 6:    $\text{ct}_{out} \leftarrow \text{MULT}(\text{ct}_{out}, \text{ct}_{temp})$
- 7: **end for**
- 8: **for**  $r \leftarrow \log |A| - 1$  to 0 **do**
- 9:    $\text{ct}_{temp} \leftarrow \text{ROTATE}(\text{ct}_{out}; 2^{r+\log |B|+\log \ell})$
- 10:    $\text{ct}_{out} \leftarrow \text{ADD}(\text{ct}_{out}, \text{ct}_{temp})$
- 11: **end for**
- 12: **return**  $\text{ct}_{out}$

---

Let  $A := \{a_i\}_{0 \leq i < |A|}$  and  $B := \{b_j\}_{0 \leq j < |B|}$  be the set of nodes and denote the corresponding binary index vectors of each set by  $\{\text{idx}_{a_0}, \text{idx}_{a_1}, \dots, \text{idx}_{a_{|A|-1}}\}$  and  $\{\text{idx}_{b_0}, \text{idx}_{b_1}, \dots, \text{idx}_{b_{|B|-1}}\}$ , respectively. For  $0 \leq i < |A|$  and  $0 \leq j < |B|$ , one can check if  $a_i$  equals to  $b_j$  with the following formula:

$$\prod_{s=0}^{\ell-1} (1 - (idx_{a_i,s} - idx_{b_j,s})^2) = \begin{cases} 1 & \text{if } a_i = b_j \\ 0 & \text{otherwise} \end{cases}.$$

By taking sum for all possible  $j$ 's, we can obtain the formula that checks if  $b_j \in A$  as follows:

$$\sum_{i=0}^{|A|-1} \prod_{s=0}^{\ell-1} (1 - (idx_{a_i,s} - idx_{b_j,s})^2) = \begin{cases} 1 & \text{if } b_j \in A \\ 0 & \text{otherwise} \end{cases}.$$

Note that the left-hand side of the above equation is highly parallelizable, and hence we can leverage the SIMD packing property of word-wise FHE. Assume that  $|A|$ ,  $|B|$ , and  $\lceil \log n \rceil$  are power-of-two (hence  $\lceil \log n \rceil = \ell$ ) and  $b = |A| \cdot |B| \cdot \ell$  for simplicity, where  $b$  is the batch size of word-wise FHE. Binary index vectors of  $A$  and  $B$  are packed in different sequences:

$$\begin{aligned} & \text{Enc}(\text{idx}_{a_0}, \dots, \text{idx}_{a_0}, \text{idx}_{a_1}, \dots, \text{idx}_{a_1}, \dots, \\ & \quad \text{idx}_{a_{|A|-1}}, \dots, \text{idx}_{a_{|A|-1}}), \\ & \text{Enc}(\text{idx}_{b_0}, \dots, \text{idx}_{b_{|B|-1}}, \text{idx}_{b_0}, \dots, \text{idx}_{b_{|B|-1}}, \dots, \\ & \quad \text{idx}_{b_0}, \dots, \text{idx}_{b_{|B|-1}}). \end{aligned}$$

When we partition  $b$  slots into  $|B|$  sub-blocks of  $|A| \cdot \ell$  slots, then we can easily see that the  $j$ -th block corresponds to the check process of the node  $b_j$ . To be precise, let  $j$ -th block the set of slots with indices in  $\bigsqcup_{0 \leq i < |A|} [i|B|\ell + j\ell, i|B|\ell + (j+1)\ell - 1]$ . Then, the subtraction of  $\text{ct}_B$  from  $\text{ct}_A$  results in  $(idx_{a_i,s} - idx_{b_j,s})$  for all  $0 \leq i < |A|$  and  $0 \leq s < \ell$  in the  $j$ -th block. Hence, we can compute  $(1 - (idx_{a_i,s} - idx_{b_j,s})^2)$  for all  $i$  and  $s$  in parallel, and the last product-and-sum step  $\sum_{i=0}^{|A|-1} \prod_{s=0}^{\ell-1}$  can be done through homomorphic addition, multiplication, and rotation. Full description is given as Algorithm 4. Refer to Appendix A for the note on general (non-power-of-two) cases.

---

**Algorithm 5** Homomorphic Bitonic Sort (BitonicSort)

---

**Require:**  $\text{ct}_A = \text{Enc}(a_0, a_1, \dots, a_{r-1}, \dots)$

**Ensure:**  $\text{ct}_{out} = \text{Enc}(\text{Sort}(a_0, a_1, \dots, a_{r-1}), \dots)$

- 1: **for**  $k = 2$  to  $r$  by multiplying by 2 **do**
- 2:   **for**  $j = k/2$  to 1 by dividing by 2 **do**
- 3:      $\text{ct}_{diff} \leftarrow \text{SUB}(\text{ct}_A, \text{ROTATE}(\text{ct}_A; j))$
- 4:      $\text{LWE}_{diff} \leftarrow \text{WtoBSchemeSwitch}(\text{ct}_{diff})$
- 5:     **for**  $i = 0$  to  $w - 1$  **do**
- 6:        $ij \leftarrow i \oplus j$
- 7:       **if**  $ij > i$  **then**
- 8:         **if**  $(i \& k) == 0$  **then**
- 9:          $\text{LWE}_{sign}[i] \leftarrow \text{SIGN}(\text{LWE}_{diff}[i])$
- 10:          $\text{LWE}_{sign}[ij] \leftarrow \text{NEGATE}(\text{LWE}_{sign}[i])$
- 11:         **else**
- 12:          $\text{LWE}_{sign}[ij] \leftarrow \text{SIGN}(\text{LWE}_{diff}[i])$
- 13:          $\text{LWE}_{sign}[i] \leftarrow \text{NEGATE}(\text{LWE}_{sign}[ij])$
- 14:         **end if**
- 15:       **end if**
- 16:     **end for**
- 17:      $\text{ct}_S \leftarrow \text{BtoWSchemeSwitch}(\text{LWE}_{sign})$
- 18:      $\text{ct}_A \leftarrow \text{Update } \text{ct}_A \text{ based on } \text{ct}_S$
- 19:   **end for**
- 20: **end for**
- 21: **return**  $\text{ct}_A$

---

**Complexity Analysis.** SetInt requires  $\frac{|A||B|\ell}{b} \cdot \log 2\ell$  ct-ct mults and  $\frac{|A||B|\ell}{b} \cdot \log |A|\ell$  rotations, and consumes  $\log \ell$  levels.

**3.2.4. Homomorphic Bitonic Sorting.** In some graph-based methods, we need a number of comparisons and/or sorting to select the next node for the next iteration. For example, in line 6 of Algorithm 1, we need to evaluate top- $w$  of input values. Since FHE computation must always be data-oblivious, we employ a data-oblivious algorithm for sorting called bitonic sorting network [27], [39]. Bitonic sorting repeatedly generates bitonic sequences and merges them to form a sorted array. Each step involves comparison of multiple input pairs and conditionally swaps them based on the comparison results.

Comparison operations in word-wise FHE schemes should be either approximated or represented by a high-degree polynomial over the real number field (CKKS) or finite field (BGV, B/FV), consuming significant multiplicative depth (See Section 2.2.1). On the other hand, bit-wise FHE allows efficient evaluation of comparison operations but suffers from high latency when handling large number of operations in parallel. To enjoy the best of both worlds, we leverage both word-wise and bit-wise FHE with scheme-switching, utilizing bit-wise FHE for efficient sign evaluations (equivalent to comparison) and word-wise FHE for batched value swaps.

For the sake of simplicity, we assume that the number of inputs  $r$  is a power of two. In each stage, we begin by evaluating the difference between pairs of values, which can be done in parallel with word-wise FHE. These differences are then converted into bit-wise FHE to compute the sign function yielding binary values. The binary results are converted back into word-wise FHE, where they

serve as masks for value-swapping (See Algorithm 5). To obtain top- $w$  instead of full sorting, we can eliminate several stages of the network that do not influence the top- $w$  results, which reduces the number of homomorphic operations including scheme switching.

We similarly utilize scheme-switching to evaluate Min/ArgMin through a tournament-based comparison method within  $\log r$  iterations. In the  $i$ -th iteration, the differences between  $2^{\log r - i - 1}$  pairs are computed using word-wise FHE. Scheme-switching from word-wise FHE to bit-wise FHE (WtoBSchemeSwitch) is then performed to evaluate the sign function in bit-wise FHE. Afterwards, we switch back from bit-wise FHE to word-wise FHE (BtoWSchemeSwitch) to reorder the values, continuing this process until the minimum or maximum result is obtained.

## 4. Instantiation to Similarity Search

In this section, we describe GraSS: An end-to-end FHE solution for secure graph-based similarity search, exploiting the core building blocks introduced in Section 3. Before querying, a client generates its own secret key  $sk$  and the set of public keys  $pk$  (including key-switching, bootstrapping and scheme-switching keys). Note that  $pk$  is broadcasted to the server and can be re-used for different queries from the same client.

The baseline protocol between client and server is as follow:

- 1) Client $\rightarrow$ Server: Client encrypts a query as  $ct_{query} := \text{Enc}(query)$  and sends it to the server.
- 2) Server $\rightarrow$ Client: Server performs similarity search with  $ct_{query}$  and send back the result  $ct_{topk}$  to the client. Client decrypts  $ct_{topk}$  and obtain the cleartext result.

### 4.1. Client/Server Input Privacy

Our protocol ensures the client that their query information is not revealed to the server, which comes from the IND-CPA security of FHE. Note that all the state-of-the-art FHE schemes [9]–[11], [13]–[15] are IND-CPA secure. On the other hand, the server-input (i.e., graph data) privacy against the client is not guaranteed by IND-CPA since the client could potentially exploit partial information from the server input during the decryption process beyond the similarity search result. Thankfully, we can hide the circuit information from the ciphertexts by statistical noise flooding [17], [18] or ciphertext sanitization through bootstrapping [19]–[22] (i.e., circuit privacy of FHE). Therefore, if the server sends back the result  $ct_{topk}$  after applying such circuit-privacy techniques, then the client cannot learn any partial information on the server’s input (graph data) except the decryption result.

### 4.2. Setup Phase

In the setup phase, a client encrypts the query data into ciphertexts and a server encodes the graph data (Table 1)

into plaintexts. These encryption and encoding are done with respect to word-wise FHE, since the similarity search algorithm starts with the distance computation which consists of highly-parallelizable polynomial arithmetic.

**Query Data Encryption (Client):** A client first generates its own secret key  $sk \in R$  and its corresponding set of public key-switching keys  $k_{sk}$ . Note that  $k_{sk}$  is broadcasted to the server before querying and is reusable for multiple queries.

For a query  $\mathbf{v} = (v_0, v_1, \dots, v_{d-1}) \in \mathbb{R}^d$ , a client encrypts each entry  $v_i$  separately as  $ct_{query,u} := \text{Enc}_{sk}(v_u, v_u, \dots, v_u)$  for  $0 \leq u < d$  and sends them to the server. To reduce the communication cost, the client can alternatively send  $ct_{query} := \text{Enc}_{sk}(v_0, v_1, \dots, v_{d-1}, 0, 0, \dots, 0)$ , and then let the server unpack  $ct_{query}$  into  $ct_{query,u}$ ’s.

**Graph Data Encoding (Server):** A server *column-wise* encodes the graph data (Table 1) into plaintexts. To be precise, denoting by  $\mathbf{v}_{s_{i,j}} = (v_{s_{i,j},0}, v_{s_{i,j},1}, \dots, v_{s_{i,j},d-1})$  and  $\mathbf{id}x_{s_{i,j}} := (idx_{s_{i,j},0}, idx_{s_{i,j},1}, \dots, idx_{s_{i,j},\lceil \log n \rceil - 1})$ , each column of Table 1 is either  $(v_{s_{i,j}})_{0 \leq i < n}$  for some  $0 \leq j < d$  (i.e., the collection of  $j$ -th vector entry of  $i$ -th neighbor) or  $(idx_{s_{i,j}})_{0 \leq i < n}$  for some  $0 \leq j < \lceil \log n \rceil$  (i.e., the collection of  $j$ -th binary index entry of  $i$ -th neighbor). In general, the number of nodes  $n$  is substantially larger than the SIMD batch size  $b$  of word-size FHE. Hence, each column is parsed into  $\lceil n/b \rceil$  vectors of the length  $b$ , and then each length- $b$  vector is encoded into a plaintext.

**Data Structure of  $Q$  and  $E$ :** For the set of candidate nodes  $Q := \{q_0, q_1, \dots, q_{w-1}\}$ , we update two ciphertexts separately:

$$\begin{aligned} ct_{Q,idx} &= \text{Enc}(\mathbf{id}x_{q_0}, \dots, \mathbf{id}x_{q_{w-1}}, \dots), \\ ct_{Q,dist} &= \text{Enc}(\text{dist}^2(\mathbf{v}, \mathbf{v}_{q_0}), \dots, \text{dist}^2(\mathbf{v}, \mathbf{v}_{q_{w-1}}), \dots), \end{aligned}$$

one as a ciphertext of binary indices and the other one as a ciphertext of distances from the query.<sup>3</sup> Both binary index and distance information of the nodes in  $Q$  are required for argmin and top- $w$  computation in Algorithm 1, while binary index is more widely used in processes including set intersection and binary-to-one-hot conversion. In the first (0-th) iteration,  $ct_{Q,idx}$  is prepared (from the server side) as a trivial ciphertext (i.e., plaintext) for pre-selected  $Q$ .

For the set of explored nodes  $E$ , we only keep a ciphertext of binary indices of the nodes in  $E$ , since we only need the information of  $E$  to check if each node  $q_i$  in  $Q$  has already been explored or not (i.e.,  $q_i \in E?$ ). Note that the binary index vectors of the input sets for SetInt need to be encrypted in a different sequence (See Algorithm 4). Hence, we update the ciphertext  $ct_E$  as

$$ct_E = \text{Enc}(\mathbf{id}x_{x_0}, \dots, \mathbf{id}x_{x_0}, \mathbf{id}x_{x_1}, \dots, \mathbf{id}x_{x_1}, \dots, \mathbf{id}x_{x_i}, \dots, \mathbf{id}x_{x_i}, \dots)$$

at  $i$ -th iteration of the graph search, where each binary index  $\mathbf{id}x_{x_i}$  is repeated for  $w$  times.

3. We assume that the SIMD batch size  $b$  is larger than  $w \cdot \lceil \log n \rceil$ . If  $b < w \cdot \lceil \log n \rceil$ , then  $\mathbf{id}x_{q_0}, \mathbf{id}x_{q_1}, \dots, \mathbf{id}x_{q_{w-1}}$  are encrypted in multiple ciphertexts.



TABLE 2: Distance and Binary Index Table of Neighborhoods

Index	Neighborhood Information			
0	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{0,0}}), \mathbf{idx}_{s_{0,0}})$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{0,1}}), \mathbf{idx}_{s_{0,1}})$	$\dots$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{0,m-1}}), \mathbf{idx}_{s_{0,m-1}})$
1	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{1,0}}), \mathbf{idx}_{s_{1,0}})$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{1,1}}), \mathbf{idx}_{s_{1,1}})$	$\dots$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{1,m-1}}), \mathbf{idx}_{s_{1,m-1}})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{n-1,0}}), \mathbf{idx}_{s_{n-1,0}})$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{n-1,1}}), \mathbf{idx}_{s_{n-1,1}})$	$\dots$	$(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{n-1,m-1}}), \mathbf{idx}_{s_{n-1,m-1}})$

### 4.3. Distance Table Computation

Before entering the iterative graph search, we pre-compute the squared distance between the (encrypted) query  $\mathbf{v} := (v_0, v_1, \dots, v_{d-1}) \in \mathbb{R}^d$  and the feature vectors  $\mathbf{v}_{s_{i,j}}$  of each (unencrypted) neighbor node  $s_{i,j}$  for  $0 \leq i < n$  and  $0 \leq j < m$ :

$$\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{i,j}}) = \sum_{u=0}^{d-1} (v_u - v_{s_{i,j},u})^2. \quad (1)$$

This naive distance computation requires  $\lceil \frac{n}{b} \rceil \cdot m \cdot d$  ct-ct mults to square  $(v_u - v_{s_{i,j},u})$  for all  $i, j$  and  $u$ . Note that the denominator  $b$  comes from the SIMD packing property of word-wise FHE that allows parallel distance computation for  $b$  nodes.

In this work, instead of Equation 1, we utilize another way to compute the distance as below:

$$\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{i,j}}) = \sum_{u=0}^{d-1} v_u^2 - \sum_{u=0}^{d-1} 2v_{s_{i,j},u} \cdot v_u + \sum_{u=0}^{d-1} v_{s_{i,j},u}^2, \quad (2)$$

In fact, Equation 2 requires more scalar multiplications and hence is less efficient than Equation 1 in cleartext computation. However, in terms of FHE computation, ct-ct mult is only required for  $\sum_{u=0}^{d-1} v_u^2$  part since the graph data  $(v_{s_{i,j},u})$  is not encrypted. The computation of  $\sum_{u=0}^{d-1} 2v_{s_{i,j},u} \cdot v_u$  is done by ct-pt mult, which is much cheaper than ct-ct mult. As a result, Equation 2 requires only  $d$  ct-ct mults (to square  $v_u$ ) and  $\lceil \frac{n}{b} \rceil \cdot m \cdot d$  ct-pt mults (to multiply  $v_u$  and  $2v_{s_{i,j},u}$ ).

Compared to Equation 1, the number of ct-ct mults to compute Equation 2 is significantly reduced by the factor  $\lceil \frac{n}{b} \rceil \cdot m$ . There exists an overhead as  $\lceil \frac{n}{b} \rceil \cdot m \cdot d$  ct-pt mults, but this additional cost is much cheaper than  $\lceil \frac{n}{b} \rceil \cdot m \cdot d$  ct-ct mults in the computation of Equation 1. Hence, utilizing Equation 2 is much more efficient to compute the distance between encrypted query and unencrypted nodes in the graph.

After applying Algorithm 6 (in Appendix 6), we obtain an encrypted distance table that stores the squared distance between the query  $\mathbf{v}$  and all the neighbor nodes  $\mathbf{v}_{s_{i,j}}$  for  $0 \leq i < n$  and  $0 \leq j < m$  in the graph. Table 2 shows the concatenation of the (encrypted) distance table and the (cleartext) binary index table.

Refer to Appendix A for the note on distance pre-computation v.s. on-the-fly computation in terms of computational cost.

### 4.4. Detailed Steps Per Iteration

As demonstrated in Algorithm 1 and Figure 1, an iteration of GraSS consists of 5 steps. First, given a set of candidate nodes  $Q$ , we find the node  $x$  that is unexplored and closest to the query. Next, we update the explored nodes set  $E$  to include  $x$ . In step 3, we retrieve the neighbor nodes  $N(x)$ . After that, we evaluate  $Q \cup N(x)$  and remove the possible duplicates. Finally, we evaluate Top- $w$  among the nodes in  $Q$ . Refer to Algorithm 6-10 in Appendix E for more details of each step.

#### 4.4.1. Step 1: Find Closest Unexplored Node in $Q$ .

Given a set of candidate nodes  $Q = \{q_0, q_1, \dots, q_{w-1}\} \subset [0, n)$ , we need to find the closest node  $x \in Q$  which has the smallest distance among all the unexplored nodes in  $Q$ .

We first need to check the explored status of each node in  $Q$  by applying the homomorphic set intersection algorithm `SetInt` described in Section 3.2.3. Following Algorithm 4, we set  $A$  as  $E$ , the set of binary index vectors of the explored nodes, and  $B$  as  $Q$ , the set of binary index vectors of the candidate nodes. The algorithm with input  $\text{ct}_E$  and  $\text{ct}_{Q,idx}$  returns a ciphertext of the explored status of each candidate node in  $Q$ :

$$\begin{aligned} \text{ct}_{Q-E} := \\ \text{Enc}((q_0 \in E?), (q_1 \in E?), \dots, (q_{w-1} \in E?), \dots). \end{aligned}$$

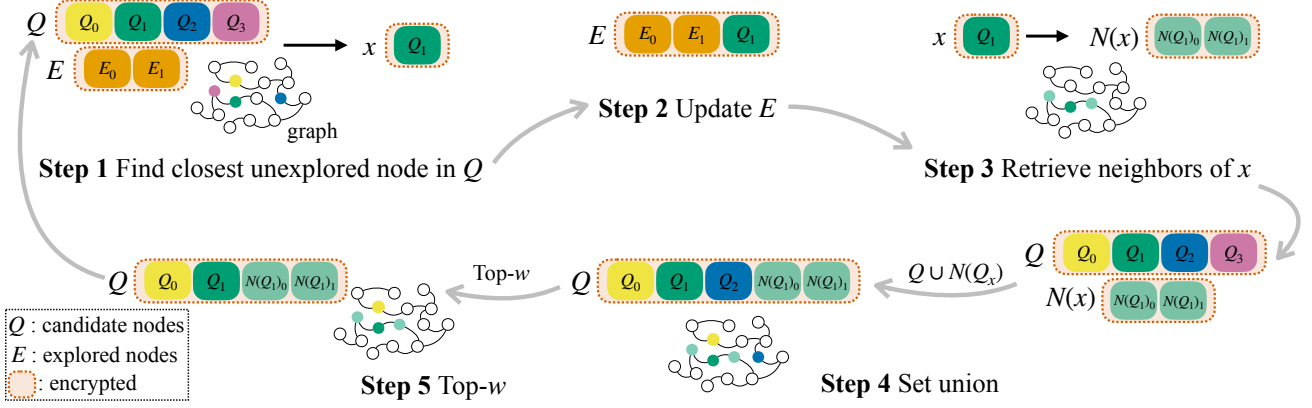
Next we use  $\text{ct}_{Q-E}$  to mask<sup>4</sup> the distance of the candidate nodes in  $Q$  (given as  $\text{ct}_{Q,dist}$ ) to make sure that explored nodes are automatically ignored during the following argmin computation. Then, we can run the argmin algorithm on the masked candidates distance and combine the argmin result with  $\text{ct}_{Q,idx}$  to obtain the binary index of the closest node  $x$  of the form  $\text{Enc}(\mathbf{0}, \dots, \mathbf{0}, \mathbf{idx}_x, \mathbf{0}, \dots, \mathbf{0})$ . Since the position of  $\mathbf{idx}_x$  is unknown, we broadcast  $\mathbf{idx}_x$  to entire slots as  $\text{Enc}(\mathbf{idx}_x, \mathbf{idx}_x, \dots, \mathbf{idx}_x)$  so that  $\mathbf{idx}_x$  is finally positioned at the known slots. For argmin computation, we utilize scheme-switching to enjoy the low latency of bit-wise FHE for comparison operations (See Section 3.2.4).

#### 4.4.2. Step 2: Update Explored Nodes $E$ .

After Step 1, we append  $x$  to the set of explored nodes  $E$ . Let  $x_i$  be the closest node  $x$  in  $i$ -th iteration. Then, we can update  $\text{ct}_E$  from the previous iteration by simply mask-and-adding the result of

4. Here, the masking procedure is not to zero-ize but to max-ize with some value  $\text{MAX} > 0$ , since we need to compute arg“min” result. In the implementation, we set sufficiently large  $\text{MAX}$  value for the max-ize process.

Figure 1: Flow of GraSS per Iteration



Step 1  $\text{Enc}(\text{idx}_{x_i}, \text{idx}_{x_i}, \dots, \text{idx}_{x_i})$ , which results in  $\text{Enc}(\text{idx}_{x_0}, \dots, \text{idx}_{x_0}, \dots, \text{idx}_{x_i}, \dots, \text{idx}_{x_i}, 0, 0, \dots)$ .

#### 4.4.3. Step 3: Neighborhood Information Retrieval.

This step intends to retrieve the neighbor nodes' information (for both nodes distance and nodes indices) of the located closest node  $x$ . Step 1 in Section 4.4.1 gives encryption of the broadcasted  $\text{idx}_x$  as  $\text{Enc}(\text{idx}_x, \text{idx}_x, \dots, \text{idx}_x)$ .

In order to apply the node index as a mask, we first convert the binary index  $\text{idx}_x$  into its one-hot encoding vector  $\mathbf{e}_x = (0, \dots, 0, 1, 0, \dots, 0) \in \{0, 1\}^n$  (See Section 3.1). This conversion can be achieved through two sub-steps: 1. Unpack  $\text{Enc}(\text{idx}_x, \text{idx}_x, \dots, \text{idx}_x)$  into  $\text{Enc}(\text{idx}_{x,u}, \text{idx}_{x,u}, \dots, \text{idx}_{x,u})$  for  $0 \leq u < \lceil \log n \rceil$  where  $\text{idx}_x := (\text{idx}_{x,0}, \dots, \text{idx}_{x, \lceil \log n \rceil - 1})$ , and then 2. apply  $\text{BintoOneHot}$  (Algorithm 2) to obtain  $\text{Enc}(\mathbf{e}_x)$ .

Next, we can use  $\text{Enc}(\mathbf{e}_x)$  to mask each column of Table 2 and extract the neighbor nodes' information of  $x$ . The masking process with  $\text{Enc}(\mathbf{e}_x)$  outputs  $\text{Enc}(0, \dots, 0, \text{idx}_{s_{x,j},i}, 0, \dots, 0)$  and  $\text{Enc}(0, \dots, 0, \text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{x,j}}), 0, \dots, 0)$  for  $0 \leq i < \lceil \log n \rceil$  and  $0 \leq j < m$ , where the slot position  $y := [x]_b$  of each  $\text{idx}_{s_{x,j},i}$  and  $\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{x,j}})$  is private. Fortunately we have already obtained the encryption of  $y$  as  $\text{Enc}(\text{idx}_{x,u}, \text{idx}_{x,u}, \dots, \text{idx}_{x,u})$  for  $0 \leq u < \lceil \log b \rceil$  as input of  $\text{BintoOneHot}$  to obtain  $\text{Enc}(\mathbf{e}_x)$ . Hence, by applying  $\text{BlindRotate}$  with input  $\text{ct}_i := \text{Enc}(0, \dots, 0, \text{idx}_{s_{x,j},i}, 0, \dots, 0)$  and  $\text{ct}_{\text{idx}_{x,u}} := \text{Enc}(\text{idx}_{x,u}, \text{idx}_{x,u}, \dots, \text{idx}_{x,u})$ , we get  $\text{Enc}(\text{idx}_{s_{x,j}}, \text{idx}_{s_{x,j}}, \dots, \text{idx}_{s_{x,j}})$  for each  $j$ . Finally, we can pack the those ciphertexts into

$$\text{ct}_{N, \text{idx}} = \text{Enc}(\text{idx}_{s_{x,0}}, \text{idx}_{s_{x,1}}, \dots, \text{idx}_{s_{x,m-1}}, \mathbf{0}, \dots, \mathbf{0}).$$

Note that the retrieval of neighborhoods' distance information can be implemented in almost the same manner. The distance information of neighborhoods are obtained as a ciphertext

$$\text{ct}_{N, \text{dist}} = \text{Enc}(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{x,0}}), \dots, \text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{x,m-1}}), 0, \dots, 0)$$

**4.4.4. Step 4: Set Union of  $Q$  and  $N(x)$ .** In this step, we want to compute the set union of the candidates nodes  $Q$  and the neighbor nodes  $N(x) = \{s_{x,0}, s_{x,1}, \dots, s_{x,m-1}\}$ . Note that the direct concatenation of the two sets of nodes can lead to duplicate nodes in the union set. Therefore we need a mechanism to eliminate the impact of potential duplicate nodes, i.e., when  $N(x)$  contains one or more nodes that already exist in  $Q$ . In our protocol, we leverage the homomorphic set intersection algorithm  $\text{SetInt}$  (see Algorithm 4) by setting  $A$  as  $Q^5$  and  $B$  as  $N(x)$ . Similar to Step 1,  $\text{SetInt}$  with input  $\text{ct}_{Q, \text{idx}}$  and  $\text{ct}_{N, \text{idx}}$  returns a ciphertext of the duplication status of each node in  $N(x)$ :

$$\begin{aligned} \text{ct}_{N-Q} &:= \\ \text{Enc}((s_{x,0} \in Q?), (s_{x,1} \in Q?), \dots, (s_{x,m-1} \in Q?), \dots) \end{aligned}$$

Next we use  $\text{ct}_{N-Q}$  to mask  $\text{ct}_{N, \text{dist}}$  to make sure that duplicated nodes are automatically ignored during the following top- $w$  computation. Finally, we concatenate  $\text{ct}_{Q, \text{dist}}$  and the masked  $\text{ct}_{N, \text{dist}}$  (resp.  $\text{ct}_{Q, \text{idx}}$  and  $\text{ct}_{N, \text{idx}}$ ) through simple homomorphic rotation and addition, which will be used as input of the last step, homomorphic top- $w$  operation.

**4.4.5. Step 5: Local Top- $w$ .** We need to extract  $w$  nodes in  $Q \cup N(x)$  based on the ascending order of the nodes' distance. The top- $w$  function takes  $\text{ct}_{Q \cup N(x), \text{dist}}$  and  $\text{ct}_{Q \cup N(x), \text{idx}}$  as inputs, which contain the distance and binary indices of the nodes in  $Q \cup N(x)$ . In the end, the function outputs the distance and binary indices of  $w$  nodes from  $Q \cup N(x)$  whose distance are the smallest. We achieve this using a top- $w$ -truncated version of homomorphic bitonic sorting  $\text{BitonicSort}$  (Algorithm 5) with  $\text{ct}_{Q \cup N(x), \text{dist}}$  as an input, which outputs the ciphertext of top- $k$  distance values and their corresponding binary indices of  $Q \cup N(x)$ . Top- $w$ -truncated bitonic sorting skips loops in line 2 of Algorithm 5 that do not influence the top- $w$  results. We also removes  $\text{SIGN}$  operations in line 9 and 12 that are unnecessary for top- $w$ . Note that  $\text{ct}_{Q \cup N(x), \text{idx}}$  can be homomorphically sorted in exactly the same manner by re-using  $\text{ct}_S$  in line 5 of Algorithm 5 that comes from  $\text{BitonicSort}(\text{ct}_{Q \cup N(x), \text{dist}})$ .

5.  $\text{ct}_{Q, \text{idx}}$  is repacked to follow the format of  $\text{ct}_A$  in Algorithm 4.

TABLE 3: Parameters and Latency for End-to-end Graph-Based Similarity Search

Operation	Deep10K	Deep100K	Deep1M	
Graph Properties				
Max Number of Iterations	24	31	34	
Search Accuracy (Recall)	0.986	0.962	0.918	
Pre-Computation Phase Latency (s)				
Step 0: Distance Table Computation	132.221	512.104	3798.05	
<i>i</i> -th Iteration Latency (s)				
Step 1: Find Closest Unexplored Node in $Q$	146.407			
Step 2: Update Explored Nodes $E$	0.884			
Step 3: Neighbor Nodes Retrieval	BintoOneHot + Masking	30.757	152.907	1759.202
	BlindRotate + Bootstrap <sup>6</sup>	542.231	557.263	611.615
Step 4: Set Union of $Q$ and $N(x)$	18.360			
Step 5: Local Top- $w$	5780			
Total	6518.639	6655.821	8316.468	
Post-Iteration Phase Latency (s)				
Compute Feature Vectors of Top- $k$	322.245	785.22	12040.895	
End-to-end Protocol Runtime (hrs)				
Total	43.584	57.674	82.944	

## 4.5. Post-Iteration Step

Let  $\text{ct}_{\text{top}k, \text{idx}} = \text{Enc}(\text{idx}_{\text{top}0}, \text{idx}_{\text{top}1}, \dots, \text{idx}_{\text{top}w-1})$  be the encrypted binary index output of the final iteration in Algorithm 1. To extract  $\mathbf{v}_{\text{top}0}, \mathbf{v}_{\text{top}1}, \dots, \mathbf{v}_{\text{top}k-1}$  homomorphically for  $k \leq w$ , we first extract the indices of  $k$  nodes among the  $w$  candidate nodes based on the ascending order of the nodes' distance. Next, we unpack  $\text{ct}_{\text{top}k, \text{idx}}$  into  $\text{ct}_{i,j} := \text{Enc}(\text{idx}_{\text{top}i,j}, \dots, \text{idx}_{\text{top}i,j})$  for  $0 \leq i < k$  and  $0 \leq j < \lceil \log n \rceil$ . Then, we perform BintoOneHot with input  $\{\text{ct}_{i,j}\}_{0 \leq j < \lceil \log n \rceil}$  to obtain  $\text{Enc}(\mathbf{e}_{\text{top}i})$  for  $0 \leq i < k$ . Finally, we use  $\text{Enc}(\mathbf{e}_{\text{top}i})$  to mask the first  $d$  columns of Table 1, which outputs the ciphertexts corresponding to  $\mathbf{v}_{s_{\text{top}i},0} = \mathbf{v}_{\text{top}i}$ . After packing the ciphertexts into  $\text{Enc}(\mathbf{v}_{\text{top}i})$  for each  $i$ , the server sends them and  $\text{ct}_{\text{top}k, \text{idx}}$  back to the client. To ensure the server-input privacy against the client, the server scheme-switches each  $\text{Enc}(\mathbf{v}_{\text{top}i})$  into bit-wise FHE ciphertexts and sends them back to the client after the ciphertext sanitization process.

## 5. Evaluation and Performance

### 5.1. Implementation and Experimental Setup

**Environment** Our implementation is written in C++ using the OpenFHE library [40] and is evaluated on machines with Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz using the Ubuntu 22.04.4 operating system. We benchmark our implementation using 16 threads.<sup>7</sup>

<sup>7</sup> Our implementation has not been optimized to support multi-threading yet; we did not put any multi-thread options outside of the OpenFHE library. An optimization to fully utilize the given threads remains as a future work.

**Datasets** We evaluated our method for various size of subsets of the widely-used Deep dataset, which contains 1 billion vectors in total derived from the final layer of convolutional neural networks [41]. Specifically, we use the first 10k, 100k, and 1 million vectors to form Deep10K, Deep100K, and Deep1M datasets, respectively.

All datasets consist of 96-dimensional vectors ( $d = 96$ ) and were pre-trained as a regular graph with degree  $m = 64$ . The graph properties in Table 3 are measured by testing 10k queries in cleartext on each dataset. We set the search window size  $w = 16$  and report the accuracy, bandwidth, and latency results for approximate top-16 search.

**FHE Parameters.** We select CKKS and FHEW as bit/word-wise FHE schemes to utilize the scheme-switching technique implemented in OpenFHE [31], [40]. For CKKS, we set the ring dimension  $N = 2^{16}$ , the scaling factor  $2^{52} \leq \Delta \leq 2^{59}$ , and the multiplicative depth 32 ( $\log Q \leq 1948$ ), which results in  $\geq 80$ -bit security. For FHEW, we used the STD128 parameter set of OpenFHE which satisfies 128-bit security and support high-precision (up to 21-bit) sign evaluation [33]:  $N_{LWE} = 1305$ ,  $t_{LWE} = 4$ ,  $q = 2^{12}$ ,  $q_{\text{large}} = 2^{22}$  and  $t_{\text{large}} = 2^{14}$  (See Section 2.2.2). Refer to Appendix D for the discussion on the concrete security of CKKS parameters.

### 5.2. Bandwidth

Once a client and a server set up a communication channel, the client first needs to broadcast a set of public keys (key-switching, bootstrapping and scheme-switching keys) to the server. These keys are sent only once at the

pre-query phase, and can later be reused across multiple queries.

During the protocol, both client and server send ciphertexts to each other just once since the protocol is non-interactive. Hence, the total communication cost per query is the size of  $c\tau_{query}$  and  $c\tau_{topk}$ , both of which are less than 1GB and even compressible to  $\sim 10$ MB (See Section 4.2).

### 5.3. Accuracy and Latency

In Table 3, we first benchmark the cleartext graph-based similarity search algorithm to get the average number of iteration and search accuracy for each dataset. We observed the perfect match between the search results from the cleartext algorithm and our FHE-based protocol. Therefore, our implementation achieves the same accuracy with the cleartext version.

Further, we benchmark the latency of each step of GraSS described in Section 4. In the end, we show the total runtime for the end-to-end graph-based similarity search. We can see from Table 3 that per iteration, the latency for most steps is *independent to  $n$*  except Step 2. Although the latency of pre-computation and post-iteration phases is  $\tilde{O}(n)$ , their one-time latency are amortized among the whole protocol run. For example, Step 5 independent to  $n$  takes the majority of whole process even for 1M dataset ( $> 60\%$ ), so total latency does not increase dramatically from 10K to 1M. Hence, our graph-based similarity search protocol is highly scalable and supports datasets of million scale while maintaining a good performance.

The sanitization process for server-input privacy is not included as a part of our implementation; however, it can be applied as a black box to the result ciphertext  $c\tau_{topk}$  only with a small computational overhead compared to the whole procedure of graph-based similarity search. For example, bit-wise FHE can achieves the circuit privacy within a second [21]. Hence, the server can scheme-switch the result ciphertexts with CKKS form into FHEW ciphertexts and send them back to the client after sanitization, which guarantees the server-input privacy against the client as well.

### 5.4. Comparison with Prior Art

Table 4 includes asymptotically estimated numbers based on the result in [27] and speedups of our work. The previous work [27] presents results for performing FHE-based kNN on  $\leq 1000$  inputs with 6-bit precision for the comparison operation in sorting network. In contrast, our work targets searches on million-scale database, which requires much higher precision (14-bit) than the previous work. To demonstrate the scalability of our approach, we projected the results from [27] to match our data size. For a fair comparison, we assumed that their implementation also uses 14-bit precision. Such estimation is evaluated following the methodology of [33]:  $256\times$  overhead to increase bit precision of comparison from 6-bit to 14-bit.

The results show that GraSS outperforms [33] for large-scale datasets: We achieve  $4.13\times$  and  $28.72\times$  speedups

TABLE 4: Comparison of End-to-end Latency Results (hrs) for Top-16 Similarity Search with Prior Art Based on FHE

Dataset Size	Latency (hrs)		Speedup
	[27] <sup>8</sup>	GraSS	
10K	23.948	43.584	-
100K	238.35	57.674	<b>4.13</b> $\times$
1M	2382.3	82.944	<b>28.72</b> $\times$

for Deep100K and Deep1M datasets, respectively. Compared to [27], our protocol demonstrates great scalability, since the kNN algorithm [27] evaluates global top- $w$  using a bitonic sorting network, which scales linearly with the dataset size  $n$ . In contrast, our graph-based algorithm utilizes an iterative method that restricts the number of inputs to the sorting network to a maximum of  $m + w$ . Our algorithm also includes some  $\tilde{O}(n)$  operations such as distance table and part of neighborhood retrieval, but those are highly parallelizable with SIMD property of CKKS, and hence  $\tilde{O}(n)$  operations are not dominant in the end-to-end runtime.

On the other hand, we observe that GraSS performs slower on the 10K dataset. This is due to the expensive scheme switching in our top- $k$  process, which is not included in the previous work. However, such approach takes advantage of both bit/word-wise FHEs, supporting high-precision and making our protocol scalable, as seen in the result for 1M dataset.

GraSS, which relies solely on FHE, exhibits relatively higher CPU computation latency compared to MPC-based solutions. However, a key advantage of our FHE-only solution is that it allows the client to transfer the query and completely disconnect until receiving the result. In contrast, MPC-based solutions require continuous data exchange throughout the process, necessitating that the client remains online. This ongoing communication demand in MPC often includes larger communication overhead between the client and server compared to FHE-based methods. For instance, our solution demonstrates about a  $155\times$  reduction in communication requirements compared to clustering of [18] for 1M dataset (12.81 MB vs. 1.99 GB). This characteristic makes our approach particularly suitable for scenarios with limited communication bandwidth or when clients cannot maintain an active online connection. Furthermore, with ongoing advancements in FHE acceleration using GPU and hardware, our framework’s computation latency has the potential to be significantly reduced while maintaining minimal communication overhead.

## 6. Conclusion

In this work, we introduced the first FHE-based solution for secure graph-based database search, GraSS. We

7. It includes both CKKS bootstrapping (for distance values) and scheme-switching-based bootstrapping (via EvalCompareSchemeSwitch with 1/2 for binary values).

8. Note that this column shows asymptotically estimated numbers, not exact numbers reported in the paper.

proposed an FHE-friendly graph structure with a novel index encoding which makes our protocol highly scalable in terms of  $n$ , by reducing the computational complexity of neighborhood retrieval process from  $O(n^2)$  to  $\tilde{O}(n)$ , for a regular graph with small degree  $m = o(n)$ . We also proposed core FHE algorithms to perform graph operations under the new graph structure and introduce the end-to-end secure protocol of graph-based similarity search.

We showed that FHE-based similarity search is even feasible for million-scale database through the proof-of-concept implementation, demonstrating  $28\times$  speedup compared to a state-of-the-art [27]. With the active research on FHE acceleration through GPU and HW (e.g.,  $1,000\times$  v.s. CPU for CKKS bootstrapping [42]), we leave the implementation of the proposed algorithms on accelerators to enable the near real-time latency as a future work.

## References

- [1] A. Blattmann, R. Rombach, K. Oktay, J. Müller, and B. Ommer, “Retrieval-augmented diffusion models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 15 309–15 324, 2022.
- [2] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. B. Van Den Driessche, J.-B. Lespiau, B. Damoc, A. Clark *et al.*, “Improving language models by retrieving from trillions of tokens,” in *International conference on machine learning*. PMLR, 2022, pp. 2206–2240.
- [3] D. Lian, H. Wang, Z. Liu, J. Lian, E. Chen, and X. Xie, “Lightrec: A memory and search-efficient recommender system,” in *Proceedings of The Web Conference 2020*, 2020, pp. 695–705.
- [4] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.
- [5] Z. Jiang, F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig, “Active Retrieval Augmented Generation,” 2023, arXiv:2305.06983 [cs].
- [6] D. Cai, Y. Wang, L. Liu, and S. Shi, “Recent Advances in Retrieval-Augmented Text Generation,” in *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2022, pp. 3417–3419.
- [7] C. Aguerrebere, I. Bhati, M. Hildebrand, M. Tepper, and T. Willke, “Similarity search in the blink of an eye with compressed indices,” *arXiv preprint arXiv:2304.04759*, 2023.
- [8] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.
- [9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” in *Proc. of ITCS*. ACM, 2012, pp. 309–325.
- [10] Z. Brakerski, “Fully homomorphic encryption without modulus switching from classical GapSVP,” in *Advances in Cryptology—CRYPTO 2012*. Springer, 2012, pp. 868–886.
- [11] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption.” *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.
- [12] C. Gentry, A. Sahai, and B. Waters, “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based,” in *Advances in Cryptology—CRYPTO 2013*. Springer, 2013, pp. 75–92.
- [13] 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.
- [14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,” in *Advances in Cryptology—ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 3–33.
- [15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology—ASIACRYPT 2017: 23rd International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.
- [16] S. Goldwasser and S. Micali, “Probabilistic encryption & how to play mental poker keeping secret all partial information,” in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 173–201.
- [17] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, “{GAZELLE}: A low latency framework for secure neural network inference,” in *27th USENIX security symposium (USENIX security 18)*, 2018, pp. 1651–1669.
- [18] H. Chen, I. Chillotti, Y. Dong, O. Poburinnaya, I. P. Razenshteyn, and M. S. Riazzi, “SANNS: Scaling up secure approximate k-nearest neighbors search,” in *USENIX Security 2020: 29th USENIX Security Symposium*, S. Capkun and F. Roesner, Eds. USENIX Association, Aug. 12–14, 2020, pp. 2111–2128.
- [19] L. Ducas and D. Stehlé, “Sanitization of the ciphertexts,” in *Advances in Cryptology—EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*. Springer, 2016, pp. 294–310.
- [20] F. Bourse and M. Izabachène, “Plug-and-play sanitization for tfhe,” *Cryptology ePrint Archive*, 2022.
- [21] K. Kluczniak, “Circuit privacy for fhew/tfhe-style fully homomorphic encryption in practice,” *Cryptology ePrint Archive*, 2022.
- [22] F. Bourse, R. Del Pino, M. Minelli, and H. Wee, “Fhe circuit privacy almost for free,” in *Annual International Cryptology Conference*. Springer, 2016, pp. 62–89.
- [23] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, “Xpir: Private information retrieval for everyone,” *Proceedings on Privacy Enhancing Technologies*, pp. 155–174, 2016.
- [24] J. P. Stern, “A new and efficient all-or-nothing disclosure of secrets protocol,” in *International conference on the theory and application of cryptology and information security*. Springer, 1998, pp. 357–371.
- [25] M. Zuber and R. Sirdey, “Efficient homomorphic evaluation of k-NN classifiers,” *Proceedings on Privacy Enhancing Technologies*, vol. 2021, no. 2, pp. 111–129, Apr. 2021.
- [26] Y. Ameur, R. Aziz, V. Audigier, and S. Bouzeffrane, “Secure and non-interactive k-NN classifier using symmetric fully homomorphic encryption,” in *Privacy in Statistical Databases*, J. Domingo-Ferrer and M. Laurent, Eds. Cham: Springer International Publishing, 2022, pp. 142–154.
- [27] K. Cong, R. Geelen, J. Kang, and J. Park, “Revisiting oblivious top- $k$  selection with applications to secure  $k$ -nn classification,” *Cryptology ePrint Archive*, Paper 2023/852, 2023, to appear at SAC 2024.

- [28] 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.
- [29] 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.
- [30] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo, "Efficient fhe bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 227–256.
- [31] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "OpenFHE: Open-source fully homomorphic encryption library," *Cryptology ePrint Archive*, Paper 2022/915, 2022, <https://eprint.iacr.org/2022/915>. [Online]. Available: <https://eprint.iacr.org/2022/915>
- [32] H. Shaul, D. Feldman, and D. Rus, "Secure k-ish nearest neighbors classifier," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 3, pp. 42–61, Jul. 2020.
- [33] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2022, pp. 130–160.
- [34] M. Dockendorf, R. Dantu, and J. Long, "Graph algorithms over homomorphic encryption for data cooperatives," in *SECRYPT*, 2022, pp. 205–214.
- [35] R. Ran, N. Xu, T. Liu, W. Wang, G. Quan, and W. Wen, "Penguin: parallel-packed homomorphic encryption for fast graph convolutional network inference," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [36] M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT 2010*. Springer, 2010, pp. 24–43.
- [37] 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.
- [38] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2021, pp. 17–28.
- [39] 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.
- [40] "OpenFHE Open-Source FHE Library," <https://github.com/openfheorg/openfhe-development>.
- [41] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.
- [42] J. Kim, W. Choi, and J. H. Ahn, "Cheddar: A swift fully homomorphic encryption library for cuda gpus," *arXiv preprint arXiv:2407.13055*, 2024.
- [43] 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.
- [44] 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 cryptography and information security*. Springer, 2019, pp. 415–445.
- [45] J. H. Cheon, D. Kim, and D. Kim, "Efficient homomorphic comparison methods with optimal complexity," in *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26*. Springer, 2020, pp. 221–256.
- [46] X. Jiang, M. Kim, K. E. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *ACM CCS 2018: 25th Conference on Computer and Communications Security*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. Toronto, ON, Canada: ACM Press, Oct. 15–19, 2018, pp. 1209–1222.
- [47] J. Kim, J. Seo, and Y. Song, "Simpler and faster bfv bootstrapping for arbitrary plaintext modulus from ckks," *Cryptology ePrint Archive*, 2024, to Appear at CCS 2024.
- [48] 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.
- [49] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 922–934.
- [50] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.
- [51] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.
- [52] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 870–881.
- [53] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.
- [54] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [55] T. Morshed, M. M. A. Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 142–153.
- [56] K. Nam, H. Oh, H. Moon, and Y. Paek, "Accelerating n-bit operations over tfhe on commodity cpu-fpga," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.
- [57] A. Putra, Prasetyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1319–1331.

- [58] L. Jiang, Q. Lou, and N. Joshi, “Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.
- [59] R. Agrawal, A. Chandrakasan, and A. Joshi, “Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 756–769.
- [60] K. Han and D. Ki, “Better bootstrapping for approximate homomorphic encryption,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2020, pp. 364–390.
- [61] M. R. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.
- [62] J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2022, pp. 521–541.
- [63] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, “High-precision bootstrapping of rms-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function,” in *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I 40*. Springer, 2021, pp. 618–647.
- [64] Y. Bae, J. H. Cheon, W. Cho, J. Kim, and T. Kim, “Meta-bits: Bootstrapping precision beyond the limit,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 223–234.

## Appendix

### 1. Compliance with the Open Science Policy

**Data Availability.** All datasets used in this research are generated from the publicly available git repository (<https://github.com/IntelLabs/VectorSearchDatasets>) following the methodology in [41].

**Code Availability and Reproducibility.** The source code for all experiments and analyses presented in this paper will be shared to Artifact Evaluation Committee (AEC) after the paper is accepted. This includes scripts for data loading/encoding/encryption and custom FHE algorithms developed for GraSS. The repository is organized to include detailed instructions for reproduction of our results, including versioning information and dependencies. This ensures that others can replicate our experiments under similar conditions.

**Preprint Submission.** We plan to publish a preprint of this work to ePrint Archive (<https://eprint.iacr.org/>) after the submission to Euro S&P 2025, which allows for early access and feedback.

**Non-polynomial Operations.** There have been active researches on how to efficiently perform non-polynomial operations such as comparison and min/max in word-wise FHE. For BGV (and B/FV), one can construct a bivariate polynomial over the finite field that outputs the comparison result of two inputs, and then homomorphically evaluate the polynomial [43]. For CKKS, a novel methodology to approximate the sign function (equivalent to

comparison) by a composite polynomial to minimize the computational complexity has been proposed in [44], [45]. Those work achieve a comparable performance with bit-wise FHE in terms of amortized running time. However, they commonly requires quite large multiplicative depth and are much less efficient than bit-wise FHE in sparse-packing case (i.e., packing less than  $b$  messages in a ciphertext).

**Ciphertext noise, level and bootstrapping.** In lattice-based FHE schemes, each ciphertext internally contains a noise that grows up for every homomorphic operation. Once the noise level reaches a certain upper bound, then no more homomorphic operation is able to be done (i.e., decryption fails). One way to mitigate the noise growth is to perform the modulus-switching algorithm [9] that reduce the exponential noise growth to a linear scale, but each modulus-switching procedure consumes a ciphertext level (from a max level  $L > 0$ ) so it cannot be a ultimate solution for the noise control. Thankfully, this bottleneck can be resolved by an algorithm called bootstrapping. Bootstrapping is technically a homomorphic evaluation of the decryption circuit so that it refreshes the old ciphertext noise by a new noise whose size is determined by the decryption circuit. Due to the noise-refreshing property, bootstrapping enables to compute arbitrary size and depth of circuit homomorphically without decryption failure. To support bootstrapping, we usually set  $N \geq 2^{16}$  and the maximal ciphertext modulus  $Q \geq 1600$ .

When  $\lceil \log n \rceil$  is not power-of-two, then we pad  $\ell - \lceil \log n \rceil$  number of zeros to each of the binary index vectors  $\text{idx}_{a_i}$  and  $\text{idx}_{b_j}$  to make their length power-of-two  $\ell$ . When  $|A|$  (resp.  $|B|$ ) is not power-of-two, we first need to assume that  $n < 2^{\lceil \log n \rceil} - 1$ . Then, we pad  $2^{\lceil \log |A| \rceil} - |A|$  (resp.  $2^{\lceil \log |B| \rceil} - |B|$ ) number of  $(1, 1, \dots, 1) \in \{0, 1\}^{\lceil \log_2 n \rceil}$  (resp.  $(1, 1, \dots, 1, 0) \in \{0, 1\}^{\lceil \log_2 n \rceil}$ ) to the set to make  $|A|$  (resp.  $|B|$ ) power-of-two.<sup>9</sup> If  $b < |A| \cdot |B| \cdot 2^{\lceil \log \ell \rceil}$ , then we partition the set  $A$  into  $|A| \cdot |B| \cdot 2^{\lceil \log \ell \rceil} / b$  subsets where each subset contains  $b / (|B| \cdot 2^{\lceil \log \ell \rceil})$  nodes, and we run Algorithm 4 for each subset of  $A$  and  $B$ . Note that the word-wise FHE batch size  $b$  is generally larger than  $|B| \cdot 2^{\lceil \log \ell \rceil}$  in practice. Note that Algorithm 4 is described for power-of-two  $|A|$ ,  $|B|$  and  $\ell$  satisfying  $|A| \cdot |B| \cdot \ell \leq b$  for simplicity.

In cleartext computation, evaluating the distance between the query and the neighbor nodes in  $N(x)$  for each iteration is the most natural way. In FHE computation, however, the distance pre-computation method works more efficiently than the on-the-fly computation due to the computational cost of the neighborhood retrieval process (Step 3 in Section 4.4.3).

The size of Table 2 (with distance and index) after the distance pre-computation is  $n \times m(1 + \lceil \log n \rceil)$ , while Table 1 (with feature vector and index) is of the size  $n \times m(d + \lceil \log n \rceil)$ . Hence, the masking process with the one-hot encoding vector in neighborhood retrieval requires  $\lceil \frac{n}{b} \rceil \cdot m(1 + \lceil \log n \rceil)$  (resp.  $\lceil \frac{n}{b} \rceil \cdot m(d + \lceil \log n \rceil)$ ) ct-pt mults

9. Here, the vectors  $(1, 1, \dots, 1), (1, 1, \dots, 1, 0)$  are the binary representation of the indices  $2^{\lceil \log_2 n \rceil} - 1, 2^{\lceil \log_2 n \rceil} - 2 > n - 1$ . We pad  $A$  and  $B$  with those “garbage” indices to make sure that the padding does not impact the set intersection result of the original  $A$  and  $B$ .

in the pre-computation (resp. on-the-fly computation) case for each iteration. In practice,  $\lceil \log n \rceil \leq 30$  is much smaller than the vector dimension  $d$  (e.g., 96), so the number of ct-pt mults required for the masking process is at least twice lower in the pre-computation case than the on-the-fly computation case.

Moreover, the computational cost of distance pre-computation itself is dominated by  $\lceil \frac{n}{b} \rceil \cdot m \cdot d$  ct-pt mults, which is even smaller than  $\lceil \frac{n}{b} \rceil \cdot m(d + \lceil \log n \rceil)$ . Therefore, the pre-computation of the distance table brings significant improvement in the efficiency compared to the naive on-the-fly distance computation.

Additionally, we modified Algorithm 5 to obtain binary indices by reusing intermediate results from the sorting process. Specifically, we generate an encryption of a permutation matrix  $P \in \{0, 1\}^{|Q| \times |Q|}$  during the distance sorting and add a homomorphic matrix multiplication between  $P$  and the binary indices. Initially, we set the matrix  $P$  as a row-wise packed identity matrix. Then, in line 18 of Algorithm 5, we reuse  $\text{ct}_S$  to conditionally swap columns of  $P$ . By the end of the loops,  $P$  becomes an encryption of a transposed permutation matrix. Finally, we transpose the matrix  $P$  and evaluate matrix multiplication between  $P_{0:w-1,0:|Q|-2}^T$  and  $\text{ct}_B = \text{Enc}(\text{id}_{x_x})$  for  $x \in Q$ . Here,  $\text{ct}_B$  is treated as a row-wise packed  $|Q| \times l$  matrix. For the matrix transpose and multiplication, we refer to [46].

## 2. Algorithmic Optimizations

**Exact FHE instead of CKKS.** We may think of using the exact FHE schemes BGV and B/FV instead of the approximate FHE scheme CKKS to get rid of the precision issue during index-related operations (e.g., neighborhood information retrieval). Currently, there is no open-source FHE library supporting scheme-switching between BGV/BFV and FHEW/TFHE. Another issue is that the level consumption during BGV/BFV bootstrapping highly depends on the plaintext modulus, while similarity search usually requires quite high precision to deal with large-scale data. Very recently, there has been proposed a new BFV bootstrapping technique [47] that exploits CKKS bootstrapping as a sub-routine and the level consumption does not depend on the plaintext modulus. Hence, it would be an interesting future work to implement 1. the new BFV bootstrapping of [47], 2. BFV-FHEW/TFHE scheme-switching, and 3. graph-based similarity search algorithm on the top of them.

**More Efficient Scheme-Switching.** Recently, there has been proposed a novel ring-packing method called HERMES [48] which offers very efficient and dynamic transformations between different ciphertext formats. For example, [48] shows that packing  $2^{15}$  LWE ciphertexts (e.g., FHEW/TFHE ciphertext) into a single RLWE ciphertext (e.g., CKKS ciphertext) with 10.2 seconds, which gives at most 41x higher throughput than the previous state-of-the-art [29], while the latency is comparable. For the parameters we implemented in this work, the target ciphertexts of scheme-switching only contains  $< 2^8$  inputs, and hence we are not able to enjoy 41x improvement on scheme-switching performance in our case. However, we

can apply this technique to our FHE solution for the case that  $m$  and  $w$  are sufficiently large.

## 3. FHE Implementation on GPU and HW accelerators

Recently, there have been massive improvements on the computational performance of FHE, including both word-wise and bit-wise FHE. For example, word-wise FHE schemes have been implemented on various platforms like GPU [42], [49], [50], FPGA [51], [52] and ASIC accelerators [53].

The state-of-the-art results show that bootstrapping in word-wise FHE, which is the most expensive operation, takes only 50ms and 0.39ms on GPU [42] and ASIC [53] respectively. These results are at least  $1,000\times$  faster compare to CPU implementations. Furthermore, this result brings down the CNN inference (ResNet-20 [54]) latency to 1.50s [42] and 99.0ms [53], which is practical for real-world use cases.

Bit-wise FHE schemes show great performance with GPU [55], FPGA [56] and ASIC [57], [58] accelerations. A state-of-the-art result with ASIC [57] accelerates programmable bootstrapping by about  $290\times$ . FPGA implementation [59], on the other hand, introduced scheme switching between word-wise and bit-wise FHE schemes to mitigate the latency of word-wise FHE bootstrapping. The work performs bootstrapping in 31 ms, which is also over  $1,000\times$  faster compare than CPU. Furthermore, they demonstrated ResNet-20 [54] inference in 0.267s. An interesting direction for future work is to incorporate a state-of-the-art FHE accelerator into our proposed FHE-based similarity search algorithm. Assuming that at least  $1,000\times$  acceleration can be achieved, combined with the high scalability of our graph-based similarity search algorithm, it would be feasible to perform similarity searches on a 1M dataset within a few minutes, making our protocol practical for real-world applications.

## 4. Note on Security Level and Bootstrapping

The bit security of word-wise FHE (CKKS) is determined by the maximal ciphertext modulus  $Q$  once the other FHE parameters including  $N$ , noise distribution, secret-key distribution, and decomposition degree  $d_{\text{num}}$  [60] are fixed, and  $Q$  is decided by the maximal multiplicative depth  $L$  and the scaling factor  $\Delta$ , as  $Q \approx \Delta^L$ . Smaller  $Q$  implies higher bit security. Our experiments on deep-96 datasets require 14 remaining-level-after-bootstrapping and 14-bit precision for bootstrapping output. As a result, we set  $L = 18 + 14 = 32^{10}$ ,  $\Delta \geq 2^{52}$ , which result in less than 128-bit security. To be precise, for multiplicative depth 32, OpenFHE automatically set 11 primes for the special modulus  $P$ , and hence  $\log PQ \simeq 2600$ . The bit security is determined by concrete security level against uSVP and hybrid dual attacks under the cost model MATZOV through lattice-estimator [61].

There are two main strategies to achieve 128-bit security: 1. Simply increase the ring dimension  $N$  from  $2^{16}$

10. The current OpenFHE implementation consumes 18 levels during CKKS bootstrapping under  $\text{LevelBudget} = \{2, 2\}$ .



to  $2^{17}$ , or 2. Apply recent new optimization techniques on CKKS bootstrapping that have not been applied to the public OpenFHE library [40] yet. The first strategy directly results in 128-bit security but accompanies  $2 \sim 4x$  computational overhead on CKKS operations and requires larger key size. In terms of the second strategy, there have been proposed some novel techniques [60], [62] to achieve higher remaining level after bootstrapping, and these techniques enables to set smaller depth parameter  $L$  which implies higher bit security from smaller  $Q \approx \Delta^L$ . Furthermore, there exist another line of work on optimizing CKKS bootstrapping in terms of output precision [62], [63]. In our experiments, to obtain 14-bit bootstrapping precision, we had to set large  $\Delta \geq 2^{52}$  and apply iterative bootstrapping [64]. Applying those optimizations would allow us to set smaller  $\Delta$  and/or avoid iterative bootstrapping, which result in higher security level and faster performance.

## 5. FHE Algorithms for GraSS

In this section, we describe the algorithms for each of the steps in Section 4:

- Algorithm 6 for Step 0: Distance table computation
- Algorithm 7 for Step 1: Find Closest Unexplored Node in  $Q$
- Algorithm 8 for Step 2: Update Explored Nodes  $E$
- Algorithm 9 for Step 3: Neighborhood Information Retrieval
- Algorithm 10 for Step 4: Set Union of  $Q$  and  $N(x)$

Note that Step 5 (Local Top- $w$ ) uses Algorithm 5 introduced in Section 3.2.

---

**Algorithm 6** Distance Table Computation

---

**Require:**  $\text{ct}_{\text{query},u} = \text{Enc}(v_u, v_u, \dots, v_u)$  for  $0 \leq u < d$ ,  $\text{pt}_{i,j,u} = \text{Ecd}(v_{s_{bi,j,u}}, v_{s_{bi+1,j,u}}, \dots, v_{s_{b(i+1)-1,j,u}})$  for  $0 \leq i < \lceil \frac{n}{b} \rceil$ ,  $0 \leq j < m$ , and  $0 \leq u < d$ .

**Ensure:**  $\text{ct}_{\text{out},i,j} = \text{Enc}(\text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{bi,j}}), \dots, \text{dist}^2(\mathbf{v}, \mathbf{v}_{s_{b(i+1)-1,j}}))$  for  $0 \leq i < \lceil \frac{n}{b} \rceil$  and  $0 \leq j < m$ .

```
1: for  $u = 0$  to  $d - 1$  do
2:    $\text{ct}_u \leftarrow \text{MULT}(\text{ct}_{\text{query},u}, \text{ct}_{\text{query},u})$ 
3:   if  $u > 0$  then
4:      $\text{ct}_0 \leftarrow \text{ADD}(\text{ct}_0, \text{ct}_u)$ 
5:   end if
6: end for
7: for  $i = 0$  to  $n/b - 1$  do
8:   for  $j = 0$  to  $m$  do
9:      $\text{ct}_{\text{out},i,j} \leftarrow \text{ct}_0$ 
10:    for  $u = 0$  to  $d - 1$  do
11:       $\text{ct}_{\text{temp}} \leftarrow \text{MULT}(\text{ct}_{\text{query},u}, 2\text{pt}_{i,j,u})$ 
12:       $\text{ct}_{\text{out},i,j} \leftarrow \text{SUB}(\text{ct}_{\text{out},i,j}, \text{ct}_{\text{temp}})$ 
13:       $\text{ct}_{\text{out},i,j} \leftarrow \text{ADD}(\text{ct}_{\text{out},i,j}, \text{pt}_{i,j,u}^2)$ 
14:    end for
15:  end for
16: end for
17: return  $\text{ct}_{\text{out},i,j}$  for  $0 \leq i < n/b$  and  $0 \leq j < m$ 
```

---

---

**Algorithm 7** Find Closest Unexplored Node in  $Q$  at  $i$ -th Iteration

---

**Require:**  $\text{ct}_E = \text{Enc}(\text{idx}_{x_0}, \dots, \text{idx}_{x_0}, \dots, \text{idx}_{x_{i-1}}, \dots, \text{idx}_{x_{i-1}}, 0, \dots, 0)$  where  $x_j$  denotes the explored node in  $j$ -th iteration,  $\text{ct}_{Q,\text{idx}} = \text{Enc}(\text{idx}_{q_0}, \text{idx}_{q_1}, \dots, \text{idx}_{q_{w-1}})$ ,  $\text{ct}_{Q,\text{dist}} = \text{Enc}(\text{dist}_{q_0}, \text{dist}_{q_1}, \dots, \text{dist}_{q_{w-1}})$ .

**Ensure:**  $\text{ct}_{\text{out}} = \text{Enc}(\text{idx}_x, \text{idx}_x, \dots, \text{idx}_x)$ .

```
1:  $\text{ct}_{Q,\text{explored}} \leftarrow \text{Broadcast}(\text{ct}_{Q,\text{idx}})$ 
2:  $\text{ct}_{Q,\text{explored}} \leftarrow \text{SetInt}(\text{ct}_E, \text{ct}_{Q,\text{idx}})$ 
3:  $\text{ct}_{Q,\text{dist}^*} \leftarrow \text{DISTRMASK}(\text{ct}_{Q,\text{dist}}, \text{ct}_{Q,\text{explored}})$ 
4:  $\text{ct}_{\text{argmin}} \leftarrow \text{ArgMin}(\text{ct}_{Q,\text{dist}^*})$ 
5:  $\text{ct}_{x,\text{idx}} \leftarrow \text{IDXMASK}(\text{ct}_{Q,\text{idx}}, \text{ct}_{\text{argmin}})$ 
6:  $\text{ct}_{\text{out}} = \text{Broadcast}(\text{ct}_{x,\text{idx}})$ 
```

---

---

**Algorithm 8** Update Explored Nodes  $E$  at  $i$ -th Iteration

---

**Require:**  $\text{ct}_E = \text{Enc}(\text{idx}_{x_0}, \dots, \text{idx}_{x_0}, \dots, \text{idx}_{x_{i-1}}, \dots, \text{idx}_{x_{i-1}}, 0, \dots, 0)$  where  $x_j$  denotes the explored node in  $j$ -th iteration,  $\text{ct}_{x,\text{idx}} = \text{Enc}(\text{idx}_{x_i}, \text{idx}_{x_i}, \dots, \text{idx}_{x_i})$ ,  $\text{pt}_{\text{mask}} = \text{Ecd}(1, \dots, 1, 0, \dots, 0)$  where first  $w \cdot \ell$  slots are 1.

**Ensure:**  $\text{ct}_E = \text{Enc}(\text{idx}_{x_0}, \dots, \text{idx}_{x_0}, \dots, \text{idx}_{x_i}, \dots, \text{idx}_{x_i}, 0, \dots, 0)$ .

```
1:  $\text{ct}_{x,\text{idx}} = \text{MULT}(\text{ct}_{x,\text{idx}}, \text{pt}_{\text{mask}})$ 
2:  $\text{ct}_{x,\text{idx}} = \text{ROTATE}(\text{ct}_{x,\text{idx}}; -i \cdot w \cdot \ell; \text{rotk})$ 
3:  $\text{ct}_E = \text{ct}_E + \text{ct}_{x,\text{idx}}$ 
```

---

---

**Algorithm 9** Neighbor Nodes Binary Index Retrieval

---

**Require:**  $\text{ct}_x = \text{Enc}(\text{idx}_x, \text{idx}_x, \dots, \text{idx}_x)$ ,  $\text{pt}_{i,j,u} = \text{Ecd}(\text{idx}_{s_{bi,j,u}}, \text{idx}_{s_{bi+1,j,u}}, \dots, \text{idx}_{s_{b(i+1)-1,j,u}})$  for  $0 \leq i < \lceil \frac{n}{b} \rceil$ ,  $0 \leq j < m$  and  $0 \leq u < \lceil \log n \rceil$

**Ensure:**  $\text{ct}_{\text{out}} = \text{Enc}(\text{idx}_{s_{x,0}}, \text{idx}_{s_{x,1}}, \dots, \text{idx}_{s_{x,m-1}}, \mathbf{0}, \dots, \mathbf{0})$

```
1:  $\{\text{ct}_{\text{idx}_{x,u}}\}_{0 \leq u < \lceil \log n \rceil} \leftarrow \text{Unpack}(\text{ct}_x)$ 
2:  $\{\text{ct}_{\text{onehot},i}\}_{0 \leq i < \lceil \frac{n}{b} \rceil} \leftarrow \text{BintoOneHot}(\{\text{ct}_{\text{idx}_{x,u}}\}_{0 \leq u < \lceil \log n \rceil})$ 
3: for  $j \leftarrow 0$  to  $m - 1$  do
4:   for  $u \leftarrow 0$  to  $\lceil \log n \rceil - 1$  do
5:      $\text{ct}_{j,u} \leftarrow \text{MULT}(\text{ct}_{\text{onehot},0}, \text{pt}_{0,j,u})$ 
6:     for  $i \leftarrow 1$  to  $\lceil \frac{n}{b} \rceil - 1$  do
7:        $\text{ct}_{j,u} \leftarrow \text{ADD}(\text{ct}_{j,u}, \text{MULT}(\text{ct}_{\text{onehot},i}, \text{pt}_{i,j,u}))$ 
8:     end for
9:   end for
10:   $\text{ct}_{\text{idx},j} \leftarrow \text{BlindRotate}(\{\text{ct}_{j,u}\}_{0 \leq u < \lceil \log n \rceil}, \{\text{ct}_{\text{idx}_{x,u}}\}_{0 \leq u < \log \ell})$ 
11: end for
12:  $\text{ct}_{\text{out}} \leftarrow \text{Pack}(\{\text{ct}_{\text{idx},j}\}_{0 \leq j < m})$ 
```

---

---

**Algorithm 10** Set Union of  $Q$  and  $N(x)$ 's Distance at  $i$ -th Iteration

---

**Require:**  $ct_{Q,idx} = \text{Enc}(\mathbf{idx}_{q_0}, \mathbf{idx}_{q_1}, \dots, \mathbf{idx}_{q_{w-1}})$ ,  $ct_{N,idx} = \text{Enc}(\mathbf{idx}_{s_{x,0}}, \mathbf{idx}_{s_{x,1}}, \dots, \mathbf{idx}_{s_{x,m-1}})$ ,  $ct_{Q,dist} = \text{Enc}(\mathbf{dist}_{q_0}, \mathbf{dist}_{q_1}, \dots, \mathbf{dist}_{q_{w-1}})$ ,  $ct_{N,dist} = \text{Enc}(\mathbf{dist}_{s_{x,0}}, \mathbf{dist}_{s_{x,1}}, \dots, \mathbf{dist}_{s_{x,m-1}})$ .

**Ensure:**  $ct_{out} = \text{Enc}(\mathbf{dist}_{Q \cup N(x)})$

- 1:  $ct_{Q,idx} \leftarrow \text{Broadcast}(ct_{Q,idx})$
  - 2:  $ct_{N,idx} \leftarrow \text{Broadcast}(ct_{N,idx})$
  - 3:  $ct_{N,duplicate} \leftarrow \text{SetInt}(ct_{Q,idx}, ct_{N,idx})$
  - 4:  $ct_{N,dist*} \leftarrow \text{DSTMASK}(ct_{N,dist}, ct_{N,duplicate})$
  - 5:  $ct_{N,dist*} \leftarrow \text{ROTATE}(ct_{N,dist*}; -w; \mathbf{rotk})$
  - 6:  $ct_{out} \leftarrow \text{ADD}(ct_{Q,dist}, ct_{N,dist*})$
-