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

In this paper, we introduce an adaptation of the counting sort algorithm that leverages the data obliviousness of the algorithm to enable the 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 takes advantage of some basic operations on TFHE's Look-Up-Tables (LUT). We have integrated these operations into RevoLUT [3], a comprehensive open-source library built on tfhe-rs [37]. 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 proves to be approximately 4 times faster than 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 [30], offers a groundbreaking approach to performing computations on encrypted data without decryption. This capability is particularly valuable in scenarios where sensitive data, such as personal health records or financial information, need to be processed by third-party services while maintaining their confidentiality. Sorting algorithms, such as QuickSort [23], MergeSort [34], and HeapSort [35], are fundamental building blocks 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. Henceforth, most of the traditional sorting methods are not directly applicable to encrypted data, requiring the development of specialized algorithms that can efficiently sort encrypted data. Recent advances in fully homomorphic encryption schemes have opened the door to practical applications where sorting on encrypted data is feasible. FHE schemes, such as TFHE [16], BGV/BFV [7, 21], and CKKS [13], enable the execution of arbitrary functions on ciphertexts, including comparisons, which are essential operations in sorting algorithms. However, comparisons remain among the most expansive operations in FHE.

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 critical for both secure model training and prediction, where data must be organized or indexed without revealing sensitive information. A famous example is the selection of the k-nearest neighbors (k-NN) from encrypted data [2, 20, 39] that we also use as an illustration of the effectiveness of our solution. There are also other innovative applications of sorting encrypted data in federated learning such as aggregating encrypted gradients [19]. 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 sorting to support queries that involve ordering or range searches. Efficient encrypted sorting enhances the functionality of encrypted databases, enabling more complex queries while ensuring data confidentiality [28].

Our contribution. This work introduces a novel approach to sorting encrypted data by harnessing the oblivious properties of TFHE's Look-Up-Tables (LUTs). Indeed, by treating LUTs as an array data structure, we can leverage certain operations in TFHE that provide efficient blind read and write capabilities. This allows us to adapt the so-called counting sort algorithm for encrypted data. This represents, to the best of our knowledge, the first comparison-free sorting algorithm for encrypted data, eliminating the need for costly comparison operations that traditionally require extra precision bits in TFHE. We showcase the practical benefits of our method by incorporating it into a tournament style top-k selection algorithm, which we then employ to build an efficient private k-NN classifier that outperforms the current state of the art.

Outline. 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, and how it 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 an analysis of both time complexity 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.

Proceedings on Privacy Enhancing Technologies 2025(3), 156–169

https://doi.org/10.56553/popets-2025-0093

2 Preliminaries

In this section, after introducing the notation used in the paper, we present the necessary background on TFHE's cryptosystem.

2.1 Notation

Let p be a power of 2. We denote by $\mathbb{Z}_p$ the space of messages and by $[\![m]\!]$ the TFHE encryption of a message $m \in \mathbb{Z}_p$. We also make use of the Kronecker delta function $\delta_{i,j}$, which 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 $\delta_i = (\delta_{i,0}, \ldots, \delta_{i,p-1}) \in \{0,1\}^p$, which contains a single 1 at index i and 0s elsewhere. Other notations are defined in the text whenever needed.

2.2 The TFHE Cryptosystem

The TFHE encryption scheme, proposed in 2016 [14, 15], 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 the TFHE cryptosystem, several types of ciphertexts are defined depending on the nature of the plaintext and the encryption method employed. We define here the different types of ciphertexts along with some notations we use in this paper.

LWE Ciphertexts. A message $m \in \mathbb{Z}_p$ can be encrypted as a LWE ciphertext $(\vec{a}, b)$ such that $b = \sum_{i=0}^{n-1} a_i \cdot s_i + \Delta m + e$ where $s = (s_0, \ldots, s_{n-1}) \in \mathbb{Z}_2^n$ is the secret key and e is sampled from a Gaussian distribution of standard deviation σ_{LWE}.

RLWE Ciphertexts. A set of message $(m_0, \ldots, m_{N-1}) \in \mathbb{Z}_p^N$ can be seen as a polynomial message $M(X)$ and can be encrypted as an RLWE ciphertext $(A(X), B(X))$ such that $B(X) = A(X) \cdot S(X) + \Delta M(X) + E(X)$ where the coefficients of E are sampled from a Gaussian distribution of standard deviation σ_{RLWE}.

LUT Ciphertexts. Additionally, [5] introduced Look-Up-Table (LUT) ciphertexts, which are essentially RLWE ciphertexts with a redundancy of $\frac{N}{p}$ for each coefficient of the polynomial $M(X)$ as shown in Figure 1. We explain later, in Section 2.2.3, why this redundancy is important.

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 $p = 8$).

The Δ term, which we call *encoding factor*, is used to encode the messages in the most significant bits of the ciphertext. The ciphertext modulus q is usually instantiated as $q = 2^{64}$ and $\Delta = \frac{q}{2p}$. This allow a bit of padding that serves to manage a well known problem of negacyclity in TFHE. The vector $\vec{a}$ in the case of LWE and $A(X)$ in the case of RLWE are commonly called the *mask*. The terms b in the case of LWE and $B(X)$ in the case of RLWE are the *body*. 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 homomorphic encryption schemes based on the LWE problem, the basic operations that can be performed on ciphertexts are as follows. Note that GLWE (General LWE) indicates that the operation applies to both LWE and RLWE ciphertexts:

- **Addition:** $([\![\star]\!]_{\text{GLWE}}, [\![\star]\!]_{\text{GLWE}}) \rightarrow [\![\star]\!]_{\text{GLWE}}$. Given two 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:** $(\star, [\![\star]\!]_{\text{GLWE}}) \rightarrow [\![\star]\!]_{\text{GLWE}}$. 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):** $([\![\star]\!]_{\text{LWE}}, [\![\star]\!]_{\text{LUT}}) \rightarrow [\![\star]\!]_{\text{RLWE}}$. This operation is used to privately rotate the polynomial $M(X)$ (encrypted as an RLWE ciphertext) by $[\![i]\!]_{\text{LWE}}$ coefficients.
- **Sample Extraction (SE):** $(\star, [\![\star]\!]_{\text{RLWE}}) \rightarrow [\![\star]\!]_{\text{LWE}}$.
  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):** $[\![\star]\!]_{\text{LWE}} \rightarrow [\![\star]\!]_{\text{LWE}}$.
  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.
- **Pub. Functional Key Switch (PFKS):** $\{[\![\star]\!]_{\text{LWE}}\} \rightarrow [\![\star]\!]_{\text{RLWE}}$.
  Introduced in [17] (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.

In our implementation, each blind rotation operation is preceded by a key switch. We denote t_{BR} as the combined execution time of these two operations, and $\mathcal{E}_{BR}$ as their cumulative impact on the ciphertext noise variance. Similarly, we use t_{PFKS} and $\mathcal{E}_{PFKS}$ to represent the execution time and noise variance impact of the PFKS operation. We omit the Sample Extraction operation from this analysis as it has negligible execution time and introduces no additional noise. These notations are summarized in Table 1.

Table 1: Notation for the time and noise impact of TFHE operations used in this paper

Operation	Time	Variance
KS + BR	t_{BR}	$\mathcal{E}_{BR}$
PFKS	t_{PFKS}	$\mathcal{E}_{PFKS}$

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 [18] 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 select the correct message m_i as long as the noise e is smaller than the redundancy.

2.3 Counting Sort and porting challenges

Counting sort is an interesting and well-known sorting algorithm, historically attributed to [32], that, unlike many other classical sorting algorithms, is not based on the use of comparisons. As such, the $\Omega(n \log n)$ lower bound on time complexity of comparison-based sorting does not apply to it [26]. 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 in our case $n = k = p$.

To highlight the challenge of porting the counting sort algorithm to the encrypted domain, we first present it in its classical form (Algorithm 1). 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 element of the input array were distinct, these would in turn be the correct indices, starting at 1, 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 algorithms's stability. The stability property refers to the ability of the sorting algorithm to maintain the original input order in the output whenever some of the ordered elements are equal.

It is worth noting that, like sorting networks, counting sort is, in essence, data oblivious. That means that it may seem FHE friendly, in the sense that porting the algorithm does not necessarily require to adapt its control flow. However, keeping its $O(n + k)$ time complexity in an oblivious implementation in the encrypted domain is a challenge. Indeed, given an array of encrypted values $m_0, \ldots, m_{p-1}$, the naive implementation of fetching the element m_i at the encrypted index i, or adding the encrypted value x to the encrypted value m_i at encrypted index i is inefficient (i.e. $O(n)$ instead of $O(1)$).

For the first operation (which we'll call blind array access), that would be computing $m \cdot \delta_i = \sum_{j=0}^{p-1} m_j \delta_{i,j}$, which relies on computing p comparisons (one for each $\delta_{i,j}$) and p multiplications/additions,

Algorithm 1: Counting Sort

Input : An array $[m_0, \ldots, m_{p-1}]$ of length p
Output : The sorted array

$C \leftarrow [0, \ldots, 0]$
// Build the count array
for $i \leftarrow 0$ **to** $p - 1$ **do**
  $C_{m_i} \leftarrow C_{m_i} + 1$
end
// Compute the running sum
for $i \leftarrow 1$ **to** $p - 1$ **do**
  $C_i \leftarrow C_i + C_{i-1}$
end
$R \leftarrow [0, \ldots, 0]$
// Reconstruct the sorted array
for $i \leftarrow p - 1$ **to** 0 **do**
  $C_{m_i} \leftarrow C_{m_i} - 1$
  $R_{C_{m_i}} \leftarrow m_i$
end
return R

leading to $O(p)$ FHE operations. Therefore, the total complexity of the sort becomes $O(p^2)$. As for the blind array add operation, it would require adding the value $x\delta_{i,j}$ into every m_j, once again requiring p comparisons and p multiplications/additions. Instead, in the remainder of this paper, we will show how both these operations can be implemented using a single Blind Rotation, assuming the encrypted messages are packed into a LUT ciphertext.

3 Related Work

Oblivious sorting. The problem of sorting encrypted data without decrypting it has been widely studied for its applications to cloud computing security and private databases. [12] investigated the effectiveness of partition-based sorting like quicksort and observed that, for FHE encrypted data, it performed worse than the much simpler bubble sort algorithm. This has been attributed to the fact that both end up making the same number of blind comparisons, but the quicksort port has more added complexity to become data oblivious, where bubble sort naturally is. To further reduce time, they proposed lazy sort, a mix of bubble sort and insertion sort that needed fewer recrypt operations.

In their review, [12] showed how porting traditional sorting algorithms to FHE affects their time complexities. The main takeaway is that simple $O(n^2)$ sorts like bubble sort or insertion sort perform essentially the same, the partition-based sorts like quicksort and mergesort, originally $O(n \log n)$, degrade when becoming data oblivious to $O(n^2)$, but the already data-oblivious sorting networks, like Batcher's odd-even merge sort [6] or the bitonic sort maintain their $O(n \log^2 n)$ time complexity.

Later, Cetin *et al.* [8–10] introduced the Direct and Greedy sort algorithms. Their method is designed to reduce the multiplicative depth of the algorithms, in order to control noise growth, and therefore optimize the wall time of otherwise quadratic algorithms by allowing selection of smaller parameter sizes and therefore perform

better. Building on this, [25] proposed a faster blind comparison algorithm, which further improves Direct Sort.

For their work on oblivious top-k selection, Cong *et al.* [20] implemented a truncated version of the batcher's odd-even merge sorting network for TFHE using the comparison operator from [39]. Even though their work applies specifically to the top-k problem, when setting $k = n$ their implementation is a good reference frame for Batcher's odd-even merge sort in TFHE.

Recently, [24] proposed an extension to the 2-way sorting network for a prime k, called the k-way sorting network. Their method reduces the depth in terms of the comparison operation from $O(log_2^2 n)$ to $O(klog_k^2 n)$, therefore improving performance for small k. Their method scales remarkly well with batching and parallel processing, but uses an approximate comparison function and is therefore inherently inexact.

The general trend we can notice from the state of the art is that algorithms that are naturally designed to be data oblivious tend to perform better in the encrypted domain. This idea is our motivation to explore in this paper another well known data oblivious sorting algorithm, the counting sort.

Table 2: Oblivious Sorting Algorithm comparison

ref	algorithm	complexity	scheme
[8]	Direct Sort	$O(n^2)$	LTV
[25]	Direct Sort	$O(n^2)$	BGV
[20]	Odd-Even Merge Sort	$O(n \log n)$	TFHE
[24]	k-way Merge Sort	$O(n \log n)$	CKKS
Ours	Counting Sort	$O(n)$	TFHE

Table 2 summarizes the complexity and cryptosystems used by both the state of the art and our contribution. Please note that implementations using the LTV, BGV and CKKS cryptosystems are batched versions of the algorithm and benefit from a SIMD approach. They therefore sort multiple arrays in better amortized time but have the disadvantage that they require the same long time if they only want to sort one array. All the schemes are exact (given good noise management) except CKKS, which is approximate by nature.

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 [2, 20, 27, 29, 31, 38, 39]. The related works closest to ours are those of [2, 20, 39] who proposed to leverage fully homomorphic encryption, and more precisely the 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 distance 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, [2] 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 use 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 [20] who unearthed an old sorting algorithm that is naturally data-oblivious and thus FHE-friendly. However, because TFHE comparators require additional padding bits, their algorithm performs worse in practice than it theoretically should.

4 Oblivious sorting algorithm

In this section, we first present read and write primitives developed in RevoLUT that we will use for building our sorting algorithm. RevoLUT [3], which stands for Rust Efficient Versatile Oblivious Look-Up-Tables, is a comprehensive open-source library built on tfhe-rs [37] for manipulating LUT ciphertexts. RevoLUT leverages LUTs as first class objects, enabling efficient oblivious operations such as array access or permutation directly within the table. We believe that RevoLUT can be of independent interest for the design of oblivious algorithms. Later in the section, we present our Blind Counting Sort algorithm, 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 in [4], Blind Array Access (BAA) is a first building block that enables access to an encrypted index in a LUT. This is achieved by using the Blind Rotate procedure, and then extracting the sample at index 0 from the resulting RLWE ciphertext.

Algorithm 2: Blind Array Access (BAA)

Input : An encrypted index $[\![i]\!]_{\text{LWE}}$
A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$
Output : A LWE ciphertext $[\![m_i]\!]_{\text{LWE}}$

1 $[\![rotated]\!]_{\text{LUT}} \leftarrow BR([\![i]\!]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
2 $[\![m_i]\!]_{\text{LWE}} \leftarrow SE(0, [\![rotated]\!]_{\text{LUT}})$
3 **return** $[\![m_i]\!]_{\text{LWE}}$

4.1.2 Blind Array Add (BAAdd). This second building block is some form of blind write operation in the LUT. For this we implemented Blind Array Add (BAAdd), 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 Add. This would double the blind rotation cost and was left aside since it is not required for our purpose, but could help for some other algorithms.

A caveat of this approach is that the $[\![x\delta_i]\!]_{\text{LUT}}$ 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.

The time cost of these operations follows directly from that of Blind Rotate (t_{BR}) and the Public Functional Key Switch (t_{PFKS}). In both cases, the cost is comparatively negligible if the provided argument is trivially encrypted. That is, the estimated time for

Algorithm 3: Blind Array Add (BAAdd)

Input : An encrypted index $[\![i]\!]_{\text{LWE}}$
A LUT ciphertext $[\![m]\!]_{\text{LUT}}$
An encrypted value $[\![x]\!]_{\text{LWE}}$
Output: A LUT ciphertext $[\![m + x\delta_i]\!]_{\text{LUT}}$

$[\![x\delta_0]\!]_{\text{LUT}} \leftarrow PFKS([\![x]\!]_{\text{LWE}})$
$[\![x\delta_i]\!]_{\text{LUT}} \leftarrow BR(-[\![i]\!]_{\text{LWE}}, [\![x\delta_0]\!]_{\text{LUT}})$
return $[\![m]\!]_{\text{LUT}} + [\![x\delta_i]\!]_{\text{LUT}}$

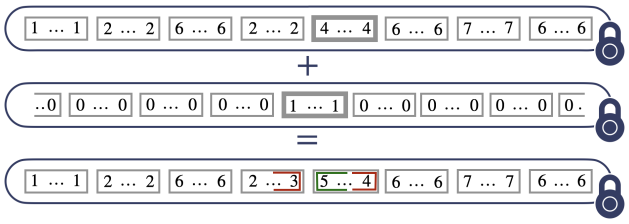

Fig. 2. Illustration of $BAAdd([\![4]\!]_{\text{LWE}}, [\![1, 2, 6, 2, 4, 6, 7, 6]\!]_{\text{LUT}}, [\![1]\!]_{\text{LWE}})$ with $p = 8$. The red areas at the boundaries of the redundancy boxes represent errors due to the noise in the LWE encryption of $[\![4]\!]_{\text{LWE}}$. If the noise in the LWE 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.

BAAdd (t_{BAAdd}) and BAA (t_{BAA}) will depend on if the rotation index is trivially encrypted ($[i]_{\text{LWE}}$) or not ($[\![i]\!]_{\text{LWE}}$), and, for BAAdd specifically, if the added value is trivially encrypted ($[x]_{\text{LWE}}$) or not ($[\![x]\!]_{\text{LWE}}$).

Table 3: Time approximations for Blind Array operations

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

4.2 Blind Counting Sort

Building on the previous blind read and write operations, we can now present the first blind sort algorithm whose number of blind rotations required (the most expensive basic operation in TFHE) scaling linearly with the size of the input array.

Algorithm. We propose Algorithm 4, porting the classical counting sort to operate on encrypted arrays represented as LUT ciphertexts. We can see that it closely follows its classical counterpart, Algorithm 1, except that the array operations are implemented using BAA (Algorithm 2) and BAAdd (Algorithm 3).

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

$$4p \cdot t_{BR} + 2p \cdot t_{PFKS}$$

Algorithm 4: Blind Counting Sort (BCS)

Input : A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$
Output: A sorted LUT

$[C]_{\text{LUT}} \leftarrow [0, \ldots, 0]_{\text{LUT}}$
for $i \leftarrow 0$ **to** $p - 1$ **do**
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  // $C_{m_i} \leftarrow C_{m_i} + 1$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [1]_{\text{LWE}})$
end
for $i \leftarrow 1$ **to** $p - 1$ **do**
  $[\![C_{i-1}]\!]_{\text{LWE}} \leftarrow BAA([i - 1]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  // $C_i \leftarrow C_i + C_{i-1}$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([i]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [\![C_{i-1}]\!]_{\text{LWE}})$
end
, $[R]_{\text{LUT}} \leftarrow [0, \ldots, 0]_{\text{LUT}}$
for $i \leftarrow p - 1$ **to** 0 **do**
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  // $C_{m_i} \leftarrow C_{m_i} - 1$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [-1]_{\text{LWE}})$
  // $R_{C_{m_i}} \leftarrow m_i$
  $[\![C_{m_i}]\!]_{\text{LWE}} \leftarrow BAA([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  $[\![R]\!]_{\text{LUT}} \leftarrow BAAdd([\![C_{m_i}]\!]_{\text{LWE}}, [\![R]\!]_{\text{LUT}}, [\![m_i]\!]_{\text{LWE}})$
end
return $[\![R]\!]_{\text{LUT}}$

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 LUTs 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 resulting LUT is the sum of many decentered LUTs, 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 freshly 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 $\sum_{i=0}^{p} i\mathcal{E}_{PFKS} = \frac{p(p-1)}{2}\mathcal{E}_{PFKS}$. In the third loop, the count LUT, gets added into p times in the same manner as in 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

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

The resulting 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

$$\left(\sum_{i=0}^{p} \mathcal{E}_{m_i}\right) + p\mathcal{E}_{PFKS} + p\mathcal{E}_{BR}$$

If we assume that all initial messages are encrypted with the same standard deviation, i.e., $\mathcal{E}_{m_i} = \sigma^2_{\text{LWE}}$ for every i, then the noise growth for the sorted LWE ciphertexts using BCS can be expressed as

$$p\left(\sigma^2_{\text{LWE}} + \mathcal{E}_{PFKS} + \mathcal{E}_{BR}\right) \tag{1}$$

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 noise growth. For p large enough, this will require bootstrapping some count values during the second loop in order to maintain the correctness of the algorithm.

4.3 Blind Top-k selection

The Blind Top-k problem asks, given a list of d encrypted elements, to return encryptions of the k smallest (or equivalently biggest) elements. This will be an important building block for our k-NN inference procedure, and so we will implement it using our proposed sorting algorithm as a sub-routine. For $k < p$, we can implement Blind Top-k selection in a tournament fashion, using BCS as a (k, p)-selector. An illustration of the blind Top-k selection is given in Figure 3 and the algorithm is given in Algorithm 5.

Algorithm 5: Blind Top-k

Input : A vector of d ciphertexts $(\llbracket m_i \rrbracket_{\text{LWE}})_{i=0}^{d-1}$.
A selection length k.
Output: A vector of k LWE ciphertexts corresponding to the k smallest elements of the input vector

$\eta \leftarrow \lceil \frac{d}{p} \rceil$ // Number of chunks
for $i \leftarrow 0$ **to** $\eta - 1$ **do**
  // Packing the elements into η LUTs ciphertexts
  $\llbracket D_i \rrbracket_{\text{LUT}} \leftarrow \text{PFKS}(\llbracket m_{ip+0} \rrbracket_{\text{LWE}}, \ldots, \llbracket m_{ip+p-1} \rrbracket_{\text{LWE}})$
  // Sorting the LUTs
  $\llbracket S_i \rrbracket_{\text{LUT}} \leftarrow \text{BCS}(\llbracket D_i \rrbracket_{\text{LUT}})$ ▷ or KV-BCS (see Section 5.3)
  // Selecting the k smallest elements
  for $j \leftarrow 0$ **to** $k - 1$ **do**
    $\llbracket R_{ik+j} \rrbracket_{\text{LWE}} \leftarrow \text{BAA}([j]_{\text{LWE}}, \llbracket S_i \rrbracket_{\text{LUT}})$
  end
end
if $\eta = 1$ **then**
  return $(\llbracket R_i \rrbracket_{\text{LWE}})_{i=0}^{k-1}$
end
return Blind Top-$k((\llbracket R_i \rrbracket_{\text{LWE}})_{i=0}^{k\eta-1}, k)$

Complexity. The base case for the method is, when given a list of up to p elements, a single call to BCS suffice, and then the first k elements of the sorted LUT can be extracted to give the answer. If more than p elements are provided, then a tournament round starts. The elements are first split into η chunks of up to p elements, which are being independantly sorted via BCS and the k first elements are extracted from each chunk. The remaining elements are passed to another tournament round, until less than p are left, at which point splitting is no longer required.

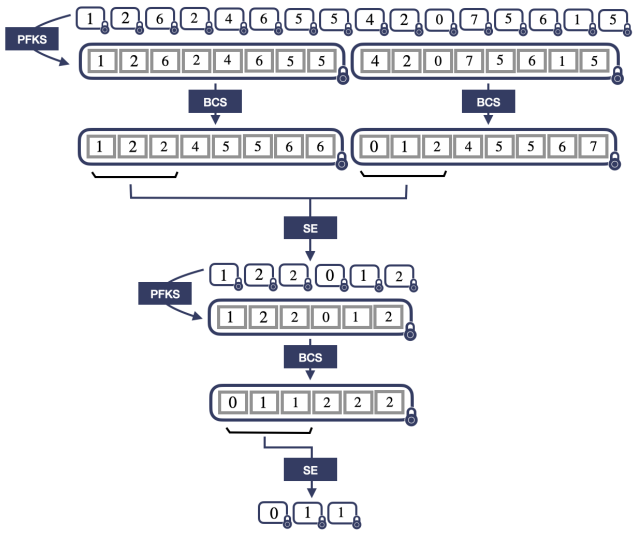

Fig. 3. Illustration of the Blind Top-k selection algorithm for $k = 3$ and $p = 8$. The input LWE ciphertexts are first split into chunks of size p. Each chunk is then packed into a LUT through a Public Functional Key Switch (PFKS) and processed by a Blind Counting Sort (BCS) to select its k smallest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

To express this more formally, we start with $u_0 = d$ elements. After completing the first round, the remaining number of elements u_1 is defined by

$$u_1 = k \cdot \left\lfloor \frac{d}{p} \right\rfloor + \min(k, \tau)$$

where $\tau = d \mod p$. This leads us to a recursive expression for the number of elements remaining after the i-th round:

$$u_{i+1} = k \cdot \left\lfloor \frac{u_i}{p} \right\rfloor + \min(k, \tau_i)$$

where $\tau_i = u_i \mod p$. Consequently, the total number of BCS calls is given by

$$U = \left(\sum_{i=0}^{r-1} \left\lfloor \frac{u_i}{p} \right\rfloor\right) + 1 \tag{2}$$

where r represents the number of rounds required until fewer than p elements remain (i.e. $u_r \leq p$).

Given that we can determine the number of BCS calls, we can also determine the number of Blind Rotations in Algorithm 5. As mentioned in Section 4.2, our BCS algorithm requires $4p$ BR operations. Therefore, focusing on the number of BR operations, we get:

$$N_{\text{BR}} = U \cdot 4p$$

Next, we also determine the number of PFKS, named N_{PFKS} in Algorithm 5. Since our BCS algorithm requires $4p$ PFKS operations and there are U calls to PFKS, we get:

$$N_{\text{PFKS}} = U \cdot (4p + 1)$$

Noise growth. If we assume that all initial messages are encrypted with the same standard deviation σ_{LWE}, then each round of the tournament will add the noise of a BCS and a PFKS to each elements. Therefore, the noise's variance of the k resulting ciphertexts for the blind Top-k is given by

$$p^r \left(\sigma^2_{\text{LWE}} + 2\mathcal{E}_{PFKS} + \mathcal{E}_{BR}\right)$$

where r is the number of rounds in the tournament.

Comparison with related work. Contrary to the sorting network of [20], our sorting algorithm does not require comparators. However, the common and most expensive operations in both algorithms are Blind Rotation (BR) and Public Functional Key Switch (PFKS), so here 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 requires no extra bit of precision relative to the size of the data. The comparator used in [20], which is based on the one in [11], does not evade this limitation. Therefore, 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 4, 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 [20] when processing data with the same precision (*i.e.* elements are from $\mathbb{Z}_{16}$).

Table 4: Running times (in ms) of Blind Rotation and PFKS in tfhe-rs [37] library for $p = 16$ and $p = 32$.

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

We can notice from the estimated running times in Figure 4 that our algorithm is very efficient for smaller value of k. For instance, for $k = 5$, our algorithm outperforms the one of [20] with $k = 3$ and $k = 5$ regardless of d.

Limitations. One drawback of our approach is that as k increases, the performance does not scale linearly - the running time grows more rapidly as k approaches p. Therefore, our approach isn't necessarily optimal when $k \in \Omega(\sqrt{d})$ contrarily to [20]. Another limitation arising from the core TFHE cryptosystem is the difficulty to process integers of more than 8 bits (i.e p has to be lower than 2^8). Moreover, because the precision and the size of the LUTs are linked in TFHE, we cannot natively process arrays larger than 256 either (even 256 wouldn't be practical because of the size of the keys). Note also that as p increases, the noise growth from the running sum of BCS leads to more errors in the count LUTs if we don't bootstrap the count values as explained in the end of Section 4.2. These errors may accumulate throughout the tournament and could result in an incorrect top-k result.

5 Private k-Nearest Neighbors classification

In this section, we present our study of applying the Blind Counting Sort algorithm to the private classification on k-Nearest Neighbors

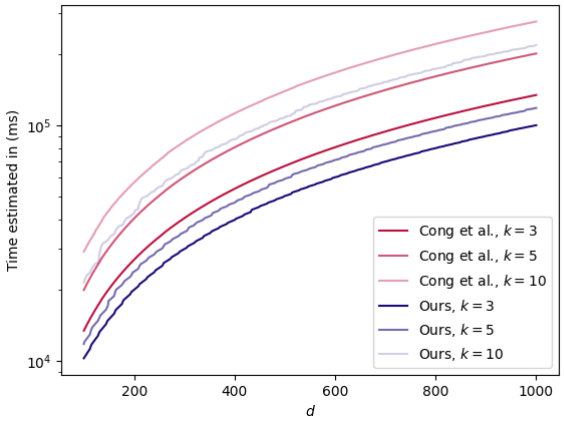

Fig. 4. Estimated running time of our blind Top-k selection algorithm compared to [20] when processing data with the same precision (*i.e.* elements are from $\mathbb{Z}_{16}$). Running times are calculated based on the number of PFKS and BR operations, with current performance metrics shown in Table 4.

use case using the top-k algorithm described in the previous section. We first describe the pipeline of the private k-Nearest Neighbors classification and then we detail each step of the pipeline. In this section, we denote γ the dimension of the feature vectors and d the number of points in the model.

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 $(m_1, \ldots, m_d)$ and the corresponding labels $(l_1, \ldots, l_d)$. 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 $d_i = ||f - m_i||$). Then, to find the k-nearest neighbors of f, it has to select the k labels corresponding to the k smallest distances and return the most frequent one selected 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, [20] 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 all the distances have been computed, in order to select the k labels associated to the k smallest distances, we use a tweaked versions of the Blind Counting Sort algorithm to implement a top-k selection on the encrypted distances and retrieve the k corresponding labels. This is detailed in Section 5.3.

Threat model. In this work, similarly to [2, 20, 39], we are placing ourselves in a scenario where a client wants to perform a k-NN classification in the cloud. Following standard assumptions in 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 in [2, 20, 39], the server deleguates 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 client 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 4 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 [20], 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 [20]. 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 [20] (Section 4.3) to reduce the precision homomorphically. In a nutshell, if the first computation made by the server is done using a plaintext modulus p_{large} (which is the case with the distance computation), the clients encrypts it data (from $\mathbb{Z}_p$) by using the encoding factor $\Delta_{\text{large}} = \frac{q}{p_{\text{large}}}$ instead of $\Delta = \frac{q}{2p}$. Then after the computation of the distances, when the server wants to reduce the plaintext modulus to p, it has to "recenter" the plaintext space by subtracting $\frac{\Delta_{\text{large}}\cdot(r-1)}{2}$ from each ciphertext. Here $r = \frac{p_{\text{large}}}{p}$ is the precision ratio between the two moduli. Finally, a bootstrapping operation is performed to reduce the noise and obtain a ciphertext in the smaller plaintext space. In our case, since the precision reduction technique is used for the distance computation only, we will use the notation p_{dist} to denote the large plaintext modulus used for the distance computation. This precision reduction increases the running time of the distance computation as it needs more bootstrapping operations (one for each distance) 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: Blind Sorting Key-Values

Once the distances are computed, we need to select the labels of the top-k elements. The idea is to use the BCS algorithm as a subroutine to implement a top-k selection from a $(k, p)-selector$ in a tournament-style fashion. The specificity of this step is that we do not need to send the k smallest distances to the client, but rather the labels associated with those. However, the top-k selection algorithm returns the k smallest distances, not the labels. To address this issue, we need to 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 and only return the labels to the client.

Time complexity. Sorting the first $\kappa \leq p$ elements (using the tweaks from Algorithm 7 in Appendix A) of λ LUT ciphertexts (using the tweaks from Algorithm 8 in Appendix B) takes time

$$(3 + \lambda)\kappa \cdot t_{BR} + (p + \lambda\kappa) \cdot t_{PFKS} \quad (3)$$

We verify that for $\lambda = 1$ and $\kappa = p$, this yields the original time complexity of BCS as presented in Section 4.2. For each round of our tournament method, κ will be equal to p for all buckets, except possibly the last, where it will be at most k if not null.

Table 5: Number of BR and PFKS for the blind top-k used in the private k-nn inference

k	d	[20]		Ours	
		BR	PFKS	BR	PFKS
3	40	186	372	190	148
	175	862	1724	995	668
	269	1332	2664	1565	1022
	457	2272	4544	2775	1794
	1000	4986	9972	6060	3846
5	40	250	500	210	156
	175	1196	2392	1215	810
	269	1856	3712	1860	1212
	457	3172	6344	3200	2054
	1000	6970	13940	7225	4582

The numbers for [20] are derived from their reported comparator count, multiplied by the 2 BR and 4 PFKS they use in their implementation. The numbers for our solution are computed by tournament simulation plugging in the time complexity formula (Equation 3) with $\lambda = 2$ and κ computed on the fly, accounting for the 2 PFKS before each call to BCS. Even though the number of BR is slightly lower for [20] for some values of k and d, recall that we use smaller parameters and that, henceforth, our Blind Rotations are much cheaper as shown in Table 4.

Algorithm 6: Private k-Nearest Neighbors (k-NN)

Input : Encrypted feature vector $[\![F]\!]_{\text{RLWE}}$ and its squared norm $[\![||f||^2]\!]_{\text{LWE}}$, the d model points $(m_1, \ldots, m_d) \in (\mathbb{Z}_p^\gamma)^d$ and their respective labels $(l_1, \ldots, l_d) \in \mathbb{Z}_p^d$, number of neighbors k

Output: The labels of the k nearest neighbors of $[\![F]\!]_{\text{RLWE}}$ encrypted as LWE ciphertexts

```
// Distance computation in Z_{p_dist}
1  for i ← 1 to d do
2      M_i(X) ← Σ_{j=0}^{γ-1} m_{i,j} · X^j
3      t ← [||m_i||^2]_LWE
4      [⟨f, m_i⟩]_LWE ← SE(γ − 1, Absorption([F]_RLWE, M_i))
5      [d_i]_LWE ← [||f||^2]_LWE − 2[⟨f, m_i⟩]_LWE + t
6  end
   // Precision reduction if p_dist > p
7  if p < p_dist then
8      for i ← 1 to d do
9          r ← p_dist / p
10         [d_i]_LWE ← [d_i]_LWE − [(Δ_dist · (r − 1)) / 2]_LWE
11         [d_i]_LWE ← Bootstrap([d_i]_LWE)
12     end
13 end
   // Top-k selection using the Key-Value BCS
14 (D, L) ← (([d_i]_LWE)_{i=1}^d, ([l_i]_LWE)_{i=1}^d)
15 Λ ← BlindTopk((D, L), k)                    ▷ With KV-BCS
16 return Λ
```

5.4 Noise and complexity analysis

Complexity. The complexity of our k-NN algorithm is essentially that of our Blind Top-k, as the distance computation is negligible and does not consume any Blind Rotations, except in the case $p_{\text{dist}} > p$ where a bootstrapping is required to reduce the precision of each distance. In this case, d Blind Rotations need to be added to N_{BR} to obtain the number of BR in the private k-NN described in Algorithm 6. Additionally, since the Key-Value BCS adds p Blind Rotations to the original BCS, its impact on the Blind Top-k leads to a total of $N_{\text{BR}}(1 + p)$ Blind Rotations. To sum up, the number of Blind Rotations in our private k-NN is

$$N_{\text{BR}} + pU + d$$

where U is given in Section 4.3 (Equation 2).

Noise growth. Our algorithm starts with inputs freshly encrypted by the client, so the variance of the noise are respectively σ^2_{RLWE} for the feature vector $[\![F]\!]_{\text{RLWE}}$ and σ^2_{LWE} for its squared norm $[\![||f||^2]\!]_{\text{LWE}}$. Then, the first operation that adds noise is the absorption of all M_i by $[\![F]\!]_{\text{RLWE}}$, and since this is a $\mathbb{Z}[X]$-linear operation, the noise variance of the results is

$$||M_i||_2^2 \cdot \sigma^2_{\text{RLWE}}$$

as stated in [16], Table 1. Then the next operation adding noise is the arithmetic operation in line 5 leading to the distances d_i with noise variance

$$\sigma^2_{\text{LWE}} + 4\left(||M_i||_2 \cdot \sigma_{\text{RLWE}}\right)^2$$

Then depending on the value of p_{dist}, we have two cases, either $p_{\text{dist}} > p$ or $p_{\text{dist}} = p$. Lets focus first on the case when $p_{\text{dist}} > p$. In this case, by design, the precision reduction step uses a bootstrap on every distance to recenter the plaintext space. This bootstrap refreshes the noise of the ciphered distances (replacing the variance of distances's noise by $\mathcal{E}_{BR}$). However, it is important to note that if the dimension of the data is large (i.e $||M_i||_2$ is large), the noise variance of the distances can be too large to be corrected by the bootstrap, thus leading to a wrong result. This phenomenon happens in one of the datasets used in our experiments (see Section 6). In the second case ($p_{\text{dist}} = p$), the precision reduction is not necessary so the noise variance is the same as before the precision reduction step. To summarize, at the line 13 of the algorithm, the distances's noise variance is given by

$$\mathcal{E}_{d_i} = \begin{cases} \mathcal{E}_{BR} & \text{if } p_{\text{dist}} > p \\ \sigma^2_{\text{LWE}} + 4\left(||M_i||_2 \cdot \sigma_{\text{RLWE}}\right)^2 & \text{if } p_{\text{dist}} = p \end{cases}$$

Finally, a Blind Top-k algorithm is performed on the distances's noise variance leading to a total noise variance of

$$p^r\left(\mathcal{E}_{d_i} + 2\mathcal{E}_{PFKS} + \mathcal{E}_{BR}\right)$$

where r is the number of rounds in the tournament.

6 Experiments

In this section, we present the experimental results of the Blind Sort algorithm and its associated private kNN selection. These experiments were performed on a computer running Ubuntu 24.04 with an Intel i9-11900KF CPU clocked at 3.5GHz and 64GB of RAM. The Blind Counting Sort algorithm and its prefix-based variant for blind Top-k selection are implemented and available in the open-source library RevoLUT [3]. For the k-NN, the code source is publicly available on GitHub[1].

6.1 Sort algorithm

The Table 6 presents the execution times for the Blind Sorting algorithm for various p denoting both the plaintext modulus and the array size. In this table, we compare the performances with numbers taken from both [25] and [20]. For [25], it is a BGV batched version of [8]'s Direct Sort using their improved comparison operator. It handles 9352 arrays simultaneously, so the total column records the wall time their algorithm takes, and the amortized column tracks the time per array. It is worth noting that, unlike with [20] and BCS, their method sorts encrypted 8-bits integers regardless of p. The performances of [20] 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 requiring an extra precision bit, sorting p integers modulo p actually requires to work on a plaintext modulus of $2p$, which makes Blind Rotations and PFKS more expensive comparatively.

Our algorithm performs much better than [25]'s total wall time, almost catching up to their amortized time. This means that many arrays (thousands) are needed for the batched method to be more effective. As for the comparison to [20], we notice that the gap scales with p, from merely 3 times faster to over 40 times faster.

[1] https://github.com/sofianeazogagh/knn

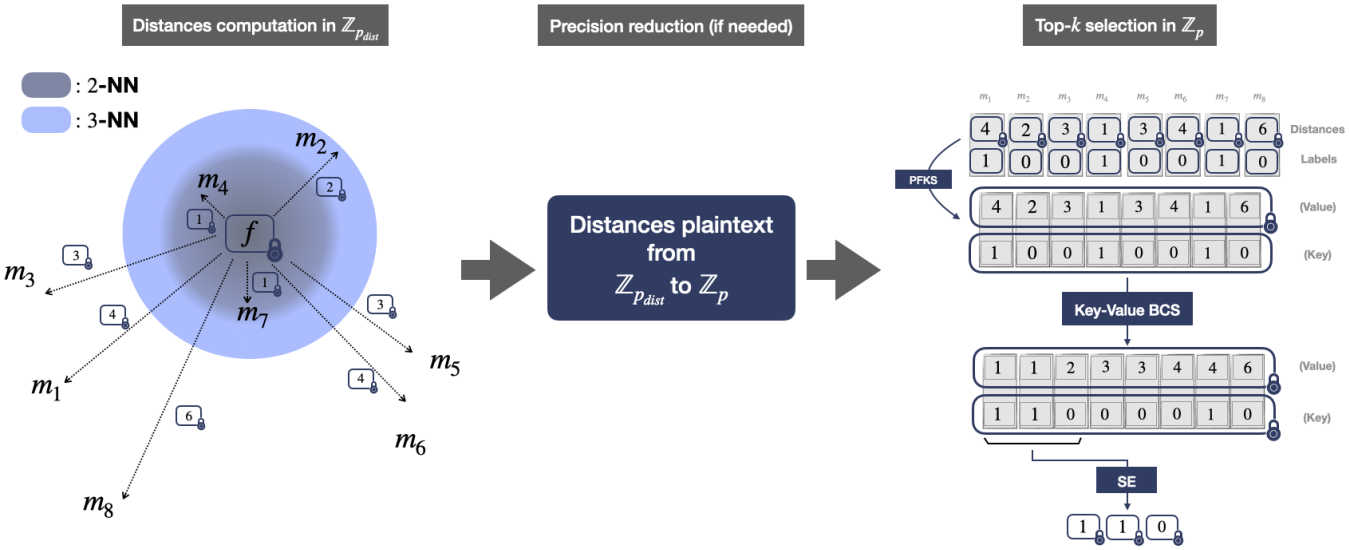

Fig. 5. The steps required to perform a private k-NN classification. The first step is the computation of the distances between the client's feature vector f and the server's model $(m_1, \ldots, m_d)$. The illustrated circle expands to encompass the first k points (i.e., the k nearest neighbors). For example, here, the slightly darker circle is for $k = 2$, while the lighter one is for $k = 3$. Depending on the dataset, an additional middle step is to reduce the precision of the distances. The second step is the selection of the k smallest distances and their associated labels.

Table 6: Computation times in seconds for sorting p elements in $\mathbb{Z}_p$. The numbers prefixed with ~ are extrapolated.

	[25]		[20]	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

In order to assess the performance of our solution for private k-Nearest Neighbors inference, we compare the execution times of both our secure k-NN algorithm and the ones of [39] and [20] applied to two datasets: Breast Cancer [36] and MNIST [1]. The breast cancer dataset features come from images of cell nuclei from a breast mass. These features capture details like size, shape, and texture of the nuclei, helping to tell apart cancerous from non-cancerous cells, so it is a binary classification problem. The MNIST dataset consists of 28x28 pixel images of handwritten digits. Each pixel can take integer values in the range 0 to 255, representing the grayscale intensity of the digit. The features of the MNIST dataset are the pixel values themselves, which are used to classify the digit into one of ten categories (0-9). For these experiments, we use the parameters presented in Table 7.

6.2.1 *Pre-processing and distances modulus.*

[2]https://docs.zama.ai/tfhe-rs/references/fine-grained-apis/shortint/parameters

Table 7: The parameters used in our experiments for the k-NN classification ensuring 128-bit of security[2]. The TFHE parameters notations used are the ones in TFHE's original paper [16].

	Parameter	Value
TFHE	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
k-NN	Dimension of breast cancer (γ)	30
	Dimension of MNIST (γ)	64
	Distances modulus breast cancer (p_{dist})	2^4
	Distances modulus MNIST (p_{dist})	2^5
	Dataset message space breast cancer	$\mathbb{Z}_2$
	Dataset message space MNIST	$\mathbb{Z}_2$
	Size of the dataset breast cancer	569
	Size of the dataset MNIST	1797

Pre-processing. Both datasets are binarized as in [20, 39]. Specifically, all the features below p are set to 0 and the features above p are set to 1. This is a usual pre-processing procedure for k-NN classification as explained in [33]. For the MNIST dataset, before binarizing, we reduced the dimension to get a 8x8 pixel image and then the value of each pixel (the grayscale value) is divided by 300 as it is done in [20, 39].

Distances modulus. For the breast cancer dataset, as $\gamma = 30$ is relatively low, we do not need to use the precision reduction techniques mentioned at the end of Section 5.2, thus $p_{\text{dist}} = p$. However, for the binarized MNIST dataset, as $\gamma = 64$ is higher, we needed to use this precision reduction technique. In our experiments, it was sufficient to compute the distances with $p_{\text{dist}} = 2^5$ and then reduce them to $p = 2^4$ for top-k selection. Recall that, in our notation, we do not include the usual padding bits in TFHE to manage negacyclicity as explained in Section 2.2.1.

6.2.2 Computation time. Hereafter, we detail the performances of the private k-NN classification for different values of k and d for both the cancer and MNIST datasets, in Tables 8 and 9 respectively. As shown in the tables, our approach significantly outperforms both [39] and [20]. This performance improvement is mainly due to our use of Blind Counting Sort which allows us to work with smaller TFHE parameters than comparison-based approaches. Indeed, while [20] needs to double the plaintext modulus to accommodate their comparison operator, our sorting algorithm operates directly on the input modulus, making basic operations like Blind Rotate and PFKS more efficient.

Table 8: Computation time in seconds for the breast cancer dataset. τ is the number of threads used for the parallelization. The numbers prefixed with ∼ are extrapolated by the authors of [20].

k	d	[39]	[20]		Ours	
			$\tau = 4$	$\tau = 1$	$\tau = 4$	$\tau = 1$
3	10	4	1.2	2.4	0.79	0.75
	30	∼18	3.7	8.2	1.87	2.77
	50	∼51	5.4	14.1	2.39	4.77
	200	∼830	18.7	56.4	7.55	19.29
5	10	∼2	1.6	2.9	0.77	0.76
	30	∼18	5.5	12.0	2.16	3.06
	50	∼52	8.3	20.6	3.41	5.75
	200	∼831	29.1	83.9	8.73	23.03

The reason some lines show a performance decrease when multi-threading versus single-threading in our implementation is that when $d \leq p$ our tournament-style algorithm does not split the input at all and we only suffer the overhead of multi-threading for no gain.

6.2.3 Accuracy. To evaluate the accuracy of our private k-NN classification, we implemented a simple version of the k-NN algorithm working in cleartext. Then, we split the dataset into two sets: one for selecting the d best points of the dataset, i.e. for training, and one for classifying the query point using the k nearest neighbors, i.e. for testing. In order to select the best d points of the dataset, we performed 100 trials and selected the points that yielded the best accuracy among these trials. We then compare the accuracy of our private k-NN classification to the clear case over 200 randomly selected queries. The accuracy results reported are averaged over the 200 clients queries.

For the breast cancer dataset, the accuracy of our private k-NN is in line with the clear case for all the values of k and d we experimented. This is explained by the fact that the noise in the encrypted labels after prediction is small enough to not change the result of the majority vote performed by the client compared to the clear case.

Table 9: Computation time in seconds for the MNIST dataset. τ is the number of threads used for the parallelization. The numbers prefixed with ∼ are extrapolated by the authors of [20].

k	d	[39]	[20]		Ours	
			$\tau = 4$	$\tau = 1$	$\tau = 4$	$\tau = 1$
3	40	30	6.2	12.9	2.41	4.33
	175	696	23.5	56.9	6.85	19.03
	269	1524	35.6	87.5	10.66	29.31
	457	4248	58.6	147.8	17.20	49.44
	1000	∼20837	124.0	323.4	34.81	109.23
5	40	∼33	8.4	18.3	2.72	4.92
	175	∼636	31.8	81.2	8.50	22.82
	269	∼1505	46.4	125.6	12.96	35.25
	457	∼4351	77.8	212.4	18.88	57.39
	1000	∼20859	164.5	464.2	39.37	125.29

For the MNIST dataset however, the accuracy is in most cases 1-2 percentage points below the clear case. The results are compiled in Table 10. Interestingly, one can notice that for $k = 3$ and $d = 40$, the accuracy of our private k-NN was higher than in the clear case. This can be explained by the fact that for some queries, the overflowing noise in the encrypted labels after prediction changes the result of the majority vote performed by the client compared to the clear case, leading to correct classification in these instances. This is especially true for small values of k as it is easier to tip the vote in those cases. Note that the accuracy can be improved by using other forms of quantization. For instance, [20] uses a ternarized version of the MNIST dataset that improves their accuracy over the binary one we used. Additionally, other quantization techniques, such as those based on the Diaconis-Freedman rules [22], could also be adopted to further improve the accuracy. The purpose of presenting the accuracy of our private k-NN is to demonstrate how much its use can degrade the accuracy of the algorithm.

Table 10: Accuracy (in % of correct classifications) of our private k-NN classifications over the MNIST dataset compared to the clear case.

		d				
k	Type	40	175	269	457	1000
3	Ours	71.33	85.67	92.33	89.67	93.67
	Clear	70.00	86.33	92.67	90.00	95.33
5	Ours	73.33	87.33	88.00	92.00	92.67
	Clear	75.67	84.67	88.33	94.67	94.00

Bandwidth. In the scenario we are considering in this paper, where the server owns the model and the client wishes to perform a k-NN classification, there is an offline phase during which the client transmits the necessary cryptographic materials to the server before

any classification takes place. In this phase, similar to [20], the client sends three keys: the bootstrapping key (BSK), the keyswitching key (KSK), and the public functional key switching key (PFKSK). And in the online phase, the client sends its query composed of one RLWE ciphertext corresponding to $[\![F(X)]\!]_{\text{RLWE}}$ and one LWE ciphertext corresponding to $[\![||f||^2]\!]_{\text{LWE}}$ (see Section 5.2 for more details). Then, after performing the prediction, the server replies with k LWE ciphertexts corresponding to the labels of the k nearest neighbors. Since we use smaller TFHE parameters than [20], our key sizes and ciphertext sizes are smaller as pointed out in Table 11. To give an idea of the total bandwidth for different instanciation of k, we provide a comparison in Table 12.

Table 11: Comparison with [20] of key sizes and ciphertext sizes.

	LWE	RLWE	BSK	KSK	PFKSK
[20]	6.7 KB	64 KB	106 MB	20.3 MB	32 MB
Ours	5.9 KB	32 KB	47.6 MB	7.5 MB	8 MB

Table 12: Bandwidth consumption during the online phase for different values of k. The offline phase bandwidth consumption is not depending on k.

Phase	k	[20]	Ours
Online Phase	3	90.8 KB	55.6 KB
	5	104.3 KB	67.4 KB
Offline Phase		158.3 MB	63.1 MB

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 4x compared to previous approaches.

Future work. We have presented BCS, an efficient and scalable oblivious sorting algorithm. However, our implementation is currently limited to work with p integers modulo p, for p a small power of two. The asymptotic gain of our linear time algorithm over the state of the art linearithmic time is only beneficial if the method can operate on larger arrays. The first step towards that would be to decouple the size of the array and element bit-width, as is typically done in the classical counting sort algorithm. Here are some research trails on how to tackle these limitations but those will be the object of future work. One way to handle larger array size could be using BCS as a comparison building block (or sub-sort routine) to build a bigger sorting network. As for the element size scaling, that would require decomposing each element into multiple smaller messages. Together, these ideas would pave the way for porting the radix sort algorithm by, for instance, representing 8-bit integers as pairs of blocks (or digits) of 4-bit messages, and sorting lexicographically by digits.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and suggestions to improve the paper. This work is supported by the DEEL Project CRDPJ 537462-18 funded by the National Science and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ), together with its industrial partners Thales Canada inc, Bell Textron Canada Limited, CAE inc and Bombardier inc[3]. Marc-Olivier Killijian is also supported by a Discovery Grant from NSERC.

References

[1] Alpaydin and Kaynak. 1998. Optical Recognition of Handwritten Digits. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50P49.

[2] 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

[3] Sofiane Azogagh, Zelma Aubin Birba, Marc-Olivier Killijian, and Félix Larose-Gervais. 2024. RevoLUT: Rust efficient versatile library for oblivious Look-Up Tables. https://github.com/sofianeazogagh/revoLUT.

[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] Ayantika Chatterjee and Indranil Sengupta. 2020. Sorting of Fully Homomorphic Encrypted Cloud Data: Can Partitioning be Effective? *IEEE Transactions on Services Computing* 13, 3 (2020), 545–558. https://doi.org/10.1109/TSC.2017.2711018

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. *International Conference*

[3] https://deel.quebec

on the Theory and Application of Cryptology and Information Security (2017), 409–437.

[14] 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

[15] 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

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

[18] 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*.

[19] 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).

[20] 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

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

[22] David Freedman and Persi Diaconis. 1981. On the histogram as a density estimator: L 2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* 57, 4 (1981), 453–476.

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

[24] 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

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

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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

[32] 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.

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

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

[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 in Algorithm 7 a way to tweak the Blind Counting Sort algorithm to sort only the first κ 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 $4\kappa \cdot t_{BR} + (\kappa + p) \cdot t_{KS}$ time.

Algorithm 7: Prefix Blind Counting Sort (PBCS)

Input : A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$
A prefix length κ
Output : A LUT whose κ first elements are sorted

$[\![C]\!]_{\text{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\text{LUT}}$
for $i \leftarrow 0$ **to** $\kappa - 1$ **do**
  // $C_{m_i} \leftarrow C_{m_i} + 1$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [1]_{\text{LWE}})$
end
for $i \leftarrow 1$ **to** $p - 1$ **do**
  // $C_i \leftarrow C_i + C_{i-1}$
  $[\![x]\!]_{\text{LWE}} \leftarrow BAA([i-1]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([i]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [\![x]\!]_{\text{LWE}})$
end
$[\![R]\!]_{\text{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\text{LUT}}$
for $i \leftarrow \kappa - 1$ **to** 0 **do**
  // $C_{m_i} \leftarrow C_{m_i} - 1$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [-1]_{\text{LWE}})$
  // $R_{C_{m_i}} \leftarrow m_i$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![C_{m_i}]\!]_{\text{LWE}} \leftarrow BAA([\![m_i]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  $[\![R]\!]_{\text{LUT}} \leftarrow BAAdd([\![C_{m_i}]\!]_{\text{LWE}}, [\![R]\!]_{\text{LUT}}, [\![m_i]\!]_{\text{LWE}})$
end
return $[\![R]\!]_{\text{LUT}}$

B Key-value Blind Counting Sort

We describe in Algorithm 8 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 λ vectors of size p by the elements of first 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 $(4 + \lambda)p \cdot t_{BR} + (2 + \lambda)p \cdot t_{KS}$ time.

Algorithm 8: Key-value Blind Counting Sort (KV-BCS)

Input : A vector of λ LUT ciphertexts
$([\![m_0^0, \ldots, m_{p-1}^0]\!]_{\text{LUT}}, \ldots, [\![m_0^{\lambda-1}, \ldots, m_{p-1}^{\lambda-1}]\!]_{\text{LUT}})$
Output : A vector of λ LUT ciphertexts sorted by the first one
$([\![m_{\pi_0}^0, \ldots, m_{\pi_{p-1}}^0]\!]_{\text{LUT}}, \ldots, [\![m_{\pi_0}^{\lambda-1}, \ldots, m_{\pi_{p-1}}^{\lambda-1}]\!]_{\text{LUT}})$

$[\![C]\!]_{\text{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\text{LUT}}$
for $i \leftarrow 0$ **to** $p - 1$ **do**
  // $C_{m_i} \leftarrow C_{m_i} + 1$
  $[\![m_i^0]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0^0, \ldots, m_{p-1}^0]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i^0]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [1]_{\text{LWE}})$
end
for $i \leftarrow 1$ **to** $p - 1$ **do**
  // $C_i \leftarrow C_i + C_{i-1}$
  $[\![x]\!]_{\text{LWE}} \leftarrow BAA([i-1]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([i]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [\![x]\!]_{\text{LWE}})$
end
for $j \leftarrow 0$ **to** $\ell - 1$ **do**
  $[\![R^j]\!]_{\text{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\text{LUT}}$
end
for $i \leftarrow p - 1$ **to** 0 **do**
  // $C_{m_i} \leftarrow C_{m_i} - 1$
  $[\![m_i^0]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0^0, \ldots, m_{p-1}^0]\!]_{\text{LUT}})$
  $[\![C]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i^0]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}}, [-1]_{\text{LWE}})$
  // $R_{C_{m_i}} \leftarrow m_i$
  $[\![C_{m_i^0}]\!]_{\text{LWE}} \leftarrow BAA([\![m_i^0]\!]_{\text{LWE}}, [\![C]\!]_{\text{LUT}})$
  for $j \leftarrow 0$ **to** $\lambda - 1$ **do**
    $[\![m_i^j]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0^j, \ldots, m_{p-1}^j]\!]_{\text{LUT}})$
    $[\![R^j]\!]_{\text{LUT}} \leftarrow BAAdd([\![C_{m_i^0}]\!]_{\text{LWE}}, [\![R^j]\!]_{\text{LUT}}, [\![m_i^j]\!]_{\text{LWE}})$
  end
end
return $([\![R^0]\!]_{\text{LUT}}, \ldots, [\![R^{\lambda-1}]\!]_{\text{LUT}})$