A non-comparison oblivious sort and its application to private k-NN

Sofiane Azogagh
azogagh.sofiane@courrier.uqam.ca
Univ Québec à Montréal
Canada

Marc-Olivier Killijian
killijian.marc-olivier.2@uqam.ca
Univ Québec à Montréal
Canada

Félix Larose-Gervais
larose-gervais.felix@courrier.uqam.ca
Univ Québec à Montréal
Canada

ABSTRACT

This paper introduces a novel adaptation of counting sort that enables sorting of encrypted data using Fully Homomorphic Encryption (FHE). Our approach represents the first known sorting algorithm for encrypted data that does not rely on comparisons. The implementation leverages some basic operations on TFHE's Look-Up-Table (LUT) . We have integrated these operations into RevoLUT [1], a comprehensive open-source library built upon tfhe-rs [37], which can be of independent interest for oblivious algorithms. We demonstrate the effectiveness of our Blind Counting Sort algorithm by developing a top-k selection algorithm and applying it to privacy-preserving k-Nearest Neighbors classification. This approach achieves approximately 5x faster performance compared to current state-of-the-art methods.

KEYWORDS

Privacy, Homomorphic encryption, Oblivious algorithm, Sort, k-Nearest Neighbors, Look-Up-Table

1 INTRODUCTION

As data security becomes increasingly critical in the era of cloud computing, the need for secure data processing methods has never been more pressing. Homomorphic encryption, first introduced by Rivest et al. in 1978 [29], presents a groundbreaking approach to performing computations on encrypted data without needing to decrypt it first. This capability is particularly valuable in scenarios where sensitive data, such as personal health records or financial information, needs to be processed by third-party services while maintaining privacy. Sorting algorithms, such as QuickSort [21], MergeSort [33], and HeapSort [35], are fundamental in computer science and are omnipresent in various applications, ranging from database management to network security. When applied to encrypted data, sorting becomes a non-trivial task due to the fact that it is a data-dependent operation. Thus, most of the traditional sorting methods are not directly applicable to encrypted data, requiring then the development of specialized algorithms that can efficiently sort encrypted data. Recent advancements in fully homomorphic encryption schemes have opened the door to practical applications where sorting on encrypted data is feasible. FHE schemes, such as TFHE [15], BGV/BFV [7, 20], and CKKS [12], enable the execution of arbitrary functions on ciphertexts, including comparisons, which are the basic operations in sorting algorithms. Sorting encrypted data plays a crucial role in privacy-preserving data analysis, especially within the realm of machine learning. In this context, sorting is often essential for secure model training or prediction, where data must be organized or indexed without revealing sensitive information. A famous example is the selection of k-nearest neighbors (k-NN) from encrypted data [3, 19, 39] that we also took to illustrate the effectiveness of our solution. Additionally, there are innovative applications such as aggregating encrypted gradients [18] in federated learning. Moreover, the development of efficient sorting algorithms for encrypted data also extends to the domain of database management. Secure databases that operate on encrypted data need to implement sorting to support queries that involve ordering or range searches. Efficient encrypted sorting algorithms enhance the functionality of encrypted databases, enabling more complex queries while ensuring data confidentiality [27].

In this paper, we delve into the recent advancements in sorting algorithms for encrypted data using homomorphic encryption. We examine the different methodologies in the literature, and provide a comprehensive explanation of our approach, which leverages certain operations of the TFHE cryptosystem. The structure of this paper is as follows: Section 2 introduces the necessary tools and background information on the TFHE cryptosystem. Section 3 reviews existing techniques and the adaptation of traditional sorting algorithms to the encrypted context. Section 4 details our proposed approach with a time complexity analysis and noise analysis. Section 6 presents the experimental results of our sorting algorithm and its application to private inference on privacy-preserving machine learning models.

2 PRELIMINARIES

In this section, after introducing the notation used in the paper, we will present the necessary background on the TFHE's cryptosystem and our contribution, the RevoLUT library, which is of significant independent interest and was used to implement our Blind Sort algorithm.

2.1 Notation

Let p be a power of 2. We denote by $\mathbb{Z}_p$ the set of messages and by $[\![m]\!]$ the TFHE encryption of a message $m \in \mathbb{Z}_p$. For N a power of 2, we define $\mathcal{R}$ as the quotient ring $\mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q$ as the same ring modulo q, that is $\mathbb{Z}_q[X]/(X^N + 1)$. Unless otherwise specified, all operations in this paper are performed in the ring $\mathcal{R}_q$. We also make use of the Kronecker delta function δ_{ij}, which

Conference'17 (), 1–9
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

equals 1 when $i = j$ and 0 otherwise. Using this notation, we can define the one-hot encoding of an integer i as the bit vector $\delta_i = (\delta_{ij})_{j=1}^{p} \in \{0,1\}^p$, which contains a single 1 at index i and 0s elsewhere. Other notations are defined in the text when needed.

2.2 The TFHE Cryptosystem

The TFHE encryption scheme, proposed in 2016 [13, 14], is based on the security of the Learning With Errors (LWE) problem and its ring variant, the Ring-LWE (RLWE) problem.

2.2.1 Ciphertext Types. In TFHE, several types of ciphertexts are defined depending on the nature of the plaintext and the encryption method employed. A commonly used type in this paper is the General LWE (GLWE) ciphertext, defined as follows:

GLWE Ciphertexts. A message $m \in \mathbb{Z}_p$ can be encrypted under the secret key $s = (s_0, \ldots, s_{k-1}) \xleftarrow{\$} \mathbb{Z}_2^k$ as a GLWE ciphertext $(a, b) \in \mathcal{R}_q^{k+1}$, where $a = (a_0, \ldots, a_{k-1}) \xleftarrow{\$} \mathcal{R}_q^k$ and $b = \sum_{i=0}^{k-1} a_i \cdot s_i + \Delta m + e$, with $\Delta = \frac{q}{p}$ and e being a noise term sampled from a Gaussian distribution. The vector a is commonly called *mask* and b *body.*

Specifically, when $N = 1$, the ciphertext is referred to as an LWE ciphertext. When $k = 1$ and $N > 1$, it is termed an RLWE ciphertext. In this case, an LWE ciphertext encrypts a message in $\mathbb{Z}_q$, while an RLWE ciphertext encrypts a polynomial in $\mathbb{Z}_q[X]$ modulo $X^N + 1$.

LUT Ciphertexts. Additionally, [5] introduced Look-Up-Table (LUT) ciphertexts, which are essentially RLWE ciphertexts that include some redundancy. A Look-Up-Table in TFHE is a vector $(m_i)_{0 \le i < p}$ of $\mathbb{Z}_p$ elements represented as a polynomial $M(X) \in \mathcal{R}_q$ of the form:

$$M(X) = \sum_{i=0}^{p-1} \sum_{j=0}^{\frac{N}{p}-1} m_i X^{i\frac{N}{p}+j}$$

This polynomial is then encrypted as an RLWE ciphertext to form a LUT ciphertext. Thus, a LUT ciphertext is a specific case of an RLWE ciphertext.

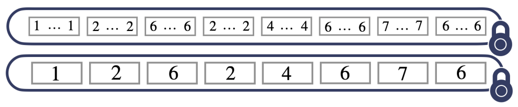

Fig. 1. Illustration of a RLWE ciphertext (top) with redundancy shown in gray boxes, which implements a LUT ciphertext (bottom) where each box represents an element in $\mathbb{Z}_p$ (here $p = 8$).

In this paper, ciphertexts are denoted within brackets to indicate their type. For instance, $[\![M]\!]_{\text{LUT}} = [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$ represents the message $M = (m_0, \ldots, m_{p-1})$ encrypted as a LUT ciphertext, while $[\![m]\!]_{\text{LWE}}$ is an LWE ciphertext and $[m]_{\text{LWE}}$ is a trivially encrypted LWE ciphertext (that is a ciphertext whose mask and noise are set to 0).

2.2.2 Classical Homomorphic Operations. As in all the homomorphic encryption schemes based on the LWE problem, the basic operations that can be performed on ciphertexts are:

- **Addition:** $([\![\star]\!]_{\text{RLWE}}, [\![\star]\!]_{\text{RLWE}}) \to [\![\star]\!]_{\text{RLWE}}$. Given two GLWE ciphertexts $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$, the addition operation computes a new GLWE ciphertext $c_3 = (a_3, b_3)$ where $a_3 = a_1 + a_2$ and $b_3 = b_1 + b_2$.
- **Absorption:** $(\star, [\![\star]\!]_{\text{RLWE}}) \to [\![\star]\!]_{\text{RLWE}}$. This operation multiplies a plaintext value m with a GLWE ciphertext $c = (a, b)$ by computing $(m \cdot a, m \cdot b)$. Note that this is the only multiplication that can be performed in TFHE (i.e the multiplication of two GLWE ciphertexts is not supported).

2.2.3 TFHE's operations. TFHE provides several building blocks for performing homomorphic operations on ciphertexts. The main operations used in this paper are:

- **Blind Rotation (BR):** $([\![\star]\!]_{\text{LWE}}, [\![\star]\!]_{\text{LUT}}) \to [\![\star]\!]_{\text{RLWE}}$. This operation is used to privately rotate the polynomial $M(X)$ (encrypted as an RLWE ciphertext) by $[\![i]\!]_{\text{LWE}}$ coefficients.
- **Sample Extraction (SE):** $(\star, [\![\star]\!]_{\text{RLWE}}) \to [\![\star]\!]_{\text{LWE}}$.
This operation extracts a coefficient from the polynomial $M(X) = \sum_{i=0}^{N-1} m_i X^i$ encrypted as an RLWE ciphertext, resulting in an LWE ciphertext $[\![m_j]\!]_{\text{LWE}}$. The LWE ciphertext is generated by selecting specific coefficients from the RLWE input.
- **Key Switching (KS):** $[\![\star]\!]_{\text{LWE}} \to [\![\star]\!]_{\text{LWE}}$.
This operation switches the secret key or parameters of an LWE ciphertext to new ones by homomorphically re-encrypting the ciphertext with a different key.
- **Public Functional Key Switch (PFKS):** $\{[\![\star]\!]_{\text{LWE}}\} \to [\![\star]\!]_{\text{RLWE}}$.
Introduced in [16] (Algorithm 2), this operation allows for the compact representation of multiple LWE ciphertexts into a single RLWE ciphertext, effectively packing several LWE ciphertexts into one.

The redundancy in a LUT ciphertext is mainly important to guarantee the correctness of the bootstrapping operation. Indeed, the LWE ciphertext used in the Blind Rotation operation serves as an index to select the correct coefficient from the LUT ciphertext. However, this LWE ciphertext incorporates a gaussian noise e which is bounded by N/p after the so-called Modulus Switching operation (see [17] for more details). This bound gives exactly the size of the redundancy of the coefficients in the RLWE ciphertext implementing the LUT. These sequences of consecutive coefficients in the RLWE ciphertext implementing a LUT are generally called *boxes.* During the (functional) bootstrapping operation, each box corresponds to a specific message m_i of the LUT ciphertext. When the Blind Rotation is performed, $[\![i]\!]_{\text{LWE}}$ points to the i-th box containing the message m_i in the LUT. Thus, the redundancy ensures that, despite the random error present in $[\![i]\!]_{\text{LWE}}$, the Sample Extraction operation will still correctly select the message m_i as long as the noise e is smaller than the redundancy. Note that, in RevoLUT, for a better noise management, we perform a Key Switching before each Blind Rotation. Hence, in the rest of the paper, whenever we refer to the cost of Blind Rotation, we implicitly include the cost of the associated Key Switching operation

3 RELATED WORK

Oblivious sorting. Historically, Batcher [6] introduced his odd-even merge sort, which has later been adapted for homomorphic encryption due to its natural data obliviousness. This algorithm is particularly well-suited for parallel processing, making it efficient for sorting large encrypted datasets. Later, Cetin [8–10] introduced the Direct and Greedy sort algorithms. They also presented a depth analysis of classical sorting algorithms such as Bubble Sort, Insertion Sort, Odd-Even Sort, Odd-Even Merge Sort, Merge Sort, and Bitonic Sort. Their work highlights the trade-offs between computational depth and efficiency in homomorphic sorting. However, the number of blind comparisons required by their algorithm scales quadratically with the size of the input array. Building on this, [23] proposed a faster comparison algorithm and applied it to the batched direct sort algorithm. This work addresses the computational bottlenecks associated with comparison operations in encrypted domains. For their work on oblivious top-k selection, [19] implemented a truncated version of the batcher's odd-even merge sorting network for TFHE using the comparison operator from [39]. In [18] proposed an efficient homomorphic trimmed sum algorithm that can be used to sort encrypted data. In the realm of secure multi-party computations, [34] proposed a secure multi-party sorting protocol based on Yao's garbled circuit. This protocol allows multiple parties to sort data without revealing the underlying values, providing a foundation for secure collaborative computations. Moreover, [25] proposed a new homomorphic sorting algorithm based on the Hardy-Littlewood-Polya rearrangement inequality. This algorithm offers remarkable performance and represents a major advancement in the field of homomorphic sorting. The use of mathematical inequalities in sorting provides a novel approach to optimizing encrypted computations. Recently, [22] proposed an efficient batched k-way sorting network using approximate comparison which scales remarkably well but is not exact.

Private k-Nearest Neighbors. The problem of finding the k nearest neighbors of a query vector in a private manner has been widely studied in the literature [3, 19, 26, 28, 30, 38, 39]. The most closely related works of this paper are those of [3, 19, 39] who proposed to leverages fully homomorphic encryption, and more precisely TFHE scheme, to perform private non-interactive k-NN inference. In [39] a method is introduced to build a matrix of closeness where the (i, j) elements of this matrix is set to 1 if the i-th point of the model is closer than the j-th point to the query vector. To build this matrix, the authors introduced an elegant way to perform a distances computation that we also use in this paper and detail in Section 5.2. This matrix is then used to compute a score between 0 and k where the higher the score, the closer vector i is to the query vector. Based on this work, [3] proposed a method to compute the most frequent labels of the k nearest neighbors of the query vector through a majority vote. To do so, they used the sum of the lines of the closeness matrix as a mask to compute the frequency of each label in the k nearest neighbors of the query vector. But the performance showed in the paper does not seem to be far from the one of [39]. To this date, the state of the art for private k-NN inference leveraging FHE is the work of [19] who unearthed an old sorting algorithm that is naturally data-oblivious and thus FHE-friendly. However, due to the lack of TFHE comparators that do not require additional padding bits, their algorithm performs worse in practice than it theoretically should.

Our contribution. Based on the oblivious read/write property enabled by TFHE's Look-Up-Tables, it allowed us to port the so-called counting sort algorithm in FHE to sort encrypted data. This has enabled us to use it as a subroutine in a top-k algorithm, which we demonstrate to be efficient when applied to k-NN inference. As far as we know it is the first attempt to port a non-comparison based sorting algorithm working on encrypted data. Therefore, by targeting this sort, we free ourselves from the need of comparators to sort encrypted elements, which was the main bottleneck in homomorphic sorting using TFHE, since it required an additional bit of precision.

4 OBLIVIOUS SORTING ALGORITHM

In this section, we first present read and write primitives developped in RevoLUT for manipulating LUT ciphertexts like arrays. Then we present our Blind Counting Sort algorithm built with them, and then detail how we used it as a subroutine in our Blind Top-k Selection algorithm for private k-NN inference.

4.1 Read and Write operations

4.1.1 Blind Array Access. Introduced by [4], Blind Array Access (BAA) is the first primitive that we required to access an encrypted index in our LUT. This is achieved by using the Blind Rotate procedure, and then extracting the sample at index 0 from the resulting RLWE ciphertext.

Algorithm 1: Blind Array Access

```
Input  : An encrypted index $[\![i]\!]_{\mathrm{LWE}}$
         A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}}$
Output : A LWE ciphertext $[\![m_i]\!]_{\mathrm{LWE}}$
1 $[\![rotated]\!]_{\mathrm{LUT}} \leftarrow BR([\![i]\!]_{\mathrm{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}})$
2 $[\![m_i]\!]_{\mathrm{LWE}} \leftarrow SE(0, [\![rotated]\!]_{\mathrm{LUT}})$
3 return $[\![m_i]\!]_{\mathrm{LWE}}$
```

4.1.2 Blind Array Increment. The second primitive required is some form of blind write operation in the LUT. For this we implemented Blind Array Increment, which adds to the i-th message of the given LUT the value of x. A Blind Array Assignment could easily be devised by first using Blind Array Access to fetch the current value, and subtract it from the given x before running Blind Array Increment. This would double the blind rotation cost and was left aside since we didn't need it for our purposes, but could help porting other algorithms.

Algorithm 2: Blind Array Increment

```
Input  : An encrypted index $[\![i]\!]_{\mathrm{LWE}}$
         A LUT ciphertext $[\![m]\!]_{\mathrm{LUT}}$
         An encrypted value $[\![x]\!]_{\mathrm{LWE}}$
Output : A LUT ciphertext $[\![m + \delta_i x]\!]_{\mathrm{LUT}}$
1 $[\![\delta_0 x]\!]_{\mathrm{LUT}} \leftarrow PFKS([\![x]\!]_{\mathrm{LWE}})$
2 $[\![\delta_i x]\!]_{\mathrm{LUT}} \leftarrow BR(-[\![i]\!]_{\mathrm{LWE}}, [\![\delta_0 x]\!]_{\mathrm{LUT}})$
3 return $[\![m]\!]_{\mathrm{LUT}} + [\![\delta_i x]\!]_{\mathrm{LUT}}$
```

The time cost of these operations follows directly from that of Blind Rotate and the Public Functional Key Switch. In both cases, the cost is comparatively negligible if the provided argument is trivially encrypted.

Table 1: Time approximations for Blind Array operations

	t_{BAI}		t_{BAA}
	$[\![x]\!]_{\text{LWE}}$	$[x]_{\text{LWE}}$	
$[\![i]\!]_{\text{LWE}}$	-	t_{KS}	-
$[i]_{\text{LWE}}$	t_{BR}	$t_{BR} + t_{KS}$	t_{BR}

A caveat of this approach is that the $[\![\delta_i x]\!]_{\text{LUT}}$ is most likely misaligned due to the noise present in the rotation index. This affects the frontiers of the redundancy boxes present in LUT ciphertexts. A way to avoid error propagation is to Sample Extract every message from the LUT and pack them in a fresh LUT.

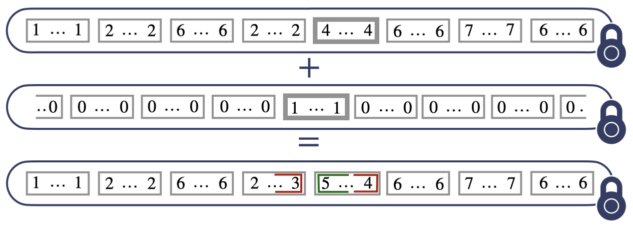

Fig. 2. Illustration of $BAI([\![4]\!]_{\text{LWE}}, [\![1,2,6,2,4,6,7,6]\!]_{\text{LUT}}, [\![1]\!]_{\text{LWE}})$ with $p = 8$. The red areas at the boundaries of the redundancy boxes represent errors due to the noise in the LWE encryption of $[\![4]\!]_{\text{LWE}}$. If the noise in the LWE ciphertext were zero, the boxes would be perfectly aligned. However, since we have no control over this noise, except that it does not exceed $(N/2p)$, we can only be certain that the center of the boxes remains accurate.

4.2 Blind Counting Sort

Counting sort is an interesting and well known sorting algorithm (historically attributed to [31]) that is not based on comparison. As such, the $\Omega(n \log n)$ lower bound on time complexity of comparison based sorting does not apply to it [24]. Instead it achieves worst-case performance (usually noted $O(n+k)$) scaling linearly with both the size of the input and its range of values. This is ideal since we use LUT ciphertexts to represent encrypted arrays of p integers modulo p, so $n = k = p$.

It is worth noting that, like sorting networks, counting sort is naturally data oblivious. That makes it FHE friendly, in the sense that porting the algorithm does not require to adapt its control flow.

4.2.1 Algorithm. We propose Algorithm 3, porting the classical counting sort to operate on encrypted arrays represented as LUT ciphertexts. The procedure can be summarized as follows:

(1) Build a count array
(2) Compute its running sum
(3) Reconstruct the sorted array

After step 2, the running sum array effectively tracks for each $i < p$ how many input elements are less than or equal to i. If each elements of the input array were distinct, these would in turn be the correct (1-based) indices where they belong in the sorted array. To account for duplicates, the running sums are decremented as they are visited, in reverse, so as to maintain the sort's stability.

Algorithm 3: Blind Counting Sort (BCS)

```
Input  : A LUT ciphertext $[\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}}$
Output : A sorted LUT
1  $[\![count]\!]_{\text{LUT}} \leftarrow [\![0, \dots, 0]\!]_{\text{LUT}}$
2  for $i \leftarrow 0$ to $p-1$ do
       // $count_{m_i} \leftarrow count_{m_i} + 1$
3      $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
4      $[\![count]\!]_{\text{LUT}} \leftarrow BAI([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [1]_{\text{LWE}})$
5  end
6  for $i \leftarrow 1$ to $p-1$ do
       // $count_i \leftarrow count_i + count_{i-1}$
7      $[\![x]\!]_{\text{LWE}} \leftarrow BAA([i-1]_{\text{LWE}}, [\![count]\!]_{\text{LUT}})$
8      $[\![count]\!]_{\text{LUT}} \leftarrow BAI([i]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [\![x]\!]_{\text{LWE}})$
9  end
10 $[\![res]\!]_{\text{LUT}} \leftarrow [\![0, \dots, 0]\!]_{\text{LUT}}$
11 for $i \leftarrow p-1$ to $0$ do
       // $count_{m_i} \leftarrow count_{m_i} - 1$
12     $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
13     $[\![count]\!]_{\text{LUT}} \leftarrow BAI([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [-1]_{\text{LWE}})$
       // $res_{count_{m_i}} \leftarrow m_i$
14     $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
15     $[\![count_{m_i}]\!]_{\text{LWE}} \leftarrow BAA([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}})$
16     $[\![res]\!]_{\text{LUT}} \leftarrow BAI([\![count_{m_i}]\!]_{\text{LWE}}, [\![res]\!]_{\text{LUT}}, [\![m_i]\!]_{\text{LWE}})$
17 end
18 return $[\![res]\!]_{\text{LUT}}$
```

Time complexity. The algorithm we propose uses primarily Blind Rotate and PFKS operations, which are the most expensive in TFHE. So we can approximate its time complexity as, ignoring the Blind Rotate calls on trivially encrypted indices, $4p \cdot (t_{BR} + t_{PFKS})$, where t_{BR} is the time it takes to execute a Blind Rotate, and t_{PFKS} the time required for a PFKS.

Box centering. Whenever a LUT gets blindly rotated, it de-centers the boxes, due to the noise in the index ciphertext. However, in this algorithm, the blindly rotated LUT are discarded and not re-used in further blind rotations, so this error margin does not increase. It is important to note that upon completion, the result LUT is the sum of many decentered LUT, and as such a few of its coefficients at the boxes frontiers are incorrect. To alleviate this issue, the caller can sample extract all boxes and re-pack a fresh centered LUT if they wish.

Noise growth analysis. In the first loop, the count LUT (initially noiseless) gets added into p times from the result of a blind rotation over a noiseless LUT, so its noise grows up to $p\mathcal{E}_{BR}$. In the second loop, the count LUT gets added into p times from the result of a packing of a noisy LWE extracted from the previous count LUT.

This gives us a noise growth of order $\sum_{i=0}^{p} i\mathcal{E}_{KS} = \frac{p(p-1)}{2}\mathcal{E}_{KS}$. In the third loop, the count LUT, gets added into p times in the same manner as the first loop (from a noiseless LUT), so it grows by an additional $p\mathcal{E}_{BR}$. Therefore, the total noise of the count LUT is bounded by

$$2p\mathcal{E}_{BR} + \frac{p(p-1)}{2}\mathcal{E}_{KS}$$

The result LUT (initially noiseless) gets added into p times from the result of a blind rotation of a packed LUT from an input LWE. Note that since the noise of a blind rotation is independant of the noise of its index, the result LUT noise is independant of the count LUT noise. Therefore the result LUT noise grows up to

$$\sum_{i=0}^{p} \mathcal{E}_{m_i} + p\mathcal{E}_{KS} + p\mathcal{E}_{BR}$$

Interestingly, the running sum part of the algorithm is the cheapest in terms of time complexity but turns out to be the source of the quadratic error growth. For p large enough, this will require bootstrapping some count values during the second loop to maintain the correctness of the algorithm.

4.3 Blind Top-k selection

For $k < p$, we can implement Blind Top-k selection in a tournament fashion, using BCS as a (k, d)-selector for $d \leq p$. An illustration of the blind Top-k selection is given in Figure 3.

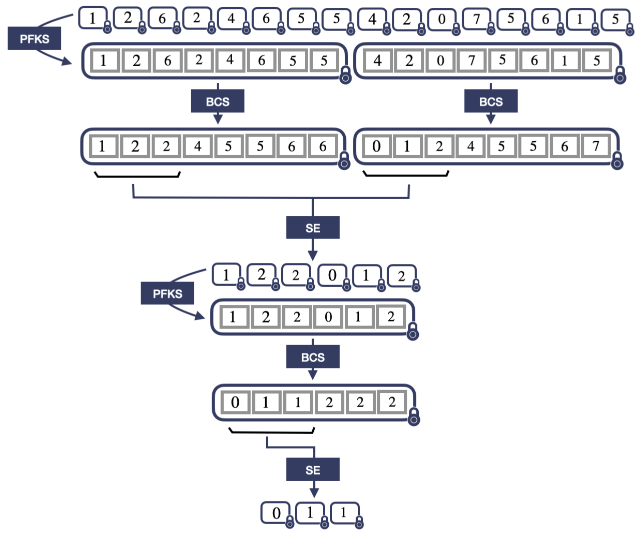

Fig. 3. Illustration of the blind Top-k selection algorithm for $k = 3$ and $p = 8$. The input LWE ciphertexts are first split into chunks of size p. Each chunk is then packed into a LUT through a Public Functional Key Switch (PFKS) and processed by a Blind Counting Sort (BCS) to select its k smallest elements using multiple Sample Extraction (SE). The selected elements from each chunk are then recursively processed until only k elements remain.

Complexity analysis. Since BCS is a non-comparison oblivious sort, it does not require comparators contrarily to [19], so we can't compare ourselves in the number of comparisons. However, the commons and most expensive operations in both algorithms are Blind Rotation (BR) and Public Functional Key Switch (PFKS), so we focus on comparing the number of BR and PFKS in both algorithms. To date, and to the best of our knowledge, there is no homomorphic comparator with TFHE that does not require an extra bit of precision. The comparator used in [19] which is the one of [11], does not escape this limitation. For instance, when processing 4-bit data ($p = 16$), they need to use 5-bit parameters ($p = 32$), which adds practical complexity to their approach.

In Table 2, we report the running times of the Blind Rotation and Public Functional Key Switch operations for two different values of p in the tfhe-rs library and in Figure 4 we give the estimated running time of our blind Top-k selection algorithm compared to [19] when processing data with the same precision (*i.e.* elements are from $\mathbb{Z}_{16}$).

Table 2: Running times (in ms) of Blind Rotation and PFKS in tfhe-rs library [37] for $p = 16$ and $p = 32$.

p	Blind Rotate (t_{BR})	Public Functional Key Switch (t_{PFKS})
16	18	3
32	43	12

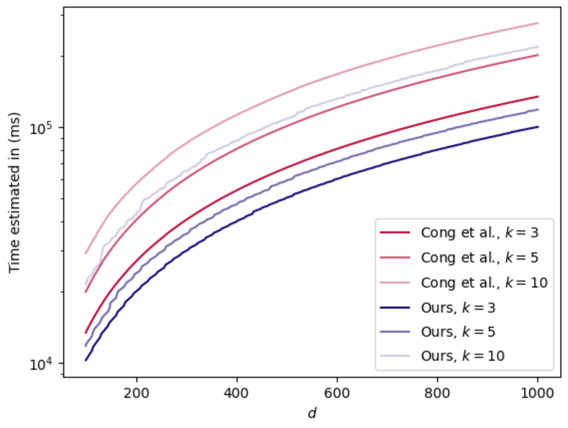

Fig. 4. Estimated running time of our blind Top-k selection algorithm compared to [19] when processing data with the same precision (*i.e.* elements are from $\mathbb{Z}_{16}$).

5 PRIVATE K-NEAREST NEIGHBORS CLASSIFICATION

In this section, we present our use case of applying the Blind Counting Sort algorithm to an efficient top-k algorithm enabling private inference on k-Nearest Neighbors. We first describe the pipeline of the private k-Nearest Neighbors classification and then we detail each step of the pipeline.

5.1 Pipeline and threat model

In a classical setting of a Machine Learning as a Service (MLaaS) platform, a client who wants to perform a k-NN classification with a classifier in the cloud will send its data (i.e a vector of features f) to the server. The server owns the model, i.e the set of points $(w_1, \ldots, w_n)$ and the corresponding labels $(l_1, \ldots, l_n)$. Thus, after receiving the input data f from the client, the server will compute the distance between f and all the points of the model (i.e $d_i = ||f - w_i||$). Then, to find the k-nearest neighbors of f, it has to select the k smallest labels corresponding to the k smallest distances and return the most frequent one by a majority vote.

To enable privacy-preserving k-NN classification, we must adapt this pipeline to work with encrypted data, specifically developing methods to compute distances and select the k smallest labels while operating on an encrypted query vector f. Regarding the distance computation, [19] adapted the method of [39] to compute the *squared distance* between an encrypted vectors and a plaintext vector using the homomorphic properties of the TFHE cryptosystem. This method is detailed in Section 5.2. Once each distances are computed, to select the k labels associated to the k smallest distances, we use the Blind Counting Sort algorithm to implement a top-k selection. This is detailed in Section 5.3.

Threat model. In this work, we are placing ourselves in a scenario where a client wants to perform a k-NN classification on the cloud. Following standard assumptions in Machine Learning as a Service (MLaaS), the server is considered honest-but-curious, meaning that it does not deviate from the protocol although it may try to infer information about the client's data. Moreover, as other works [19, 39], the server deleguate the majority vote at the end of the top-k selection to the client. Hence, the client learns more information about the server's model than in a classical setting. One can argue that if a malicious clients wants to infer information about the server's model, it would be better to perform the majority vote on the server side. A simple way to do it, is to homomorphically count the frequency of each label in the top-k selection, as done in the first step of Algortihm 3 and then perform an homomorphic argmax on this frequency array.

5.2 Distances computation using TFHE

Before the computation of the distances, the client's feature vector $f = (f_0, f_1, \ldots, f_{\gamma-1})$ must be encoded and encrypted in a particular way to enable the server to compute the squared distances. Indeed, as explained in [19], the squared distance between two vectors f and m is given by

$$d_i = ||f - m||^2 = ||f||^2 - 2\langle f, m\rangle + ||m||^2$$

This gives a sort of "symmetric" formula where the left term $||f||^2$ is owned by the client and the right term $||m||^2$ is owned by the server. Thus each party can precompute their part of the formula independently. The challenge lies in computing the middle term of the formula, $2\langle f, m\rangle$. This can be done by using a polynomial multiplication as shown in [19]. More formally, if we set

$$F(X) = \sum_{i=0}^{\gamma-1} f_i \cdot X^i \text{ and } M(X) = \sum_{i=0}^{\gamma-1} m_{\gamma-i-1} \cdot X^i$$

The $\gamma-1$ coefficient of the polynomial product $F(X)\cdot M(X)$ is exactly $\langle f, m\rangle$ (a more detailed proof is given in the appendix of [39]). To support that in the encrypted domain, the client produces $c = [\![F(X)]\!]_{\text{RLWE}}$ and sends it to the server. Then, the server performs an Absorption between c and $M(X)$, and SampleExtract the $\gamma - 1$ coefficient of the resulting RLWE ciphertext to get $[\![\langle f, m\rangle]\!]_{\text{LWE}}$. The server can then compute the distances by adding the three terms of the squared distance formula :

$$[\![d_i]\!]_{\text{LWE}} = [\![||f||^2]\!]_{\text{LWE}} - 2[\![\langle f, m\rangle]\!]_{\text{LWE}} + [\![||m||^2]\!]_{\text{LWE}}$$

This simple method to compute the distance is extremly efficient, taking less than 1% of the total computation time of the k-NN algorithm. However, for certain datasets where γ is large, to avoid a noise explosion we need either to increase the plaintext modulus p or to use the method explained in [19] (Section 4.3) to reduce the precision homomorphically. This precision reduction increases the running time of the distance computation as it needs more bootstrapping operations but has the advantage of allowing keeping the plaintext modulus p as lower as possible to reduce the computational costs of the sorting algorithm.

5.3 Selecting the k-Nearest Neighbors

Once the distances are computed, we can use the BCS algorithm as a subroutine to implement a top-k selection as detailed in Section 5.3. The difficulty of this step is that we don't want to send the k smallest distances to the client, but rather the labels associated to them. However, the top-k selection algorithm returns the k smallest distances, not the labels. To address this issue, we can see the sorting process as a permutation of elements and tweak the BCS algorithm to mirror this permutation onto the corresponding labels. By doing so, at the end of the tournament, we obtain the top-k distances along their associated labels. In terms of complexity, this method adds p BR and p PFKS to the original BCS algorithm. The detailed algorithm of this key-value BCS is described in the Appendix B.

6 EXPERIMENTS

In this section, we present the experimental results of our Blind Sort algorithm and our private kNN selection based on it. All experiments are performed on a computer running Ubuntu 24.04 with an Intel i9-11900KF CPU clocked at 3.5GHz and 64GB of RAM.

6.1 Sort algorithm

Here are the execution times for our proposed sorting algorithm. For our algorithm, we denote p the plaintext modulus and the array size.

We compare first with numbers taken from [23] where they implemented in BGV a batched version of [8]'s Direct Sort using their improved comparison operator. They are sorting 9352 arrays simultaneously, so the total column records the wall time their algorithm takes, and the amortized column tracks the time per array. Contrarily to the other two, their method sorts encrypted 8-bits integers regardless of p.

The numbers for [19] are obtained by running their implementation of Batcher's odd-even merge sorting network (not truncated) on the same hardware as ours. Due to their comparison operator costing a precision bit, to sort p integers modulo p they actually need to work on a plaintext modulus of $2p$, which makes the basic Blind Rotate and PFKS operations more expensive comparatively.

Table 3: Computation times in seconds for sorting p elements in $\mathbb{Z}_p$. The numbers prefixed with ∼ are extrapolated.

	Iliashenko [23]		Cong [19]	BCS
p	total	amortized		
4	186.28	0.02	∼0.1	0.1
8	867.46	0.09	0.95	0.32
16	3652.23	0.39	8.65	0.83
32	14769.23	1.579	77.96	3.79
64	60351.02	6.453	833.79	17.76
128	∼246232	∼26	∼8913	125.5

6.2 Private k-Nearest Neighbors inference

We compare the execution times of our secure k-NN algorithm with the ones of [39] and [19]. They have both been applied to the Breast Cancer dataset [36] and the MNIST dataset [2].

Table 4: The TFHE parameters used in our experiments for the k-NN classification. The notation used are the one in TFHE's original paper [15]

Parameter	Value
LWE dimension (n)	742
RLWE polynomial degree (N)	2048
LWE standard deviation (σ_{LWE})	2^{40}
RLWE standard deviation (σ_{RLWE})	2^{2}
Decomp params bootstrapping (g, ℓ)	$(2^{23}, 1)$
Decomp params KS (g, ℓ)	$(2^{3}, 5)$
Decomp params PFKS (g, ℓ)	$(2^{23}, 1)$
Ciphertext modulus (q)	2^{64}
Plaintext modulus (p)	2^{4}
Plaintext modulus MNIST	2^{5}
Plaintext modulus breast cancer	2^{4}
Dataset message space MNIST	$\mathbb{Z}_2$
Dataset message space breast cancer	$\mathbb{Z}_2$

Pre-processing and precision reduction. Both datasets are binarized as in [39] and [19]. Specifically, all the features below p are set to 0 and the features above p are set to 1. This is a current pre-processing step for k-NN classification as explained in [32]. For the breast cancer dataset, as $\gamma = 30$ is relatively low, the plaintext modulus match the message space, so we can compute the squared distances between the query and the model points without any precision reduction. However, for the MNIST dataset, even after we use the precision reduction technique mentioned at the end of the Section 5.2 to compute squared distances. This is because the MNIST dataset has larger feature values and dimensions, making the exact computation more expensive in terms of noise. In both cases, we achieve better running times than previous works while maintaining comparable accuracy levels.

Table 5: Computation time in seconds for the breast cancer dataset.

k	d	[39]	[19]		Ours	
			$\tau = 4$	$\tau = 1$	$\tau = 4$	$\tau = 1$
3	10	4	1.8	3.2	0.79	0.75
	30	∼18	5.0	11.5	1.87	2.77
	50	∼51	7.4	19.0	2.39	4.77
	200	∼830	25.5	76.0	7.55	19.29
5	10	∼2	2.2	4.2	0.77	0.76
	30	∼18	7.5	16.7	2.16	3.06
	50	∼52	11.6	28.8	3.41	5.75
	200	∼831	40.2	114.6	8.73	23.03

Table 6: Computation time in seconds for the MNIST dataset.

k	d	[39]	[19]		Ours	
			$\tau = 4$	$\tau = 1$	$\tau = 4$	$\tau = 1$
3	40	30	8.7	17.5	2.41	4.33
	175	696	31.9	78.1	6.85	19.03
	269	1524	47.4	119.5	10.66	29.31
	457	4248	78.9	202.3	17.20	49.44
	1000	∼20837	168.0	441.1	34.81	109.23
5	40	∼33	11.6	25.5	2.72	4.92
	175	∼636	43.1	112.7	8.50	22.82
	269	∼1505	62.7	173.0	12.96	35.25
	457	∼4351	105.0	291.1	18.88	57.39
	1000	∼20859	227.5	642.3	39.37	125.29

7 CONCLUSION

In this paper, we introduced the first oblivious sorting algorithm that operates directly on encrypted data without requiring any comparisons between ciphertexts. By leveraging this novel sorting approach, we developed an efficient top-k algorithm and demonstrated its effectiveness through a k-nearest neighbors implementation that significantly outperforms the state-of-the-art. The adaptation of the counting sort algorithm to the encrypted domain was made possible through the RevoLUT library and its powerful LUT read and write operations. The key contribution of our work lies in eliminating the need for ciphertext comparisons, which removes the requirement for additional precision bits in the representation. This allows us to work with exactly the same precision as the input data, leading to more efficient computations while maintaining the same level of accuracy. Our experimental results on both the breast cancer and MNIST datasets demonstrate substantial performance improvements, with speedups of up to 5x compared to previous approaches.

ACKNOWLEDGMENTS

TODO

REFERENCES

[1] 2024. RevoLUT: Rust efficient versatile library for oblivious Look-Up Tables. https://github.com/sofianeazogagh/revoLUT.
[2] Alpaydin and Kaynak. 1998. Optical Recognition of Handwritten Digits. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50P49.

[3] Yulliwas Ameur, Rezak Aziz, Vincent Audigier, and Samia Bouzefrane. 2022. Secure and Non-interactive k-NN Classifier Using Symmetric Fully Homomorphic Encryption. In *Privacy in Statistical Databases: International Conference, PSD 2022, Paris, France, September 21–23, 2022, Proceedings* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 142–154. https://doi.org/10.1007/978-3-031-13945-1_11

[4] Sofiane Azogagh, Victor Delfour, Sébastien Gambs, and Marc-Olivier Killijian. 2022. PROBONITE: PRivate One-Branch-Only Non-Interactive decision Tree Evaluation. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 23–33. https://doi.org/10.1145/3560827.3563377

[5] Sofiane Azogagh, Victor Delfour, and Marc-Olivier Killijian. 2024. Oblivious Turing Machine. In *2024 19th European Dependable Computing Conference (EDCC)*. IEEE, 17–24.

[6] Kenneth E Batcher. 1968. Sorting networks and their applications. In *1968 AFIPS Spring Joint Computer Conference*. IEEE, 307–314.

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Leveled fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[8] Gizem S Çetin, Yarkın Doröz, Berk Sunar, and Erkay Savaş. 2015. Depth optimized efficient homomorphic sorting. In *Progress in Cryptology–LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings 4*. Springer, 61–80.

[9] Gizem S Çetin, Yarkın Doröz, Berk Sunar, and Erkay Savaş. 2015. Low depth circuits for efficient homomorphic sorting. *Cryptology ePrint Archive* (2015).

[10] Gizem S Cetin, Erkay Savaş, and Berk Sunar. 2020. Homomorphic sorting with better scalability. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2020), 760–771.

[11] Olive Chakraborty and Martin Zuber. 2022. Efficient and accurate homomorphic comparisons. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 35–46.

[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. *International Conference on the Theory and Application of Cryptology and Information Security* (2017), 409–437.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. 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, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. *IACR Cryptol. ePrint Arch.* (2018), 421. https://eprint.iacr.org/2018/421

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[17] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*.

[18] Antoine Choffrut, Rachid Guerraoui, Rafael Pinot, Renaud Sirdey, John Stephan, and Martin Zuber. 2023. Practical Homomorphic Aggregation for Byzantine ML. *arXiv preprint arXiv:2309.05395* (2023).

[19] Kelong Cong, Robin Geelen, Jiayi Kang, and Jeongeun Park. 2023. Revisiting Oblivious Top-k Selection with Applications to Secure k-NN Classification. Cryptology ePrint Archive, Paper 2023/852. https://eprint.iacr.org/2023/852

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[21] C. A. R. Hoare. 1962. Quicksort. *Computer Journal* 5, 1 (1962), 10–15.

[22] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. 2021. Efficient Sorting of Homomorphic Encrypted Data With k-Way Sorting Network. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4389–4404. https://doi.org/10.1109/TIFS.2021.3106167

[23] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies* 2021, 3 (2021), 246–264.

[24] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[25] Neeta B Malvi and N Shylashree. 2024. Innovative Homomorphic Sorting of Environmental Data in Area Monitoring Wireless Sensor Networks. *IEEE Access* (2024).

[26] Jeongsu Park and Dong Hoon Lee. 2018. Privacy Preserving k-Nearest Neighbor for Medical Diagnosis in e-Health Cloud. *Journal of healthcare engineering* 2018, 1 (2018), 4073103.

[27] Raluca Ada Popa, Catherine M S Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 85–100.

[28] Yinian Qi and Mikhail J Atallah. 2008. Efficient privacy-preserving k-nearest neighbor search. In *2008 The 28th International Conference on Distributed Computing Systems*. IEEE, 311–319.

[29] Ronald L Rivest, Burton S Kaliski, and Yair Yacobi. 1978. Data encryption standard. *Advances in Cryptology* (1978).

[30] Hong Rong, Hui-Mei Wang, Jian Liu, and Ming Xian. 2016. Privacy-Preserving k-Nearest Neighbor Computation in Multiple Cloud Environments. *IEEE Access* 4 (2016), 9589–9603. https://doi.org/10.1109/ACCESS.2016.2633544

[31] Harold Herbert Seward. 1954. *Information sorting in the application of electronic digital computers to business operations*. Ph. D. Dissertation. Massachusetts Institute of Technology. Department of Electrical Engineering.

[32] David Bingham Skalak. 1997. *Prototype selection for composite nearest neighbor classifiers*. University of Massachusetts Amherst.

[33] John Von Neumann. 1960. Design and analysis of computer algorithms. *IBM Journal of Research and Development* 4, 1 (1960), 43–63.

[34] Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. 2010. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (Beijing, China) *(ASIACCS '10)*. Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/1755688.1755716

[35] J. W. J. Williams. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (1964), 347–348.

[36] William Wolberg, Olvi Mangasarian, Nick Street, and W. Street. 1993. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B.

[37] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs.

[38] Yandong Zheng, Rongxing Lu, Songnian Zhang, Jun Shao, and Hui Zhu. 2024. Achieving Practical and Privacy-Preserving kNN Query Over Encrypted Data. *IEEE Transactions on Dependable and Secure Computing* 21, 6 (2024), 5479–5492. https://doi.org/10.1109/TDSC.2024.3376084

[39] Martin Zuber and Renaud Sirdey. 2021. Efficient homomorphic evaluation of k-NN classifiers. *Proceedings on Privacy Enhancing Technologies* (2021).

A PREFIX BLIND COUNTING SORT

We decribe here a way to tweak the Blind Counting Sort algorithm to sort only the first k elements of a given LUT. This prefixed version of the Blind Counting Sort algorithm is essentially the same, except the first and last loops are truncated. This version requires approximately $4k \cdot t_{BR} + (k+p) \cdot t_{KS}$ time.

Algorithm 4: Prefix Blind Counting Sort (PBCS)

```
Input  : A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}}$
         A prefix length $k$
Output : A LUT whose $k$ first elements are sorted
1  $[\![count]\!]_{\mathrm{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\mathrm{LUT}}$
2  for $i \leftarrow 0$ to $k-1$ do
       // $count_{m_i} \leftarrow count_{m_i} + 1$
3      $[\![m_i]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}})$
4      $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([\![m_i]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [1]_{\mathrm{LWE}})$
5  end
6  for $i \leftarrow 1$ to $p-1$ do
       // $count_i \leftarrow count_i + count_{i-1}$
7      $[\![x]\!]_{\mathrm{LWE}} \leftarrow BAA([i-1]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}})$
8      $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([i]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [\![x]\!]_{\mathrm{LWE}})$
9  end
10 $[\![res]\!]_{\mathrm{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\mathrm{LUT}}$
11 for $i \leftarrow k-1$ to $0$ do
       // $count_{m_i} \leftarrow count_{m_i} - 1$
12     $[\![m_i]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}})$
13     $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([\![m_i]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [-1]_{\mathrm{LWE}})$
       // $res_{count_{m_i}} \leftarrow m_i$
14     $[\![m_i]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\mathrm{LUT}})$
15     $[\![count_{m_i}]\!]_{\mathrm{LWE}} \leftarrow BAA([\![m_i]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}})$
16     $[\![res]\!]_{\mathrm{LUT}} \leftarrow BAI([\![count_{m_i}]\!]_{\mathrm{LWE}}, [\![res]\!]_{\mathrm{LUT}}, [\![m_i]\!]_{\mathrm{LWE}})$
17 end
18 return $[\![res]\!]_{\mathrm{LUT}}$
```

B KEY-VALUE BLIND COUNTING SORT

We decribe here a way to tweak the Blind Counting Sort algorithm to sort any array of tuples by their first element. More precisely, it can be used to sort a set of ℓ vectors of size p by the first element of each vector. This tensorized version of the Blind Counting Sort algorithm is basically the same as the original one, except in the last loop where we need to apply the permutation on all the other elements of the tuples represented as LUT ciphertexts. This version requires $(4+\ell)p \cdot t_{BR} + (2+\ell)p \cdot t_{KS}$.

Algorithm 5: Key-value Blind Counting Sort

```
Input  : A vector of $\ell$ LUT ciphertexts
         $([\![m_0^0, \ldots, m_{p-1}^0]\!]_{\mathrm{LUT}}, \ldots, [\![m_0^{\ell-1}, \ldots, m_{p-1}^{\ell-1}]\!]_{\mathrm{LUT}})$
Output : A vector of $\ell$ LUT ciphertexts sorted by the first one
         $([\![m_{\pi(0)}^0, \ldots, m_{\pi(p-1)}^0]\!]_{\mathrm{LUT}}, \ldots, [\![m_{\pi(0)}^{\ell-1}, \ldots, m_{\pi(p-1)}^{\ell-1}]\!]_{\mathrm{LUT}})$
1  $[\![count]\!]_{\mathrm{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\mathrm{LUT}}$
2  for $i \leftarrow 0$ to $p-1$ do
       // $count_{m_i} \leftarrow count_{m_i} + 1$
3      $[\![m_i^0]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0^0, \ldots, m_{p-1}^0]\!]_{\mathrm{LUT}})$
4      $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([\![m_i^0]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [1]_{\mathrm{LWE}})$
5  end
6  for $i \leftarrow 1$ to $p-1$ do
       // $count_i \leftarrow count_i + count_{i-1}$
7      $[\![x]\!]_{\mathrm{LWE}} \leftarrow BAA([i-1]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}})$
8      $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([i]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [\![x]\!]_{\mathrm{LWE}})$
9  end
10 for $j \leftarrow 0$ to $\ell-1$ do
11     $[\![res^j]\!]_{\mathrm{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\mathrm{LUT}}$
12 end
13 for $i \leftarrow p-1$ to $0$ do
       // $count_{m_i} \leftarrow count_{m_i} - 1$
14     $[\![m_i^0]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0^0, \ldots, m_{p-1}^0]\!]_{\mathrm{LUT}})$
15     $[\![count]\!]_{\mathrm{LUT}} \leftarrow BAI([\![m_i^0]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}}, [-1]_{\mathrm{LWE}})$
       // $res_{count_{m_i}} \leftarrow m_i$
16     $[\![count_{m_i^0}]\!]_{\mathrm{LWE}} \leftarrow BAA([\![m_i^0]\!]_{\mathrm{LWE}}, [\![count]\!]_{\mathrm{LUT}})$
17     for $j \leftarrow 0$ to $\ell-1$ do
18         $[\![m_i^j]\!]_{\mathrm{LWE}} \leftarrow BAA([i]_{\mathrm{LWE}}, [\![m_0^j, \ldots, m_{p-1}^j]\!]_{\mathrm{LUT}})$
19         $[\![res^j]\!]_{\mathrm{LUT}} \leftarrow$
             $BAI([\![count_{m_i^0}]\!]_{\mathrm{LWE}}, [\![res^j]\!]_{\mathrm{LUT}}, [\![m_i^j]\!]_{\mathrm{LWE}})$
20     end
21 end
22 return $([\![res^j]\!]_{\mathrm{LUT}})_{j=0}^{\ell}$
```