SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search

Hao Chen
Microsoft Research
haoche@microsoft.com

Ilaria Chillotti
KU Leuven
ilaria.chillotti@kuleuven.be

Yihe Dong
Microsoft Research
Yihe.Dong@microsoft.com

Oxana Poburinnaya
Boston University
oxanapob@bu.edu

Ilya Razenshteyn
Microsoft Research
ilyaraz@microsoft.com

M. Sadegh Riazi
UC San Diego
Sadegh's email here

***Abstract*—We present new secure protocols for approximate k-nearest neighbor search (k-NNS) over the Euclidean distance in the semi-honest model, which scale to massive datasets. One of the new ingredients is a circuit for the approximate top-k selection from n numbers that is built from $O(n + \mathrm{poly}(k))$ comparators. Using this circuit as a subroutine, we design new k-NNS algorithms and two corresponding secure protocols: 1) optimized linear scan; 2) clustering-based *sublinear time* algorithm.**

The new secure protocols utilize a combination of additively-homomorphic encryption, garbled circuits and oblivious RAM. Along the way, we introduce various optimizations to these primitives, which drastically improve concrete efficiency.

We evaluate the new protocols empirically and show that they are able to handle datasets that are significantly larger than in the prior work. For instance, running on two standard Azure instances within the same availability zone, for a dataset of 96-dimensional descriptors of 10 000 000 images, we can find 10 nearest neighbors with average accuracy 0.9 in under 10 seconds improving upon prior work by at least *two orders of magnitude*.

I. Introduction

The *k-Nearest Neighbor Search problem* (k-NNS from now on) can be defined as follows. For a given dataset $X \subset \mathbb{R}^d$ lying in a d-dimensional space, and a query point $\mathbf{q} \in \mathbb{R}^d$, the goal is to find k data points closest (with respect to the Euclidean distance) to the query. To improve the search efficiency, one typically relaxes the k-NNS problem in two ways. First, one allows the answer to be *approximate* (i.e., the returned set of k points should contain most but not necessarily all of the true k closest points). Second, one may allow to *preprocess* a dataset computing some auxiliary information, which can be later used to speed up the query procedure.

The k-NNS has many applications in modern data analysis, including web search, face recognition, recommendation systems, advertisement matching, drug design, DNA analysis, plagiarism detection, motion planning, spell checking, machine learning and other areas. One typically starts with a dataset and, using domain expertise together with machine learning tools, produces *feature vector representation* of the dataset. Then, *similarity search* queries ("find k objects most similar to the query") directly translate to k-NNS queries in the feature space. Let us note, as a side remark, that one standard modern technique for producing feature vectors is to train a deep neural network and then read off the feature values from one of the layers [72]. Even though some applications of k-NNS benefit from non-Euclidean distances [6], the overwhelming majority of applications utilize Euclidean distance or cosine similarity, which can be modeled as Euclidean distance on a unit sphere. Usually such reduction to the Euclidean geometry can be done by learning an appropriate feature representation.

When it comes to applications dealing with sensitive data, such as medical, biological or financial data, the privacy of the information contained in the dataset and the queries needs to be ensured. Such settings include: face recognition [34], [62], biometric identification [35], [10], [28], patient data search in a hospital [63], [6] and many others. One can naturally pose the *Secure k-NNS* problem, where a *server* stores the dataset $X \subset \mathbb{R}^d$, and a *client* holds one or several query points $q \in \mathbb{R}^d$. We would like the client to learn k data points (approximately) closest to q such that the server learns nothing about the query or the result, while the client should not learn anything about the dataset besides the answer to the query.

From the theoretical viewpoint, problems like these are well-understood: one can use secure two-party computation protocols [68], [41] or homomorphic encryption [56], [27], [37], [36], [23], [38]. However, the known generic constructions of these primitives as of today seldom lead to practically efficient protocols for concrete tasks. As a result, secure k-NNS has been thoroughly studied on its own: see Section I-B for an overview.

In this paper, we design and evaluate two new highly-efficient and secure k-NNS protocols. The first protocol is a secure implementation of the (heavily-optimized) *linear scan*, where we compute distances from the query to all the data points, and then choose k smallest ones. The second protocol is based on a new *sublinear-time* k-NNS algorithm, which avoids computing all the distances. The new algorithm is based on *clustering:* at a very high level, during the preprocessing phase, we cluster the dataset, and then during the query stage, we search for closest points in several clusters that are the closest to the query point. Let us point out that—despite the long line of prior work on secure k-NNS—the present paper is the first, where a sublinear-time k-NNS algorithm is implemented securely.

Security guarantee. The security of approximate k-NNS can be defined in several ways. In this work, we follow the standard approach in secure two-party computation, and require that the secure protocol does not reveal more than what is revealed by the *outputs* of a *plaintext* approximate k-NNS algorithm[1].

We remark that in the clustering-based protocol, the client does learn the *hyperparameters:* for example, the total number of clusters and the number of clusters the protocol processes during the query stage (see Section III-H for more details). Even though the hyperparameters can a priori be arbitrary, the client can expect the server to set them in a way that optimizes the performance of the overall computation. Note that this situation is similar to the line of work on secure inference of neural networks (e.g. [46], [60]), where the hyperparameters of the underlying neural network, such as number of layers and number of nodes in each layer, are revealed to both parties. We leave the task of analyzing the potential leakage from the hyperparameters or hiding them from the client to future work.

We prove simulation-based security of our protocols in the semi-honest model, where both parties follow the protocol specification while trying to infer information about the input of the other party from the received messages. This is an appropriate model for parties that in general trust each other (e.g., two companies or hospitals) but need to run a secure protocol due to legal restrictions. For instance, arguably most of the cases of secure multi-party computation deployed in practice operate in the semi-honest model: computing gender pay gap [17], sugar beets auctions [19], and others. Besides, any semi-honest protocol can be reinforced to be maliciously secure (when parties can actively tamper with the sent messages), though usually it incurs a significant performance hit [40]. Thus, obtaining a semi-honest protocol for a task is a first natural step towards malicious security.

A. Our contributions

a) Plaintext approximate k-NNS algorithms tailored to secure computation: There is a huge body of work on k-NNS algorithms (both theoretical and practical): see [5], [65], [45] for an overview of the area. However, those protocols are not tailored to secure computation. For instance, consider the task of hiding the database access pattern, which is necessary to prevent the server learning information about the query. In algorithms which access all data points in a *coherent* way – e.g. in those which do a linear scan – this is not an issue; however, (non-secure) algorithms that currently perform the best [54] are not scanning the entire dataset, and therefore one would have to employ oblivious RAM (ORAM) to hide the access pattern (see Section II-A). The issue is that the best-performing k-NNS algorithms, which are based on following paths in certain carefully constructed graphs, are highly adaptive: addresses of memory accesses highly depend on the content of the previous memory accesses. Hence, when implemented securely, such algorithms would require many rounds of interaction, each protected by ORAM, which makes them inefficient. Another issue which greatly affects performance is that the algorithm from [54] and related ones needs to compute one *individual* distance at a time, rather than computing distances to large sets of points at once. This does not play well with certain optimizations, such as batching the distance computation using lattice-based AHE.

These observations give us two natural ways for solving our problem. Our first algorithm performs a linear scan of the dataset to compute all distances to the query point and returns the k closest points. To achieve good performance, we employ a number of algorithmic and implementational optimizations. In particular, we introduce an efficient circuit that performs approximate top-k selection which greatly impacts the overall search performance.

Our second algorithm has sublinear complexity in the dataset size and it is specifically designed to perform relatively few non-adaptive memory accesses and compute distances to many points at once. The starting point is a classic *clustering-based* approach, which appears in [44] and relies on the k-*means* clustering of the dataset. In short, during the query stage, we find several clusters, whose centers are closest to the query, and choose closest points from these clusters as an answer. The problem with this algorithm is that clusters found using k-means are typically highly unbalanced in cardinality, which degrades the performance since we would have to pad all clusters to the same size to avoid information leakage. In order to rectify this issue, we perform clustering iteratively at different scales, making sure the resulting clusters are balanced in size. See Section III-B for more details.

b) Approximate top-k selection: Both of our algorithms rely extensively on the top-k selection: given a sequence of n numbers of b bits each, find k smallest of them[2]. In order to implement top-k selection in a secure way, we need to design a Boolean *circuit* that performs the top-k selection. If $k = 1$, this can be easily done in optimal $O(bn)$ gates. For $k > 1$, the question becomes more interesting. In all of the prior work, only naïve circuits of sizes $O(bnk)$ or $O(b^2n)$ have been used. One can also use sorting networks and compute top-k in $O(bn \log k)$ gates. We invite the reader to Section III-C for a more thorough overview of the prior state of the art.

In this work, we show a new *randomized* circuit for top-k selection with only $O(b \cdot (n + \mathrm{poly}(k)))$ gates, which outputs the correct result with high probability. The circuit is simple and it gives a large boost to empirical performance. As a result, even our version of the linear scan already significantly improves upon the prior work on secure k-NNS for say $k = 10$. We also provide theoretical analysis of the accuracy of the circuit.

c) Mixed protocols for secure k-NNS: Both of our algorithms utilize two major subroutines: computing distances between a query and a list of points, and top-k selection. In case of clustering-based algorithm, we also require random memory accesses. We implement distance computation using lattice-based additively-homomorphic encryption (AHE), top-k selection using garbled circuits (GC) and random access via distributed oblivious RAM (DORAM). Combining all the three primitives allows us to achieve better overall performance than using only one or two of them.

For AHE, we use the SEAL library [55] which implements the Brakerski/Fan-Vercauteren (BFV) scheme [21], [36]. For garbled circuits we use our own implementation of Yao's

[1] For an alternative definition of security in this setting, see [43].

[2] Sometimes, it is useful to return IDs along with the values, but for now we ignore this issue for clarity.

protocol [68], and for DORAM we implement the Floram [32] in the read-only mode.

d) Optimizing the cryptographic primitives: We have made several optimizations to the underlying cryptorgraphic primitives to improve efficiency of our protocol. Most notably, in the Floram construction [32], we replace AES with Kreyvium [26], which yields a speed-up by more than an order of magnitude. When we use AHE for distance computation, we utilize the SIMD capabilities of the BFV scheme via coefficient-wise packing instead of the commonly used packing technique via Chinese Remainder Theorem, to simultaneously compute many distances. Our approach not only avoids expensive homomorphic rotation operations, but also allows setting the plaintext modulus to a power of two. The latter makes the top-k part of the algorithm faster compared to using a prime plaintext modulus, since we can avoid performing expensive reductions modulo a prime in garbled circuits.

e) Efficient implementation: We implement our protocols in 7400 lines of C++ code and 2300 lines of Python code and evaluate them on two datasets: SIFT [53] (1 000 000 image descriptors) and more modern Deep1B [9] (1 000 000 000 image descriptors obtained using deep neural networks, from which we subsample 1 and 10 million). We require to return 10 nearest neighbors so that on average 9 of them are correct (i.e., accuracy is 0.9, which is the level of accuracy routinely adopted in practice). We find that the clustering-based algorithm is faster than the linear scan, up to more than an order of magnitude. Yet our linear scan protocol is already faster than prior works by at least another order of magnitude due to a more efficient top-k circuit.

Overall, we show the first practically efficient secure implementation of a *sublinear-time* NNS algorithm, and our work is the first to handle datasets of the scale of tens of millions points, whereas we are not aware of any prior work, which runs secure NNS on datasets more than several thousand points. Finally, let us note that the set of primitives developed in this paper should be sufficient to implement many other k-NNS algorithms such as the ones based on locality-sensitive hashing (LSH) [4]. We plan to investigate this direction in the future work.

B. Related work

The works [34], [62], [35], [10], [28] consider the secure computation scenarios that can be mapped to the k-NNS problem for $k = 1$, with the exception that [10] returns all matches with distance below a given threshold. While these works employ different techniques, they share some common properties: first, they perform linear scan over the database. Second, these works use the Paillier AHE scheme [57] for distance computation (except for [28], which uses secret sharing schemes). In contrast, we use a more recently developed packed lattice-based AHE scheme which significantly reduces the computation cost. Moreover, all experiments done in these works have the dataset size to be at most 5 000.

Several works implemented secure algorithms tailored for NNS. The work [63] assumes that both dataset and query belong to the client and the goal is to outsource the k-NNS computation to a server, while client only pays a minimal cost per query. This was done using FHE but resulted in significant inefficiency. In contrast, our work assumes that the database belongs to the server, and both client and server are allowed to perform non-negligible computation. The work of [6] considers approximate NNS problem in a setting very similar to ours. They focus on a biological application, which requires NNS with respect to the *edit distance*. The number of points in their dataset is relatively small (at most several thousands), so the top-k selection can be done in a straightforward way (using $O(nk)$ comparisons). We explore a different regime for the NNS problem, which is arguably more relevant for practice: the number of points n is large (in the order of millions or more), the dimension d is not too high (several hundreds), and the distance of choice is Euclidean (for instance, this holds for by now standard and very popular k-NNS benchmarks [7]). In this regime, as it turns out, the top-k computation is a vast bottleneck.[3]

The work [64] implements the entire k-NNS computation using garbled circuits, which results in prohibitive network communication unless the dimension d is small (besides, they consider Hamming distance which is much more garbled circuit-friendly than the Euclidean distance[4]). The work of [59] provides a secure k-NNS solution based on the BMR protocol [13] in the *multiparty* setting where the database is distributed among different parties and another party wants to find the k nearest neighbors among all databases. Finally, the work [58] provides an extremely efficient secure NNS protocol in a different security model in which several clients use a specific hash functions and store hashes of their data on an *untrusted* server. The scheme introduces a trade-off between the search quality and an upper bound on the information leakage from hashes. In contrast, our protocols avoid any information leakage beyond the search result and the hyperparameters.

C. Organization.

In Section II, we recall some background information on the cryptographic primitives used in this work. We introduce our plaintext k-NNS algorithms in Section III and the corresponding secure protocols in section IV. We present implementation details and performance results in Section V. Finally, we conclude with discussions of future directions in Section VI.

II. Preliminaries

A. Distributed oblivious RAM (DORAM)

As we have mentioned, previous solutions for secure k-NNS require computing distance between the query point and all points in the database. This linear complexity is undesirable, in particular for large databases. In fact, this problem is ubiquitous in secure computation involving large datasets: most of the existing secure computation techniques only handle computation in the circuit model, whereas in practice, many computations are efficient in the RAM model and a direct translation of RAM programs into circuits may incur large overhead.

[3] Interestingly, when trying to implement (non-secure) k-NNS on a GPU, the top-k selection is a bottleneck as well [45].

[4] Since the Euclidean distance requires *multiplications,* which are known to be expensive in terms of the number of gates.

One of the constructions that we use in this work in order to achieve sublinear search complexity is oblivious RAM (ORAM). ORAM was first proposed by Goldreich and Ostrovsky [42] to allow a client to outsource data storage to a server, and later perform efficient read/write operations without leaking addresses to the server.

ORAM can be used in the context of secure computation, and the corresponding version is called distributed ORAM (DORAM). In this scenario, the access address is *secret-shared* among two parties that are executing the secure computation protocol and neither client nor the server know the value of the address. The goal is to retrieve data at a secret address and use it within some secure computation protocol without revealing address or data to either party. One typically requires the complexity (communication and/or computation) of the retrieval procedure to be sublinear in the database size. There are many known DORAM constructions [67], [66], [71], [33], among which we chose *Floram* [33] for efficiency reasons. In this work, we use Floram in *read-only* mode, and we further enhance its performance through careful optimizations. At a high level, we implement and use two subroutines for DORAM:

- $\mathsf{DORAM.Init}(1^\lambda, DB) \rightarrow (k_A, k_B, \overline{DB})$. This step creates a masked version of the database ($\overline{DB}$) from the plaintext version (DB) and outputs two secret keys k_A and k_B, one to each party.
- $\mathsf{DORAM.Read}(\overline{DB}, k_A, k_B, i_A, i_B)$ $\rightarrow (DB[i]_A, DB[i]_B)$.
This subroutine performs the read operation where address i is secrete-shared between two parties as $i_A \oplus i_B = i$. Both parties acquire a XOR-share of the database content $DB[i]$.

In Section IV-C, we describe these subroutines and various optimizations in a greater detail.

B. Additive homomorphic encryption (AHE)

We utilize a lattice-based additive homomorphic encryption (AHE) scheme to securely compute the Euclidean distance between two points. For our purposes, it suffices to use a private-key AHE scheme, consisting of the following randomized algorithms:

- $\mathsf{AHE.KeyGen}(1^\lambda) \rightarrow sk$. Given security parameter λ, generate a secret key used for encryption and decryption.
- $\mathsf{AHE.Enc}(sk, m) \rightarrow c$. Encrypt a message m to a ciphertext c.
- $\mathsf{AHE.Add}(c_1, c_2) \rightarrow c_3$. Given encryptions of m_1, m_2, output an encryption of $m_1 + m_2$.
- $\mathsf{AHE.CMult}(c, \mu) \rightarrow c'$. Given an encryption of m and a scalar μ, return an encryption of $m \cdot \mu$.
- $\mathsf{AHE.CAdd}(c, m') \rightarrow c'$. Given an encryption of m and a plaintext m', return an encryption of $m + m'$.
- $\mathsf{AHE.Dec}(sk, c) \rightarrow m$. Decrypt the plaintext message m.

We require our AHE scheme to satisfy standard correctness and two security properties: IND-CPA security and *circuit privacy*, which means that a ciphertext generated from Add, CAdd and CMult operations should not reveal any information about the operations to the secret key owner, other than the resulting plaintext message. This is required for privacy, since in our case the server will input its secret values into CAdd and CMult. We chose to use the BFV scheme, and we achieve circuit privacy through noise flooding in the same fashion as [46].

C. Garbled Circuits

Garbled circuit (GC) is a technique first proposed by Yao in [68] for achieving generic secure two-party computation. Abstractly, a garbled circuit of a given Boolean circuit f is a triple (F, e, d), where F is the garbled circuit, e is an encoding function and d is a decoding information. One party, the *garbler*, randomly samples (F, e, d) and encodes its input x_1 as $X_1 = e(x_1)$. Then, the two parties execute an *oblivious transfer* [47] so that the second party, the *evaluator*, obtains $X_2 = e(x_2)$ while the garbler learns nothing about x_2. Finally, the garbler sends the evaluator F, d, X_1, who then evaluates the garbled circuit, obtains $Y = F(X_1, X_2)$, and converts it to the final output $y = d(Y) = f(x_1, x_2)$ of the computation.

Many improvements to GC have been proposed in literature, such as free XORs [48] and half-gates [69]. In addition, we use the fixed-key block cipher optimization for garbling and evaluation [14]. Using Advanced Encryption Standard (AES) as the block cipher, we leverage Intel AES instructions to perform faster garbling and evaluation.

D. k-means clustering

One of our algorithms uses the k-means clustering algorithm [52] as a subroutine. It is a simple heuristic, which finds a clustering $X = C_1 \cup C_2 \cup \ldots \cup C_k$ into disjoint subsets $C_i \subseteq X$, and centers $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k \in \mathbb{R}^d$, which approximately minimize the following objective function:

$$\sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \|\mathbf{c}_i - \mathbf{x}\|^2.$$

It is immediate that for a given cluster C_i, the optimal choice of center is the *mean* of the points in C_i.

k-means clustering is implemented with repeated Lloyd iterations [52] as follows. The cluster centers $\{\mathbf{c}_i\}$ are randomly initialized in the beginning. During each iteration, each point is attached to the nearest center with respect to the Euclidean distance. Cluster centers are recalculated at the end of an iteration by averaging over the points in each cluster. The algorithm stops either when the center-assignments converge, or when a preset maximum number of iterations is reached.

III. Plaintext k-NNS algorithms

A. High-level overview

In this paper, we present efficient and secure implementations of the following two algorithms.

a) Algorithm 1*:* The first algorithm is a heavily optimized implementation of the straightforward linear scan: we compute distances from the query point to *all* the data points, and then (approximately) select k_{nn} data points closest to the query. At a high level, we will implement distance computation using AHE, while the selection step is done using garbled circuits.

To speed up this protocol, we employ the following optimization. Computing top-k naïvely would require a circuit consisting of $O(nk)$ comparators. Instead, we use a new algorithm for an approximate selection of top-k, which allows for a smaller circuit size (see section III-C) and will help us later when we implement the top-k selection securely using garbled circuits.

b) Algorithm 2*:* The second algorithm is based on the k-means clustering (see Section II-D) and, unlike the first one, has *sublinear* query time. We now give a simplified version of the algorithm, and in Section III-B we explain why this simplified version is inadequate and provide a full description that leads to efficient implementation.

At a high level, we first compute k-means clustering of the server's dataset with $k = k_{\mathrm{c}}$ clusters. Each cluster $1 \le i \le k_{\mathrm{c}}$ is associated with its *center* $\mathbf{c}_i \in \mathbb{R}^d$. During the query stage, we find $1 \le u \le k_{\mathrm{c}}$ centers that are closest to the query, where u is a parameter to be chosen. Then we compute k_{nn} data points from the corresponding u, and return IDs of these points as a final answer.

B. Balanced clustering and stash

To implement the above Algorithm 2 securely without linear cost, we use secure DORAM for retrieval of clusters. In order to prevent leaking the size of each cluster, we need to set the memory block size equal to the size of the *largest* cluster in the clustering. This can be very inefficient, if the clustering at hand is not very balanced, i.e., the largest cluster is much larger than a *typical* cluster. Unfortunately, this is exactly what we observed in our experiments. Thus, we need a mechanism to mitigate imbalance of clusters. Below we describe one such approach, which constitutes the *actual* version of Algorithm 2 we securely implement. With cluster balancing, our experiments achieve $3.3\times$ to $4.95\times$ reduction of maximum cluster sizes for different datasets.

We start with specifying the desired largest cluster size $1 \le m \le n$ and an auxiliary parameter $0 < \alpha < 1$, where n denotes the total number of data points. Then, we find the smallest k (recall k denotes the number of centers) such that in the clustering of the dataset X found by the k-means clustering algorithm at most α-fraction of the dataset lies in clusters of size more than m. Then we consider all the points that belong to the said large clusters, which we denote by X', setting $n' = |X'| \le \alpha n$, and apply the same procedure recursively to X'. Specifically, we find the smallest k such that the k-means clustering of X' leaves at most $\alpha n'$ points in clusters of size more than m. We then cluster these points etc. The algorithm terminates whenever every cluster has size $\le m$.

At the end of the algorithm, we have $\widetilde{T}$ *groups* of clusters that correspond to disjoint subsets of the dataset (as a side remark, we note that one always has $\widetilde{T} \le \log_{1/\alpha} n$). We denote the number of clusters in the i-th group by k_{c}^i, the clusters themselves by $C_1^i, C_2^i, \ldots, C_{k_{\mathrm{c}}^i}^i \subseteq X$ and their centers by $c_1^i, c_2^i, \ldots, c_{k_{\mathrm{c}}^i}^i \in \mathbb{R}^d$. During the query stage, we find u^i clusters from the i-th group with the centers closest to the query point, then we retrieve all the data points from the corresponding $\sum_{i=1}^{\widetilde{T}} u^i$ clusters, and finally from these retrieved points we select k_{nn} data points that are closest to the query.

We now describe one further optimization that helps to speed up the resulting k-NNS algorithm even more. Namely, we collapse last several groups into a special set of points, which we call a *stash*, denoted by $S \subseteq X$. Unlike clusters from the remaining groups, we perform *linear scan* on the stash. We denote $s = |S|$ the stash size and $T \le \widetilde{T}$ the number of remaining groups of clusters that are not collapsed.

The motivation for introducing the stash is that the last few groups are usually pretty small, so in order for them to contribute to the overall accuracy meaningfully, we need to retrieve most of the clusters from them. But this means many DORAM accesses which are less efficient than the straightforward linear scan.

Note that while the simplified version of Algorithm 2 from Section III-A is well-known and very popular in practice (see, e.g., [44], [45]), our modification of the algorithm in this section, to the best of our knowledge, is new. Let us reiterate that the clustering-based k-NNS algorithms are *not* the fastest on the CPU[5], but they are a perfect match for secure computation.

C. Approximate top-k selection

In both of our algorithms, we rely extensively on the following *top-k selection* subroutine: given a list of n numbers $x_1, x_2, \ldots, x_n$, find $k \le n$ smallest list elements in the sorted order. Let us denote the corresponding function, which outputs a tuple of size k, by $\mathrm{MIN}_n^k(x_1, x_2, \ldots, x_n)$. In the RAM model, computing MIN_n^k is a well-studied problem, and it is by now a standard fact that it can be computed in time $O(n + k \log k)$ [18]. However, to perform top-k selection securely, we need to implement it as a Boolean *circuit*. Suppose that all the list elements are b-bit integers. Then the desired circuit has bn inputs and bk outputs. To improve efficiency, it is desirable to design a circuit for MIN_n^k with as few gates as possible.

a) The naïve construction: A naïve circuit for MIN_n^k performs $O(nk)$ comparisons and hence consists of $O(bnk)$ gates. Roughly, it keeps a sorted array of the current k minima. For every x_i, it uses a "for" loop to insert x_i into its correct location in the array, and discards the largest item to keep it of size k.

b) Sorting networks: Another approach is to first sort the array and then take the first k elements. We could use a sorting network such as AKS [1], with $O(bn \log n)$ gates, which is better than the naïve bound whenever $k \gg \log n$. The number of gates can be further reduced to $O(bn \log k)$ by splitting the input array into subsets of size k, and then

[5] However, they are known to be extremely efficient on GPU [45] due to reasons similar to the ones considered in this paper.

repeatedly merging two subsets into one of size k consisting of the k smallest elements from the union of the two arrays. The merge operation can be done in $O(bk \log k)$ gates using the AKS sorting network, and we need to perform it $O(n/k)$ times, which gives a total of $O(nk \log k)$ gates. This is asymptotically better than the naïve method for any super-constant value of k. However, the hidden constant factor for AKS sorting network is prohibitively high. At the same time, Batcher's sorting network [11] has slightly worse complexity $O(n \log^2 n)$, but very good concrete efficiency. Plugging it into the above construction yields the bound of $O(bn \log^2 k)$ gates, which is still better than the naïve approach for super-constant k.

c) Approximate randomized selection: We are not aware of any circuit for MIN_n^k with $O(bn)$ gates unless k is a constant (such bound—if true—would have been optimal, since the input size is bn bits). Instead, we propose a *randomized* construction of a circuit with $O(bn)$ gates. We start with shuffling the inputs in a *uniformly random order*. Namely, instead of $x_1, x_2, \ldots, x_n$, we consider the list $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$, where π is a uniformly (pseudo-)random permutation of $\{1, 2, \ldots, n\}$. We require the output to be "approximately correct" (more on the precise definitions later) with high probability over π for every particular list $x_1, x_2, \ldots, x_n$.

We proceed by partitioning the input list into $l \leq n$ bins of size n/l as follows:

$$\begin{aligned}
&U_1 = \{x_{\pi(1)}, \ldots, x_{\pi(n/l)}\}, \\
&U_2 = \{x_{\pi(n/l+1)}, \ldots, x_{\pi(2n/l)}\}, \\
&\ldots, \\
&U_l = \{x_{\pi((l-1)n/l+1)}, \ldots, x_{\pi(n)}\}.
\end{aligned}$$

Our circuit works in two stages: first, we compute the minimum within each bin $M_i = \min_{x \in U_i} x$, then we output $\mathrm{MIN}_l^k(M_1, M_2, \ldots, M_l)$ as a final result using the naïve circuit for MIN_l^k. The circuit size is $O(b \cdot (n + kl))$, which is $O(bn)$ whenever $kl = O(n)$.

Intuitively, if we set the number of bins l to be large enough, the above circuit should output a high-quality answer with high probability over π. We state and prove two theorems formalizing this intuition in two different ways. We defer the proofs to Appendix C.

Theorem 1. *There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ such that for every n, $0 < \delta < \delta_0$, and $k \geq k_0(\delta)$, one can set the number of bins $l = k/\delta$ such that the intersection $\mathcal{I}$ of the output of our circuit with $\mathrm{MIN}_n^k(x_1, x_2, \ldots, x_n)$ contains at least $(1-\delta)k$ entries in expectation over the choice of π.*

This bound yields a circuit of size $O(b \cdot (n + k^2/\delta))$.

Theorem 2. *There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ such that for every n, $0 < \delta < \delta_0$, and $k \geq k_0(\delta)$, one can set the number of bins $l = k^2/\delta$ such that the output of our circuit is* exactly $\mathrm{MIN}_n^k(x_1, x_2, \ldots, x_n)$ *with probability at least $1-\delta$ over the choice of π.*

This yields a circuit of size $O(b \cdot (n + k^3/\delta))$, which is worse than the previous bound, but the corresponding correctness guarantee is stronger.

In some applications, it is enough to output a binary vector of length n with exactly k ones on the positions that correspond to the k smallest entries of the list. It was known how to do this in $O(b^2 n)$ gates [24], and we show how to improve this bound to the optimal $O(bn)$ gates. Such a circuit can be used for the linear scan, but for the clustering-based algorithm, we need to return k smallest entries explicitly. Due to this requirement and also the fact that the new $O(bn)$-sized circuit has a higher hidden constant than the above randomized construction, we decided not to implement it. For completeness and for the future reference, we describe the new circuit in Appendix D.

D. Approximate distances

To speed up the top-k selection further, instead of exact distances, we will be using *approximate* distances. Namely, instead of storing full b-bit distances, we discard r low-order bits, and the overall number of gates in the selection circuit becomes $O((b - r) \cdot (n + kl))$.

For the clustering-based algorithm, we set r differently when we select closest cluster centers and when we select closest data points, which allows for a more fine-grained parameter tuning.

E. Putting it together

We now give a high-level summary of our algorithms and in the next two sections we provide a more detailed description. For the linear scan, we use the approximate top-k selection to return the k_{nn} IDs after computing distances between query and all points in the database.

For the clustering-based algorithm, we use approximate top-k selection for retrieving u^i clusters in i-th group for all $i \in \{1, \ldots, T\}$. Then, we compute the closest k_{nn} points from the query to all the retrieved points using the naive algorithm. Meanwhile, we compute the approximate top-k with $k = k_{\mathrm{nn}}$ between query and the stash. Finally, we compute and output the k_{nn} closest points from the above $2k_{\mathrm{nn}}$ candidate points.

Note that in the clustering-based algorithm, we use exact top-k selection for retrieved points and approximate selection for cluster centers and stash. The main reason is that the approximate selection requires a random shuffle of the input values. The corresponding permutation can be known only by the server and not by the client to ensure that there is no additional leakage when the algorithm is implemented securely. Jumping ahead to the secure protocol in the next section, the points we retrieve from the clusters will be secret-shared. Thus, performing approximate selection on retrieved points would require a secure two-party shuffling protocol, which is expensive. Therefore, we run a naïve circuit for exact computation of top-k for the retrieved points.

F. Hyperparameters

Here we list the hyperparameters used by our algorithms. See Figure 4 and Figure 5 for the values that we use for various datasets.

Main hyperparameters:

- n is the number of data points

- d is the dimension
- k_{nn} is the number of data points we need to return as an answer
- T is the number of *groups* of clusters
- k_{c}^i is the total number of clusters for the i-th group, $1 \leq i \leq T$
- m is the *largest* cluster size
- u^i is the number of closest clusters we retrieve for the i-th group, $1 \leq i \leq T$
- $u_{\text{all}} = \sum_{i=1}^{T} u^i$ is the total number of clusters we retrieve.
- s is the *stash* size
- l^i is the number of bins we use to speed up the selection of closest clusters for the i-th group, $1 \leq i \leq T$
- l_{s} is the number of bins we use to speed up the selection of closest points for the stash
- b_{c} is the number of bits necessary to encode one *coordinate*
- b_{d} is the number of bits necessary to encode one *distance* ($b_{\text{d}} = 2b_{\text{c}} + \lceil \log_2 d \rceil$)
- b_{cid} is the number of bits necessary to encode the ID of a *cluster* ($b_{\text{cid}} = \left\lceil \log_2 \left(\sum_{i=1}^{T} k_{\text{c}}^i \right) \right\rceil$)
- b_{pid} is the number of bits necessary to encode the ID of a *point* ($b_{\text{pid}} = \lceil \log_2 n \rceil$)
- r_{c} is the number of bits we discard when computing distances to *centers of clusters*, $0 \leq r_{\text{c}} \leq b_{\text{d}}$
- r_{p} is the number of bits we discard when computing distances to *points*, $0 \leq r_{\text{p}} \leq b_{\text{d}}$

Additional hyperparameters:

- α is the allowed fraction of points in large clusters during the preprocessing
- N is the ring dimension in BFV scheme;
- q is the ciphertext modulus in BFV scheme;
- $t = 2^{b_{\text{d}}}$ is the plaintext modulus in BFV scheme and the modulus for secret-shared distances.

G. Pseudocode

We now present the pseudocode for plaintext algorithms. The algorithms use functions MIN (which returns the smallest element and its ID) and NAIVETOPK (which returns k smallest elements together with their IDs). We refer to Section III-F for the hyperparameters used in the below pseudocode. For a point $p \in X$, we denote $\text{ID}(p)$ its ID.

Algorithm 1 Plaintext linear scan

```
function PLAINLINEARSCANKNNS($\mathbf{q}$)
    # The algorithm depends on: $r_{\text{p}}$, $k_{\text{nn}}$, $l_{\text{s}}$
    for $i \leftarrow 1, \ldots, n$ do
        $d_i \leftarrow \|\mathbf{q} - \mathbf{p}_i\|^2$
        $d_i \leftarrow \lfloor \frac{d_i}{2^{r_{\text{p}}}} \rfloor$
    end for
    $(v_1, \text{ID}_1), \ldots, (v_{k_{\text{nn}}}, \text{ID}_{k_{\text{nn}}}) \leftarrow$
        $\leftarrow$ APPROXTOPK$((d_1, 1), \ldots, (d_n, n), k_{\text{nn}}, l_{\text{s}})$
    return $\text{ID}_1, \ldots, \text{ID}_{k_{\text{nn}}}$
end function
```

Algorithm 2 Plaintext clustering-based algorithm

```
function PLAINCLUSTERINGKNNS($\mathbf{q}$)
    # The algorithm depends on:
    # partition into clusters $C_j^i$ and the stash $S$
    # $k_{\text{nn}}$, $r_{\text{p}}$, $r_{\text{c}}$, $u^i$, $l^i$, $l_{\text{s}}$
    for $i \leftarrow 1, \ldots, T$ do
        for $j \leftarrow 1, \ldots, k_{\text{c}}^i$ do
            $d_j^i \leftarrow \|\mathbf{q} - \mathbf{c}_j^i\|^2$
            $d_j^i \leftarrow \lfloor \frac{d_j^i}{2^{r_{\text{c}}}} \rfloor$
        end for
        $(v_1, \text{ID}_1^i), \ldots, (v_{u^i}, \text{ID}_{u^i}^i) \leftarrow$
            $\leftarrow$ APPROXTOPK$((d_1^i, 1), \ldots, (d_{k_{\text{c}}^i}^i, k_{\text{c}}^i), u^i, l^i)$
    end for
    $C \leftarrow \bigcup_{1 \leq i \leq T} \bigcup_{1 \leq j \leq u_i} C_{\text{ID}_j^i}^i$
    for $\mathbf{p} \in C \cup S$ do
        $d_{\mathbf{p}} \leftarrow \|\mathbf{q} - \mathbf{p}\|^2$
        $d_{\mathbf{p}} \leftarrow \lfloor \frac{d_{\mathbf{p}}}{2^{r_{\text{p}}}} \rfloor$
    end for
    $(a_1, \widetilde{\text{ID}}_1), \ldots, (a_{k_{\text{nn}}}, \widetilde{\text{ID}}_{k_{\text{nn}}}) \leftarrow$
        $\leftarrow$ NAIVETOPK$(\{(d_{\mathbf{p}}, \text{ID}(\mathbf{p}))\}_{\mathbf{p} \in C}, k_{\text{nn}})$
    $(a_{k_{\text{nn}}+1}, \widetilde{\text{ID}}_{k_{\text{nn}}+1}), \ldots, (a_{2k_{\text{nn}}}, \widetilde{\text{ID}}_{2k}) \leftarrow$
        $\leftarrow$ APPROXTOPK$(\{(d_{\mathbf{p}}, \text{ID}(\mathbf{p}))\}_{\mathbf{p} \in S}, k_{\text{nn}}, l_{\text{s}})$
    $(v_1, \widehat{\text{ID}}_1), \ldots, (v_{k_{\text{nn}}}, \widehat{\text{ID}}_{k_{\text{nn}}})) \leftarrow$
        $\leftarrow$ NAIVETOPK$((a_1, \widetilde{\text{ID}}_1), \ldots, (a_{2k_{\text{nn}}}, \widetilde{\text{ID}}_{2k_{\text{nn}}}), k_{\text{nn}})$
    return $\widehat{\text{ID}}_1, \ldots, \widehat{\text{ID}}_{k_{\text{nn}}}$
end function
```

Algorithm 3 Approximate top-k selection

```
function APPROXTOPK($(x_1, \text{ID}_1), \ldots, (x_n, \text{ID}_n)$, $k$, $l$)
    $\pi \leftarrow$ random permutation of $\{1, 2, \ldots, n\}$
    for $i \leftarrow 1 \ldots l$ do
        $(M_i, \widetilde{\text{ID}}_i) \leftarrow$
            $\leftarrow$ MIN$(\{(x_{\pi(i \cdot n/l+j)}, \text{ID}_{\pi(i \cdot n/l+j)})\}_{j=1}^{n/l})$
    end for
    return NAIVETOPK($(M_1, \widetilde{\text{ID}}_1), \ldots, (M_l, \widetilde{\text{ID}}_l)$, $k$)
end function
```

H. What does the output of our algorithms leak?

We briefly discuss the potential leakage from the *output* of our k-NNS algorithms. Note that this discussion is independent from the privacy guarantee of our secure protocols, which have no leakage beyond the output. We note that quantifying the exact amount of leaked information is likely to be challenging and we leave this (admittedly, very important) question for the future work.

One can naturally consider what the exact answer for a k-NNS query leaks. For example, [50] shows that for *low-dimensional* datasets, one can approximately reconstruct the database after issuing sufficiently many k-NNS queries. Here, we remark that the current techniques of database recovery from k-NNS query results still do not scale well to high-dimension data considered in this work. We also note that by asking many queries *adaptively,* the client can recover the k-NN graph of the dataset, which contains lots of valuable information about the data, including community structure. To prevent this, one needs to restrict the client in the number of queries and the allowed degree of adaptivity.

Next, the leakage can occur due to our answers being approximate. For instance, just asking the same query several times, we will be potentially receiving different answers due to the randomness used in the approximate top-k selection.

IV. SECURE PROTOCOL FOR k-NNS

In this section, we describe the new secure protocols. For the formal specification, see Appendix F. For the security proofs, see Appendix E.

A. Overview of our protocol

We give a high-level overview of our secure protocols implementing the plaintext algorithms from the previous section, followed by description of individual subroutines (AHE, GC, and DORAM). We start with the clustering-based protocol. For the illustration of the protocol, see Figure 1.

Recall that the server's input to the protocol is a partition of the dataset X into clusters and a stash

$$X = [\bigcup_{i=1}^{T} \bigcup_{j=1}^{k_{\mathrm{c}}^{i}} C_j^i] \cup S,$$

such that each cluster C_i^j has size at most m. Let c_j^i denote the center of C_j^i. Our protocol works in the following stages:

Setup. The server and the client execute $\mathsf{DORAM.Init}$ to insert all clusters from all groups into DORAM, with one cluster in each block. Clusters are padded by very far points to reach size m.

Query. This stage consists of the following steps.

1) The server performs $T + 1$ independent random shuffles necessary for the approximate top-k: on each of the T groups of the cluster centers and stash points.
2) For each group of clusters $i \in \{1, \ldots, T\}$,
    - The client and server use AHE with noise flooding to compute secret shares of $d_j^i = ||\mathbf{q} - \mathbf{c}_j^i||^2$ for all j.
    - Client and server run approximate top-k selection algorithm from Section III-C using garbled circuits, with $k = u_i$, and output secret shares of u_i cluster indices. Before running top-k, distances are truncated by r_{c} bits (within the circuit).
3) Client and server input the secret shares of the $u_{\mathrm{all}} = \sum_{i=1}^{T} u_i$ indices $(i_1, j_1), \ldots, (i_{u_{\mathrm{all}}}, j_{u_{\mathrm{all}}})$ obtained in previous step into $\mathsf{DORAM.Read}$ to retrieve all points in $C := C_{j_1}^{i_1} \cup \cdots \cup C_{j_{u_{all}}}^{i_{u_{all}}}$ in secret-shared form.
4) Use AHE with noise flooding to compute secret shares of distances between q and all points in $C \cup S$.
5) Use garbled circuit to securely evaluate a naïve top-k circuit, compute secret shares of IDs and distances of k_{nn} closest points in C to the query. The distances are truncated by r_{p} bits.
6) Use garbled circuit to securely evaluate the approximate top-k circuit from Section III-C to compute secret shares of IDs and distances of k_{nn} closest points in S to the query. The distances are truncated by r_{p} bits.
7) Use garbled circuit to evaluate the naïve top-k selection circuit which takes as input secret shares of the above $2k_{\mathrm{nn}}$ points (and truncated distances) and outputs the IDs of the closest k_{nn} points to the client's query.

Now, our linear scan protocol can be obtained by setting the stash equal to the entire database, i.e. $S = X$, and skipping the clustering and DORAM altogether. Then, we execute step (3) and (6) in the above query procedure to obtain a list of k_{nn} IDs.

B. Distance computation from AHE

It is well-known that secure computation of Euclidean distances can be done using AHE. Among the existing AHE schemes, we select the lattice-based Brakerski/Fan-Vercauteren (BFV) scheme [22], [36] with the nice property that it supports efficient single-instruction-multiple-data (SIMD) operations on encrypted vectors. This allows us to compute distances from the query point to many points of the dataset at once. The idea of using the BFV scheme to perform fast secure linear operations is in the same spirit as [46]. However, compared to [46], our approach avoids expensive ciphertext *rotations*. Also, we modify the SIMD encoding technique to fit our scenario, notably removing the restriction on the plaintext modulus and perform computation modulo a power of two instead. The benefit of computation modulo powers of two (as opposed to modulo prime p) is that it allows us to later avoid a costly addition modulo p transformation inside a garbled circuit when reconstructing distances from secret shares. Thus, our approach is more efficient and more compatible with the garbled circuit components of our protocols.

More precisely, in order to enable SIMD operation such as elementwise multiplication of vectors in the BFV scheme, we need to work with plaintexts consisting of integers modulo some prime $p \equiv 1 \mod 2N$, where N is the ring dimension parameter. However, we observe that our distance computation protocol only requires efficient multiplication between *scalars* and vectors. Therefore we can drop the requirement on the

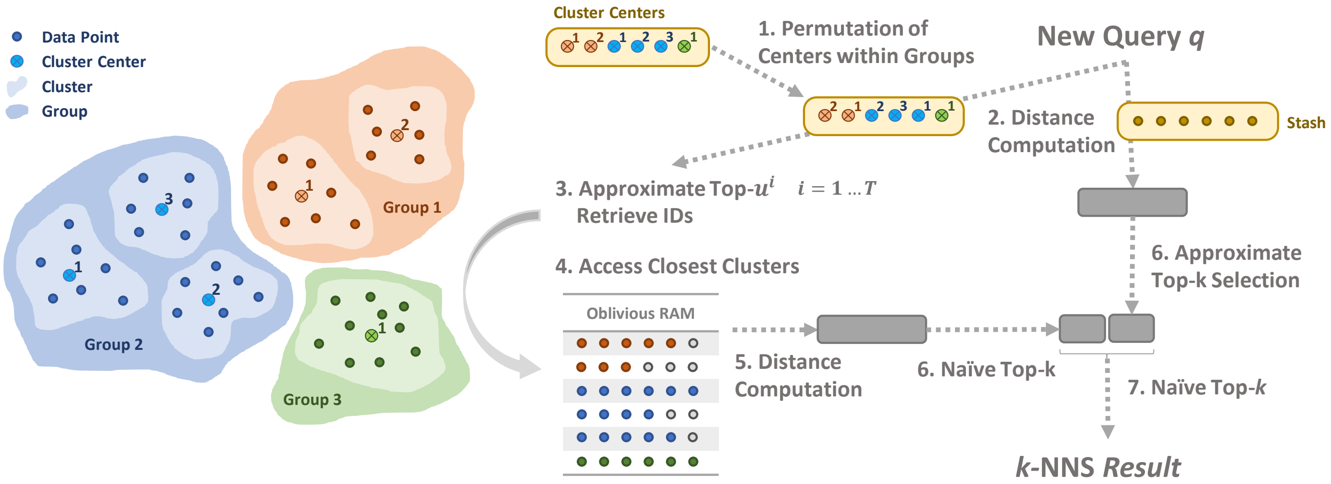


Fig. 1: Global computation and data flow of SANNS.

plaintext modulus and perform computation modulo some power of two without losing efficiency.

We now describe our distance computation protocol in more detail. Recall that plaintext space of the BFV scheme is a polynomial ring $R_t := \mathbb{Z}_t[x]/(x^N + 1)$, where we take N to be a power of 2 and t an integer modulus. So a plaintext is represented as a polynomial with degree less than N and coefficients in $\mathbb{Z}_t$. Then, the client encodes each coordinate of the query separately into the constant coefficient in R_t, i.e., we encode the query $\mathbf{q} = (\mathbf{q}[1], \dots, \mathbf{q}[d]) \in \mathbb{R}^d$ as $f_i = \mathbf{q}[i] + 0 \cdot x + \cdots + 0 \cdot x^{N-1}$ for each $1 \leq i \leq d$. For the sake of simplicity, assume that the server has exactly N points $\mathbf{p}_1, \dots, \mathbf{p}_N$. It encodes these points into d plaintexts, each encoding one coordinate of all points, resulting in

$$g_i = p_{1,i} + p_{2,i}x + \cdots + p_{N,i}x^{N-1}, \quad 1 \leq i \leq d.$$

Then, we could verify that $\sum_{i=1}^{d} f_i g_i = \sum_{j=1}^{N} \langle \mathbf{q}, \mathbf{p}_j \rangle x^{j-1}$. That is, we could compute N dot products using d homomorphic scalar multiplications and additions. Our protocol works by letting the client encrypt each f_i into a ciphertext c_i and send to the server; then the server uses AHE.CMult and AHE.Add to compute a ciphertext encrypting $h(x) = \sum_{j=1}^{N} \langle \mathbf{q}, \mathbf{p}_j \rangle x^{j-1}$. The server then samples a random polynomial $r(x)$ and uses AHE.CAdd to compute encryption of $h(x) + r(x)$, which it sends back to the client. The client then decrypts the ciphertext to obtain $h(x) + r(x)$, and the server keeps $r(x)$; in other words, the client and the server hold secret shares of $\langle \mathbf{q}, \mathbf{p}_j \rangle$ modulo t. Then, secret shares of Euclidean distances can be reconstructed via local operations, using the identity $\|\mathbf{q} - \mathbf{p}_j\|^2 = \|\mathbf{q}\|^2 - 2\langle \mathbf{q}, \mathbf{p}_j \rangle + \|\mathbf{p}_j\|^2$.

We need to slightly modify the above routine when computing distances of points retrieved from DORAM. Here the server does not hold points in the clear: instead, the client and server secret share the points and their squared Euclidean norms. We use $\langle x \rangle_C$ and $\langle x \rangle_S$ to denote the client and server's shares of a private input x, such that $x = \langle x \rangle_C + \langle x \rangle_S \mod t$. Then, we only need to compute dot products securely between $\mathbf{q}$ and each $\langle \mathbf{p}_j \rangle_S$, since

$$\begin{aligned} &\|\mathbf{q} - \mathbf{p}_j\|_2^2 \\ &= \|\mathbf{q}\|_2^2 - 2\langle \mathbf{q}, \langle \mathbf{p}_j \rangle_C \rangle + \langle \|\mathbf{p}_j\|_2^2 \rangle_C - 2\langle \mathbf{q}, \langle \mathbf{p}_j \rangle_S \rangle + \langle \|\mathbf{p}_j\|_2^2 \rangle_S. \end{aligned}$$

C. Point Retrievals Using DORAM

To retrieve points from given clusters, we use *Floram*, a DORAM construction proposed by Doerner and Shelat [32]. Here, we briefly explain the functionality of Floram and refer the reader to the original paper [32] for details.

In Floram, both parties hold *identical* copies of the masked database. Let the plaintext database be DB, block at address i be $DB[i]$, and the masked database be $\overline{DB}$. We set:

$$\overline{DB}[i] = DB[i] \oplus PRF_{k_A}(i) \oplus PRF_{k_B}(i),$$

where PRF is a pseudo-random function, k_A is a secret key owned by A and k_B is similarly owned by B. At a high level, Floram's retrieval functionality consists of the two main parts: token generation using Functional Secret Sharing (FSS) [39] and data unmasking from the PRFs. In Floram, FSS is used to securely generate two bit vectors (one for each party) u^A and u^B such that individually they look random, yet $u_j^A \oplus u_j^B = 1$ iff $j = i$, where i is the address we are retrieving. Then, party A computes $\bigoplus_j u_j^A \cdot \overline{DB}[i]$ and, likewise, party B computes $\bigoplus_j u_j^B \cdot \overline{DB}[i]$. The XOR of these two values is simply $\overline{DB}[i]$. To recover the desired value $DB[i]$, the parties use a garbled circuit to compute the required PRFs and XOR to remove the masks[6].

We implemented Floram with a few optimizations. The first two are proposed by the Floram paper itself. The third optimization reduces the overhead of FSS evaluation. We also propose to use a PRF based on Kreyvium [26] instead of AES as was done in [33]. Last but not least, we reduce the number of interactions between two parties when accessing the database at several different indices. We discuss these optimizations below.

[6] The retrieved block can be either returned to one party, or secret-shared between the parties via the same garbled circuit

Constant PRG (CPRG). The bottleneck of FSS is many evaluations of a pseudo-random generator (PRG) within GC. Doerner and Shelat [32] propose an optimization that replaces secure evaluation of PRG with $\log_2 n$ simple secure computations. As a result of this technique, the round complexity is increased to $\log_2 n$ per access but all PRG evaluations (required by FSS) are performed in plaintext.

Tree trimming. The second optimization proposed for *read* operations in Floram is to avoid evaluating last several layers in FSS tree at a cost of small computational overhead. However, the overhead is quickly paid off due to the exponential growth of the last layers in FSS tree. We refer the reader to the Floram paper [32] for a more detailed explanation.

Precomputing OT. Recall that with CPRG technique, two parties have to execute the GC protocol $\log_2 n$ times iteratively which in turn requires $\log_2 n$ set of Oblivious Transfers (OTs). Performing consecutive OTs can significantly slow down the FSS evaluation. In order to mitigate the overhead, we use Beaver OT precomputation protocol [12] which allows to perform all necessary OTs on random values in the beginning of FSS evaluation with a very small additional communication for each GC invocation.

Kreyvium as PRF. In original Floram, PRF is implemented using Advanced Encryption Standard (AES). While computing AES is fast in plaintext due to Intel AES instructions, it requires many AND gates to be evaluated within a garbled circuit. Thus, we propose a more efficient solution based on stream ciphers. In particular, we implement our PRF using Kreyvium [26] which requires significantly fewer number of AND gates (see Appendix B for various related trade-offs). However, evaluating Kreyvium in plaintext during the initial database masking adds large overhead compared to AES. To mitigate the overhead, we pack multiple (512 in our case) invocations of Kreyvium and evaluate them simultaneously by using AVX-512 instructions provided by Intel CPUs.

Multi-address access. All of the aforementioned optimizations improve the performance of a single access. Accessing the database at k different locations, requires $k \log_2 n$ number of interactions. If these memory accesses are non-adaptive, then the same process can be implemented much more efficiently by fusing all of the access procedures reducing the number of rounds to merely $\log_2 n$.

D. Top-k selection using Garbled Circuits

For the top-k selection, the client and server start with secret shares of n distances and IDs. At a high level, we implement secure top-k selection by plugging the circuit described in Section III-C into Yao's garbled circuits [68]. We make some further optimizations to improve the performance. First, instead of working with exact distances, we truncate them, which allows us to reduce the circuit size significantly (see Section III-D). The truncation is done by simply discarding some lower order bits after adding the secret shares in the garbled circuit. The second optimization comes from the implementation side. Using generic MPC frameworks such as ABY [28] ends up being problematic for us, since such frameworks require storing the entire circuit explicitly with accompanying bloated data structures. However, our top-k circuit is highly structured (i.e., it is a composition of a certain small circuit with itself many times), which allows us to work with it looking at one small part at a time. This means that the memory consumption of the garbling and the evaluation algorithms is essentially independent of n, which makes them much more cache-efficient and as a result much faster. To accomplish this, we use our own GC implementation with most of the standard optimizations [13], [49], [14], [70][7], which allows us to save more than an order of magnitude in both time and memory usage compared to ABY.

V. Implementation and Performance Results

A. Environment

We perform the evaluation on two Azure `F72s_v2` instances (with 72 *virtual* CPUs and 144 GB of RAM each) hosted in the "West US 2" availability zone. We evaluate our algorithms in 1 and 72 threads (for the query procedure, the preprocessing and OT phases are always single-thread). We implement networking using ZeroMQ: the latency between instances ends up being around 0.5 ms, while the throughput ranges between 374 MB/s on a single thread and 3.30 GB/s on 72 threads. We also perform an experiment on two instances hosted in "West US 2" and "East US" availability zones. In that case, the networking is a good deal slower: the latency is 34 ms and the throughput ranges between 36 MB/s for a single thread, and 2.0 GB/s for 72 threads. We use g++ 7.3.0, Ubuntu 18.04, SEAL 2.3.1 [55] and libOTe [61] for the OT phase (in the single-thread mode due to unstable behavior when run in several threads). We implement balanced clustering as described in Section III-B using PyTorch and run it on four NVIDIA Tesla V100 GPUs. It is done once per dataset and takes several hours (with the bottleneck being the vanilla k-means clustering described in Section II-D).

B. Datasets

We evaluate our algorithms as well as baselines on two datasets: SIFT ($n = 1\,000\,000$, $d = 128$) is a standard dataset of image descriptors [53] that can be used to compute similarity between images; Deep1B ($n = 1\,000\,000\,000$, $d = 96$) is also a dataset of image descriptors [9], which is more modern and are feature vectors obtained by passing images through a deep neural network (for more details see the original paper [9]). We conduct the evaluation on two subsets of Deep1B that consist of the first $1\,000\,000$ and $10\,000\,000$ images, which we label Deep1B-1M and Deep1B-10M, respectively. SIFT comes with $10\,000$ sample queries which we use for evaluation; for Deep1B-1M and Deep1B-10M, we use a sample of $10\,000$ data points, which we remove from the dataset, as queries. For all the datasets we use Euclidean distance to measure similarity between points. Note that the Deep1B-1M and Deep1B-10M datasets are normalized to lie on the unit sphere.

Note that both SIFT and Deep1B have been extensively used in nearest neighbors benchmarks. In particular, SIFT is a part of ANN Benchmarks [8], where a large array of NNS algorithms has been thoroughly evaluated. Deep1B has been used for evaluation of NNS algorithms in, e.g., [9], [45], [54] and a number of other places.

[7] For oblivious transfer, we use libOTe [61]

	Threads	Algorithm	Overall query	ORAM	Top-k	Distances	OT phase	Preprocessing
SIFT	1	Linear scan	35.4 s 4.52 GB	None	15.6 s 4.42 GB	19.8 s 98.8 MB	2.99 s 950 MB	None
		Clustering	8.63 s 1.77 GB	4.38 s 1.07 GB	1.98 s 645 MB	2.22 s 56.7 MB	0.63 s 166 MB	12.9 s 484 MB
	72	Linear scan	6.15 s 4.52 GB	None	2.54 s 4.42 GB	3.08 s 98.8 MB	N/A	None
		Clustering	**2.36 s** 1.79 GB	0.92 s 1.07 GB	1.00 s 666 MB	0.35 s 56.7 MB	N/A	N/A
Deep1B-1M	1	Linear scan	30.0 s 4.50 GB	None	15.1 s 4.42 GB	14.9 s 86.2 MB	3.07 s 950 MB	None
		Clustering	7.44 s 1.59 GB	3.87 s 921 MB	1.86 s 621 MB	1.67 s 44.1 MB	0.59 s 153 MB	11.0 s 407 MB
	72	Linear scan	6.02 s 4.50 GB	None	2.66 s 4.42 GB	2.87 s 86.2 MB	N/A	None
		Clustering	**2.33 s** 1.61 GB	0.91 s 921 MB	1.07 s 640 MB	0.33 s 44.1 MB	N/A	N/A
Deep1B-10M	1	Linear scan	390 s 47.9 GB	None	187 s 47.4 GB	203 s 518 MB	32.6 s 10.4 GB	None
		Clustering	31.6 s 5.53 GB	18.0 s 3.12 GB	7.23 s 2.35 GB	6.33 s 59.4 MB	1.83 s 576 MB	86.3 s 3.72 GB
	72	Linear scan	75.9 s 47.9 GB	None	54.0 s 47.4 GB	17.0 s 518 MB	N/A	None
		Clustering	**6.37 s** 5.59 GB	2.94 s 3.12 GB	2.74 s 2.41 GB	0.68 s 59.4 MB	N/A	N/A

Fig. 2: Performance of our algorithms on two "West US 2" Azure instances. We show the break down of the running time and communication between the parts of the algorithm. "Overall query time" does not include the OT phase, which is done once per query. Preprocessing is done once per client. We run OT and preprocessing in a single thread. Also we measure overall query time as the maximum between server and client, but measure the parts on the server.

C. *Parameters*

Accuracy. In our experiments, we require the algorithms to return $k_{\mathrm{nn}} = 10$ nearest neighbors and measure accuracy as the average of the number of correctly returned points over the set of queries (we refer to this later as "10-NN accuracy"). We evaluate our algorithms requiring that the 10-NN accuracy is at least 0.9, which is a level of accuracy considered to be acceptable in practice.

Quantization of coordinates. For SIFT, coordinates of points and queries are already small integers between 0 and 255, so we leave them as is. For Deep1B, the coordinates are real numbers, and we quantize them to 8-bit integers uniformly between the minimum and the maximum coordinates for the dataset. In experiments, such quantizations barely affect the 10-NN accuracy compared to using the true floating point coordinates.

Cluster size balancing. As noted in Section III-B, our cluster balancing algorithm achieves the crucial bound over the maximum cluster size needed for efficient ORAM retrieval of candidate points. In our experiments, for SIFT, Deep1B-10M, and Deep1B-1M, the balancing algorithm reduced the maximum cluster size by factors of $4.95\times$, $3.67\times$, and $3.31\times$, respectively.

Parameter choices. We initialized the BFV scheme with parameters $N = 2^{13}$, $t = 2^{23}$ and a 180-bit modulus q. For the parameters such as standard deviation error and secret key distribution we use SEAL default values. These parameters allow us to use the noise flooding technique to provide 108 bits of statistical circuit privacy[8]. We used the LWE estimator[9] by Albrecht et al. [2] to estimate the security level of the scheme, which suggests 141 bits of security.

Let us describe how we set the hyperparameters of our algorithms. Both of our algorithms (especially the clustering-based) have quite a few moving parts that nontrivially affect the overall performance. See Section III-F for the full list of hyperparameters, below we list the one that affect the performance for both of our algorithms:

- Both algorithms depend on n, d, k_{nn}, which depend on the dataset and our requirements;
- Besides that, linear scan depends on l_{s}, b_{c} and r_{p},
- The clustering-based algorithm depends on T, k_{c}^{i}, m, u^{i}, s, l^{i}, l_{s}, b_{c}, r_{c} and r_{p}, where $1 \leq i \leq T$.

For both of the algorithms, we use the *total number of AND gates* in the top-k and the ORAM circuits as a proxy for both communication and running time. Moreover, for simplicity we neglect the FSS part of ORAM, since it does not affect the performance much. We refer the reader to Appendix A for the exact formulas used in our cost model. Overall, we search for the hyperparameters that yield 10-NN accuracy at least 0.9

[8] We refer the reader to [46] for details on the noise flooding technique

[9] We used the most recent commit (3019847) from https://bitbucket.org/malb/lwe-estimator

	Threads	Algorithm	Overall query	ORAM	Top-k	Distances	OT phase	Preprocessing
SIFT	1	Linear scan	130 s	None	103.7 s	24.9 s	30.2 s	None
		Clustering	61.8 s	41.3 s	16.09 s	3.56 s	4.95 s	23.6 s
	72	Linear scan	21.5 s	None	4.45 s	13.5 s	N/A	None
		Clustering	11.5 s	3.70 s	5.30 s	2.01 s	N/A	N/A
Deep1B-1M	1	Linear scan	125 s	None	104 s	20.1 s	23.9 s	None
		Clustering	47.1 s	27.6 s	16.4 s	3.09 s	4.56 s	20.2 s
	72	Linear scan	20.5 s	None	4.43 s	12.9 s	N/A	None
		Clustering	11.2 s	3.78 s	5.29 s	1.90 s	N/A	N/A
Deep1B-10M	1	Linear scan	1400 s	None	1190 s	204 s	250 s	None
		Clustering	172 s	103 s	58.3 s	10.1 s	14.5 s	165 s
	72	Linear scan	211 s	None	186 s	16.8 s	N/A	None
		Clustering	29.7 s	9.00 s	16.4 s	4.04 s	N/A	N/A

Fig. 3: Similar to Figure 2, but now the instances are hosted on “West US 2” and “East US”, so the running times are higher due to the slower networking (which, however, benefits from multi-threaded communication). We do not report communication, since it’s the same to Figure 2.

(approximately) minimizing the total number of AND-gates. In Figure 4 and Figure 5, we summarize the parameters we use for both of our algorithms on each of the datasets.

D. Evaluation

Figure 2 shows the running times and communication volumes for both of our algorithms evaluated on SIFT, Deep1B-1M and Deep1B-10M run on two “West US 2” instances. Since the OT phase and per-client preprocessing are implemented only in a single-thread regime, we mark the respective entries in the “multi-thread” rows with “N/A”. Let us note however that both of these phases should be easily parallelizable. We find that the clustering-based algorithm consistently outperforms the linear scan, both in terms of the running time and the communication, both for 1 and 72 threads. On Deep1B-10M, the gap for the respective characteristics exceeds an order of magnitude. It is interesting that for a single thread and a single query, the clustering-based algorithm beats the linear scan *even taking the per-client preprocessing time into account*. We do not report the timing of hyperparameter tuning and clustering, since it needs to be done only once per dataset. We also evaluate our algorithms on a slower network connection: between a “West US 2” and an “East US” instance, see Figure 3. The results are qualitatively similar to Figure 2.

Let us now compare the numbers we obtain with two baselines. First, we use the arithmetic mode of ABY [28] to compute distances from a query to the data points. We find that on SIFT it takes 620 seconds and 167 GB of communication, which is significantly worse than what can be achieved by AHE (2.22 s, 56.7 MB). On Deep-1B-1M, ABY takes about the same time, and on Deep-1B-10M, it consumes more than all the available RAM on our instances, but it is likely to be an order of magnitude slower. Second, we evaluate the naïve top-k circuit that consists of $O(nk)$ comparisons that has been used in the prior work (e.g., in [64]) using our GC implementation. On SIFT it takes 147 seconds and 24.7 GB of communication, while our better circuit takes merely 15.6 seconds and 4.42 GB of communication, improving by almost an order of magnitude. We note that the gap in communication is around 5x due to the fact that we compute distances from secret shares, which becomes one of the bottlenecks for our faster top-k selection.

We summarize the running times of our algorithms as well as the baselines on Figure 6.

VI. Conclusions and future directions

In this work, we design new secure computation protocols for approximate k-Nearest Neighbors Search between a client holding a query and a server holding a database, with the Euclidean distance metric. Our solution combines several state-of-the-art cryptographic primitives such as lattice-based additively homomorphic encryption, FSS-based distributed ORAM and garbled circuits with various optimizations. Underlying one of our protocols is a new sublinear-time plaintext approximate k-NNS algorithm tailored to secure computation. Notably, it is the first sublinear-time k-NNS protocol implemented securely. Our performance results show that our solution scales well to massive datasets consisting of up to ten million points.

We highlight some directions for future work:

- Our construction can be proved secure in the semi-honest model, but it would be interesting to extend our protocols to protect against malicious adversaries, where the client or the server can deviate from the protocol in order to learn about the other party’s data or manipulate the output of the other party. An interesting compromise is the covert model, where a cheating party is guaranteed to be caught with, say, 10% probability.
- We used Kreyvium instead of AES in order to reduce communication between the parties, but when the cipher needs to be evaluated in the clear, AES is still more efficient thanks to optimized hardware implementation. It would be interesting to investigate on improvements of the plaintext implementation of Kreyvium.
- It would be interesting to implement other sublinear k-NNS algorithms securely, most notably Locality-Sensitive Hashing (LSH) [4], which has *provable* sublinear query time and is widely used in practice.

	Linear scan			Clustering		
Parameter	SIFT	Deep1B-1M	Deep1B-10M	SIFT	Deep1B-1M	Deep1B-10M
l_{s}	8334	8334	83	262	210	423
b_{c}	8	8	8	8	8	8
r_{p}	8	8	9	8	8	8

Fig. 4: (Near-)optimal hyperparameters that are used both by linear scan and the clustering-based algorithm.

Parameter	SIFT	Deep1B-1M	Deep1B-10M
T	4	5	6
k_{c}^{i}	50810 25603 9968 4227	44830 25867 11795 5607 2611	209727 107417 39132 14424 5796 2394
m	20	22	48
u^{i}	50 31 19 13	46 31 19 13 7	88 46 25 13 7 7
s	31412	25150	50649
l^{i}	458 270 178 84	458 270 178 84 84	924 458 178 93 84 84
r_{c}	5	5	5
α	0.56	0.56	0.56

Fig. 5: (Near-)optimal hyperparameters that are specific to the clustering-based algorithm.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM, 1983, pp. 1–9.

[2] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[3] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, 2015, pp. 430–454.

[4] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems*, 2015, pp. 1225–1233.

[5] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate nearest neighbor search in high dimensions," *arXiv preprint arXiv:1806.09823*, 2018.

[6] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin, "Privacy-preserving search of similar patients in genomic data," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 104–124, 2018.

[7] M. Aumüller, E. Bernhardsson, and A. Faithfull, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," in *International Conference on Similarity Search and Applications*. Springer, 2017, pp. 34–49.

[8] ——, "Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, 2019.

[9] A. Babenko and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.

[10] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti *et al.*, "Privacy-preserving fingercode authentication," in *Proceedings of the 12th ACM workshop on Multimedia and security*. ACM, 2010, pp. 231–240.

[11] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[12] D. Beaver, "Precomputing oblivious transfer," in *Annual International Cryptology Conference*. Springer, 1995, pp. 97–109.

[13] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *STOC*, vol. 90, 1990, pp. 503–513.

[14] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 478–492.

[15] D. J. Bernstein, "The chacha family of stream ciphers," https://cr.yp.to/chacha.html.

[16] ——, "The salsa20 family of stream ciphers," in *New Stream Cipher Designs - The eSTREAM Finalists*, 2008, pp. 84–97.

[17] A. Bestavros, A. Lapets, and M. Varia, "User-centric distributed solutions for privacy-preserving analytics," *Communications of the ACM*, 2017.

[18] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, 1973.

[19] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter *et al.*, "Secure multiparty computation goes live," in *International Conference on Financial Cryptography and Data Security*. Springer, 2009, pp. 325–343.

[20] J. Boyar and R. Peralta, "A small depth-16 circuit for the AES s-box," in *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, 2012, pp. 287–298.

[21] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.

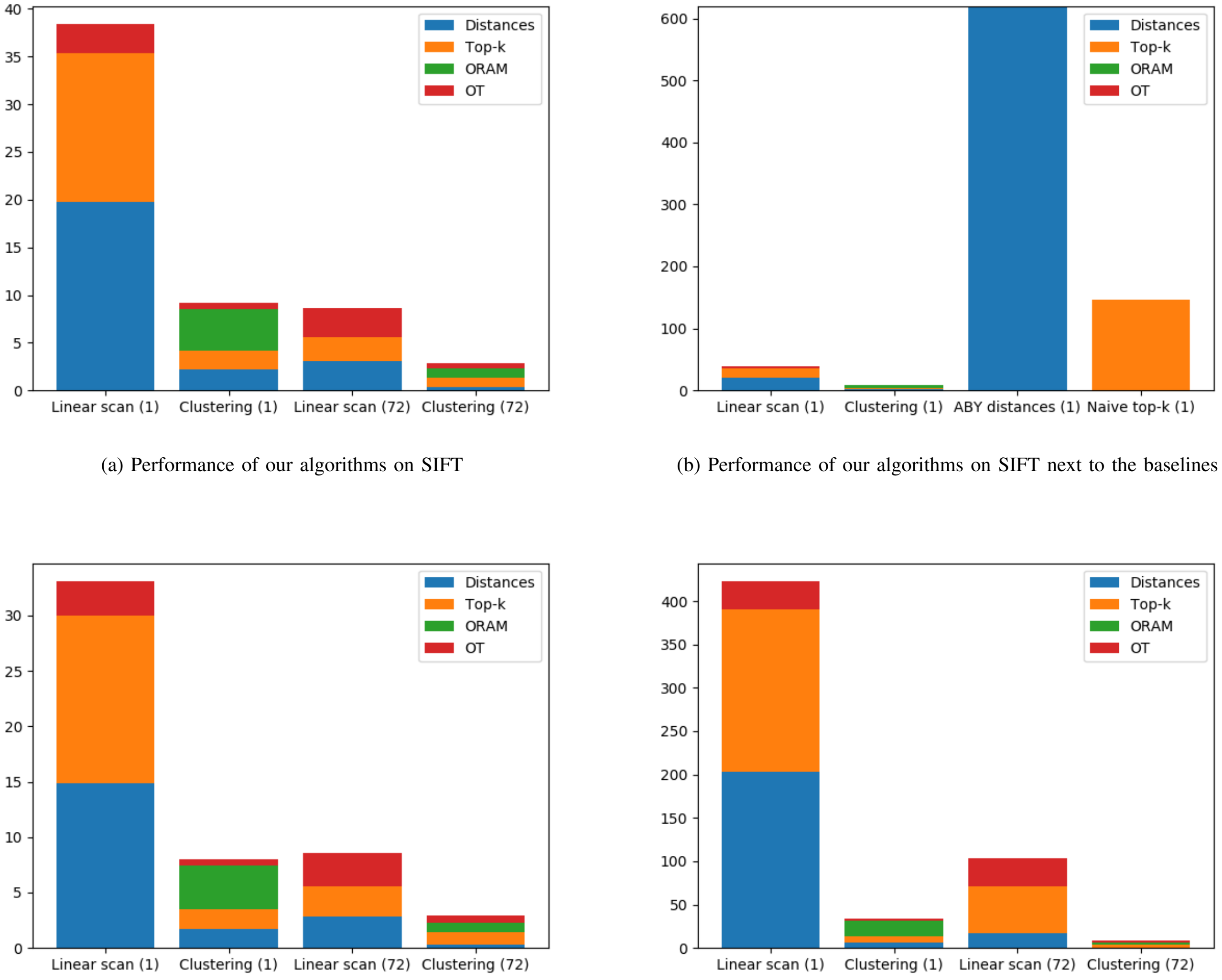
[22] ——, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, 2012, pp. 868–886.

[23] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. of ITCS*. ACM, 2012, pp. 309–325.

[24] M. Burkhart and X. Dimitropoulos, "Fast privacy-preserving top-k queries using secret sharing," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, 2010, pp. 1–7.

[25] C. D. Cannière and B. Preneel, "Trivium," in *New Stream Cipher Designs - The eSTREAM Finalists*, 2008, pp. 244–266.

[26] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey, "Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression," in *Fast Software En-*



(a) Performance of our algorithms on SIFT

(b) Performance of our algorithms on SIFT next to the baselines

(c) Performance of our algorithms on Deep1B-1M

(d) Performance of our algorithms on Deep1B-10M

Fig. 6: Comparison of our algorithms run on 1 and 72 threads on two "West US 2" instances. The y-axis is the running time (in seconds). OT phase is always run single-threaded. For SIFT we compare our algorithms with distances computed in ABY as well as the naïve top-k circuit.

cryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers, 2016, pp. 313–333.

[27] I. Damgård, M. Geisler, and M. Krøigaard, "Efficient and secure comparison for on-line auctions," in *Australasian Conference on Information Security and Privacy*. Springer, 2007, pp. 416–430.

[28] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[29] P. Diaconis and D. Freedman, "Finite exchangeable sequences," *The Annals of Probability*, pp. 745–764, 1980.

[30] J. Doerner, "The absentminded crypto kit," https://bitbucket.org/jackdoerner/absentminded-crypto-kit.

[31] J. Doerner and A. Shelat, "Floram: The floram oblivious ram implementation for secure computation," https://gitlab.com/neucrypt/floram.

[32] ——, "Scaling ORAM for secure computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 523–535.

[33] ——, "Scaling oram for secure computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 523–535.

[34] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2009, pp. 235–253.

[35] D. Evans, Y. Huang, J. Katz, and L. Malka, "Efficient privacy-preserving biometric identification," in *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS*, 2011.

[36] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[37] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[38] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 75–92.

[39] N. Gilboa and Y. Ishai, "Distributed point functions and their applications," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014, pp. 640–658.

[40] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 218–229.

[41] ——, "How to play any mental game or A completeness theorem for protocols with honest majority," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, 1987, pp. 218–229.

[42] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[43] P. Indyk and D. Woodruff, "Polylogarithmic private approximations and efficient matching," in *Theory of Cryptography Conference*. Springer, 2006, pp. 245–264.

[44] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[45] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[46] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium*. USENIX Association, 2018.

[47] J. Kilian, "Founding crytpography on oblivious transfer," in *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1988, pp. 20–31.

[48] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, 2008, pp. 486–498.

[49] ——, "Improved garbled circuit: Free xor gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.

[50] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Data recovery on encrypted databases with k-nearest neighbor query leakage," in *Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage*. IEEE, p. 0.

[51] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography*. Springer, 2017, pp. 277–346.

[52] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[53] D. G. Lowe *et al.*, "Object recognition from local scale-invariant features." in *iccv*, vol. 99, no. 2, 1999, pp. 1150–1157.

[54] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[55] W. Microsoft Research, Redmond, "Simple Encrypted Arithmetic Library," http://sealcrypto.org, 10 2018, SEAL 3.0.

[56] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, 1999, pp. 223–238. [Online]. Available: https://doi.org/10.1007/3-540-48910-X_16

[57] ——, "Public-key cryptosystems based on composite degree residuosity classes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238.

[58] M. S. Riazi, B. Chen, A. Shrivastava, D. Wallach, and F. Koushanfar, "Sub-linear privacy-preserving near-neighbor search with untrusted server on large-scale datasets," *arXiv preprint arXiv:1612.01835*, 2016.

[59] M. S. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar, "MPCircuits: Optimized circuit generation for secure multi-party computation," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019.

[60] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018, pp. 707–721.

[61] P. Rindal, "libOTe: an efficient, portable, and easy to use Oblivious Transfer Library," https://github.com/osu-crypto/libOTe.

[62] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Efficient privacy-preserving face recognition," in *International Conference on Information Security and Cryptology*. Springer, 2009, pp. 229–244.

[63] H. Shaul, D. Feldman, and D. Rus, "Scalable secure computation of statistical functions with applications to k-nearest neighbors," *arXiv preprint arXiv:1801.07301*, 2018.

[64] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Compacting privacy-preserving k-nearest neighbor search using logic synthesis," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[65] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *arXiv preprint arXiv:1408.2927*, 2014.

[66] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 850–861.

[67] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "SCORAM: oblivious ram for secure computation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 191–202.

[68] A. C.-C. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167.

[69] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole - reducing data transfer in garbled circuits using half gates," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pp. 220–250.

[70] ——, "Two halves make a whole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[71] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: efficient random access in multi-party computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 218–234.

[72] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 586–595.

Appendix

A. Cost model

Here we describe the cost model we use to tune the hyperparameters. We focus on the clustering-based algorithm, since tuning the linear scan can be seen as an easy special case (all points in the stash etc.). We heavily use the notation introduced in Section III-F.

There are three main steps of the algorithm:

1) Compute closest u^i from the overall k_c^i centers for the i-th *group* for $1 \leq i \leq T$;
2) Retrieve $u_{\text{all}} = \sum_{i=1}^{T} u^i$ clusters from ORAM (of m points each);
3) Compute k_{nn} closest points from the union of the stash (of s points) and $m \cdot u_{\text{all}}$ retrieved points.

1) Number of AND gates: As a proxy for the total cost, we use the number of AND gates in the circuits. For ORAM retrieval, we *do not* count AND gates necessary for the functional secret sharing.

1) Cost of computing closest centers of clusters:

$$\sum_{i=1}^{T}\Big(k_{\text{c}}^{i}\cdot b_{\text{d}}+(k_{\text{c}}^{i}+u^{i}\cdot l^{i})\cdot\big(2\cdot(b_{\text{d}}-r_{\text{c}})+b_{\text{cid}}\big)\Big)$$

2) Cost of ORAM retrieval (modulo FSS):

$$u_{\text{all}}\cdot\Big(6\cdot\big(1152+m\cdot(d\cdot b_{\text{c}}+b_{\text{d}}+b_{\text{pid}})\big)+m\cdot(d+1)\cdot b_{\text{d}}\Big)$$

3) Cost of computing closest points (the final answer):

$$\big(s+m\cdot u_{\text{all}}\big)\cdot b_{\text{d}}+\Big(s+m\cdot u_{\text{all}}+k_{\text{nn}}\cdot\big(l_{\text{s}}+m\cdot u_{\text{all}}\big)\Big)\times$$
$$\times\,\big(2\cdot(b_{\text{d}}-r_{\text{p}})+b_{\text{pid}}\big)$$

Our cost is defined as the sum of the three above expressions. Next, for completeness we list the formulae for the numbers of inputs and outputs for the server and client for all of the three parts. These quantities affect the communication, but we do not include them in the cost we optimize since they affect the computation time less than the number of AND gates.

1) Closest centers
   - Server's inputs:
   $$\sum_{i=1}^{T}\big(k_{\text{c}}^{i}\cdot(b_{\text{d}}+b_{\text{cid}})+u^{i}\cdot b_{\text{cid}}\big)$$
   - Client's inputs:
   $$\sum_{i=1}^{T}k_{\text{c}}^{i}\cdot(b_{\text{d}}+b_{\text{cid}})$$
   - Client's outputs:
   $$\sum_{i=1}^{T}u^{i}\cdot b_{\text{cid}}$$
2) DORAM retrieval
   - Server's inputs:
   $$u_{\text{all}}\cdot\Big(b_{\text{cid}}+128+$$
   $$+m\cdot\big(d\cdot b_{\text{c}}+b_{\text{d}}+b_{\text{pid}}+(d+1)\cdot b_{\text{d}}+b_{\text{pid}}\big)\Big)$$
   - Client's inputs:
   $$u_{\text{all}}\cdot\big(b_{\text{cid}}+128+m\cdot(d\cdot b_{\text{c}}+b_{\text{d}}+b_{\text{pid}})\big)$$
   - Client's outputs:
   $$u_{\text{all}}\cdot m\cdot\big((d+1)\cdot b_{\text{d}}+b_{\text{pid}}\big)$$
3) Closest points
   - Server's inputs:
   $$(s+u_{\text{all}}\cdot m)\cdot(b_{\text{d}}+b_{\text{pid}})+k_{\text{nn}}\cdot b_{\text{pid}}$$
   - Client's inputs:
   $$(s+u_{\text{all}}\cdot m)\cdot(b_{\text{d}}+b_{\text{pid}})$$
   - Client's outputs:
   $$k_{\text{nn}}\cdot b_{\text{pid}}$$

B. Stream Ciphers as PRF

In the original Floram construction [32], the PRF and the PRG used in the read-only process are chosen by the authors to be AES-128. Indeed, AES-128 is a block cipher that has been largely studied by the cryptographic community. The implementations of the scheme are highly optimized (less than 5000 non-free gates per block [20]) and its security is often used as a standard term of comparison. The implementation of Floram [31], [30] uses the optimized AES-128 and proposes two alternative symmetric encryption schemes: the streams Salsa20 [16] and its variant Chacha20 [15].

However, other symmetric ciphers can be used to obtain an efficient PRF/PRG. In particular, we looked for a PRF with low number of AND gates in order to decrease the communication between the parties when it is evaluated in GC (in the Free-XOR setting). Between the block ciphers, one of the most promising constructions is LowMC [3], which has a small number of AND gates per output bit. Between the stream ciphers, instead, Trivium [25] and its variant Kreyvium [26] captured our attention. They are flexible in terms of input and output size, since there is no fixed block size to respect, and their evaluation is very efficient in terms of AND gates per output bit of stream.

Trivium belongs to the 2008 eSTREAM portfolio. It presents a simple construction, needing only 3 AND gates per bit of stream produced, plus $3\cdot 1152$ initialization AND gates executed once per stream. Trivium uses a secret key and an IV of size 80-bits each and achieves 80-bits of security. The scheme uses three registers, which are initialized with the key, the IV and some additional fixed bits. At each round, three temporary variables are computed by adding or multiplying some fixed elements in the register (9 XORs and 3 ANDs per round): at the end of each round, every register is rotated by 1 position, one element is discarded and one temporary value is appended. The first 1152 (this number is chosen for security reasons) rounds are the initialization rounds and they do not produce any stream. After the initialization, every round outputs one bit of stream, equal to the XOR of the three temporary values.

Krevium was presented in 2015 as a 128-bits secure variant of Trivium, as a solution particularly suited for homomorphic-ciphertext compression: the construction uses longer keys and IVs (128 bits each), 2 additional registers and a few additional XOR gates per round, but keeps the same amount of AND gates per bit of stream produced and for the initialization phase.

AES-128 needs about 5000 AND gates to produce 128 bits of stream, while Trivium and Kreyvium need $3\cdot(1152+N)$ AND gates, where N is the size of the input/output of the stream. The difference is not impressive when the input blocks are of size 128, but the gap between the two ciphers increases when the size of inputs increases, since the stream cipher only needs 3 more AND gates per bit of input. For AES-128 the number of AND gates per bit remains constant (about 39 AND gates per output bit) while in Kreyvium it decreases to about 3 AND gates per bit of stream (see Table I).

The inputs we use in our construction have different sizes. For small datasets, every input is about 2.7 kB while for large datasets the inputs are about 5 or 6 kB. We compute 2 PRFs per input, so the actual number of AND gates in Table I should be doubled.

	128 bits	2.7 kB	6 kB
AES-128	5000 AND (39 AND/bit)	865000 AND (39.1 AND/bit)	1920000 AND (39.06 AND/bit)
Chacha20	20480 AND (160 AND/bit)	901120 AND (40.7 AND/bit)	1966080 AND (40 AND/bit)
Kreyvium	3840 AND (30 AND/bit)	69810 AND (3.15 AND/bit)	150912 AND (3.07 AND/bit)

TABLE I: Estimates on the number of AND gates for ciphers AES-128, Chacha20 and Kreyvium for different input sizes. The estimates for Chacha20 refer to a naive implementation of the scheme: we believe that the scheme would be more efficient in terms of non trivial gates in practice, but we did not found such optimal estimates in the literature. We do not report the number of AND gates for LowMC: they should be comparable to the estimates we have for Kreyvium for an optimal choice of the parameters.

While our approach is more efficient in GC with respect to Floram, the plaintext evaluation of Kreyvium is slower than the (highly optimized) hardware implementation of AES. In order to mitigate this issue, we vertically batch 512 bits and we compute multiple streams in parallel (using AVX-512), so we are able to process several hundreds of Mega Bytes of information per second in single core.

C. Proofs of correctness for approximate top-k

In this section, we give proofs for Theorem 1 and Theorem 2.

Proof of Theorem 1: First, suppose that we assign a bin for each element uniformly and *independently*. For this sampling model, it is not hard to see that the desired expectation of the size of the intersection $\mathcal{I}$ is:

$$\begin{aligned}&\mathrm{E}\left[|\mathcal{I}|\right]\\&= l \cdot \Pr[U_i \text{ contains at least one of the top-}k \text{ elements}]\\&= l \cdot \left(1 - \left(1 - \frac{1}{l}\right)^k\right),\end{aligned}$$

where the first step follows from the linearity of expectation, and the second step is an immediate calculation. Suppose that $l = k/\delta$, where $\delta > 0$ is sufficiently small, and suppose that $k \to \infty$. Then, continuing the calculation,

$$\begin{aligned}l \cdot \left(1 - \left(1 - \frac{1}{l}\right)^k\right) &= \frac{k}{\delta} \cdot \left(1 - e^{k \cdot \ln\left(1 - \frac{\delta}{k}\right)}\right)\\&= \frac{k}{\delta}\left(1 - e^{-\delta + O(1/k)}\right) = \frac{k \cdot (1 - e^{-\delta})}{\delta} + O(1)\\&\geq \frac{k \cdot \left(\delta - \frac{\delta^2}{2}\right)}{\delta} + O(1) = k \cdot \left(1 - \frac{\delta}{2}\right) + O(1),\end{aligned}$$

where the first step is immediate, the second step uses the Taylor series of $\ln x$, the third step uses the Taylor series of e^x, the fourth step uses the inequality $e^{-x} \leq 1 - x + \frac{x^2}{2}$, which is true for sufficiently small positive x, and the last step is immediate.

To argue about the actual sampling process, where instead of uniform and independent assignment, we shuffle elements and partition them into l blocks of size n/l, we use the main result of [29]. Namely, it is true that the probability

$$\Pr[U_i \text{ contains at least one of the top-}k \text{ elements}]$$

can change by at most $O(1/l)$ when passing between two sampling processes. This means that the overall expectation changes by at most $O(1)$, and is thus still at least:

$$k \cdot \left(1 - \frac{\delta}{2}\right) + O(1).$$

For a fixed δ, this expression is at least $(1 - \delta)k$, whenever k is sufficiently large. ■

Proof of Theorem 2: As in the proof of the previous theorem, we start with a simpler sampling model, where bins are assigned independently. Suppose that $\delta > 0$ is fixed and k tends to infinity. We set $l = k^2/\delta$. In that case, one has:

$$\begin{aligned}&\Pr[\text{all top-}k \text{ elements end up into different bins}]\\&= \left(1 - \frac{1}{l}\right) \cdot \left(1 - \frac{2}{l}\right) \cdot \ldots \cdot \left(1 - \frac{k-1}{l}\right)\\&= \left(1 - \frac{\delta}{k^2}\right) \cdot \left(1 - \frac{2\delta}{k^2}\right) \cdot \ldots \cdot \left(1 - \frac{(k-1) \cdot \delta}{k^2}\right)\\&= \exp\left(\ln\left(1 - \frac{\delta}{k^2}\right) + \ln\left(1 - \frac{2\delta}{k^2}\right) + \ldots\right.\\&\quad\left. + \ln\left(1 - \frac{(k-1) \cdot \delta}{k^2}\right)\right)\\&= \exp\left(-\frac{\delta(1 + 2 + \ldots + (k-1))}{k^2} + O\left(\frac{1}{k}\right)\right)\\&= e^{-\delta/2} + O\left(\frac{1}{k}\right) \geq 1 - \frac{\delta}{2} + O\left(\frac{1}{k}\right),\end{aligned}$$

where the fourth step uses the Taylor series of $\ln x$ and the sixth step uses the inequality $e^{-x} \geq 1 - x$. The final bound is at least $1 - \delta$ provided that k is large enough.

Now let us prove that for the actual sampling procedure (shuffling and partitioning into l blocks of size n/l), the probability of top-k elements being assigned to different bins *can only increase*, which implies the desired result. To see this, let us denote c_i the bin of the i-th of the top-k elements. One clearly has:

$$\Pr[\text{all top-}k \text{ elements end up into different bins}] = \sum_{\text{distinct } j_1, j_2, \ldots, j_k} \Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k].$$

Thus, it is enough to show that any probability of the form

$$\Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k],$$

where $j_1, j_2, \ldots, j_k$ are distinct, can only increase when passing to the actual sampling model. This probability can be factorized as follows:

$$\begin{aligned}&\Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k]\\&= \Pr[c_1 = j_1] \cdot \Pr[c_2 = j_2 \mid c_1 = j_1] \cdot \ldots\\&\cdot \Pr[c_k = j_k \mid c_1 = j_1 \wedge \ldots \wedge c_{k-1} = j_{k-1}].\end{aligned}$$

For the simplified sampling model, each of these conditional probabilities is equal to $1/l$ due to the independence of c_i. However, for the actual model, they are larger: if we condition on t equalities, then the probability is equal to $\frac{n}{l(n-t)}$. This implies the required monotonicity result.

■

D. Optimal circuit for implicit top-k

Recall that our goal is, given n numbers each consisting of b bits, to find k smallest numbers in the following form: the output of a circuit it a binary vector with exactly k ones at the positions that correspond to the smallest elements. Such representation was used in [24] and [6].

Previously it was known how to achieve this in $O(b^2 n)$ gates as follows: we need to find a threshold y such that $|\{i\colon x_i \leq y\}| = k$, after that finding the result can be trivially done in $O(bn)$ gates by comparing every number with y. We can find y using binary search, which takes b iterations, and for each iteration we compare every number with a current guess for y, which takes $O(bn)$ gates, resulting in $O(b^2 n)$ gates overall.

Now we show how to improve this construction to the optimal $O(bn)$ gates. Instead of running the full binary search for y, we will be computing it bit-by-bit starting from the most significant one. In order to do this, we maintain a binary vector a_i of "alive" elements of the list, initially $a_i \equiv 1$. To figure out i-th bit of y, we count how many alive elements of the list have 0 as the i-th bit; let us denote this number by c_0. This counting can be done in $O(n)$ gates using a binary tree of adders. Next we compare c_0 with k: if $k \leq c_0$, then we zero out the entries of a for the elements of the list with the i-th bit being 1, otherwise, we zero out the entries with the i-th bit being 0 and subtract c_0 from k. All of these operations can be implemented in $O(n)$ gates, and there are b iterations in total. Overall, this circuit can be seen as a hybrid between radix sort and a randomized selection algorithm.

E. Security proofs

We prove simulation-based security for our protocols for approximate k-NNS. First, we recall the definition of two party computation and simulation-based security for semi-honest adversaries. The definitions are taken from [51].

Definition 1. *A two-party functionality is a possibly randomized function*

$$f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*,$$

where $f = (f_1, f_2)$. That is, for every pair of inputs $x, y \in \{0,1\}^n$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x,y)$ and the second party (with input y) wishes to obtain $f_2(x,y)$.

Let π be a protocol computing the functionality f, i.e., by honestly executing π via possibly multiple rounds of local computations and sending messages, the two parties learn f_1 and f_2 when π completes. The *view* of the i-th party during an execution of π on (x,y) and security parameter λ is denoted by

$$\mathrm{View}_{\pi,i}(x,y,\lambda)$$

and equals the party i's input with its internal randomness, plus all messages it receives during the protocol.

Definition 2. *Let $f = (f_1, f_2)$ be a functionality and let π be a protocol that computes f. We say that π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms S_1 and S_2 (often called simulators) such that*

$$S_1(1^\lambda, x, f_1(x,y)) \approx \mathrm{View}_{\pi,1}(x,y,\lambda)$$

and

$$S_2(1^\lambda, y, f_2(x,y)) \approx \mathrm{View}_{\pi,2}(x,y,\lambda).$$

Here $\approx$ means computational indistinguishability.

Next, we recall the security assumption, namely Ring learning-with-errors (RLWE), specialized to the power of two cyclotomic rings, which serves as the underlying assumption of the AHE scheme we use.

Definition 3 (decision-RLWE problem)**.** *For security parameter λ and a power of two integer n depending on λ, set $R = \mathbb{Z}[x]/(x^n + 1)$. Let $q = q(\lambda) \geq 2$ be an integer. For a random element $s \in R_q$ and a distribution $\chi = \chi(\lambda)$ over R, denote with $A^{(q)}_{s,\chi}$ the distribution obtained by choosing a uniformly random element $a \leftarrow R_q$ and a noise term $e \leftarrow \chi$ and outputting $(a, [a \cdot s + e]_q)$. The Decision-RLWE problem is to distinguish between the distribution $A^{(q)}_{s,\chi}$ and the uniform distribution $U(R_q^2)$.*

Lemma 1. *Assuming the average-case hardness of decision-RLWE problem for parameters λ, n, q, χ. Then the following two distributions are in-distinguishable. For any fixed message m, the first distribution is $\mathsf{AHE.Enc}(sk, m)$ where sk is the output of $\mathsf{AHE.Keygen}$, and the second distribution is $\mathsf{AHE.Enc}(sk, 0)$.*

Proof: We include the proof for reader's convenience. Let $s = sk \in R_q$. Note that $\mathsf{AHE.Enc}(s, 0) = (a, as + e + \Delta m)$ for some integer $\Delta \approx q/t$, polynomial $a \leftarrow U(R_q)$ and $e \leftarrow \chi$. From the decision-RLWE assumption, the pair $(a, as + e)$ is computationally insdistinguishable from uniform, hence for any m, the pair $(a, as+e+\Delta m)$ is also indistinguishable from uniform. This proves the claim. ■

1) Ideal Fucntionalities: First, we define the ideal functionalities that our protocol achieves. Note that the two protocols have slightly different ideal functionalities. We will denote them by $\mathcal{F}_{\mathrm{ANN_{cl}}}$ (for clustering) and $\mathcal{F}_{\mathrm{ANN_{ls}}}$ (for linear scan).

Parameters: number of elements n, dimension d, bits of precision b_c.

- Input: client inputs a query $\mathbf{q} \in \mathbb{R}^d$ and server inputs database $DB = [(\mathbf{p}_i, \mathrm{ID}_i)]_{i=1}^n$. Note that points are truncated to b_c bits.
- Output: returns output of Algorithm 1 to client.

Fig. 7: Ideal functionality $\mathcal{F}_{\mathrm{ANN_{ls}}}$

2) Ideal functionalities for subroutines: Note that we used garbled circuit to achive $\mathcal{F}_{\mathrm{TOPk}}$ and $\mathcal{F}_{\mathrm{aTOPk}}$, and we used the FLORAM construction [32] to implement $\mathcal{F}_{\mathrm{DROM}}$ securely. We refer the reader to the referenced papers for the full security proof of these sub-protocols. Below we give the definition of the three ideal functionalities:

Parameters: number of elements n, dimension d, bits of precision b_c, and clustering-based hyperparameters T, $k_{\rm c}^i$, m, u^i, s, l^i, $l_{\rm s}$, $b_{\rm c}$, $r_{\rm c}$ and $r_{\rm p}$.

- Input: client inputs a query $\mathbf{q} \in \mathbb{R}^d$ and server inputs database $DB = [(\mathbf{p}_i, \mathrm{ID}_i)]_{i=1}^n$. The points are truncated to b_c bits.
- Output: returns output of Algorithm 2 to client.

Fig. 8: Ideal functionality $\mathcal{F}_{\mathrm{ANN_{cl}}}$

Parameters: array size m, modulus 2^t, truncation bit size r.

- On input A_c from the client, store A_c.
- On input A_s, $idlist$ from the server, store A_s and $idlist$.
- When both A_c and A_s are received, compute $A = (A_s + A_c) \mod 2^t = (a_1, \ldots, a_n)$. and $A' = \mathrm{NAIVETOPK}(a_1, \ldots, a_n, idlist)$. Sample an array R of size k with random entries, output $A' + R$ to the client, and output R to the server

Fig. 9: Ideal functionality $\mathcal{F}_{\mathrm{TOPk}}$

Parameters: array size m, modulus 2^t, truncation bit size r.
Extra parameter: $returnDist \in \{0, 1\}$ (if set to true, return (dist, id) instead of just id.)

- On input A_c from the client, store A_c.
- On input A_s and $idlist$ from the server, store A_s and $idlist$.
- When both A_c and A_s are received, compute $A = A_s + A_c \mod 2^t = (a_1, \ldots, a_n)$. and set $a'_i = [a_i/2^r]$. Let $B = \mathrm{APPROTOPK}(a'_1, \ldots, a'_n, idlist)$. Sample an array R of size k with random entries modulo 2^t, output $B + R$ to the client and R to the server.

Fig. 10: Ideal functionality $\mathcal{F}_{\mathrm{aTOPk}}$

Parameters: Database size n, bit-length of each data block b.

- Init: on input (Init, DB) from the server, it stores DB.
- Read: on input (Read, i_c) and (Read, i_s) from both client and server, it samples a random $R \in \{0,1\}^b$. Then it outputs $DB[(i_s + i_c) \mod n] \oplus R$ to client and outputs R to server.

Fig. 11: Ideal functionality $\mathcal{F}_{\mathrm{DROM}}$

3) Security proofs:

Theorem 3. *Assuming the hardness of the decision-RLWE problem, our linear scan protocol $\Pi_{\mathrm{ANN_{ls}}}$ securely implements the functionality $\mathcal{F}_{\mathrm{ANN_{ls}}}$ in the $\mathcal{F}_{\mathrm{aTOPk}}$ hybrid model, with semi-honest adversaries.*

Proof: First, we construct a simulator for the client. The simulator generates a key sk for the AHE scheme and sends sk to the client. Then, it simulates the server's first message as $\mathsf{AHE.Enc}(sk, r_i)$ for random values r_i. From the circuit privacy property of the AHE scheme, this is indistinguishable from the first message in the real protocol. Next, the simulator simply feeds $\{r_i\}$ to the ideal functionality $\mathcal{F}_{\mathrm{aTOPk}}$ and forwards the output to the client. This completes the simulation.

Next, we construct a simulator for the server. The simulator generates a key sk for the AHE scheme. The first message from the client to the server consists of the encryptions $\mathsf{AHE.Enc}(sk, \mathbf{q}[i])$ in the real protocol. Instead, the simulator just sends $\mathsf{AHE.Enc}(sk, 0)$ for $1 \le i \le d$. From Lemma 1, these views are indistinguishable.

Next, the simulator generates a random sequence $R = (r_1, \ldots, r_n)$ of values and forwards that to the server. This completes the simulation. ■

Theorem 4. *Assuming the hardness of the decision-RLWE problem, our clustering protocol $\Pi_{\mathrm{ANN_{cl}}}$ securely implements the $\mathcal{F}_{\mathrm{ANN_{cl}}}$ functionality in the ($\mathcal{F}_{\mathrm{TOPk}}$, $\mathcal{F}_{\mathrm{aTOPk}}$, $\mathcal{F}_{\mathrm{DROM}}$)-hybrid model in the prescence of semi-honest adversaries.*

Proof: Again correctness is easy to verify. We first descrbie simulator for the client. First, the simulator generates a secret key sk for the AHE scheme and sends sk to the client. Next, the simulator sends blocks of zero to $\mathcal{F}_{\mathrm{DROM}}.\mathsf{Init}$. Then, on receiving the query message from the client, the simulator does the following: for each i, j, it samples random values r_{ij} and generates $\mathsf{AHE.Enc}(sk, r_{ij})$. Using a similar argument as in the previous proof, these ciphertexts are indistinguishable from the client's view of the first step of the secure protocol. Then, the simulator forwards the r_{ij} to $\mathcal{F}_{\mathrm{aTOPk}}$ and gets back secret shares of indices, namely $[i_1], \ldots, [i_u]$. Then, it feeds these indices shares to $\mathcal{F}_{\mathrm{DROM}}.\mathsf{Read}$ and forwards the output to the client. Also, it samples random messages s_i and sends $\mathsf{AHE.Enc}(sk, s_i)$ to the client. Later, when the simulator receives the shares $m \cdot u_{\mathrm{all}} + s$ of (point, ID) pairs from the client, it samples $m \cdot u_{\mathrm{all}} + s$ random pairs of values and send the first $m \cdot u_{\mathrm{all}}$ values to $\mathcal{F}_{\mathrm{TOPk}}$ and the last s values to $\mathcal{F}_{\mathrm{aTOPk}}$. Then, it forwards the output to the client. Since the intermediate values revealed to the client are all independent uniformly random values, the view generated from simulator is indistinguishable from the real view.

Now, the simulator for server works in almost the same fashion, with the difference that in contrast to the real client which sends $\mathsf{AHE.Enc}(sk, \mathbf{q}_i)$ for $1 \le i \le d$, the simulator simply sends d encryption of zeros. This is indistinguishable from Lemma 1.

■

F. Secure protocols

We formally specify our secure protocols implementing the functionalities defined in previous section. Figure 12 formally specifies our linear scan protocol. Figure 13 formally specifies our clustering based protocol.

Public Parameters: coefficient bit length b_c, number of items in the database n, dimension d, AHE ring dimension N, plain modulus t.
Inputs: client inputs query $\mathbf{q} \in \mathbb{R}^d$; server inputs n points $\mathbf{p}_1, \ldots, \mathbf{p}_n \in \mathbb{R}^d$ and a list of n IDs $idlist$.

1) Client calls $\mathsf{AHE.Keygen}$ to get sk.
2) Both client and server discretize and normalize their points into $\mathbf{q}'$ and $\mathbf{p}'_i \in \mathbb{Z}^d_{2^{b_c}}$.
3) Client sends $c_i = \mathsf{AHE.Enc}(sk, \mathbf{q}'[i])$ for $1 \le i \le d$ to the server.
4) Server sets $p_{ik} = \mathbf{p}'_{kN+1}[i] + \mathbf{p}'_{kN+2}[i]x + \cdots + \mathbf{p}'_{(k+1)N}[i]x^{N-1}$, samples random vector $\mathbf{r} \in \mathbb{Z}^n_t$ and computes homomorphically

$$f_k = \sum_{i=1}^{d} c_i \cdot \mathbf{p}_{ik} + \mathbf{r}[kN : (k+1)N]$$

for $1 \le k \le \lceil n/N \rceil$.
5) Server sends f_k to the client. The client decrypts all f_k and obtains vector $\mathbf{s} \in \mathbb{Z}^n_t$.
6) Client sends $-2\mathbf{s} + ||q'||^2 \cdot (1, 1, \ldots, 1)$ to $\mathcal{F}_{\mathrm{aTOPk}}$; server sends $idlist$ and $(-2r_i + ||p'_i||^2)_i$ to $\mathcal{F}_{\mathrm{aTOPk}}$. Client gets back $[\mathbf{id}]_c \in \mathbb{Z}^k_t$, and server gets back $[\mathbf{id}]_s \in \mathbb{Z}^k_t$.
7) Server sends the vector $[\mathbf{id}]_s$ to client, who outputs $\mathbf{id} = [\mathbf{id}]_c + [\mathbf{id}]_s \mod t$.

Fig. 12: Protocol $\Pi_{\mathrm{ANN}_{\mathrm{ls}}}$

Public Parameters: coefficient bit length b_c, number of items in the database n, dimension d, AHE ring dimension N, plain modulus t.
Clustering hyperparameters: T, k^i_{c}, m, u^i, s, l^i, l_{s}, b_{c}, r_{c} and r_{p}.
Inputs: client inputs query $\mathbf{q} \in \mathbb{R}^d$; server inputs T groups of clusters with each cluster of size up to m, and a stash S of size s, for a total of n points; server also inputs a list of n IDs $idlist$, and u_{all} cluster centers c^i_j.

1) Client calls $\mathsf{AHE.Keygen}$ to get sk.
2) Both client and server discretize and normalize their points plus the clueter centers into $\mathbf{q}'$ and $\mathbf{p}'_i \in \mathbb{Z}^d_{2^{b_c}}$.
3) Server sends all clusters, where points are accompanied by ID and sqwuared norms of point, with one block per cluster, to $\mathcal{F}_{\mathrm{DROM}}.\mathsf{Init}$, padding with dummy points if necessary to reach size m for each block.
4) The server performs two independent random shuffles on the cluster centers and stash points.
5) For each $i \in \{1, \ldots, T\}$,
   - The client and server use line 3-5 in Figure 12 to compute secret shares of vector $\mathbf{d}_i$ where $\mathbf{d}_i[j] = ||\mathbf{q} - \mathbf{c}^i_j||^2_2$.
   - Client sends its share $\langle \mathbf{d}_i \rangle_C$ to $\mathcal{F}_{\mathrm{aTOPk}}$ with $k = u_i$; Server sends its share $\langle \mathbf{d}_i \rangle_S$ of distances and the corresponding IDs to $\mathcal{F}_{\mathrm{aTOPk}}$. Client and server outputs secret shares of a vector $\mathbf{ind}_i$ of u_i indices.
6) Client and server input the secret shares of the $u_{\mathrm{all}} = \sum_{i=1}^{T} u_i$ indices $(i_1, j_1), \ldots, (i_{u_{\mathrm{all}}}, j_{u_{\mathrm{all}}})$ obtained in previous step into $\mathcal{F}_{\mathrm{DROM}}.\mathsf{Read}$, to retrieve secret shared list of tuples $(\mathbf{p}, \mathrm{ID}(\mathbf{p}), ||\mathbf{p}||^2)$ of all points in $C := C^{i_1}_{j_1} \cup \cdots \cup C^{i_{u_{all}}}_{j_{u_{all}}}$.
7) Client and server use line 3-6 in Figure 12 to get secret shares of a distance vector $\mathbf{d} = \langle \mathbf{d} \rangle_C + \langle \mathbf{d} \rangle_S$ (calling $\mathcal{F}_{\mathrm{aTOPk}}$ with returnDist = True). Client and server outputs secret shares of a list of tuples $(d^{Cluster}_i, \mathrm{ID}^{Cluster}_i))^k_{i=1}$.
8) For the stash S, client and server use line 3-6 in Figure 12 (with returnDist = True). Client and server outputs secret shares of a list of tuples $(d^{Stash}_i, \mathrm{ID}^{Stash}_i))^k_{i=1}$.
9) Client and server inputs the union of the secret shares of (point, id) pairs obtained from previous two steps into $\mathcal{F}_{\mathrm{TOPk}}$, and outputs secret shares of IDs of the closest k points to the client's query.
10) Server sends its secret shares of IDs to the client, who outputs the final list of IDs.

Fig. 13: Protocol $\Pi_{\mathrm{ANN}_{\mathrm{cl}}}$