Revisiting Oblivious Top-k Selection with Applications to Secure k-NN Classification

Kelong Cong[1]*, Robin Geelen[2], Jiayi Kang[2], and Jeongeun Park[2]

[1] Zama, Paris, France
kelong.cong@zama.ai
[2] COSIC, ESAT, KU Leuven, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. An oblivious Top-k algorithm selects the k smallest elements from d elements while ensuring the sequence of operations and memory accesses do not depend on the input. In 1969, Alekseev proposed an oblivious Top-k algorithm with complexity $O(d \log^2 k)$, which was later improved by Yao in 1980 for small $k \ll \sqrt{d}$.
In this paper, we revisit the literature on oblivious Top-k and propose another improvement of Alekseev's method that outperforms both for large $k = \Omega(\sqrt{d})$. Our construction is equivalent to applying a new truncation technique to Batcher's odd-even sorting algorithm. In addition, we propose a combined network to take advantage of both Yao's and our technique that achieves the best concrete performance, in terms of the number of comparators, for any k. To demonstrate the efficiency of our combined Top-k network, we present a secure non-interactive k-nearest neighbors classifier using homomorphic encryption as an application. Compared with the work of Zuber and Sirdey (PoPETS 2021) where oblivious Top-k was realized with complexity $O(d^2)$, our experimental results show a speedup of up to 47 times (not accounting for difference in CPU) for $d = 1000$.

Keywords: Top-k selection · Homomorphic encryption · Machine learning · Nearest neighbors · Sorting networks · TFHE.

1 Introduction

Outsourcing computation has been a popular solution to resolve modern conflicts between large data collection versus 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

* Work done while the author was at COSIC, ESAT, KU Leuven.

[10,18,37,31,14,17,40,15,36]. With promising speedup from hardware accelerators [33,34,3,21,4], 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. In other words, the data secrecy guaranteed by FHE comes at the cost of increased computational complexity, even if the overhead of FHE is low.

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, suppose we want to sort d encrypted elements homomorphically. Then 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 $O(d \log d)$. In contrast, Batcher's odd-even merge sort [25] is a data-oblivious algorithm with time complexity $O(d \log^2 d)$, and it can be realized homomorphically with the same complexity.

Additionally, the most appropriate cost measure depends on the FHE scheme. In BGV [6], BFV [5,20] and CKKS [11], controlling the multiplicative depth (i.e., the number of consecutive multiplications) of the sorting algorithm is crucial. On the other hand, optimizing the algorithmic complexity is more important in the TFHE [12] scheme.

Oblivious Top-k selection. Given d elements, a Top-k algorithm selects its k smallest (or largest) elements, while the output elements are not necessarily sorted. Top-k selection is an important building block for various applications, in which the k most important records in a huge information space (consisting of d records) are extracted by defining proper scoring functions and returning those with the best ranks. Widely used examples include the k-nearest neighbors classification technique [26], recommender systems [23] and genetic algorithms [30].

This work focuses on Top-k algorithms that are data-oblivious. Since oblivious programs can be visualized as networks of low-level modules, the terms *Top-k network* and *oblivious Top-k* are used interchangeably. Appendix A introduces the basics of the network visualization.

Research into oblivious Top-k has a long history. In this work, we revisit the complexity upper bounds for the oblivious Top-k algorithm derived by Alekseev in 1969 [2], which was then improved by Yao in 1980 [39]. Alekseev proposed a procedure to select the Top-k out of $2k$ elements, which can be generalized into Top-k out of d elements with complexity $O(d \log^2 k)$ [25]. This method not only provides better asymptotic complexity than the recent realizations but also outperforms those in practice. Yao, on the other hand, introduced an unbalanced recursive procedure in [39, Lemma 3.2] to improve Alekseev's for small $k \ll \sqrt{d}$.

These results, however, have not received much attention so far. Recent realizations of oblivious Top-k in secure computation either use an oblivious sorting algorithm [40,38,24] or call oblivious Min (or Max) k times repeatedly [8,22,7]. The complexity in the former method is not parameterized by k since it always contains redundant comparisons, and the latter method results in complexity $O(kd)$, which grows linearly in k.

1.1 Our contributions

Firstly, we revisit Yao's recursive approach for oblivious Top-k selection. We observe that not all parameters d and k can be reduced to the recursive base case ($k = 1$ or $k = 2$) in [39, Theorem 3.1], Therefore, we fix this by handling one more case using symmetry. This construction results in a Top-k network with complexity $O(d \log k)$ for small $k \ll \sqrt{d}$.

Secondly, we also propose another improvement of Alekseev's method independent of Yao's. Specifically, we improve Alekseev's order-preserving merging procedure from $O(k \log^2 k)$ to $O(k \log k)$ by truncating Batcher's merge. As such, our network is essentially a truncated version of Batcher's odd-even sorting network with complexity $O(d \log^2 k)$, where redundant comparisons are removed.

Since our truncated method is better for large $k = \Omega(\sqrt{d})$ and Yao's method is better for small k, our third contribution is to introduce a combined method which takes advantage of both. For concrete values of k and d, the combined network recursively calls our truncated merge method or Yao's method, depending on which one uses fewer comparators. As such, the complexity of our combined network is upper bounded by the better method of Yao's and our truncated Top-k, and slightly improves on those methods for some parameters. If k and d are known in advance, the combined network can be preprocessed by the server.

Lastly, to demonstrate the efficiency of our combined Top-k network, we present non-interactive and secure k-Nearest Neighbors (k-NN) classification as an application. We use the TFHE homomorphic encryption scheme to handle large multiplicative depth efficiently, and our protocol is implemented in the TFHE-rs[3] library. Compared with prior work [40], where oblivious Top-k was realized with complexity $O(d^2)$, our experimental results show a speedup of up to 47 times (not accounting for difference in CPU) for $d = 1000$ and $k = 3$.

1.2 Related work

Oblivious Top-*k*. To simplify the explanation, we denote an oblivious Top-k out of d procedure as a (k, d)-selector. In 1969, Alekseev [2] proposed a $(k, 2k)$-selector using sorting as a subprocedure. This method was generalized to arbitrary d [25], which achieves a complexity of $(\lceil d/k \rceil - 1)(2S(k) + k)$ comparators

[3] https://github.com/zama-ai/tfhe-rs

for oblivious Top-k selection. We use $S(k)$ to denote the number of comparators in k-sorting (sorting k elements).

The above complexity was improved by Yao in 1980. Specifically, in the proof of [39, Theorem 3.1], an unbalanced recursive procedure was introduced, which yields better networks for small k. This recursive procedure will be discussed in detail in Section 2.3.

Surprisingly, the constructions above have been ignored in research over the past few decades. To our knowledge, recent data-oblivious Top-k solutions can be categorised into three types:

– The first type applies a sorting algorithm directly, and then discards the $d-k$ irrelevant elements. For example, in a recent homomorphic k-NN realization, Zuber and Sirdey [40] use sorting of complexity $O(d^2)$ to achieve Top-k.
– The second type repeatedly finds the Min (or Max) using the so-called tournament method [22], where inputs are compared pairwise and the "winner" proceeds to the next stage. This requires $O(kd)$ comparisons in total.
– The third type is from hardware-related research [35]. This work builds a bitonic Top-k algorithm by removing the unnecessary comparisons in bitonic sorting, thereby achieving $O(d \log^2 k)$. The number of comparators in bitonic sorting is always higher than Batcher's odd-even merge sort, but it is useful in hardware designs due to a more cache-friendly memory access pattern.

Secure k-NN classification. Chen et al. [8] proposed two secure k-NN classifiers based on a mixture of homomorphic encryption and secure multi-party computation. However, both versions use an approximate circuit for Top-k selection and therefore do not necessarily return the nearest neighbors. Additionally, their protocols are interactive 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 [40], who also propose a non-interactive k-NN algorithm based on the TFHE scheme. The authors use a specialized, FHE-friendly approach to perform the Top-k selection step which is asymptotically worse than standard sorting algorithms. Our approach differs in this key step where we identify Top-k selection networks that involve fewer comparison operators.

2 Top-k selection networks

2.1 Revisiting Alekseev's Top-k network

Alekseev [2] proposed a merging procedure to select the Top-k out of $2k$ elements: partition and sort two length-k arrays to obtain $\{x_0, \dots, x_{k-1}\}$ and $\{x_k, \dots, x_{2k-1}\}$, then compare and interchange

$$x_0 \text{ with } x_{2k-1},\ x_1 \text{ with } x_{2k-2},\ \dots,\ x_{k-1} \text{ with } x_k. \tag{1}$$

This can be generalized into Top-k out of d elements by partitioning the inputs into $\lceil d/k \rceil$ length-k arrays and applying Alekseev's procedure (two k-sortings and k comparisons) $\lceil d/k \rceil - 1$ times as in the tournament procedure [25]. It solves the Top-k problem using $(\lceil d/k \rceil - 1)(2S(k) + k)$ comparators, where $S(k)$ is the number of comparators for k-sorting. Realizing $S(k)$ with practical sorting networks (e.g., Batcher's odd-even merge sort) leads to an asymptotic complexity of $S(k) = O(k \log^2 k)$, so this Top-k network consists of $O(d \log^2 k)$ comparators.

Reinterpretation as order-preserving merging. Let (d_1, d_2, k)-merge denote an order-preserving merge where the inputs are two sorted arrays of length d_1 and d_2, and the output is the sorted Top-k out of $d_1 + d_2$ elements. Then Alekseev's Top-k procedure can be reinterpreted into three steps:

1. sort $\lceil d/k \rceil$ length-k arrays;
2. apply $\lceil d/k \rceil - 2$ times (k, k, k)-merge in the tournament manner, where each merge consists of the procedure from (1) and a k-sorting of the output;
3. apply the procedure from (1).

In step 2, each (k, k, k)-merge has complexity $S(k) + k$. Using practical sorting networks such as Batcher's odd-even sorting network leads to complexity $O(k \log^2 k)$ for (k, k, k)-merge.

2.2 Our truncated sorting network

Our order-preserving merging. We achieve (k, k, k)-merges differently: we observe that Batcher's odd-even merge with inputs of length d_1 and d_2 is an order-preserving merge with an output array of length $d_1 + d_2$ (see Appendix A for details). Since only the Top-k smallest elements are of interest, performing Batcher's odd-even merge directly would be excessively costly. Instead, we generalize the merge procedure into Algorithm 1, which outputs at most k elements by removing redundant comparisons.

Theorem 1. *The truncated (k, k, k)-merge contains $O(k \log k)$ comparators and has a comparison depth of $O(\log k)$.*

Note that our order-preserving merge procedure is not only asymptotically better than Alekseev's $O(k \log^2 k)$, but also better in practice: as Figure 1 shows, it contains fewer comparisons for a small value of $k = 3$.

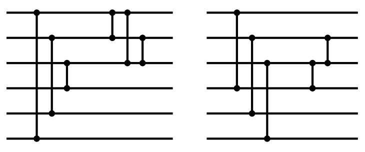
(a) Alekseev's (b) Our network

Fig. 1: Two constructions of a $(3, 3, 3)$-merge network.

Algorithm 1 Our truncated (d_1, d_2, k)-merge

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

Our truncated sorting network. Realizing step 2 in Alekseev's Top-k with our truncated (k, k, k)-merge is essentially a truncated version of Batcher's odd-even sorting algorithm, where the Top-k elements are selected in a recursive approach. As can be seen in Algorithm 2, we split the initial array into two parts, find the Top-k elements of these two parts recursively, and then call Algorithm 1 to compute the final result.

Moreover, our Algorithm 2 also improves the input partitioning in Alekseev's step 1. Specifically, we observe that the truncated network is 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). As an example, the resulting network for $d = 16$ and $k = 3$ is shown in Figure 2, where each box represents a merging procedure.

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

Proof. For the ease of asymptotic analysis, we restrict the parameters d and k to powers of two. In this case, the full algorithm reduces to Batcher's odd-even

Algorithm 2 Our truncated odd-even merge sort

```
Input: Array $\mathbf{x}$ (of size $d > 0$) and truncation parameter $k > 0$
Output: Sorted array that contains the entries of $\mathbf{x}$, or its $k$ smallest entries if $k < d$
 1: function SORT($\mathbf{x}, k$)
 2:     if $d = 1$ then
 3:         return $\mathbf{x}$
 4:     else                                        ▷ Sort two chunks separately and merge
 5:         $i \leftarrow$ CHUNKSIZE($d, k$)
 6:         $\mathbf{v} \leftarrow$ SORT(($\mathbf{x}_0, \ldots, \mathbf{x}_{i-1}$), $k$)
 7:         $\mathbf{w} \leftarrow$ SORT(($\mathbf{x}_i, \ldots, \mathbf{x}_{d-1}$), $k$)
 8:         return MERGE($\mathbf{v}, \mathbf{w}, k$)
 9:     end if
10: end function
11: function CHUNKSIZE($d, k$)                    ▷ Compute size of first chunk
12:     $\mu \leftarrow 2^{\lceil \log k \rceil}$
13:     if $d \le \mu$ then
14:         return $\lceil d/2 \rceil$
15:     else
16:         return $\mu \cdot \lceil d/(2\mu) \rceil$
17:     end if
18: end function
```

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 Theorem 1, the comparison depth is

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

and the total number of comparisons is

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

Note that our Algorithm 2 always outputs sorted results, but the output of the Top-k problem does not need to be sorted. Therefore, two more optimizations are incorporated in the implementation: (1) if $k > d/2$, we can exchange the roles of k and $d - k$ (this will be explained in more detail in Section 2.3); (2) the last merge box can be replaced by Alekseev's merge procedure [2], where only $d_1 + d_2 - k$ comparators are used.

Comparison with related work. To the best of our knowledge, there exist three Top-k methods of complexity $O(d \log^2 k)$: Alekseev's procedure (Section 2.1), a method based on bitonic sorting [35], and our method from Algorithm 2. Despite the same asymptotic complexity, our algorithm has the fewest

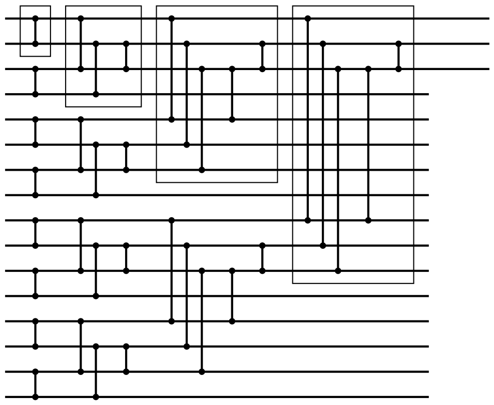

Fig. 2: Our network for finding the 3 smallest values out of 16, which has 35 comparators and depth 9. Boxes visualize our truncated $(d, d, 3)$-merge for $d = 1, 2, 3, 3$ from the leftmost to the rightmost box.

comparators, following the explanation in Algorithm 2 and Section 1.2. An example of $d = 40$ is presented in Figure 3. Note that the monotonicity in our Top-k is a result of our input partitioning optimization in Section 2.2.

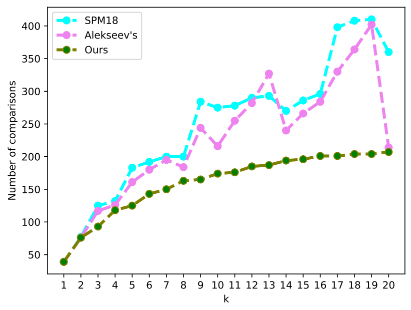

Fig. 3: The number of comparisons $U(k, d)$ in selecting the Top-k elements out of $d = 40$ using the method in SPM18 [35], Alekseev's method and our truncated network. The number of comparisons is shown only for $1 \leq k \leq d/2$ as $U(k, d) = U(d - k, d)$ by symmetry.

2.3 Yao's Top-k selection network revisited

We investigate another idea from Yao [39], who designed a Top-k selection network via direct recursion. The idea is as follows: first, using $\lfloor d/2 \rfloor$ comparators, the input array $\mathbf{x}$ is partitioned into two halves $\mathbf{a} = (\mathbf{a}_1, \ldots, \mathbf{a}_{\lceil d/2 \rceil})$ and

$\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_{\lfloor d/2 \rfloor})$ such that $\mathbf{a}_i < \mathbf{b}_i$ for $i = 1, \ldots \lfloor d/2 \rfloor$. At most $\lfloor k/2 \rfloor$ elements of the desired output will be in array $\mathbf{b}$, so we can use a $(\lfloor k/2 \rfloor, \lfloor d/2 \rfloor)$-selector to find those elements. Then the output of this selector and array $\mathbf{a}$ are given to a $(k, \lceil d/2 \rceil + \lfloor k/2 \rfloor)$-selector, which produces the final result. An example construction of a $(4, 9)$-selector is given in Figure 4.

Theorem 3 (Yao [39]). *The comparator count $U_Y(k, d)$ for Top-k using Yao's recursion satisfies*

$$U_Y(k, d) \leq d\lceil \log(k+1) \rceil + c_k (\log d)^{\lceil \log((k+1)/3) \rceil},$$

where $c_k = O(k^{2+\lambda_k})$ and $\lambda_k = O(\log \log k)$.

For small $k \ll \sqrt{d}$, the complexity $U_Y(k, d)$ is dominated by the first term $d\lceil \log(k+1) \rceil$. This is asymptotically lower than the complexity of $O(d \log^2 k)$ (see Section 2.2). However, this is not true for $k = \Omega(\sqrt{d})$, because the second term in $U_Y(k, d)$ is asymptotically larger than $O(d \log^2 k)$.

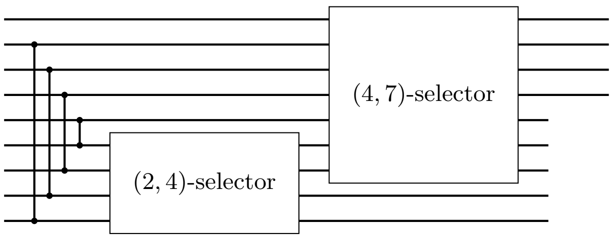

Fig. 4: Recursive construction of a $(4, 9)$-selector using Yao's method.

The pseudocode for Yao's method is given in Algorithm 3. Next to the recursion described above, multiple special cases are handled:

- If $k = 1$ or $k = 2$, we directly use the tournament method (pseudocode for the tournament method is omitted for brevity).
- We observe that if $k > d/2$, one can reduce the number of comparators by exchanging the roles of k and $d-k$: instead of finding the k smallest elements, we find the $d-k$ largest elements and return the remaining ones (made explicit in the pseudocode by the set difference on Line 8). The reverted functionality is written as YaoSwap, and the only differences compared to Algorithm 3 are the following: the output of each comparator is swapped (including those in Tour), such that the largest value is put on the top wire and the smallest value on the bottom wire; each call to Yao is replaced by YaoSwap and vice versa.

Algorithm 3 Yao's Top-k selection network

```
Input: Array $\mathbf{x}$ (of size $d > 0$) and truncation parameter $0 \leq k \leq d$
Output: Array that contains the $k$ smallest entries of $\mathbf{x}$
 1: function YAO($\mathbf{x}, k$)
 2:     if $k = 1$ then
 3:         $\mathbf{x} \leftarrow$ TOUR($\mathbf{x}$)
 4:     else if $k = 2$ then
 5:         $(\mathbf{x}_0, \mathbf{x}_2, \ldots, \mathbf{x}_{d-1}) \leftarrow$ TOUR($\mathbf{x}_0, \mathbf{x}_2, \ldots, \mathbf{x}_{d-1}$)
 6:         $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{d-1}) \leftarrow$ TOUR($\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{d-1}$)
 7:     else if $k > d/2$ then                                  ▷ Exchange $k$ with $d - k$
 8:         return $\{\mathbf{x}\} \setminus$ YAOSWAP($\mathbf{x}, d - k$)
 9:     else if $k \neq 0$ then                          ▷ Denote $\mathbf{x}_{i \ldots j} = (\mathbf{x}_i, \ldots, \mathbf{x}_j)$
10:         base $\leftarrow d - 2\lfloor d/2 \rfloor - 1$
11:         for $i \leftarrow 1$ to $\lfloor d/2 \rfloor$ do
12:             $(\mathbf{x}_{\text{base}+i}, \mathbf{x}_{d-i}) \leftarrow$ COMPARE($\mathbf{x}_{\text{base}+i}, \mathbf{x}_{d-i}$)
13:         end for
14:         $\mathbf{x}_{\lceil d/2 \rceil \ldots d-1} \leftarrow$ YAO($\mathbf{x}_{\lceil d/2 \rceil \ldots d-1}, \lfloor k/2 \rfloor$)
15:         $\mathbf{x}_{0 \ldots \lceil d/2 \rceil + \lfloor k/2 \rfloor - 1} \leftarrow$ YAO($\mathbf{x}_{0 \ldots \lceil d/2 \rceil + \lfloor k/2 \rfloor - 1}, k$)
16:     end if
17:     return TRUNCATE($\mathbf{x}, k$)
18: end function
```

2.4 Combined network

Since our truncated method is better for large $k = \Omega(\sqrt{d})$ and Yao's method is better for small k, we combine them into one oblivious Top-k network for improved performance. More specifically, the combined network recursively calls our truncated merge method or Yao's method, depending on which one uses fewer comparators. The pseudocode of this combined method differs from Algorithm 3 in only one aspect: in the "else" case on Line 9, it first computes the number of comparators for a (k, d)-selector using our truncation technique, and then uses our selector if it reduces the number of comparators with respect to the normal flow. The new pseudocode is omitted here. As Figure 5 shows, the complexity of our combined network is upper bounded by the better method of Yao's and our truncated Top-k, and slightly improves on those methods for some parameters.

3 Our k-NN protocol instantiated with TFHE

We introduce our application of Top-k to secure k-NN, which consists of two phases: computation of the squared distances and finding the k closest vectors, together with their corresponding class labels. Commonly used TFHE notations and symbols for the k-NN classification are summarized in Table 1, and necessary TFHE building blocks are introduced in Appendix B. This section describes how these two phases are realized and glued together.

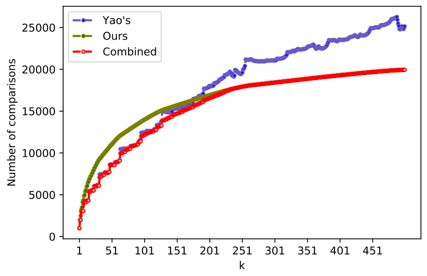

Fig. 5: $U(k,d)$ in selecting the Top-k elements out of $d = 1000$ using the network with our truncated merge, the network with Yao's recursion and the combined method. The number of comparisons is shown only for $1 \leq k \leq d/2$ as $U(k,d) = U(d-k,d)$ by symmetry.

3.1 The protocol

Squared distance computation. We generalize the method of Zuber and Sirdey [40] to compute the squared distance between a target and a model vector. 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:

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

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

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

Table 1: List of symbols for the TFHE scheme and the k-NN classification.

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

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

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

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

$$c'' = c' - 2M(X) \cdot c + M'(X). \tag{3}$$

The result computed in (3) 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 (3) and we obtain

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

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

Precision reduction (optional). The squared distances may be computed using a large plaintext modulus (t_{dist}), but the input of programmable bootstrapping (PBS) expects a small plaintext modulus (we need $t_{\mathsf{sort}} \ll 2N$). 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 $(m_i/2) \cdot \Delta_{\mathsf{sort}} = m_i \cdot \Delta_{\mathsf{dist}}$. 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 the evaluation of Section 4, we show that precision reduction is necessary for MNIST but not for the breast cancer dataset.

The Top-k selection network. To instantiate our Top-k selection network for the privacy-preserving k-NN application, we need a comparator that also outputs $\arg\min$ and $\arg\max$ (which represent the label) in addition to the minimum and maximum. This comparator is visualized in Figure 6 and its instantiation is described in Appendix B.5.

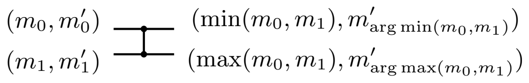
(m_0, m'_0)
(m_1, m'_1)
$(\min(m_0, m_1), m'_{\arg\min(m_0,m_1)})$
$(\max(m_0, m_1), m'_{\arg\max(m_0,m_1)})$

Fig. 6: 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.

Using the squared distance values as the scoring function, we then apply our combined Top-k network composed of augmented comparators. The output of this phase is a set of k LWE ciphertexts that encrypt the predicted class labels, which 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.

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. We refer to the noise analysis for our building blocks (Algorithm 5 [16] and Algorithm 6 [13]). Based on that, 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 $O(\log d \cdot \log k)$ 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.

Security. 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 IND-CPA secure FHE: as the data seen by the server is always encrypted, it cannot learn the privacy-sensitive input from the client, which is guaranteed by the underlying security of FHE. 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. 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. We do not consider model privacy. Observe that model privacy can never be reached perfectly, because part of the model leaks through the query result.

4 Evaluation

4.1 Implementation and experimental setup

Our prototype implementation is written in the Rust programming language using the TFHE-rs[4] library. The source code can be found on GitHub.[5] All experiments in this section are executed on machines with Intel(R) Core(TM) i9-9900 CPU @ 3.10 GHz using the Ubuntu 20.04 operating system. Our implementation supports multi-threading in the sorting network, i.e., if two comparators are on the same level in the network, then they may be executed in parallel. All aspects of Section 3 are implemented.

Our experiment uses (a reduced version of) two datasets: the MNIST[6] and breast cancer[7] datasets. We preprocess the MNIST dataset in two ways: (1) the images are downsized to 8×8 pixels which are feature vectors of length $\gamma = 64$; (2) elements in every feature vector are converted to ternary values. The breast cancer dataset has $\gamma = 32$ and we preprocess the feature vectors to use binary values. This kind of preprocessing is similar [40] where the authors also convert the MNIST images to 8×8 pixels and divide values by 300.[8]

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 average prediction/inference over 200 randomly selected test vectors using our secure k-NN algorithm.

The TFHE parameters are given in Table 2. These parameters are adapted from TFHE-rs.[9] We make a distinction between the plaintext modulus for distance computation (t_{dist}) and sorting (t_{sort}). That is, if $t_{\mathsf{dist}} \neq t_{\mathsf{sort}}$, then the precision reduction step from Section 3 needs to be used. Our definition of the plaintext modulus includes the padding bit. This extra padding bit is necessary

[4] `https://github.com/zama-ai/tfhe-rs`

[5] `https://github.com/KULeuven-COSIC/ppknn`

[6] `https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits`

[7] `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`

[8] In [40], low-precision floating point representation is used instead of integers. Dividing by 300 is similar to right shifting by 8 bits, which is a form of quantization.

[9] `https://github.com/zama-ai/tfhe-rs/blob/release/0.1.x/tfhe/src/shortint/parameters/mod.rs`

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 [1].

Table 2: The TFHE parameters used in our experiments. Note that when the homomorphic computation is done, the most significant bit is reserved for padding. Hence if $t = 2^i$, then the actual message uses $i - 1$ bits.

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 6 (g, ℓ)	$(2^{22}, 1)$
KS key—at the end of PBS (g, ℓ)	$(2^{3}, 6)$
KS key—ksk in Algorithm 5 (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	$\mathbb{Z}_2$

The sections below primarily reports the computation time and bandwidth. Memory usage is not detailed since it is not a significant overhead in our construction. Namely, our biggest experiment ($d = 1000$, $k = \sqrt{d}$) used only 700 MB of memory. Bootstrapping and key switching keys dominate the memory usage.

4.2 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 [40]. Modulo the difference in CPU (we estimate that our CPU is at most two times faster than theirs), the wall-clock time ranges from 1.7× to over 47× faster than prior work [40] while maintaining a good level of accuracy.

The main reason for our acceleration is the performance gain in the costly Top-k selection step, which is up to 10× more expensive for our parameter set. In the delta-matrix method of Zuber and Sirdey [40], 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. As such, building the matrix itself requires $(d^2 - d)/2$ comparison operations, then additional scoring operations are needed. In comparison, our Top-k network scales linearly with d and quadratically with $\log k$.

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 that 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 to 6 bits before running the selection network. The computation times prefixed with $\sim$ are estimated using extrapolation. The number of parallel threads is τ.

		Duration (s)			Comparators		Accuracy	
k	d	[40]	$\tau = 4$	$\tau = 1$	[40]	Ours	Clear	FHE
3	40	30	8.7	17.5	780	93	0.81	0.79
	175	696	31.9	78.1	15225	431	0.94	0.94
	269	1524	47.4	119.5	36046	666	0.95	0.94
	457	4248	78.9	202.3	104196	1136	0.98	0.97
	1000	$\sim$ 20837	168.0	441.1	499500	2493	0.98	0.96
5	40	$\sim$ 33	11.6	25.5	780	125	0.74	0.73
	175	$\sim$ 636	43.1	112.7	15225	598	0.92	0.90
	269	$\sim$ 1505	62.7	173.0	36046	928	0.94	0.93
	457	$\sim$ 4351	105.0	291.1	104196	1586	0.97	0.97
	1000	$\sim$ 20859	227.5	642.3	499500	3485	0.98	0.96
$\lfloor\sqrt{d}\rfloor$	40	$\sim$ 33	13.1	28.1	780	143	0.75	0.74
	175	$\sim$ 639	68.4	171.8	15225	1015	0.89	0.89
	269	$\sim$ 1516	117.7	310.4	36046	1789	0.95	0.94
	457	$\sim$ 4402	209.0	530.2	104196	3412	0.95	0.94
	1000	$\sim$ 21410	455.8	1252	499500	9121	0.98	0.97

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.

4.3 Bandwidth

Both our solution and [40] require bootstrapping for the computation, so evaluation keys should be sent at the setup phase. This costs 160 MB in our case,

Table 4: Computation time and accuracy for the breast cancer dataset. No precision reduction is performed. The computation times prefixed with $\sim$ are estimated using extrapolation. The number of parallel threads is τ.

		Duration (s)			Comparators		Accuracy	
k	d	[40]	$\tau = 4$	$\tau = 1$	[40]	Ours	Clear	FHE
3	10	4	1.8	3.2	45	18	0.94	0.92
	30	~ 18	5.0	11.5	435	68	0.94	0.94
	50	~ 51	7.4	19.0	1225	118	0.94	0.94
	200	~ 830	25.5	76.0	19900	493	0.95	0.94
5	10	~ 2	2.2	4.2	45	21	0.91	0.88
	30	~ 18	7.5	16.7	435	91	0.95	0.93
	50	~ 52	11.6	28.8	1225	161	0.96	0.95
	200	~ 831	40.2	114.6	19900	685	0.96	0.96
$\lfloor\sqrt{d}\rfloor$	200	~ 836	69.9	185.7	19900	1234	0.95	0.95

smaller than 200 MB of [40]. We use three different evaluation keys: two key switching keys (53.5 MB) and one bootstrapping key (107 MB) which takes the dominant part of the whole key size. On the other hand, the previous work uses two different bootstrapping keys. Those are necessary for the main computation, so they have larger memory consumption. We note that the evaluation key size is not considered as online bandwidth in both works, since it is only sent once and reused for the repeated computation.

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 6.7 KB. As an optimization, we can easily pack k LWE ciphertexts into an RLWE ciphertext almost for free as long as $k \leq N$ [9]. We can also reduce the size of the answer by switching the modulus $\log q$ from 64 bits to 32 bits [6], and reducing the degree of the polynomials $N = 4096$ to 1024 by key switching. The resulting answer has a size of 8 KB, which is smaller than k LWE ciphertexts for $k \geq 2$.

5 Conclusion and future directions

Top-k selection algorithms are broadly used in various applications, and secure computation highly benefits from the obliviousness of Top-k selection. We revisited the constructions by Alekseev (1969) and Yao (1980), and then proposed additional improvements for $k = \Omega(\sqrt{d})$. Our resulting combined Top-k network has complexity $O(d \log^2 k)$ in general and $O(d \log k)$ for small $k \ll \sqrt{d}$.

The efficiency of our combined Top-k network is demonstrated with an application: homomorphic k-NN classification. Compared with the state of the art [40], where oblivious Top-k was realized with complexity $O(d^2)$, our experimental results show a speedup of up to 47 times.

Future directions. In our TFHE instantiation of k-NN, to work with the restricted plaintext space, we quantize values (of 8 bits or more) down to binary or ternary values. 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. [27].

Although our combined Top-k network has the best performance compared to existing methods, it does not give the optimal asymptotic complexity $O(d \log k)$ for all parameters. Further improvements would therefore be interesting. Moreover, many secure computation applications include oblivious Top-k as a building block. It would also be interesting to incorporate our solution to improve the performance of those applications.

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, and by the Defence Advanced Research Projects Agency (DARPA) under contract No. HR001120S0032. In addition, this work is supported in part by the European Commission through the Horizon 2020 research and innovation program Belfort ERC Advanced Grant 101020005. 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 authors and do not necessarily reflect the views of the ERC, DARPA, the U.S. Government, the European Union, CyberSecurity Research Flanders or the FWO. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

The authors would also like to thank Frederik Vercauteren for giving feedback on this paper.

erc
European Research Council
Established by the European Commission

References

1. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015), http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml
2. Alekseev, V.E.: Sorting algorithms with minimum memory. Cybernetics **5**(5), 642–648 (1969)
3. 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

4. Bertels, J., Beirendonck, M.V., Turan, F., Verbauwhede, I.: Hardware acceleration of FHEW. In: Jenihhin, M., Kubátová, H., Metens, N., Raik, J., Ahmed, F., Belohoubek, J. (eds.) 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2023, Tallinn, Estonia, May 3-5, 2023. pp. 57–60. IEEE (2023). https://doi.org/10.1109/DDECS57882.2023.10139347, https://doi.org/10.1109/DDECS57882.2023.10139347
5. 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
6. 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
7. Chakraborty, O., Zuber, M.: Efficient and accurate homomorphic comparisons. Cryptology ePrint Archive, Report 2022/622 (2022), https://eprint.iacr.org/2022/622
8. 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)
9. 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
10. 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
11. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23. pp. 409–437. Springer (2017)
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.A. (eds.) Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12716, pp. 1–19. Springer (2021). https://doi.org/10.1007/978-3-030-78086-9_1, https://doi.org/10.1007/978-3-030-78086-9_1
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., Nicolas, G., Park, J.: Poster: Panacea - stateless and non-interactive oblivious RAM. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023. pp. 3585–3587. ACM (2023). https://doi.org/10.1145/3576915.3624388, https://doi.org/10.1145/3576915.3624388
17. 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
18. 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
19. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. 2001 (2009)
20. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
21. 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
22. 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
23. Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender systems: an introduction. Cambridge University Press (2010)
24. 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
25. Knuth, D.E.: The art of computer programming: Volume 3: Sorting and Searching. Addison-Wesley Professional (1998)
26. Kramer, O.: K-Nearest Neighbors, pp. 13–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38652-7_2, https://doi.org/10.1007/978-3-642-38652-7_2
27. 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
28. 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
29. 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)
30. Mitchell, M.: An introduction to genetic algorithms. MIT press (1998)

31. 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
32. 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
33. Samardzic, N., Feldmann, A., Krastev, A., Devadas, S., Dreslinski, R.G., Peikert, C., Sánchez, D.: F1: A fast and programmable accelerator for fully homomorphic encryption. In: MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021. pp. 238–252. ACM (2021). https://doi.org/10.1145/3466752.3480070, https://doi.org/10.1145/3466752.3480070
34. Samardzic, N., Feldmann, A., Krastev, A., Manohar, N., Genise, N., Devadas, S., Eldefrawy, K., Peikert, C., Sánchez, D.: Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In: Salapura, V., Zahran, M., Chong, F., Tang, L. (eds.) ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022. pp. 173–187. ACM (2022). https://doi.org/10.1145/3470496.3527393, https://doi.org/10.1145/3470496.3527393
35. Shanbhag, A., Pirk, H., Madden, S.: Efficient top-k query processing on massively parallel hardware. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 1557–1570. ACM (2018). https://doi.org/10.1145/3183713.3183735, https://doi.org/10.1145/3183713.3183735
36. 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
37. 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)
38. 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)
39. Yao, A.C.C.: Bounds on selection networks. SIAM Journal on Computing **9**(3), 566–582 (1980)
40. 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

A Data-oblivious algorithms

An oblivious algorithm can be visualized as a so-called *network*. This section explains the basics of networks and gives two examples: the tournament algorithm [19, Chapter 9] and Batcher's (d_1, d_2)-merging algorithm [25, Chapter 5].

A network comprises of interconnected *basic modules*. In the case of sorting and selection networks, this basic module is a comparator. Figure 7 shows the network of odd-even merge sort for $d = 4$, where inputs enter from the left and a vertical line compares two elements. The comparator swaps the inputs if the first one is greater than the second. 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 number of consecutive comparisons on the longest path from input to output.

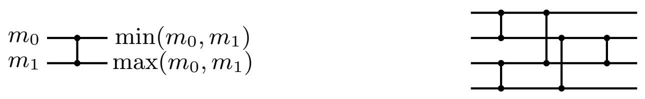

Fig. 7: The basic module and the network of odd-even merge sort for $d = 4$.

A.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 the HE literature [22,7]. 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 requires $d - 1$ comparisons and the network has depth $\lceil \log d \rceil$.

Repeating the tournament method k times, each time over the remaining data, results in a (k, d)-selector. This approach needs $O(kd)$ comparison operations and has depth $O(k \log d)$.

A.2 Batcher's merging network

A crucial component of many sorting networks (and our Top-k selection network) is a *merge* procedure. Given two sorted arrays of size d_1 and d_2, then a (d_1, d_2)-merging algorithm produces a sorted array that contains the same elements.

Batcher's merging network is specified in Algorithm 4. This algorithm is based on a recursive decomposition of the problem: first, the input arrays are split into even- and odd-index components. Then the even- and odd-index arrays are merged separately via two recursive calls. One can prove that, after these recursive calls, the smallest element is in array $\mathbf{v}$, the second and third smallest elements are either in array $\mathbf{v}$ or $\mathbf{w}$ (yet not in the same one), and so on. As such, the result can be reconstructed by pairwise comparison of the elements of the recursive calls.

Theorem 4. *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 $\mathbf{y}$
 1: function MERGE($\mathbf{x}, \mathbf{y}$)
 2:     if $d_1 \cdot d_2 = 0$ then
 3:         return $(\mathbf{x}, \mathbf{y})$
 4:     else if $d_1 \cdot d_2 = 1$ then
 5:         return COMPARE($\mathbf{x}_0, \mathbf{y}_0$)
 6:     else                                   ▷ Merge even- and odd-index components
 7:         $\mathbf{v} \leftarrow$ MERGE($(\mathbf{x}_0, \dots, \mathbf{x}_{2\lceil d_1/2 \rceil - 2}), (\mathbf{y}_0, \dots, \mathbf{y}_{2\lceil d_2/2 \rceil - 2})$)
 8:         $\mathbf{w} \leftarrow$ MERGE($(\mathbf{x}_1, \dots, \mathbf{x}_{2\lfloor d_1/2 \rfloor - 1}), (\mathbf{y}_1, \dots, \mathbf{y}_{2\lfloor d_2/2 \rfloor - 1})$)
 9:         $\mathbf{z} \leftarrow (\mathbf{v}_0, \mathbf{w}_0, \mathbf{v}_1, \mathbf{w}_1, \dots)$
10:         for $i \leftarrow 1$ to $\lfloor (\text{size}(\mathbf{z}) - 1)/2 \rfloor$ do
11:             $(\mathbf{z}_{2i-1}, \mathbf{z}_{2i}) \leftarrow$ COMPARE($\mathbf{z}_{2i-1}, \mathbf{z}_{2i}$)
12:         end for
13:         return $\mathbf{z}$
14:     end if
15: end function
16: function COMPARE($x, y$)                  ▷ Comparator module
17:     return $(\min(x, y), \max(x, y))$
18: end function
```

B TFHE building blocks

B.1 Notations

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 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 $\mathbf{g} = (1, g, \dots, g^{\ell-1})^\top$ and a gadget matrix $\mathbf{G} = \mathbf{I}_2 \otimes \mathbf{g}$, where $\otimes$ denotes the Kronecker product. 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 $\mathbf{a} \in \mathcal{R}_q^2$.

B.2 TFHE ciphertexts and basic operations

The TFHE scheme [13] uses four ciphertext types based on the (ring) learning with errors problem [32,28]. 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 $\mathsf{Err}(c)$, and we use $\mathsf{Err}(c, i)$ to refer to its i-th coefficient. The ciphertext types are:

- $\mathsf{LWE}_{\mathbf{s}}(m) = (a_1, \ldots, a_n, b) \in \mathbb{Z}_q^{n+1}$, where $b = \sum_{i=1}^{n} -a_i \cdot s_i + \Delta \cdot m + e$. 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.
- $\mathsf{RLWE.Trivial.Noiseless}(m) = (0, \Delta \cdot m) \in \mathcal{R}_q^2$ 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.
- $\mathsf{RGSW}_s(m) = \mathbf{Z} + \mathbf{G} \cdot m \in \mathcal{R}_q^{2\ell \times 2}$, 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 is 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 $\mathbf{C} = \mathsf{RGSW}_s(m_1)$ and $c = \mathsf{RLWE}_s(m_2)$, and outputs $c' = \mathbf{C}^\top \cdot \mathbf{G}^{-1}(c) = \mathsf{RLWE}_s(m_1 \cdot m_2)$.

B.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, which is similarly explained in [16]. This method is specified in Algorithm 5 and can be applied to the N LWE ciphertexts encrypting $\{m_0, \ldots, m_{N-1}\}$ separately. The RLWE outputs can be further packed into one RLWE ciphertext encrypting $m_0 + m_1 \cdot X + \ldots + m_{N-1} \cdot X^{N-1}$ as follows:

$$\sum_{i=0}^{N-1} X^i \cdot \mathsf{RLWE}_s(m_i) = \mathsf{RLWE}_s\left(\sum_{i=0}^{N-1} m_i \cdot X^i\right). \tag{5}$$

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 5. One can also take $p < N$ LWE ciphertexts when some entries of $\{m_0, \ldots, m_{N-1}\}$ are duplicated. In that case, we only need p calls to Algorithm 5.

Algorithm 5 Key switching from one LWE ciphertext to a 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
```

B.4 Programmable bootstrapping

The TFHE scheme 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 6, and we refer to Micciancio and Polyakov [29] for a detailed discussion including comparison to the FHEW accumulator.

Importantly, the secret key in Algorithm 6 depends on the ciphertext, so we introduce a different symbol for each one of them: the input ciphertext $\mathbf{c}$ is encrypted with secret key $\mathbf{z}$; the accumulator ACC is encrypted with secret key s; and the output ciphertext is encrypted with secret key $\mathbf{s}$ (which is derived from s during sample extraction). After bootstrapping, going back to the original key $\mathbf{z}$ is possible via an LWE-to-LWE key switching procedure.

Programmable bootstrapping can evaluate a negacyclic function $f : \mathbb{Z}_t \to \mathbb{Z}_t$ (i.e., it has to satisfy $f(m+t/2) = -f(m)$) on the encrypted input. 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 defined in the original paper [14]. In our case, we encode the initial accumulator as

$$\mathsf{Init}_f(b') = \mathsf{RLWE.Trivial.Noiseless}\left(T(X) \cdot X^{-b'}\right).$$

The polynomial $T(X)$ is referred to as the *test polynomial*, and it typically contains some redundancy which is necessary to decode noisy ciphertexts. In our case, we perform LWE-to-RLWE key switching to compute an encryption of $T(X)$ in order to initialize the accumulator. This requires in total $t/2$ calls to Algorithm 5 (and even fewer if some function values of f are duplicated).

Algorithm 6 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:     $b' \leftarrow \lfloor (2N/q) \cdot b \rceil$
3:     $\mathsf{ACC} \leftarrow \mathsf{Init}_f(b')$
4:     for $i \leftarrow 1$ to $n$ do
5:         $a'_i \leftarrow \lfloor (2N/q) \cdot a_i \rceil$
6:         $\mathsf{ACC} \leftarrow \mathsf{ACC} + (X^{-a'_i} - 1) \cdot (\mathsf{bk}_i \boxdot \mathsf{ACC})$
7:     end for
8:     return $\mathbf{c}' \leftarrow \mathsf{SampleExtraction}(\mathsf{ACC}, 0)$
9: end function
```

B.5 Comparison operations

Comparing two encrypted numbers can be done with programmable bootstrapping. For example, Zuber and Chakraborty [7] proposed two homomorphic comparison operators to build min and arg min functions. 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.

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 need 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 $\mathbf{c}' = \mathbf{c}_0 - \mathbf{c}_1 = \mathsf{LWE}_{\mathbf{s}}(m)$, where $m = m_0 - m_1$. This ciphertext encrypts a positive number 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 $\mathbf{c}' = \mathsf{LWE}_{\mathbf{s}}(m)$, 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 used $f(m) = -f(m - t/2) = -m_0$ for $t/4 < m < t/2$. 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.