Efficient and Secure k-NN Classification from Improved Data-Oblivious Programs and Homomorphic Encryption

Kelong Cong, Robin Geelen, Jiayi Kang, and Jeongeun Park

COSIC, ESAT, KU Leuven, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. The k-nearest neighbors classifier is a simple machine learning algorithm with applications in image recognition, finance, medical diagnosis and so on. It involves a measurement which is compared against a database of preclassified vectors, so that the result depends on the k vectors in the database that are closest to the measurement. In the client-server model, this classification process can be outsourced to an external party that offers machine learning as a service, where the measurement is sent in the form of a query. However, this raises privacy concerns if sensitive information is contained in the query.

We design a secure and non-interactive version of the k-nearest neighbors classifier, based on fully homomorphic encryption, which does not leak any information about the query to the server. Our algorithm is instantiated with the TFHE homomorphic encryption scheme, and the selection of the top-k elements is done with a novel strategy based on a type of data-oblivious algorithm—sorting networks. Compared to prior work from PoPETs 2021, the asymptotic complexity is improved from $\mathcal{O}(d^2)$ to $\mathcal{O}(d \log^2 k)$, where d is the number of entries in the k-NN model. Experimental results show that the proposed protocol can be up to 16 times faster (not accounting for difference in CPU) than previous approaches for a moderately sized database.

Keywords: Homomorphic encryption · Machine learning · k-nearest neighbors · Sorting networks.

1 Introduction

Outsourcing computation has been a popular solution to resolve modern conflicts between large data collection versus the limited local storage and computational power. Stimulated by regulations such as the General Data Protection Regulation (GDPR), data confidentiality received growing attention in outsourced computation. Fully Homomorphic Encryption (FHE) is a powerful cryptographic technique that allows arbitrary computations over encrypted data without decrypting intermediate values. This property enables secure computations that are *non-interactive* and propels FHE into a key privacy preserving technology [11,17,42,33,14,16,45,15,41]. With promising speedup from hardware accelerators [36,37,4,23,5], which can be up to three orders of magnitude faster than CPUs, FHE can soon provide feasible solutions for a wider range of real-world, privacy preserving applications.

Despite its promising potential, developing efficient FHE programs remains difficult. An important part of the inefficiency is the amplification in computation complexity when a plaintext program is converted into a program operating on the corresponding FHE ciphertexts. For example, in the homomorphic evaluation of the if-else paradigm, each conditional statement needs to be executed. By extension, when traversing a binary-tree, the full tree is touched instead of a single path. Programs that consist of nested branching are therefore impractical to realize homomorphically. In other words, the data secrecy guaranteed by FHE comes at the cost of increased computational complexity.

Data-oblivious algorithms in FHE Fortunately, the increase in computational complexity does not apply to *data-oblivious* programs where the sequence of operations and memory accesses do not depend on the input. Therefore, data-oblivious programs can be directly translated to their low-level FHE analogue. In this sense, describing a high-level algorithm in a data-oblivious manner is an FHE-friendly paradigm.

As an example, imagine we want to sort d encrypted elements homomorphically. Assuming we can efficiently implement a comparison operator in FHE, which sorting algorithm should we use? Quicksort and heapsort turn out to be not data-oblivious, and realizing those homomorphically is impractical despite their optimal time complexity of $\mathcal{O}(d \log d)$. In contrast, odd-even merge sort [27] is a data-oblivious algorithm with time complexity $\mathcal{O}(d \log^2 d)$. As such, it can be realized homomorphically with the same complexity.

The sequence of data-oblivious operations can be visualized as a *network*. For example, Figure 1 shows the network of odd-even merge sort for $d = 4$, where inputs enter from the left and a vertical line compares two elements. By counting the number of vertical lines and the number of vertical lines in series, we know that the algorithm has 5 comparators and a depth of 3. Here the depth refers to the maximum number of consecutive comparisons on any path from input to output.

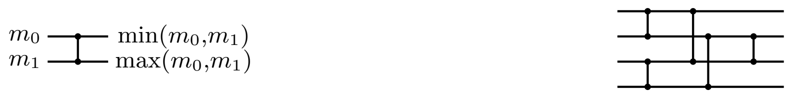

Fig. 1. The basic module and the network representation of odd-even merge sort for $d = 4$.

Depending on the used FHE scheme, the relevant algorithmic properties are different. In BGV [7] and BFV [6,22], controlling the depth is crucial, whereas optimizing the algorithmic complexity is more important in the TFHE [12] scheme.

Secure outsourcing and k-NN Privacy issues cannot be avoided when machine learning services happen in the cloud, especially when private/confidential data such as medical/financial records are involved. In this case, we need to solve the secure outsourcing problem, where a client wishes to keep its input private against a server, and the server does not want to reveal its data to the client, apart from what is requested. In particular, this work focuses on the k-Nearest Neighbors (k-NN) classification algorithm using an improved data-oblivious program.

The k-NN algorithm finds k "closest" neighbors of a given target vector among a set of vectors with respect to the Euclidean distance. In the classification use case, the k-NN algorithm takes as input a target feature vector with unknown class and then outputs the k nearest neighboring feature vectors with their corresponding classes. The final classification result can be obtained via majority voting. Due to its simplicity, non-parametric approach and high accuracy, k-NN is widely chosen as an effective solution for classification problems.

Solutions have been proposed based on cryptographic primitives such as homomorphic encryption [45], oblivious transfer (OT) [19], and garbled circuits (GC) combined with additively homomorphic encryption [9]. More information can be found in Section 1.2. Although the computation and communication cost have improved significantly over the years, using techniques such as OT and/or GC incurs inevitable interactions between a server and a client, which does not perfectly fit into the aim of our scenario. A recent work based solely on FHE [45] provides a non-interactive protocol, but it suffers from high computational overhead due to its complex instantiation.

1.1 Threat model and scenario

Similarly to previous works [9,45], our threat model considers a semi-honest (honest-but-curious) server that follows the protocol correctly, but tries to obtain information of the client from publicly known data. This goal can be achieved with FHE: as the data seen by the server is always encrypted, it cannot learn the privacy sensitive input from the client. The considered scenario is visualized in Figure 2. To reach our security goals, the client encrypts its query before sending it to the server. The database itself is owned by the server and therefore does not need to be encrypted. We do not explicitly consider model privacy since it is the same as in the plaintext setting, without FHE.

In summary, our primary goal is to design a non-interactive k-NN algorithm that is secure against a semi-honest server, i.e., ensuring client privacy.

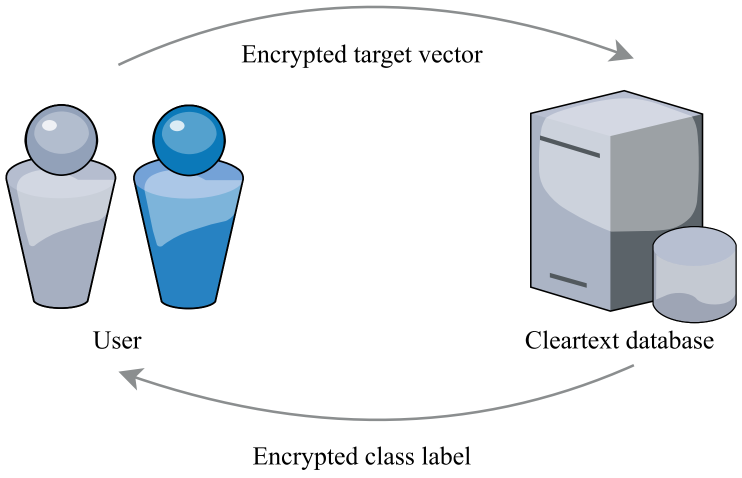

Fig. 2. Illustration of the considered scenario. A client sends an encrypted target vector (the query) to a cloud server. Then the server evaluates the secure k-NN algorithm and returns the encrypted class label.

1.2 Related work

Related art for secure *k*-NN classifiers Chen et al. [9] proposed two secure k-NN classifiers based on a mix of homomorphic encryption and multi-party computation. Their first classifier performs a linear scan of all data points and evaluates a top-k selection circuit to find the closest ones. Their second classifier is a clustering-based algorithm and has sublinear time complexity in the database size. However, it should be noted that both versions use an approximate circuit for top-k selection and therefore do not necessarily return the nearest neighbors. The protocol computes distances with BFV homomorphic encryption [6,22], and retrieval of the nearest neighbors is based on several primitives such as garbled circuits, secret sharing and distributed oblivious RAM. Although they provide faster runtimes than our method, the considered scenario is different. In particular, their protocol is interactive due to the use of multi-party computation, which makes it less suitable for outsourced computation in the context of cloud computing.

The most closely related work is that of Zuber and Sirdey [45], who also propose a non-interactive k-NN algorithm based on the TFHE scheme. The authors follow a different approach where they homomorphically construct a 0/1-matrix (the so-called delta-matrix) that contains a row and column for each vector in the database. More specifically, for each pair of distances between the client's target vector and two database vectors, this delta-matrix indicates which distance is the largest. Then they compute a sum over each column of the delta-matrix and check whether the database vector is among the k closest ones by comparison to a threshold. However, the time complexity of their method is quadratic in d, which makes it rather impractical for large databases. Moreover, the computed result is an encrypted 0/1-vector of dimension d, which results in a communication cost that is linear in d. To reduce the communication cost from $\mathcal{O}(d)$ to $\mathcal{O}(k)$, the delta-matrix method can be extended with a compression technique [3]. However, this enhanced delta-matrix method still has a quadratic computation complexity in the database size. In this work, we bring down the execution time to a linear function of the database database size (assuming k is constant). The communication cost of our solution is proportional to k. More details are given in Section 1.3.

Other related works are either very slow or rely on totally different security models. For example, Shaul et al. [39] implement a secure k-NN algorithm based on BGV homomorphic encryption [7], but report an execution time of around 3 hours on the breast cancer dataset. A completely different approach is taken by the SCONEDB model via a scalar-product-preserving encryption scheme [44]. However, this protocol computes the query result in the clear, which leaks useful information to the server. Another very recent paper proposes a lightweight k-NN solution, but it needs to distribute trust between two non-colluding servers [38].

Related art for data-oblivious algorithms and networks Theoretically, every RAM program of complexity $\mathcal{O}(T)$ can be converted into a data-oblivious circuit of size $\mathcal{O}(T^3 \log T)$ [40,34,32]. This principle gives an upper bound on the data-oblivious realization of RAM programs. As an extensively-studied problem, sorting has data-oblivious solutions lower than this upper bound. In fact, the complexity lower bound of sorting d elements is $\mathcal{O}(d \log d)$ comparisons, and the data-oblivious AKS network [1] is asymptotically optimal. Despite its theoretical significance, the AKS network is inefficient for practical values of d due to the large constant term hidden in the Big O notation. Leighton and Plaxton [28] describe a faster $\mathcal{O}(d \log d)$ sorting network, but their randomized method does not always sort correctly.

Practically used sorting networks have complexity $\mathcal{O}(d \log^2 d)$, which include Batcher's odd-even merge sort and bitonic sort [27]. The former always outperforms the latter in practice and is a popular choice in secure computation [43,32,26]. The structure of Batcher's odd-even merge sort is doubly recursive: *comparators* are called recursively to form a *merge* function, which is recursively called in the whole sorting procedure.

The doubly recursive structure gives data-oblivious solutions not only to sorting but also to the problem of *aggregation* (i.e., adding the weights that belong to the same item). As noted by Jönsson et al. [26], using a basic module called AGGREGATE-IF-EQUAL, one can build an aggregation network of complexity $\mathcal{O}(d \log^2 d)$, supposing there are d item-weight pairs to aggregate.

1.3 Our contributions

The goal of our work is to solve the secure, non-interactive k-NN problem efficiently, with a simpler instantiation based solely on FHE. We summarize our contributions as follows:

- We design a secure k-NN classifier for outsourced computation. Our protocol uses the TFHE [13] homomorphic encryption scheme for instantiating efficient min/max function and achieve non-interactivity. Unlike prior work [45], we use the classic approach called linear scan. Here the k

Table 1. List of symbols that will be used in this paper.

Meaning	Symbol
The LWE/RLWE dimension	n/N
The standard deviation of the noise	σ
The gadget base/size	g/ℓ
The plaintext/ciphertext modulus	t/q
The size of the database	d
The vector dimension of the database	γ
The desired number of nearest neighbors	k

smallest values are determined by comparing the distances between the given target vector and the d database vectors.

- In the subroutine of finding the top-k smallest elements out of d, we propose a novel data-oblivious algorithm that has a time complexity of $\mathcal{O}(d \log^2 k)$ (compared to $\mathcal{O}(d^2)$ in prior work [45]). Our algorithm is constructed via a truncation technique applied to Batcher's odd-even sorting network that removes the redundant comparisons. It is, to the best of our knowledge, the first protocol to leverage sorting networks in the design of a secure k-NN classifier.
- Our truncation technique is not limited to the sorting network, but naturally extends to data-oblivious algorithms with doubly recursive structures. For example, applying the truncation technique to the aggregation network optimizes the complexity from $\mathcal{O}(d \log^2 d)$ into $\mathcal{O}(d \log^2 z)$, where z is the number of different items in the list.
- We implement our protocol using the tfhe-rs library and evaluate it with the MNIST and the breast cancer dataset from [20]. Bar the differences in hardware, our best result is more than an order of magnitude faster than prior work.

1.4 Roadmap

Section 2 introduces all the building blocks used in this work. Section 3 gives a detailed description of data-oblivious programs as a form of networks, with a focus on sorting networks. Next, in Section 4, we improve the networks from the prior section using our novel truncation technique. This technique is applied in Section 5 to construct an efficient and secure k-NN algorithm, and the results (experimental and theoretical) are presented in Section 6. Finally, we conclude in Section 7.

2 Building blocks

2.1 Notations

The most commonly used symbols are summarized in Table 1. Some of these symbols will only be introduced in later sections. Note that the first part of this table gives the parameters for the TFHE scheme and the second part is related to the k-NN problem.

The dot product of two vectors $\mathbf{v}$ and $\mathbf{w}$ is denoted by $\langle \mathbf{v}, \mathbf{w} \rangle$. For a vector $\mathbf{x}$, we denote by $\mathbf{x}_i$ its i-th component scalar and by $\|\mathbf{x}\|$ its infinity norm. The logarithm function with base 2 is written as $\log(\cdot)$. We use the rings $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$, where q and N are positive integers. The i-th coefficient of a polynomial $M(X)$ is denoted by M_i. The variance of a random variable Z is denoted by $\mathsf{Var}(Z)$. Given a base g and a decomposition parameter $\ell \leq \lceil \log_g(q) \rceil$, we define a gadget vector $\mathbf{g} = (1, g, \ldots, g^{\ell-1})^\top$ and a gadget matrix $\mathbf{G} = \mathbf{I}_2 \otimes \mathbf{g}$.

The gadget decomposition function is written as $\mathbf{g}^{-1}(\cdot)$, and it satisfies $\langle \mathbf{g}^{-1}(a), \mathbf{g} \rangle \approx a \pmod q$ for all $a \in \mathcal{R}_q$. The result has small entries, i.e., $\|\mathbf{g}^{-1}(a)\| \leq g/2$. This can be extended entry-wise to vectors, that is, $\mathbf{G}^\top \cdot \mathbf{G}^{-1}(\mathbf{a}) \approx \mathbf{a} \pmod q$ with $\|\mathbf{G}^{-1}(\mathbf{a})\| \leq g/2$ for all $\mathbf{a} \in \mathcal{R}_q^2$.

2.2 TFHE ciphertexts and basic operations

The TFHE scheme [13] uses four ciphertext types based on the (ring) learning with errors problem [35,30]. Each ciphertext contains a noise or error term e that is added during encryption (sampled from a distribution χ_σ with standard deviation σ; a different σ may be used depending on the ciphertext type) and removed during decryption. The error of a ciphertext c is denoted by $\mathsf{Err}(c)$. The ciphertext types are defined as follows:

- $\mathsf{LWE}_{\mathbf{s}}(m) = (a_1, \ldots, a_n, b) \in \mathbb{Z}_q^{n+1}$, where $b = \sum_{i=1}^{n} -a_i \cdot s_i + \Delta m + e$. The message $m \in \mathbb{Z}_t$ (with $t \ll q$) is encoded in the ciphertext under a scaling factor $\Delta = q/t$. We call t the plaintext modulus and q the ciphertext modulus. For LWE ciphertexts, the secret key is a vector $\mathbf{s} = (s_1, s_2, \ldots, s_n)$.
- $\mathsf{RLWE}_s(m) = (a, b) \in \mathcal{R}_q^2$, where $b = -a \cdot s + \Delta m + e$. Both the message $m \in \mathcal{R}_t$ and the secret key s are polynomials. We use $c[1]$ and $c[2]$ to refer to the entries of a ciphertext c.
- $\mathsf{RLWE.Trivial.Noiseless}(m) = (0, \Delta \cdot m) \in \mathcal{R}_q^2$ is a ciphertext where all randomness (including the noise) is set to 0. It can be computed by a party that does not know the secret key.
- $\mathsf{RLWE}'_s(m) = \mathbf{Z} + [\mathbf{0}, \mathbf{g}] \cdot m \in \mathcal{R}_q^{\ell \times 2}$, where $\mathbf{Z}$ is a matrix, each row of which is an $\mathsf{RLWE}_s(0)$ encryption. The message and secret key have the same format as in the RLWE case. We use $\mathbf{C}[1]$ and $\mathbf{C}[2]$ to refer to the columns of a ciphertext $\mathbf{C}$.
- $\mathsf{RGSW}_s(m) = \mathbf{Z} + \mathbf{G} \cdot m \in \mathcal{R}_q^{2\ell \times 2}$, where $\mathbf{Z}$ is a matrix, each row of which is an $\mathsf{RLWE}_s(0)$ encryption. The message and secret key have the same format as in the RLWE case. In practice, however, messages are typically restricted to the form $m = \pm X^v$ or $m = 0$.

To distinguish between these types, LWE ciphertexts will be written as $\mathbf{c}$, RLWE ciphertexts as c and $\mathsf{RLWE}'/\mathsf{RGSW}$ ciphertexts as $\mathbf{C}$.

The exact definition of the encryption and decryption procedures is not relevant to this paper and therefore omitted here. Instead, we focus on the homomorphic operations that TFHE provides. Each of the following operations are computed over the ciphertext space and has a corresponding effect over the plaintext space:

- $\mathsf{SampleExtraction}(c, i) \to \mathbf{c}$: this procedure extracts one coefficient of a message polynomial encrypted as an RLWE ciphertext into an LWE ciphertext. It takes $c = \mathsf{RLWE}_s(M(X))$ and an index $0 \leq i < N$, and outputs $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(M_i)$, where M_i is the i-th coefficient of $M(X)$. The entries of the LWE key $\mathbf{s}$ will be equal to the coefficients of the RLWE key s.
- $M(X) \cdot c \to c'$: this procedure multiplies a plaintext polynomial by an RLWE ciphertext. Specifically, it takes $M(X) \in \mathcal{R}_t$ and $c = \mathsf{RLWE}_s(m)$, and outputs $c' = \mathsf{RLWE}_s(M(X) \cdot m)$.
- $c_1 + c_2 \to c'$: this procedure adds two RLWE ciphertexts. Specifically, it takes $c_1 = \mathsf{RLWE}_s(m_1)$ and $c_2 = \mathsf{RLWE}_s(m_2)$, and outputs $c' = \mathsf{RLWE}_s(m_1 + m_2)$. Note that this procedure can also take one ciphertext and one plaintext polynomial instead of two ciphertexts.
- $\mathbf{C} \boxdot c \to c'$: this procedure computes the *external product* between an RGSW and RLWE ciphertext. Specifically, it takes takes $\mathbf{C} = \mathsf{RGSW}_s(m_1)$ and $c = \mathsf{RLWE}_s(m_2)$, and outputs $c' = \mathbf{C}^\top \cdot \mathbf{G}^{-1}(c) = \mathsf{RLWE}_s(m_1 \cdot m_2)$.

An analogous multiplication and addition procedure exists for LWE ciphertexts and plaintexts, but it is not discussed here. Each of the above operations causes a certain amount of noise growth. This means that the output noise with be larger than the input noise. The only operation that reduces noise is called programmable bootstrapping and will be explained in Section 2.4.

2.3 LWE-to-RLWE key switching

Apart from the basic ciphertext operations defined in the previous section, TFHE relies on a key switching procedure. Key switching can be used to change the secret key, the ciphertext type and the parameters of the input ciphertext. We explain a key switching method due to Chillotti et al. [13] that converts a set of N LWE ciphertexts encrypting $(m_0, \ldots, m_{N-1})$ to an RLWE ciphertext that encrypts $m_0 + m_1 \cdot X + \ldots + m_{N-1} \cdot X^{N-1}$.

First, we introduce a key switching method that converts one LWE ciphertext into an RLWE ciphertext encrypting the same message. This method is specified in Algorithm 1 and is applied to the set of N LWE ciphertexts separately. Then, given N RLWE ciphertexts as the output Algorithm 1, we pack them into one RLWE ciphertext encrypting $m_0 + m_1 \cdot X + \ldots + m_{N-1} \cdot X^{N-1}$ as follows:

$$\mathsf{RLWE}_s\left(\sum_{i=0}^{N-1} m_i \cdot X^i\right) = \sum_{i=0}^{N-1} X^i \cdot \mathsf{RLWE}_s(m_i). \tag{1}$$

This process does not require expensive homomorphic operations because it only rearranges the coefficients of the RLWE ciphertexts. Therefore, the main cost of this key switching method comes from N iterations of Algorithm 1. Finally, we note one can also take $p < N$ LWE ciphertexts and duplicate some entries of $(m_0, \ldots, m_{N-1})$. In that case, we only need p calls to Algorithm 1.

Algorithm 1 Key switching one LWE ciphertext to RLWE

```
Input: $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(m) = (a_1, \ldots, a_n, b)$ where $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathsf{ksk} = \{\mathsf{ksk}_i = \mathsf{RLWE}'_s(s_i)\}_{i \in [n]}$
Output: $c = \mathsf{RLWE}_s(m)$
1: function KeySwitch($\mathbf{c}, \mathsf{ksk}$)
2:     for $i \leftarrow 1$ to $n$ do
3:         $c_i \leftarrow (\langle \mathbf{g}^{-1}(a_i), \mathsf{ksk}_i[1] \rangle, \langle \mathbf{g}^{-1}(a_i), \mathsf{ksk}_i[2] \rangle)$
4:     end for
5:     return $c \leftarrow (\sum_{i=1}^n c_i[1], b + \sum_{i=1}^n c_i[2])$
6: end function
```

Noise analysis of Algorithm 1. The noise contained in ksk_i is sampled from χ_σ which has variance σ^2, and we assume that the noise follows a subgaussian distribution. Let $\mathbf{e}_i$ be the noise contained in ksk_i, then the noise of c_i is equal to $\langle \mathbf{g}^{-1}(a_i), \mathbf{e}_i \rangle$ and therefore $\mathsf{Var}(\mathsf{Err}(c_i)) \leq g^2 \cdot \ell \cdot \mathsf{Var}(\mathbf{e}_i)$.[1] It follows that

$$\begin{aligned} \mathsf{Var}(\mathsf{Err}(c)) &\leq n \cdot \mathsf{Var}(\mathsf{Err}(c_i)) + \mathsf{Var}(\mathsf{Err}(\mathbf{c})) \\ &\leq n \cdot g^2 \cdot \ell \cdot \mathsf{Var}(\mathbf{e}_i) + \mathsf{Var}(\mathsf{Err}(\mathbf{c})) \\ &= n \cdot g^2 \cdot \ell \cdot \sigma^2 + \mathsf{Var}(\mathsf{Err}(\mathbf{c})). \end{aligned}$$

[1] Here we use $\mathsf{Err}(c_i)$ and $\mathsf{Var}(\mathbf{e}_i)$ for the largest error and largest variance among all coefficients, respectively.

We note that $\mathbf{c}$ can be either a fresh LWE ciphertext or an output of an earlier computation.

When repeating Algorithm 1 multiple times and combining the result in (1), the noise variances get added. Writing the resulting ciphertext as c', we have

$$\mathsf{Var}(\mathsf{Err}(c')) \leq n \cdot N \cdot g^2 \cdot \ell \cdot \sigma^2 + \mathsf{Var}(\mathsf{Err}(\mathbf{c}_i)),$$

where $\mathbf{c}_i$ is the input ciphertext with largest error. Note that the errors of the input ciphertexts are not added, because each one is located at a different coefficient. This analysis has made the implicit assumption that the coefficients of all ciphertexts are centered and uncorrelated. This assumption, called the independence heuristic, is standard in TFHE [13].

2.4 Programmable bootstrapping

All previous homomorphic operations increase the ciphertext's noise component. If the noise of a ciphertext is too large, it cannot be correctly decrypted anymore. To overcome this limitation, all FHE schemes can be equipped with a bootstrapping procedure that reduces the noise. The TFHE schemes applies bootstrapping to LWE ciphertexts, and it has one extra capability on top of noise reduction: it can evaluate a function on the encrypted input for free. It is therefore referred to as programmable bootstrapping, and the evaluated function is called a lookup table [13].

From a high level, the idea is to decrypt the input LWE ciphertext homomorphically using an RGSW/RLWE accumulator scheme. The input of programmable bootstrapping is a ciphertext $\mathbf{c} = \mathsf{LWE}_{\mathbf{z}}(m)$ and a bootstrapping key bk, which is essentially an RGSW encryption of $\mathbf{z}$. The full procedure is specified in Algorithm 2, and we refer to Micciancio and Polyakov [31] for a detailed discussion including comparison to the FHEW accumulator.

Importantly, the encryption parameters and the secret key in Algorithm 2 depend on the ciphertext, so we introduce a different symbol for each one of them: the input ciphertext $\mathbf{c}$ is encrypted with ciphertext modulus q and secret key $\mathbf{z}$; the accumulator ACC is encrypted with ciphertext modulus Q and secret key s; and the output ciphertext is encrypted with ciphertext modulus Q and secret key $\mathbf{s}$ (which is derived from s during sample extraction). Note that the output ciphertext $\mathbf{c}'$ of bootstrapping is encrypted under the secret key s. However, going back to the original key $\mathbf{z}$ is possible via an LWE-to-LWE key switching procedure.

Programmable bootstrapping can evaluate any negacyclic function $f : \mathbb{Z}_t \to \mathbb{Z}_t$ (i.e., it has to satisfy $f(m + t/2) = -f(m)$ for all m) on the encrypted input.[2] It is assumed that t is even and $t \ll 2N$. If the function f is known, the initialization of the accumulator can simply be done as

$$T(X) = \sum_{i=0}^{N-1} f\left(\left\lfloor \frac{i \cdot t}{2N} \right\rceil\right) \cdot X^i, \tag{2}$$

$$\mathsf{Init}_f(b) = \mathsf{RLWE.Trivial.Noiseless}\left(T(X) \cdot X^{-b}\right).$$

The polynomial $T(X)$ is referred to as the *test polynomial*, and it typically contains some redundancy which is necessary to decode noisy ciphertexts. That is, the coefficients of (2) encode all function values of f in total $2N/t$ times. In general, $T(X)$ is a trivial-noiseless encryption, but it is also possible that the function values of f are given as LWE ciphertexts (e.g., if f is not known to

[2] Here we make the simplifying assumption that the domain and codomain of f have the same plaintext modulus t, but this need not necessarily be the case.

the server). In that case, we can perform LWE-to-RLWE key switching to compute an encryption of (2) in order to initialize the accumulator. This requires in total N calls to Algorithm 1 (and even fewer if some function values of f are duplicated).

Algorithm 2 Programmable bootstrapping

```
Input: $\mathbf{c} = \mathsf{LWE}_{\mathbf{z}}(m) = (a_1, \dots, a_n, b)$ where $\mathbf{z} = (z_1, \dots, z_n)$, $\mathsf{bk} = \{\mathsf{bk}_i = \mathsf{RGSW}_s(z_i)\}_{i \in [n]}$ and a negacyclic
    function $f$
Output: $\mathbf{c}' = \mathsf{LWE}_{\mathbf{s}}(f(m))$
1: function BOOTSTRAP($\mathbf{c}, \mathsf{bk}, f$)
2:     $\mathsf{ACC} \leftarrow \mathsf{Init}_f(b)$
3:     for $i \leftarrow 1$ to $n$ do
4:         $\mathsf{ACC} \leftarrow \mathsf{ACC} + (X^{-a_i} - 1) \cdot (\mathsf{bk}_i \boxdot \mathsf{ACC})$
5:     end for
6:     return $\mathbf{c}' \leftarrow \mathsf{SampleExtraction}(\mathsf{ACC}, 0)$
7: end function
```

Noise analysis of Algorithm 2. The noise growth of programmable bootstrapping has been theoretically analyzed [13,31]. Let σ^2 be the noise variance of bk, then the noise of $\mathbf{c}'$ is equal to

$$\mathsf{Var}(\mathsf{Err}(\mathbf{c}')) \leq n \cdot N \cdot g^2 \cdot \ell \cdot \sigma^2 + \mathsf{Var}(\mathsf{Err}(\mathsf{ACC}_{\mathsf{init}})),$$

where $\mathsf{ACC}_{\mathsf{init}}$ is the initial accumulator. Here we assume a binary secret key distribution. Note that the noise growth is *additive* and depends only on the parameters of the encryption scheme.

2.5 Homomorphic computation of the squared distance

One building block of k-nearest neighbors classification is computation of the squared distance between an encrypted target vector and a cleartext data point. We adapt the method of Zuber and Sirdey [45] to compute the squared distance between a target vector and a model vector efficiently. Their method actually computes the *difference* between two squared distances, but we need the squared distance itself to be compared in the sorting network.

We are given one target vector $c \in \mathcal{R}_q^2$ (the client's encrypted input), which is an RLWE ciphertext that encodes $\mathbf{v} \in \mathbb{Z}_t^\gamma$. And we have a model vector $\mathbf{w} \in \mathbb{Z}_t^\gamma$ stored in the database. We assume that the model vector is given in cleartext since the server owns the database in our scenario. The goal here is to compute $\|\mathbf{v} - \mathbf{w}\|_2^2 = \|\mathbf{v}\|_2^2 - 2 \cdot \langle \mathbf{v}, \mathbf{w} \rangle + \|\mathbf{w}\|_2^2$ homomorphically. To do this, the model vector is encoded in two ways:

$$M(X) = \sum_{i=0}^{\gamma-1} \mathbf{w}_{\gamma-i-1} \cdot X^i,$$

$$M'(X) = \left(\sum_{i=0}^{\gamma-1} \mathbf{w}_i^2 \right) \cdot X^{\gamma-1}.$$

The target vector $\mathbf{v}$ is encrypted as

$$c = \mathsf{RLWE}_s \left(\sum_{i=0}^{\gamma-1} \mathbf{v}_i \cdot X^i \right). \tag{3}$$

However, the computation is much easier if additional information about $\|\mathbf{v}\|_2^2$ is provided by the client. Therefore,

$$c' = \mathsf{RLWE}_s\left(\left(\sum_{i=0}^{\gamma-1}\mathbf{v}_i^2\right)\cdot X^{\gamma-1}\right)$$

is also given to the computing party. The squared distance between the encrypted target vector c and the model vector $\mathbf{w}$ can now be computed as

$$c'' = c' - 2M(X)\cdot c + M'(X). \tag{4}$$

The result computed in (4) is an RLWE ciphertext that encrypts a polynomial, the $(\gamma-1)$-th coefficient of which gives us the squared distance. Therefore, we run $\mathsf{SampleExtract}(c'', \gamma-1)$ to get $\mathsf{LWE}_{\mathbf{s}}(\|\mathbf{v}-\mathbf{w}\|_2^2)$ (which works correctly assuming that $\gamma \leq N$).

An optimization It is sufficient for the k-NN application to compute the squared distances between target and model vectors up to a certain constant. In particular, since the ciphertext c' is identical for each squared distance, it can simply be removed from (4) and we obtain

$$c'' = -2M(X)\cdot c + M'(X). \tag{5}$$

This reduces the communication between client and server with 50% as now only one RLWE ciphertext is sent.

2.6 Comparison operations

Comparing two encrypted numbers can be done with programmable bootstrapping. For example, Zuber and Chakraborty [8] proposed two homomorphic comparison operators to build min and arg min functions. However, apart from the minimum and its argument, our protocol also requires the maximum and its argument. We therefore implement a different algorithm as explained in this section.

Assume that we are given four ciphertexts $\mathbf{c}_0 = \mathsf{LWE}_{\mathbf{s}}(m_0)$ and $\mathbf{c}_1 = \mathsf{LWE}_{\mathbf{s}}(m_1)$, and their corresponding labels $\mathbf{c}_0' = \mathsf{LWE}_{\mathbf{s}}(m_0')$ and $\mathbf{c}_1' = \mathsf{LWE}_{\mathbf{s}}(m_1')$. We want to compute four results:

- An LWE encryption of $\min(m_0, m_1)$.
- An LWE encryption of $\max(m_0, m_1)$.
- An LWE encryption of m_i' with $i = \arg\min(m_0, m_1)$.
- An LWE encryption of m_i' with $i = \arg\max(m_0, m_1)$.

First, we homomorphically compute the difference of the squared distances as $\mathbf{c}' = \mathbf{c}_0 - \mathbf{c}_1 = \mathsf{LWE}_{\mathbf{s}}(m_0 - m_1)$. This ciphertext encrypts a positive number if and only if $m_1 < m_0$. Note that the min function outputs either m_0 or m_1, and the arg min function outputs the corresponding m_0' or m_1'. Therefore, we can encode these numbers into two test polynomials for programmable bootstrapping. That is, the input ciphertext of bootstrapping is set to $\mathbf{c}' = \mathsf{LWE}_{\mathbf{s}}(m) = \mathsf{LWE}_{\mathbf{s}}(m_0 - m_1)$, which basically serves as a selector. The minimum can now be computed with the function

$$f(m) = \begin{cases} m_0 & \text{if } -t/4 < m \leq 0 \\ m_1 & \text{if } 0 \leq m < t/4, \end{cases} \tag{6}$$

Here we only consider the domain $(-t/4, t/4)$ to guarantee that f is negacyclic. The test polynomial can now be constructed as

$$T(X) = \sum_{i=0}^{N/2-1} m_1 \cdot X^i - \sum_{i=N/2}^{N-1} m_0 \cdot X^i,$$

where we have used $f(m) = -f(m - t/2) = -m_0$ for $t/4 \le m < t/2$. Similarly, the test polynomial for arg min can be constructed by replacing m_0 and m_1 with m_0' and m_1' in (6). Note that these four values are actually encrypted, so both test polynomials are obtained via LWE-to-RLWE key switching on $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_0'$ and $\mathbf{c}_1'$.

Finally, we note that the maximum of the two numbers can be computed as $\max(m_0, m_1) = m_0 + m_1 - \min(m_0, m_1)$. The arg max function can be evaluated in a similar way.

3 Visualizing data-oblivious algorithms as networks

FHE algorithms do not contain branches on encrypted data. Indeed, if an FHE algorithm did contain data-dependent branches, the process of executing this branch would leak the encrypted value. Such an algorithm, where the control flow and memory access pattern do not depend on the input data itself, is called *data-oblivious*.

An oblivious sequence of operations can be visualized as a so-called *network*. This section explains the basics of this visualization technique and further gives two examples of comparison networks: the tournament algorithm [18, Chapter 9] and Batcher's (d_1, d_2)-merging algorithm [27, Chapter 5].

A network comprises of interconnected *basic modules*. Figure 3 shows the structure of a basic module where inputs enter at the left and computations are represented by a vertical line in between the two inputs. In general, the outputs of a module (the two lines at the right) are computed as arbitrary functions f_0 and f_1 evaluated on the inputs. The outputs can again be used as inputs to another module. This work discusses sorting networks, where the most common basic module is the *comparator* of Figure 4. The comparator swaps the inputs if the first one is greater than the second.

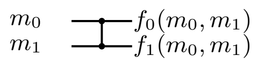

Fig. 3. General basic module

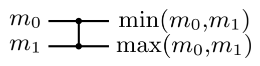

Fig. 4. Comparator

3.1 The tournament network

The tournament method is a data-oblivious algorithm to find the minimum (or maximum) of an array of d elements, which has received attention from HE works [25,8]. This algorithm divides the input in pairs, compares each of these pairs, and then the "winner" of each comparison proceeds to the next stage. As such, it only requires $d - 1$ comparisons, and the network has depth $\lceil \log d \rceil$. A different interpretation is as follows: the minimum element is constructed by splitting the initial array into two, finding the minimum of each part recursively, and then returning their minimum. Figure 5 shows the tournament network for $d = 8$.

Observe that we can find the k smallest elements out of d by performing the tournament method k times, each time over the remaining data. This approach needs $\mathcal{O}(kd)$ comparison operations and has depth $\mathcal{O}(k \log d)$.

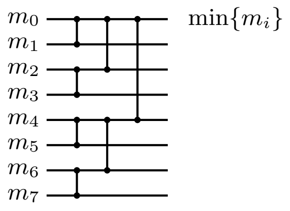

Fig. 5. The tournament network for finding the minimum of 8 elements.

3.2 Batcher's odd-even sorting network

Batcher's sorting algorithm uses a similar idea as the tournament method of decomposing the initial array in two parts. Then each part is sorted separately and the two sorted arrays are merged into one. The pseudocode of sorting is given in Algorithm 3. The resulting network for sorting 16 elements is shown in Figure 6, where each box represents a merging procedure.

The merging subroutine of sorting is specified in Algorithm 4. This algorithm is also based on a recursive decomposition of the problem: the input arrays are split into their even- and odd-index components. Then the even- and odd-index components are merged separately via two recursive calls. Finally, the result is constructed by pairwise comparison of the elements of the recursive calls. These pairwise comparisons are necessary to recombine the even and odd components, and one can prove that it results in a sorted array.

Algorithm 3 Batcher's odd-even merge sort

```
Input: An array $\mathbf{x}$ (of size $d > 0$)
Output: Sorted array that contains the same entries as $\mathbf{x}$
 1: function SORT($\mathbf{x}$)
 2:     $d \leftarrow$ size($\mathbf{x}$)
 3:     if $d = 1$ then
 4:         return $\mathbf{x}$
 5:     else                                        ▷ Sort two chunks separately and merge
 6:         $\mathbf{v} \leftarrow$ SORT($\mathbf{x}_0, \dots, \mathbf{x}_{\lceil d/2 \rceil - 1}$)
 7:         $\mathbf{w} \leftarrow$ SORT($\mathbf{x}_{\lceil d/2 \rceil}, \dots, \mathbf{x}_{d-1}$)
 8:         return MERGE($\mathbf{v}, \mathbf{w}$)
 9:     end if
10: end function
```

Property 1. The $(2^i, 2^i)$-merge contains $2^i \cdot i + 1$ comparators and has a comparison depth of $i + 1$.

Algorithm 4 Batcher's (d_1, d_2)-merge

```
Input: Two sorted arrays $\mathbf{x}$ (of size $d_1$) and $\mathbf{y}$ (of size $d_2$)
Output: Sorted array that contains the entries of $\mathbf{x}$ and $\mathbf{y}$
function MERGE($\mathbf{x}, \mathbf{y}$)
    $d_1 \leftarrow \text{size}(\mathbf{x}),\ d_2 \leftarrow \text{size}(\mathbf{y})$
    if $d_1 \cdot d_2 = 0$ then
        return $(\mathbf{x}, \mathbf{y})$
    else if $d_1 \cdot d_2 = 1$ then
        return COMPARE($\mathbf{x}_0, \mathbf{y}_0$)
    else                                                    ▷ Merge even- and odd-index components
        $\mathbf{v} \leftarrow$ MERGE($(\mathbf{x}_0, \dots, \mathbf{x}_{2\lceil d_1/2\rceil-2}), (\mathbf{y}_0, \dots, \mathbf{y}_{2\lceil d_2/2\rceil-2})$)
        $\mathbf{w} \leftarrow$ MERGE($(\mathbf{x}_1, \dots, \mathbf{x}_{2\lfloor d_1/2\rfloor-1}), (\mathbf{y}_1, \dots, \mathbf{y}_{2\lfloor d_2/2\rfloor-1})$)
        $\mathbf{z} \leftarrow (\mathbf{v}_0, \mathbf{w}_0, \mathbf{v}_1, \mathbf{w}_1, \dots)$
        for $i \leftarrow 1$ to $\lfloor(\text{size}(\mathbf{z}) - 1)/2\rfloor$ do
            $(\mathbf{z}_{2i-1}, \mathbf{z}_{2i}) \leftarrow$ COMPARE($\mathbf{z}_{2i-1}, \mathbf{z}_{2i}$)
        end for
        return $\mathbf{z}$
    end if
end function
function COMPARE($x, y$)                                    ▷ Comparator module of Figure 4
    return $(\min(x, y), \max(x, y))$
end function
```

Proof. The base case $i = 0$ needs 1 comparison of depth 1. Now suppose the property holds for $i - 1$, then following Algorithm 4, $(2^i, 2^i)$-merge needs

$$(2^{i-1} \cdot (i-1) + 1) \cdot 2 + (2^i - 1) = 2^i \cdot i + 1$$

comparisons and has depth $i + 1$. This completes the proof.

Theorem 1. *Batcher's odd-even sorting network for an array of size d has time complexity $\mathcal{O}(d \cdot \log^2 d)$ and depth $\mathcal{O}(\log^2 d)$.*

Proof. Assuming d is a power of two and following Property 1, the total comparison depth is

$$1 + 2 + \ldots + \log d \approx \frac{\log^2 d}{2} = \mathcal{O}(\log^2 d).$$

The total number of comparisons is

$$\sum_{i=1}^{\log d} \frac{d}{2^i}(2^{i-1}(i-1)+1) \approx \sum_{i=1}^{\log d} \frac{d}{2} \cdot i \approx \frac{d \log^2 d}{4} = \mathcal{O}(d \log^2 d).$$

The result for d not a power of two follows immediately, because both the time complexity and depth are bounded by the values for the next power of two.

4 Our top-k selection network

The key step of the k-NN method is returning the class labels for the k smallest distances out of d. For this purpose, we propose an algorithm with a much lower depth and a better time complexity than previous work [45].

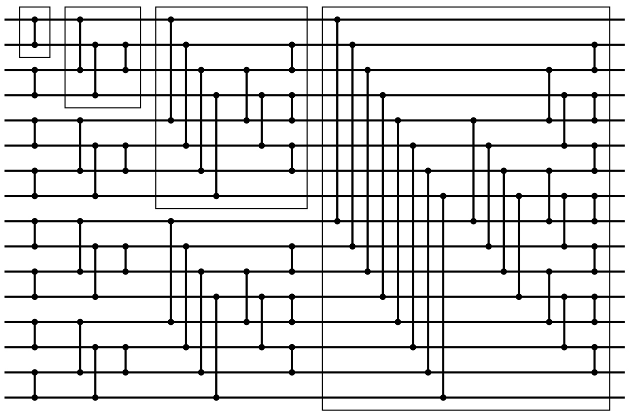

Fig. 6. Batcher's odd-even sorting network for an array of 16 elements, which has 63 comparators and depth 10. Boxes visualize Batcher's $(2^i, 2^i)$-merge for $i = 0, 1, 2, 3$ from the leftmost to the rightmost box.

Our use case does not require to output the full sorted array, but only the top-k smallest elements. Running Batcher's odd-even sorting network would therefore be suboptimal. Instead, we generalize the merging step from Algorithm 4 into Algorithm 5, which outputs k elements at most. This is achieved by truncating both the input and output array up to k elements, and instantiating the recursive calls with a lower value of k. The following property follows directly from Property 1:

Property 2. The truncated (k, k, k)-merge contains $\mathcal{O}(k \log k)$ comparators and has a comparison depth of $\mathcal{O}(\log k)$.

Just like sorting, finding the top-k smallest elements out of d can be done with a recursive approach: we split the initial array into two parts, find the k lowest elements of these two parts recursively, and then call Algorithm 5 to compute the final result. In fact, this can be interpreted as a truncated version of Batcher's odd-even sorting algorithm. The pseudocode of our top-k selector is given in Algorithm 6. The resulting network for $d = 16$ and $k = 4$ is shown in Figure 7, where each box represents a merging procedure.

An important difference between our top-k selector and the original sorting algorithm is the computation of the chunk size. In Algorithm 3, the input array is always split into two chunks of size $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$. However, from our experiments, we observe that the truncated version can be slightly more efficient if the chunk size is chosen as a multiple of $\mu = 2^{\lceil \log k \rceil}$. We therefore use the following heuristic: if $d > \mu$, the first chunk's size is computed as a multiple of μ that is close to $d/2$. Otherwise, the first chunk's size is equal to $\lceil d/2 \rceil$. The second chunk simply consists of the remaining elements (i.e., the ones that are not in the first chunk).

Theorem 2. *Our network for finding the k lowest elements out of d has time complexity $\mathcal{O}(d \log^2 k)$ and depth $\mathcal{O}(\log d \cdot \log k)$.*

Proof. We restrict the parameters d and k to powers of two. In this case, the full algorithm reduces to Batcher's odd-even sorting network until obtaining d/k sorted arrays of size k, and then performing the (k, k, k)-merge recursively as in the tournament method. Using Property 2 and following a similar reasoning as in the proof of Theorem 1, the comparison depth is

$$1 + 2 + \ldots + \log k + \mathcal{O}(\log k) \cdot \log \frac{d}{k} = \mathcal{O}(\log d \cdot \log k).$$

Algorithm 5 Truncated Batcher's (d_1, d_2, k)-merge

Input: Two sorted arrays $\mathbf{x}$ (of size d_1) and $\mathbf{y}$ (of size d_2) and a truncation parameter $k > 0$
Output: Sorted array that contains the entries of $\mathbf{x}$ and $\mathbf{y}$, or their k smallest entries if $k < d_1 + d_2$

```
 1: function MERGE($\mathbf{x}, \mathbf{y}, k$)
 2:     $\mathbf{x} \leftarrow$ TRUNCATE($\mathbf{x}, k$), $\mathbf{y} \leftarrow$ TRUNCATE($\mathbf{y}, k$)
 3:     $d_1 \leftarrow \text{size}(\mathbf{x})$, $d_2 \leftarrow \text{size}(\mathbf{y})$
 4:     if $d_1 \cdot d_2 = 0$ then
 5:         $\mathbf{z} \leftarrow (\mathbf{x}, \mathbf{y})$
 6:     else if $d_1 \cdot d_2 = 1$ then
 7:         $\mathbf{z} \leftarrow$ COMPARE($\mathbf{x}_0, \mathbf{y}_0$)
 8:     else                                        ▷ Merge even- and odd-index components
 9:         $\mathbf{x}^e \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_{2\lceil d_1/2 \rceil - 2})$, $\mathbf{x}^o \leftarrow (\mathbf{x}_1, \ldots, \mathbf{x}_{2\lfloor d_1/2 \rfloor - 1})$
10:         $\mathbf{y}^e \leftarrow (\mathbf{y}_0, \ldots, \mathbf{y}_{2\lceil d_2/2 \rceil - 2})$, $\mathbf{y}^o \leftarrow (\mathbf{y}_1, \ldots, \mathbf{y}_{2\lfloor d_2/2 \rfloor - 1})$
11:         $\mathbf{v} \leftarrow$ MERGE($\mathbf{x}^e, \mathbf{y}^e, \lfloor k/2 \rfloor + 1$)
12:         $\mathbf{w} \leftarrow$ MERGE($\mathbf{x}^o, \mathbf{y}^o, \lfloor k/2 \rfloor$)
13:         $\mathbf{z} \leftarrow (\mathbf{v}_0, \mathbf{w}_0, \mathbf{v}_1, \mathbf{w}_1, \ldots)$
14:         for $i \leftarrow 1$ to $\lfloor (\text{size}(\mathbf{z}) - 1)/2 \rfloor$ do
15:             $(\mathbf{z}_{2i-1}, \mathbf{z}_{2i}) \leftarrow$ COMPARE($\mathbf{z}_{2i-1}, \mathbf{z}_{2i}$)
16:         end for
17:     end if
18:     return TRUNCATE($\mathbf{z}, k$)
19: end function
20: function TRUNCATE($\mathbf{x}, k$)                       ▷ Truncate array to $k$ elements
21:     if size($\mathbf{x}$) $> k$ then
22:         $\mathbf{x} \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_{k-1})$
23:     end if
24:     return $\mathbf{x}$
25: end function
```

Algorithm 6 Truncated Batcher's odd-even merge sort

Input: An array $\mathbf{x}$ (of size $d > 0$) and a truncation parameter $k > 0$
Output: Sorted array that contains the same entries as $\mathbf{x}$, or its k smallest entries if $k < d$

```
 1: function SORT($\mathbf{x}, k$)
 2:     $d \leftarrow \text{size}(\mathbf{x})$
 3:     if $d = 1$ then
 4:         return $\mathbf{x}$
 5:     else                                    ▷ Sort two chunks separately and merge
 6:         $i \leftarrow$ CHUNKSIZE($d, k$)
 7:         $\mathbf{v} \leftarrow$ SORT($\mathbf{x}_0, \ldots, \mathbf{x}_{i-1}, k$)
 8:         $\mathbf{w} \leftarrow$ SORT($\mathbf{x}_i, \ldots, \mathbf{x}_{d-1}, k$)
 9:         return MERGE($\mathbf{v}, \mathbf{w}, k$)
10:     end if
11: end function
12: function CHUNKSIZE($d, k$)                        ▷ Compute size of first chunk
13:     $\mu \leftarrow 2^{\lceil \log k \rceil}$
14:     if $d \leq \mu$ then
15:         return $\lceil d/2 \rceil$
16:     else
17:         return $\mu \cdot \lceil d/(2\mu) \rceil$
18:     end if
19: end function
```

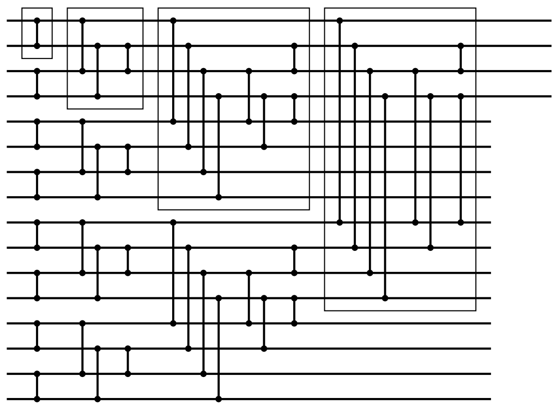

Fig. 7. Our network for finding the smallest 4 values out of 16, which has 44 comparators and depth 9. Boxes visualize truncated Batcher's $(2^i, 2^i, 4)$-merge for $i = 0, 1, 2, 3$ from the leftmost to the rightmost box.

Similarly, we obtain the total number of comparisons as

$$\sum_{i=1}^{\log k} \frac{d}{2^i}(2^{i-1}(i-1)+1) + \mathcal{O}(k \log k) \cdot \frac{d}{k} \approx \sum_{i=1}^{\log k} \frac{d}{2} \cdot i + \mathcal{O}(d \log k)$$
$$= \mathcal{O}(d \log^2 k).$$

4.1 The basic module of our network

The k-NN application has inputs of the form (m_i, m'_i), where m_i is the distance between the i-th model vector and the target vector, and m'_i is the class label of the i-th vector. The basic module of our network is therefore the *augmented* comparator visualized in Figure 8. The augmented comparator does not only compute the minimum and maximum, but also their corresponding class labels.

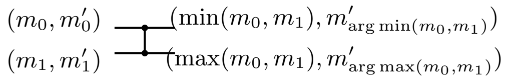
(m_0, m'_0)
$(\min(m_0, m_1), m'_{\arg\min(m_0,m_1)})$
(m_1, m'_1)
$(\max(m_0, m_1), m'_{\arg\max(m_0,m_1)})$

Fig. 8. An augmented comparator, where $\arg\min(m_0, m_1)$ and $\arg\max(m_0, m_1)$ refer to the indices (either 0 or 1) of the minimum and maximum element, respectively.

5 Our k-NN protocol

The k-NN application consists of two phases: computation of the squared distances and finding the k closest ones, together with their corresponding class labels. This section describes how these two phases are glued together.

5.1 The protocol

Squared distance computation First, the client encrypts the target vector using Equation (3) from Section 2.5 and sends it to the server. Then, for each model vector, the server evaluates the formula in (5) and extracts the $\gamma-1$-th coefficient to compute its squared distance. The result of the distance computation should satisfy $\|\mathbf{v}-\mathbf{w}\|_2^2 < t/4$ and we explain the reasoning in Section 5.1.

Precision reduction (optional) The squared distances may be computed using a large plaintext modulus, but the input of programmable bootstrapping (PBS) expects a small plaintext modulus (because we need $t \ll 2N$). If the two plaintext moduli are different, we need to perform a precision reduction, which can be done with one subtraction and one bootstrapping operation for every squared distance. The subtraction is necessary because we need to "recenter" the plaintext space. For example, consider plaintext moduli $t_{\mathsf{dist}} = 2 \cdot t_{\mathsf{sort}}$, and their scaling factors $2 \cdot \Delta_{\mathsf{dist}} = \Delta_{\mathsf{sort}}$. Encoded plaintexts of the form $(m_i \cdot \Delta_{\mathsf{dist}}, (m_i + 1) \cdot \Delta_{\mathsf{dist}})$ are mapped to $(m_i/2) \cdot \Delta_{\mathsf{sort}}$ since we want to reduce the precision by one bit in this example. Before bootstrapping, the center of $(m_i \cdot \Delta_{\mathsf{dist}}, (m_i + 1) \cdot \Delta_{\mathsf{dist}})$ needs to be at $(m_i/2) \cdot \Delta_{\mathsf{sort}} = m_i/\Delta_{\mathsf{dist}}$. As such, we need to subtract $\Delta_{\mathsf{dist}}/2$ from the initial plaintext and then perform bootstrapping with the identity function. This method easily generalizes to the case where t_{sort} is any multiple of t_{dist}.

The precision reduction step is only necessary if γ is high or if the precision of every element in the feature vector is large in comparison to t_{sort}. In Section 6, we show that precision reduction is necessary for one dataset but not for the other one.

Truncated sorting network The LWE ciphertexts of the previous phase are used as input to the truncated sorting network. Here the augmented comparators are built as explained in Section 2.6. This explains why the encoding needs to satisfy $\|\mathbf{v}-\mathbf{w}\|_2^2 < t/4$: in that way, the difference between two squared distances lies in the interval between $-t/4$ and $t/4$, and can therefore be unambiguously interpreted as a positive or negative number in the comparator. If the squared distance does not lie between $-t/4$ and $t/4$, then the result would overflow into the padding bit and cause errors. The output of this phase is a set of k LWE ciphertexts that encrypt the predicted class labels, and those are sent back to the client for decryption. Finally, the client computes the most common class label in the clear via majority voting. This is acceptable for most use cases as typically k is much smaller than d.

5.2 Noise growth of our protocol

Programmable bootstrapping is used to lower the noise level of its input, and computing a non-linear function at the same time. However, even though homomorphic comparisons are implemented with bootstrapping, the squared distances are never refreshed during the sorting phase. This is because the initial accumulator is generated by LWE-to-RLWE key switching, and is therefore a noisy ciphertext. Using the noise analyses from Section 2, we find that each comparison adds a noise variance of at most $2n \cdot N \cdot g^2 \cdot \ell \cdot \sigma^2$. To compute the output noise, this variance is multiplied by the depth of the circuit, which is $\mathcal{O}(\log d \cdot \log k)$ due to additive noise growth. For both datasets tested in the next section, the output noise remains at least 10 bits below the 64-bit ciphertext modulus. Hence this is sufficient to support a plaintext precision of 10 bits without requiring extra bootstrapping operations.

5.3 Security

The target vector is sent by a client encrypted with TFHE, which is IND-CPA secure (this is equivalent to semantic security). After homomorphic operation (via our protocol) with the server's data, the final output is also encrypted under the client's key. Therefore, query privacy is immediate from the IND-CPA property of TFHE. As we mentioned earlier in Section 1.1, we do not explicitly consider model privacy. Observe that model privacy can never be reached perfectly, because part of the model leaks through the query result.

6 Evaluation

6.1 Implementation and experimental setup

Our prototype implementation is written in the Rust programming language using the tfhe-rs[3] library. The source code can be found on GitHub.[4] We include the dependency manifest (`Cargo.lock`) and scripts used to run the experiments to aid reproducibility. All experiments are executed on machines with Intel(R) Core(TM) i9-9900 CPU @ 3.10 GHz using the Ubuntu 20.04 operating system. Only a single thread is used for all experiments.

Our experiment uses two datasets: the MNIST[5] and breast cancer[6] datasets. The MNIST dataset contains images of 8×8 pixels ($\gamma = 64$), which are the feature vectors. We preprocess the feature vectors to use ternary values. The breast cancer dataset has $\gamma = 32$ and we preprocess the feature vectors to use binary values.

We run our privacy preserving k-NN protocol using different values of d and k for both datasets and report the timing, accuracy and bandwidth results below. All experiments are done with the best feature vectors as the model. This is done by creating 10,000 plaintext models at random and selecting the one that gives the highest accuracy when evaluated on all the possible test vectors. Then we perform prediction/inference on 200 randomly selected test vectors using our secure k-NN algorithm. Therefore, the reported timing results are averaged over 200 executions.

The TFHE parameters are given in Table 2. These parameters are adapted from tfhe-rs.[7] We make a distinction between the plaintext modulus for sorting (t_{sort}) and distance computation (t_{dist}). That is, if $t_{\mathsf{sort}} \neq t_{\mathsf{dist}}$, then the precision reduction step from Section 5 needs to be used. Our definition of the plaintext modulus includes the padding bit. This extra padding bit is necessary to satisfy negacyclicity when the data is encoded [14]. For example, if the plaintext modulus is $t = 2^6$, then the message space is 5 bits since one bit is reserved for padding. The parameter choice from Table 2 guarantees 128 bits of security [2].

6.2 Computation time

The computation time and accuracy probabilities for the MNIST dataset are shown in Table 3, together with the results taken (and extrapolated) from [45]. Modulo the difference in CPU (we estimate that our CPU is faster by at most a factor of 2), the wall-clock time is between 1.5 to 16 times faster than prior work [45] while maintaining a good level of accuracy. The reason why our

[3] `https://github.com/zama-ai/tfhe-rs`
[4] `https://github.com/KULeuven-COSIC/ppknn`
[5] `https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits`
[6] `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`
[7] `https://github.com/zama-ai/tfhe-rs/blob/release/0.1.x/tfhe/src/shortint/parameters/mod.rs`

Table 2. The TFHE parameters used in our experiments. Note that when the homomorphic computation is done, the most significant bit is reserved for the sign. Therefore, if $t = 2^x$, then the actual message space is 2^{x-1}.

Parameter	Value
LWE dimension (n)	856
RLWE polynomial degree (N)	4096
LWE standard deviation (σ_{LWE})	2^{44}
RLWE standard deviation (σ_{RLWE})	2^2
PBS key—bk in Algorithm 2 (g, ℓ)	$(2^{22}, 1)$
KS key—at the end of PBS (g, ℓ)	$(2^3, 6)$
KS key—ksk in Algorithm 1 (g, ℓ)	$(2^{23}, 1)$
Ciphertext modulus (q)	2^{64}
Plaintext modulus (t_{sort})	2^6
Plaintext modulus MNIST (t_{dist})	2^9
Plaintext modulus breast cancer (t_{dist})	2^6
Dataset message space (MNIST)	$\mathbb{Z}_3$
Dataset message space (breast cancer)	$\mathbb{Z}_2$

performance gain becomes more apparent for larger d is because we do not have quadratic growth in computational complexity. Additionally, this experiment demonstrates the effect of precision reduction. Starting with 9 bits of precision for the distance computation, we reduce to 6 bits before the start of the sorting network. From the results, we see this has very little effect on accuracy (the "Clear accuracy" column does not have the precision reduction step).

Table 3. Computation time and accuracy for the MNIST dataset. The distance computation is performed using 9 bits of precision, then it is converted down to 6 bits before running our sorting network. The computation times prefixed with $\sim$ are estimated using extrapolation.

		Duration (s)		Accuracy	
k	d	[45]	Ours	Clear	FHE
3	40	30	21.1	0.78	0.76
	175	696	94.3	0.93	0.91
	228	1098	122.4	0.94	0.94
	269	1524	151.4	0.96	0.94
	457	4248	261.4	0.96	0.96
5	40	$\sim$ 33	27.7	0.78	0.76
	175	$\sim$ 636	128.2	0.92	0.90
	228	$\sim$ 1081	166.5	0.94	0.92
	269	$\sim$ 1505	204.8	0.94	0.94
	457	$\sim$ 4351	346.2	0.96	0.94

Similarly, the computation time and accuracy probabilities for the breast cancer dataset are presented in Table 4. For this dataset, there is no precision reduction step (i.e., $t_{\mathsf{dist}} = t_{\mathsf{sort}}$), because γ is low, the feature vectors are somewhat sparse and we preprocess the data to have binary feature vectors. Since our plaintext modulus is only 6 bits (one bit is reserved as the padding bit and another bit is reserved for the sign), the squared distance cannot exceed 4 bits. As such, we still have some errors when compared to the plaintext algorithm since the distance computation may overflow into the padding bit occasionally. Fortunately, the overflow does not happen often and our FHE accuracy closely trails the plaintext accuracy.

Table 4. Computation time and accuracy for the breast cancer dataset. No precision reduction is performed in this experiment. The computation times prefixed with $\sim$ are estimated using extrapolation.

		Duration (s)		Accuracy	
k	d	[45]	Ours	Clear	FHE
3	10	4	3.9	0.93	0.91
	30	$\sim$ 18	13.6	0.94	0.94
	50	$\sim$ 51	22.9	0.94	0.92
5	10	$\sim$ 2	5.1	0.94	0.92
	30	$\sim$ 18	18.7	0.94	0.94
	50	$\sim$ 51	32.9	0.95	0.94

6.3 Bandwidth

In our scenario, which [45] also considered, the client sends its target vector encrypted in an RLWE ciphertext under its own secret key. Then the server sends k class labels as its answer. Given the parameters in Table 2, our concrete query size is 64KB. However, as we defined in Section 2.2, an RLWE ciphertext (a, b) consists of a uniformly random part a. Therefore, the client can create a small seed to generate this part pseudorandomly, and then send this seed to the server instead. This is a very common optimization method [13,33], which halves the query size down to 32KB.

After executing our protocol, the server returns the k selected labels, which are in the form of LWE ciphertexts, therefore the answer size would be k times 6.7KB. As an optimization, we can easily pack k LWE ciphertexts into an RLWE ciphertext as long as $k \leq N$ (almost for free) [10]. We can also reduce the size of the answer by switching the modulus $\log q$ from 64 bits to 32 bits [7], and reducing the degree of the polynomials $N = 4096$ to 1024 by key switching. The resulting answer will have a size of 8KB, which is always smaller than k LWE ciphertexts for $k \geq 2$.

Zuber and Sirdey [45] considered an additional scenario where the database is encrypted by the server (data owner) and sent to the client (querier). Then the client runs the protocol with its own cleartext target vector. Therefore, the bandwidth is linear in the model size d (compared to a constant bandwidth in our protocol). However, comparing these two numbers is not completely fair since the proposed scenario is different.

6.4 Computational complexity

In this section, we compare the computational complexity of our sorting-based approach to previous works. The asymptotic complexities are summarized in Table 5 and the concrete computational cost, given as the number of programmable bootstrapping (PBS) operations, is shown in Figure 9. We do not count LWE-to-RLWE key switching, as we observe that bootstrapping is up to four times as expensive. Moreover, we consider exclusively the sorting step as it is computationally heavier than distance computation (up to $10\times$ more expensive for our parameter set).

First, there is the delta-matrix method of Zuber and Sirdey [45] in which a $d \times d$ matrix is constructed. Each element at position (i, j) in the matrix is 0 if the target vector is closer to the i-th model vector than to the j-th vector, and 1 otherwise. This method has quadratic complexity in the database size d. More specifically, building the matrix itself requires $(d^2 - d)/2$ bootstrapping operations. Then a so-called scoring operation is evaluated. This is done using roughly $d^2/(m-k)$ bootstrapping operations, where m is the number of ciphertexts that can be added before bootstrapping is called.

A second method is repeating the tournament method from Section 3.1 multiple times, which is a naive extension of a recently proposed min/arg min operator [8]. This approach requires in total $\mathcal{O}(kd)$ comparison operations. On the other hand, our approach requires only $\mathcal{O}(d \log^2 k)$ comparison operations, which scales better in terms of the parameter k. In practice, the value of k is typically chosen between 3 and 10, or sometimes up to $\sqrt{d}$ [24]. This leads to a significant improvement over the extended tournament method in terms of computational complexity.

Table 5. The complexity (number of bootstrappings) of finding the k smallest elements out of d using three algorithms: the delta-matrix method, performing the tournament network k times (denoted by Tourn. $k\times$), and our network. The additional parameter m was set to 65 as indicated in [45].

Delta-matrix method [45]	Tourn. $k\times$	Our network
$(d^2 - d)/2 + d^2/(m-k)$	$\mathcal{O}(kd)$	$\mathcal{O}(d \log^2 k)$

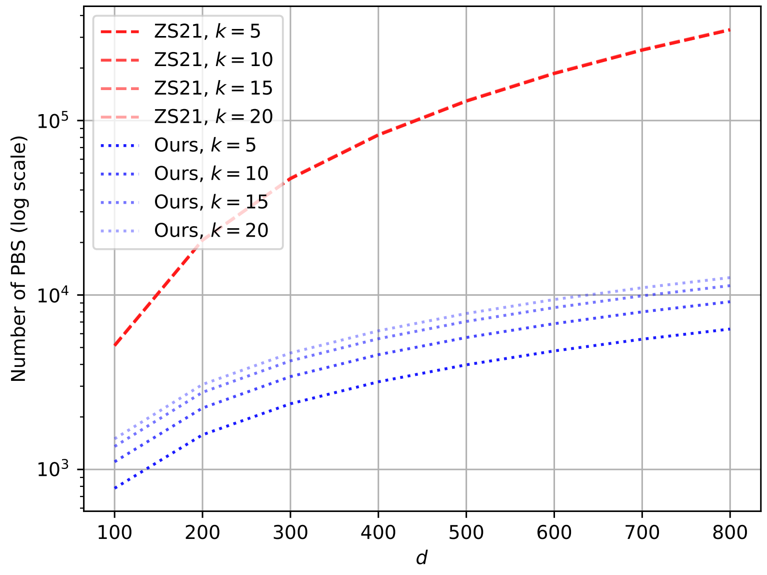

Fig. 9. Comparison of the number of PBS between our work and [45], denoted by ZS21 in the legend. Note that the red lines for [45] are stacked.

7 Discussion and conclusion

This work uses a truncated sorting network to compute the top-k smallest elements of an array for the secure k-NN classification problem. Our approach is asymptotically better (improved from $\mathcal{O}(d^2)$ to $\mathcal{O}(d \log^2 k)$), moving us closer to the computational complexity of plaintext algorithms. We also implemented our algorithm and it is up to $16\times$ faster than prior work. Additionally, our truncation technique may be applied to (weighted) homomorphic voting which we discuss below.

Homomorphic voting In our protocol, the truncated sorting network gives k pairs of encrypted distances and class labels that are sorted by distance. Without loss of generality, we denote them by $\mathcal{M}_k = \{(m_i, m'_i) \mid 1 \leq i \leq k\}$. The class labels in $\mathcal{M}_k$ are sent to the client, who performs a majority voting after decryption to obtain the predicted class. Another possibility is to perform homomorphic majority voting by the server, which is achievable with complexity $\mathcal{O}(k \log^2 k)$ as follows:

1. Assign a unit weight to each class label and sort the pairs $\{(m'_i, 1) \mid 1 \leq i \leq k\}$ according to the class labels.
2. Use the aggregation network to sum the weights of each distinct class label separately.
3. Find the class label which has the highest weight using the tournament method.

Steps (1) and (3) have a time complexity of $\mathcal{O}(k \log^2 k)$ and $\mathcal{O}(k)$, respectively. In step (2), the aggregation network has complexity $\mathcal{O}(k \log^2 k)$, which can be optimized with our truncation technique into $\mathcal{O}(k \log^2 z)$ if the number of different classes z is less than k. We omit further details.

The voting can also be distance-weighted, i.e., a model vector that is closer to the target vector can be given a higher weight. After the truncated sorting network, the class labels $\{m'_i \mid 1 \leq i \leq k\}$ are sorted by distance, so a suitable weight for m'_i is $k-i+1$ [21]. By changing the weight assignment in step (1) to $k-i+1$ (instead of a unit for each label), we can easily realize the distance-weighted voting in a data-oblivious manner.

Realizing majority voting in plaintext by the client or homomorphically by the server is a trade-off: the former has $\mathcal{O}(k)$ communication complexity, while the latter reduces the response size into $\mathcal{O}(1)$, at the cost of an extra $\mathcal{O}(k \log^2 k)$ computation for the server. If k is large (but still much smaller than d), it is better to use homomorphic majority voting as bandwidth can be significantly reduced, while an extra server cost of $\mathcal{O}(k \log^2 k)$ is negligible compared to the sorting network.

Future directions One limitation of TFHE is the restriction on the plaintext space. In our protocol, we work around this by quantizing the original values, which might be 8 bits or more, down to binary or ternary values. Of course, this step affects the accuracy of our secure k-NN protocol. In the future, we hope to investigate techniques that would support plaintexts with large precision, for example as proposed by Liu et al. [29].

Additionally, k-means clustering shares some similarities with k-NN since it also involves distance computation and sorting (to find the nearest centroid). Therefore, another future direction is to apply our techniques to the secure k-means clustering problem.

Acknowledgements This work was supported by CyberSecurity Research Flanders with reference number VR20192203. Additionally, this work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC

Pacific) under contract No. FA8750-19-C-0502. Robin Geelen is funded by Research Foundation – Flanders (FWO) under a PhD Fellowship fundamental research (project number 1162123N).

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA, the US Government, Cyber Security Research Flanders or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: An o(n log n) sorting network. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing. pp. 1–9 (1983)
2. Albrecht, M., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9** (10 2015). https://doi.org/10.1515/jmc-2015-0016
3. Ameur, Y., Aziz, R., Audigier, V., Bouzefrane, S.: Secure and non-interactive-nn classifier using symmetric fully homomorphic encryption. In: International Conference on Privacy in Statistical Databases. pp. 142–154. Springer (2022)
4. Beirendonck, M.V., D'Anvers, J.P., Verbauwhede, I.: FPT: a fixed-point accelerator for torus fully homomorphic encryption. Cryptology ePrint Archive, Report 2022/1635 (2022), https://eprint.iacr.org/2022/1635
5. Bertels, J., Beirendonck, M.V., Turan, F., Verbauwhede, I.: Hardware acceleration of fhew (2023), https://eprint.iacr.org/2023/618, https://eprint.iacr.org/2023/618
6. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_50
7. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090262
8. Chakraborty, O., Zuber, M.: Efficient and accurate homomorphic comparisons. Cryptology ePrint Archive, Report 2022/622 (2022), https://eprint.iacr.org/2022/622
9. Chen, H., Chillotti, I., Dong, Y., Poburinnaya, O., Razenshteyn, I.P., Riazi, M.S.: SANNS: Scaling up secure approximate k-nearest neighbors search. In: Capkun, S., Roesner, F. (eds.) USENIX Security 2020. pp. 2111–2128. USENIX Association (Aug 2020)
10. Chen, H., Dai, W., Kim, M., Song, Y.: Efficient homomorphic conversion between (ring) LWE ciphertexts. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 21, Part I. LNCS, vol. 12726, pp. 460–479. Springer, Heidelberg (Jun 2021). https://doi.org/10.1007/978-3-030-78372-3_18
11. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1243–1255. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3134061
12. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 3–33. Springer, Heidelberg (Dec 2016). https://doi.org/10.1007/978-3-662-53887-6_1
13. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (Jan 2020). https://doi.org/10.1007/s00145-019-09319-x
14. Chillotti, I., Joye, M., Paillier, P.: Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In: Dolev, S., Margalit, O., Pinkas, B., Schwarzmann, A. (eds.) Cyber Security Cryptography and Machine Learning. pp. 1–19. Springer International Publishing, Cham (2021)
15. Cong, K., Das, D., Nicolas, G., Park, J.: Panacea: Non-interactive and stateless oblivious RAM. Cryptology ePrint Archive, Report 2023/274 (2023), https://eprint.iacr.org/2023/274
16. Cong, K., Das, D., Park, J., Pereira, H.V.L.: SortingHat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 563–577. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560702
17. Cong, K., Moreno, R.C., da Gama, M.B., Dai, W., Iliashenko, I., Laine, K., Rosenberg, M.: Labeled PSI from homomorphic encryption with reduced computation and communication. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1135–1150. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484760
18. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. 2001 (2009)
19. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: NDSS 2015. The Internet Society (Feb 2015)

20. Dua, D., Graff, C.: UCI machine learning repository (2017), http://archive.ics.uci.edu/ml
21. Dudani, S.A.: The distance-weighted k-nearest-neighbor rule. IEEE Transactions on Systems, Man, and Cybernetics (4), 325–327 (1976)
22. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
23. Geelen, R., Beirendonck, M.V., Pereira, H.V.L., Huffman, B., McAuley, T., Selfridge, B., Wagner, D., Dimou, G., Verbauwhede, I., Vercauteren, F., Archer, D.W.: BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption. Cryptology ePrint Archive, Report 2022/657 (2022), https://eprint.iacr.org/2022/657
24. Hassanat, A.B., Abbadi, M.A., Altarawneh, G.A., Alhasanat, A.A.: Solving the problem of the k parameter in the knn classifier using an ensemble learning approach (2014)
25. Iliashenko, I., Zucca, V.: Faster homomorphic comparison operations for BGV and BFV. PoPETs **2021**(3), 246–264 (Jul 2021). https://doi.org/10.2478/popets-2021-0046
26. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122 (2011), https://eprint.iacr.org/2011/122
27. Knuth, D.E.: The art of computer programming: Volume 3: Sorting and Searching. Addison-Wesley Professional (1998)
28. Leighton, T., Plaxton, C.G.: Hypercubic sorting networks. SIAM Journal on Computing **27**(1), 1–47 (1998)
29. Liu, Z., Micciancio, D., Polyakov, Y.: Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part II. LNCS, vol. 13792, pp. 130–160. Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22966-4_5
30. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (May / Jun 2010). https://doi.org/10.1007/978-3-642-13190-5_1
31. Micciancio, D., Polyakov, Y.: Bootstrapping in fhew-like cryptosystems. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 17–28 (2021)
32. Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., Boneh, D.: Privacy-preserving matrix factorization. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 801–812 (2013)
33. Park, J., Tibouchi, M.: SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In: Chen, L., Li, N., Liang, K., Schneider, S.A. (eds.) ESORICS 2020, Part II. LNCS, vol. 12309, pp. 86–106. Springer, Heidelberg (Sep 2020). https://doi.org/10.1007/978-3-030-59013-0_5
34. Pippenger, N., Fischer, M.J.: Relations among complexity measures. Journal of the ACM (JACM) **26**(2), 361–381 (1979)
35. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th ACM STOC. pp. 84–93. ACM Press (May 2005). https://doi.org/10.1145/1060590.1060603
36. Samardzic, N., Feldmann, A., Krastev, A., Devadas, S., Dreslinski, R., Peikert, C., Sanchez, D.: F1: A fast and programmable accelerator for fully homomorphic encryption. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. p. 238–252. MICRO '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3466752.3480070, https://doi.org/10.1145/3466752.3480070
37. Samardzic, N., Feldmann, A., Krastev, A., Manohar, N., Genise, N., Devadas, S., Eldefrawy, K., Peikert, C., Sanchez, D.: Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 173–187. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3470496.3527393, https://doi.org/10.1145/3470496.3527393
38. Servan-Schreiber, S., Langowski, S., Devadas, S.: Private approximate nearest neighbor search with sublinear communication. In: 2022 IEEE Symposium on Security and Privacy. pp. 911–929. IEEE Computer Society Press (May 2022). https://doi.org/10.1109/SP46214.2022.9833702
39. Shaul, H., Feldman, D., Rus, D.: Secure k-ish nearest neighbors classifier. PoPETs **2020**(3), 42–61 (Jul 2020). https://doi.org/10.2478/popets-2020-0045
40. Stephen A. Cook, R.A.R.: Time bounded random access machines. Journal of Computer and System Sciences **7**, 354–375 (1973)
41. Stoian, A., Frery, J., Bredehoft, R., Montero, L., Kherfallah, C., Chevallier-Mames, B.: Deep neural networks for encrypted inference with tfhe. Cryptology ePrint Archive, Paper 2023/257 (2023), https://eprint.iacr.org/2023/257, https://eprint.iacr.org/2023/257
42. Tueno, A., Boev, Y., Kerschbaum, F.: Non-interactive private decision tree evaluation. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 174–194. Springer (2020)

43. Wang, G., Luo, T., Goodrich, M.T., Du, W., Zhu, Z.: Bureaucratic protocols for secure two-party sorting, selection, and permuting. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. pp. 226–237 (2010)
44. Wong, W.K., Cheung, D.W.l., Kao, B., Mamoulis, N.: Secure knn computation on encrypted databases. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 139–152 (2009)
45. Zuber, M., Sirdey, R.: Efficient homomorphic evaluation of k-NN classifiers. PoPETs **2021**(2), 111–129 (Apr 2021). https://doi.org/10.2478/popets-2021-0020