SANNS: Scaling Up Secure Approximate *k*-Nearest Neighbors Search

Hao Chen, *Microsoft Research;* Ilaria Chillotti, *imec-COSIC KU Leuven & Zama;* Yihe Dong, *Microsoft;* Oxana Poburinnaya, *Simons Institute;* Ilya Razenshteyn, *Microsoft Research;* M. Sadegh Riazi, *UC San Diego*

https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hao

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search

Hao Chen
Microsoft Research

Ilaria Chillotti
imec-COSIC KU Leuven & Zama

Yihe Dong
Microsoft

Oxana Poburinnaya
Simons Institute

Ilya Razenshteyn
Microsoft Research

M. Sadegh Riazi
UC San Diego

Abstract

The k-Nearest Neighbor Search (k-NNS) is the backbone of several cloud-based services such as recommender systems, face recognition, and database search on text and images. In these services, the client sends the query to the cloud server and receives the response in which case the query and response are revealed to the service provider. Such data disclosures are unacceptable in several scenarios due to the sensitivity of data and/or privacy laws.

In this paper, we introduce SANNS, a system for secure k-NNS that keeps client's query and the search result confidential. SANNS comprises two protocols: an optimized linear scan and a protocol based on a novel *sublinear time* clustering-based algorithm. We prove the security of both protocols in the standard semi-honest model. The protocols are built upon several state-of-the-art cryptographic primitives such as lattice-based additively homomorphic encryption, distributed oblivious RAM, and garbled circuits. We provide several contributions to each of these primitives which are applicable to other secure computation tasks. Both of our protocols rely on a new circuit for the approximate top-k selection from n numbers that is built from $O(n+k^2)$ comparators.

We have implemented our proposed system and performed extensive experimental results on four datasets in two different computation environments, demonstrating more than $18-31\times$ *faster response time* compared to optimally implemented protocols from the prior work. Moreover, SANNS is the first work that scales to the database of 10 million entries, pushing the limit by more than *two orders of magnitude*.

1 Introduction

The *k-Nearest Neighbor Search problem* (k-NNS) is defined as follows. For a given n-point dataset $X \subset \mathbb{R}^d$, and a query point $\mathbf{q} \in \mathbb{R}^d$, find (IDs of) k data points closest (with respect to the Euclidean distance) to the query. The k-NNS has many applications in modern data analysis: one typically starts with a dataset (images, text, etc.) and, using domain expertise together with machine learning, produces its *feature vector representation*. Then, *similarity search* queries ("find k objects most similar to the query") directly translate to k-NNS queries in the feature space. Even though some applications of k-NNS benefit from non-Euclidean distances [6], the overwhelming majority of applications (see [7] and the references therein) utilize Euclidean distance or cosine similarity, which can be modeled as Euclidean distance on a unit sphere.

When it comes to applications dealing with sensitive information, such as medical, biological or financial data, the privacy of both the dataset and the queries needs to be ensured. Therefore, the "trivial solution" where the server sends the entire dataset to the client or the client sends the plaintext query to the server would not work, since we would like to protect the input from both sides. Such settings include: face recognition [30,60], biometric identification [9,23,31], patient data search in a hospital [6, 62] and many others. One can pose the *Secure k-NNS* problem, which has the same functionality as the k-NNS problem, and at the same time preserves the privacy of the input: the server—who holds the dataset—should learn nothing about the query or the result, while the client—who has the query—should not learn anything about the dataset besides the k-NNS result.

Secure k-NNS is a heavily studied problem in a variety of settings (see Section 1.2 for the related work). In this paper, we consider one of the most conservative security requirements of *secure two-party computation* [32], where the protocol is not allowed to reveal anything beyond the *output* of the respective *plaintext* k-NNS algorithm. Note that we do not rely on a trusted third party (which is hardly practical) or trusted hardware such as Intel SGX[1] (which is known to have major security issues: see, e.g., [66]).

In this paper, we describe SANNS: a system for fast processing of secure k-NNS queries that works in the two-party

[1] While the trust model of cryptographic solutions is based on computational hardness assumptions, Trusted Execution Environments (TEE)-based methodologies, such as Intel SGX, require remote attestation before the computation can begin. As a result, TEE-based solutions need to trust the hardware vendor as well as TEE implementation.

secure computation setting. The two main contributions underlying SANNS are the following. First, we provide an improved secure protocol for the top-k selection. Second, we design a new k-NNS algorithm tailored to secure computation, which is implemented using a combination of Homomorphic Encryption (HE), Garbled Circuits (GC) and Distributed Oblivious RAM (DORAM) as well as the above top-k protocol. Extensive experiments on real-world image and text data show that SANNS achieves a speed-up of up to 31× compared to (carefully implemented and heavily optimized) algorithms from the prior work.

Trust model We prove simulation-based security of SANNS 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. Most of the instances of secure multi-party computation deployed in the real world operate in the semi-honest model: computing gender pay gap [15], sugar beets auctions [17], and others. Our protocol yields a substantial improvement over prior works under the same trust model. Besides, any semi-honest protocol can be reinforced to be maliciously secure (when parties are allowed to tamper actively with the sent messages), though it incurs a significant performance overhead [35].

1.1 Specific Contributions

Underlying SANNS are two new algorithms for the k-NNS problem. The first one is based on *linear scan,* where we compute distances to all the points, and then select the k closest ones. The improvement comes from the new top-k selection protocol. The second algorithm has *sublinear* time avoiding computing all the distances. At a high level, it proceeds by clustering the dataset using the k-means algorithm [47], then, given a query point, we compute several closest clusters, and then compute k closest points within these clusters. The resulting points are *approximately* closest; it is known that approximation is necessary for *any* sublinear-time k-NNS algorithm [57][2]. In order to be suitable for secure computation, we introduce a new cluster *rebalancing* subroutine, see below. Let us note that among the *plaintext* k-NNS algorithms, the clustering approach is far from being the best [7], but we find it to be particularly suitable for secure computation.

For both algorithms, we use Additive Homomorphic Encryption (AHE) for secure distance computation and garbled circuit for the top-k selection. In case of our sublinear-time algorithm, we also use DORAM to securely retrieve the clusters closest to the query. For AHE, we use the SEAL library [52] which implements the Brakerski/Fan-Vercauteren (BFV) scheme [33]. For GC we use our own implementation of Yao's protocol [70] with the standard optimizations [11, 12, 41, 71], and for DORAM we implement Floram [27] in the read-only mode.

Our specific contributions can be summarized as follows:

- We propose a novel mixed-protocol solution based on AHE, GC, and DORAM that is tailored for secure k-NNS and achieves more than 31× performance improvement compared to prior art with the same security guarantees.
- We design and analyze an improved circuit for approximate top-k selection. The secure top-k selection protocol within SANNS is obtained by garbling this circuit. This improvement is likely to be of independent interest for a range of other secure computation tasks.
- We create a clustering-based algorithm that outputs *balanced* clusters, which significantly reduces the overhead of oblivious RAMs for secure random accesses.
- We build our system and evaluate it on various real-world datasets of text and images. We run experiments on two computation environments that represent fast and slow network connections in practice.
- We make several optimizations to the AHE, GC, and DORAM cryptographic primitives to improve efficiency of our protocol. Most notably, in Floram [27], we substitute block cipher for stream cipher, yielding a speed-up by more than an order of magnitude.

1.2 Related Work

To the best of our knowledge, all prior work on the secure k-NNS problem in the secure two-party computation setting is based on the *linear scan*, where we first compute the distance between the query and all of n database points, and then select k smallest of them. To contrast, our clustering-based algorithm is *sublinear*, which leads to a substantial speed-up. We classify prior works based on the approaches used for distance computation and for top-k selection.

Distance computation SANNS computes distances using the BFV scheme [33]. Alternative approaches used in the prior work are:

- Paillier scheme [54] used for k-NNS in [9, 29–31, 60]. Unlike the BFV scheme, Paillier scheme does not support massively vectorized SIMD operations, and, in general, is known to be much slower than the BFV scheme for vector/matrix operations such as a batched Euclidean distance computation: see, e.g., [40].
- OT-based multiplication is used for k-NNS in [23] for $k = 1$. Compared to the BFV scheme, OT-based approach requires much more communication, $O(n+d)$ vs. $O(nd)$, respectively, while being slightly less compute-intensive. In our experiments, we find that the protocol from [53] that is carefully tailored to the matrix operations (and is, thus, significantly faster than the generic one used in [23]) is as fast as AHE on the fast network, but significantly slower on the slow network.

[2] At the same time, approximation is often acceptable in practice, since feature vectors are themselves merely approximation of the "ground truth"

Top-k selection SANNS chooses k smallest distances out of n by garbling a new top-k circuit that we develop in this work. The circuit is built from $O(n+k^2)$ comparators. Alternative approaches in the prior work are:

- The naive circuit of size $\Theta(nk)$ (c.f. Algorithm 1) was used for k-NNS in [6, 61, 64]. This gives asymptotically a factor of k slow-down, which is significant even for $k = 10$ (which is a typical setting used in practice).
- Using homomorphic encryption (HE) for the top-k selection. In the works [62, 63], to select k smallest distances, the BGV scheme is used, which is a variant of the BFV scheme we use for distance computations. Neither of the two schemes are suitable for the top-k selection, which is a highly non-linear operation. A more suitable HE scheme for this task would have been TFHE [22], however, it is still known to be slower than the garbled circuit approach by at least three orders of magnitude.

We can conclude the discussion as follows: our experiments show that for $k = 10$, even the linear scan version of SANNS is at up to $3.5\times$ faster than all the prior work *even if we implement all the components in the prior work using the most modern tools* (for larger values of k, the gap would increase). However, as we move from the linear scan to the sublinear algorithm, this yields additional speed-up up to $12\times$ at a cost of introducing small error in the output (on average, one out of ten reported nearest neighbors is incorrect).

All the prior work described above is in the semi-honest model except [61] (which provides malicious security). The drawback, however, is efficiency: the algorithm from [61] can process one query for a dataset of size 50 000 in several hours. Our work yields an algorithm that can handle 10 million data points in a matter of seconds. All the other prior work deals with datasets of size at most 10 000. Thus, by designing better algorithms and by carefully implementing and optimizing them, we scale up the datasets one can handle efficiently by *more than two orders of magnitude.*

Other security models Some prior work considered the secure k-NNS problem in settings different from "vanilla" secure two-party computation. Two examples are the works [58, 69]. The work [69] is under the two-server setting, which is known to give much more efficient protocols, but the security relies on the assumption that the servers do not collude. At the same time, our techniques (e.g., better top-k circuit and the balanced clustering algorithm) should yield improvements for the two-server setting as well. In the work [58], a very efficient sublinear-time protocol for secure approximate k-NNS is provided that provides a trade-off between privacy and the search quality. One can tune the privacy parameter to limit the information leakage based on the desired accuracy threshold. As a result, their protocol can leak more than approximate k-NNS results, i.e., one can estimate the similarity of two data points based on the hash values (see Section 5 of [58] for a formal bound on the information leakage).

1.3 Applications of Secure k-NNS

SANNS can potentially impact several real-world applications. At a high-level, our system can provide a an efficient mechanism to retrieve similar elements to a query in any two-party computation model, e.g., database search, recommender systems, medical data analysis, etc. that provably does not leak anything beyond (approximate) answers. For example, our system can be used to retrieve similar images within a database given a query. We analyze the efficiency of our system in this scenario using the SIFT dataset which is a standard benchmark in approximate nearest-neighbor search [48]. Additionally, we consider Deep1B which is a dataset of image descriptors [8]. We run SANNS on a database as big as *ten million* images, whereas the prior work deals with datasets of size at most 50 000. As another application of secure k-NNS consider privacy-preserving text search, which has been rigorously studied in the past [21, 37, 50, 55, 65]. One group of these solutions support (multi)-keyword search [21, 50, 65]: a client can receive a set of documents which include all (or subset of) keywords queried by the clients. In a more powerful setting, text *similarity* search can be performed where all documents that are semantically similar to a given document can be identified while keeping the query and the database private [37, 55]. In this context, we evaluate SANNS on the Amazon reviews text database [51].

2 Preliminaries

2.1 Secret Sharing

In this work, we use a combination of secure computation primitives to solve the k-NNS problem. We connect these primitives via secret sharing, which comes in two forms: an *arithmetic* secret sharing of a value $x \in \mathbb{Z}_t$ is a pair $(\langle x\rangle_C, \langle x\rangle_S)$ of random values subject to $\langle x\rangle_C + \langle x\rangle_S \equiv x \mod t$, whereas a *Boolean* (or XOR) secret sharing of $x \in \{0,1\}^\tau$ is a pair of random strings subject to $\langle x\rangle_C \oplus \langle x\rangle_S = x$.

2.2 Distributed Oblivious RAM (DORAM)

Previous solutions for secure k-NNS require computing distance between the query point and all points in the database, which is undesirable for large databases. In order to avoid this linear cost, we utilize a distributed version of oblivious RAM (DORAM). In this scenario, two parties hold secret shares of an array, and they can perform oblivious read and write operations, with *secret-shared* indices. Typically one requires the communication cost to be sublinear in the array size. There are many known DORAM constructions [27, 67, 68, 72], among which we choose Floram [27] 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:

- $\mathsf{DORAM.Init}(1^\lambda, DB) \to (k_A, k_B, \overline{DB})$. 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. Here λ is a security parameter.
- $\mathsf{DORAM.Read}(\overline{DB}, k_A, k_B, i_A, i_B) \to (DB[i]_A, DB[i]_B)$. This subroutine performs the read operation where address i is secret-shared between two parties as $i_A \oplus i_B = i$. Both parties acquire a XOR-share of $DB[i]$.

In Section 4.3, we describe these subroutines and various optimizations in a greater detail.

2.3 Additive Homomorphic Encryption (AHE)

A (private-key) additive homomorphic encryption (AHE) scheme is private-key encryption scheme with three additional algorithms Add, CAdd and CMult, which supports adding two ciphertexts, and addition / multiplication by constants. We require our AHE scheme to satisfy standard IND-CPA security and *circuit privacy*, which means that a ciphertext generated from Add, CAdd and CMult operations should not leak more information about the operations to the secret key owner, other than the decrypted message. This is required since in our case the server will input its secret values into CAdd and CMult. We chose to use the BFV scheme [33], and we achieve circuit privacy through noise flooding [40].

2.4 Garbled Circuit (GC)

Garbled circuit (GC) is a technique first proposed by Yao in [70] for achieving generic secure two-party computation for arbitrary Boolean circuits. Many improvements to GC have been proposed in literature, such as free-XOR [41] and half-gates [71]. In addition, we use the fixed-key block cipher optimization for garbling and evaluation [12]. Using Advanced Encryption Standard (AES) as the block cipher, we leverage Intel AES instructions for faster garbling procedure.

2.5 k-means Clustering

One of our algorithms uses the k-means clustering algorithm [47] 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 minimizes the objective function $\sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \|\mathbf{c}_i - \mathbf{x}\|^2$.

3 Plaintext k-NNS Algorithms

Optimized linear scan Our first algorithm is a heavily optimized implementation of the 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 top-k selection is done using GC.

Computing top-k naïvely would require a circuit built from $O(nk)$ comparators. Instead, we propose a new algorithm for an approximate selection of top-k, which allows for a smaller circuit size (see section 3.1) and will help us later when we implement the top-k selection securely using garbled circuit.

Clustering-based algorithm The second algorithm is based on the k-means clustering (see Section 2.5) and, unlike our first algorithm, has *sublinear* query time. We now give a simplified version of the algorithm, and in Section 3.3 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 $k = k_{\text{c}}$ clusters. Each cluster $1 \le i \le k_{\text{c}}$ is associated with its *center* $\mathbf{c}_i \in \mathbb{R}^d$. During the query stage, we find $1 \le u \le k_{\text{c}}$ 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-many centers, and return IDs of these points as a final answer.

3.1 Approximate Top-k Selection

In both of our algorithms, we rely extensively on the following *top-k selection* functionality which we denote by $\text{MIN}_n^k(x_1, x_2, \ldots, x_n)$: given a list of n numbers $x_1, x_2, \ldots, x_n$, output $k \le n$ smallest list elements in the sorted order. We can also consider the augmented functionality where each value is associated with an ID, and we output the IDs together with the values of the smallest k elements. We denote this augmented functionality by $\overline{\text{MIN}}_n^k$. 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 $O(n + k \log k)$ [16]. 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 required 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.

The naïve construction A naïve circuit for MIN_n^k performs $O(nk)$ comparisons and hence consists of $O(bnk)$ gates. Algorithm 1 gives such a circuit (to be precise, it computes the augmented functionality $\overline{\text{MIN}}_n^k$, but can be easily changed to compute only MIN_n^k). 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.

Sorting networks Another approach is to employ sorting networks (e.g., AKS [1] or the Zig-Zag sort [36]) with $O(bn \log n)$ gates, which can be further improved to $O(bn \log k)$. However, these constructions are not known to be practical.

Approximate randomized selection We are not aware of any circuit for MIN_n^k with $O(bn)$ gates unless k is a constant ($O(bn)$ gates is optimal since the input has bn bits). Instead,

Algorithm 1 Naive Top-k Computation

function NAIVETOPK($(x_1,\mathrm{ID}_1),\ldots,(x_n,\mathrm{ID}_n)$, k)
  $OPT = [\mathsf{MAXVAL}]_k$
  $idlist = [0]_k$
  for $i \leftarrow 1 \ldots n$ **do**
    $x \leftarrow x_i$, $idx \leftarrow \mathrm{ID}_i$
    for $j \leftarrow 1 \ldots k$ **do**
      $b \leftarrow (x < OPT[j])$
      $(OPT[j], x) = \mathrm{MUX}(OPT[j], x, b)$
      $(idlist[j], idx) = \mathrm{MUX}(idlist[j], idx, b)$
    end for
  end for
  return $(OPT, idlist)$
end function
function $\mathrm{MUX}(a_1, a_2, b)$
  # Returns (a_1, a_2) for $b = 0$, and (a_2, a_1) for $b = 1$
  return $(a_1 + (a_2 - a_1) \cdot b, a_2 + (a_1 - a_2) \cdot b)$
end function

Algorithm 2 Approximate top-k selection

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

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 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 $l \le n$ bins of size n/l as follows: $U_1 = \{x_{\pi(1)}, \ldots, x_{\pi(n/l)}\}$, $U_2 = \{x_{\pi(n/l+1)}, \ldots, x_{\pi(2n/l)}\}$, $\ldots$, $U_l = \{x_{\pi((l-1)n/l+1)}, \ldots, x_{\pi(n)}\}$. 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. *Suppose the input list $(x_1, \ldots, x_n)$ is in uniformly random order. There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ with the following property. For every n, $0 < \delta < \delta_0$, and $k \ge k_0(\delta)$, one can set the number of bins $l = k/\delta$ such that the intersection I of the output of Algorithm 2 with $\overline{\mathrm{MIN}}_n^k(x_1, x_2, \ldots, x_n)$ contains at least $(1-\delta)k$ entries in expectation over the choice of π.*

Algorithm 3 Plaintext linear scan

function LINEARSCANKNNS($\mathbf{q}, \{\mathbf{p}_i\}_{i=1}^n$, ID)
  # Uses hyperparameters r_p, k_nn, l_s from Figure 1
  Randomly permute the set $\{\mathbf{p}_i\}$
  for $i \leftarrow 1, \ldots, n$ **do**
    $d_i \leftarrow \|\mathbf{q} - \mathbf{p}_i\|^2$
    $d_i \leftarrow \lfloor \frac{d_i}{2^{r_\mathrm{p}}} \rfloor$
  end for
  $(v_1, \mathrm{ID}_1), \ldots, (v_{k_\mathrm{nn}}, \mathrm{ID}_{k_\mathrm{nn}}) \leftarrow$
   APPROXTOPK$(d_1, \mathrm{ID}(\mathbf{p}_1), \ldots, (d_n, \mathrm{ID}(\mathbf{p}_n), k_\mathrm{nn}, l_\mathrm{s})$
  return $\mathrm{ID}_1, \ldots, \mathrm{ID}_{k_\mathrm{nn}}$
end function

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

Theorem 2. *Suppose the input list $(x_1, \ldots, x_n)$ is in uniformly random order. There exists $\delta_0 > 0$ and a positive function $k_0(\delta)$ with the following property. For every n, $0 < \delta < \delta_0$, and $k \ge k_0(\delta)$, one can set the number of bins $l = k^2/\delta$ such that the output of Algorithm 2 is* exactly *$\overline{\mathrm{MIN}}_n^k(x_1, x_2, \ldots, x_n)$ 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.

3.2 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 depending on whether we select closest cluster centers or closest data points, which allows for a more fine-grained parameter tuning.

3.3 Balanced Clustering and Stash

To implement the clustering-based k-NNS algorithm securely while avoiding linear cost, we use 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. This can be very inefficient, if clusters are not very balanced, i.e., the largest cluster is much larger than a *typical* cluster. Unfortunately, this is exactly what we observe 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 the clustering-based algorithm we securely implement. With cluster balancing, our

	Parameter	Description
Dataset	n	number of data points in the dataset
	d	dimensionality of the data points
	k_{nn}	number of data points we need to return as an answer
Clustering Algorithm	T	number of *groups* of clusters
	k_{c}^i	total number of clusters for the i-th group, $1 \le i \le T$
	m	*largest* cluster size
	u^i	number of closest clusters we retrieve for the i-th group, $1 \le i \le T$
	$u_{\text{all}} = \sum_{i=1}^{T} u^i$	total number of clusters we retrieve
	l^i	is the number of bins we use to speed up the selection of closest clusters for the i-th group, $1 \le i \le T$
	α	the allowed fraction of points in large clusters during the preprocessing
Stash	s	size of the *stash*
	l_{s}	number of bins we use to speed up the selection of closest points for the stash
Bitwidth	b_{c}	number of bits necessary to encode one *coordinate*
	b_{d}	number of bits necessary to encode one *distance* ($b_{\text{d}} = 2b_{\text{c}} + \lceil \log_2 d \rceil$)
	b_{cid}	number of bits necessary to encode the index of a *cluster* ($b_{\text{cid}} = \left\lceil \log_2 \left(\sum_{i=1}^{T} k_{\text{c}}^i \right) \right\rceil$)
	b_{pid}	number of bits for ID of a *point*
	r_{c}	number of bits we discard when computing distances to *centers of clusters*, $0 \le r_{\text{c}} \le b_{\text{d}}$
	r_{p}	number of bits we discard when computing distances to *points*, $0 \le r_{\text{p}} \le b_{\text{d}}$
AHE	N	the ring dimension in BFV scheme
	q	ciphertext modulus in BFV scheme
	$t = 2^{b_{\text{d}}}$	plaintext modulus in BFV scheme and the modulus for secret-shared distances

Figure 1: List of hyperparameters.

experiments achieve 3.3× to 4.95× 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 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 $C_1^i, C_2^i, \ldots, C_{k_{\text{c}}^i}^i \subseteq X$ and their centers by $c_1^i, c_2^i, \ldots, c_{k_{\text{c}}^i}^i \in \mathbb{R}^d$. 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 $S \subseteq X$. In contrast to clusters from the remaining groups, to search the stash, we perform *linear scan*. 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 3 is well-known and very popular in practice (see, e.g., [38, 39]), our modification of the algorithm in this section, to the best of our knowledge, is new. It is interesting to observe that in the "plaintext world", clustering algorithm is far from being the best for k-NNS (see [7] for the benchmarks), but several of its properties (namely, few non-adaptive memory accesses and that it requires computing many distances at once) make it very appealing for the secure computation.

3.4 Putting It All Together

We now give a high-level summary of our algorithms and in the next section 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 top-k algorithm. Meanwhile, we compute the approximate top-k with $k = k_{\text{nn}}$ among distances between query and the stash. Finally, we compute and output the k_{nn} closest points from the above $2k_{\text{nn}}$ 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 input values to be shuffled. 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

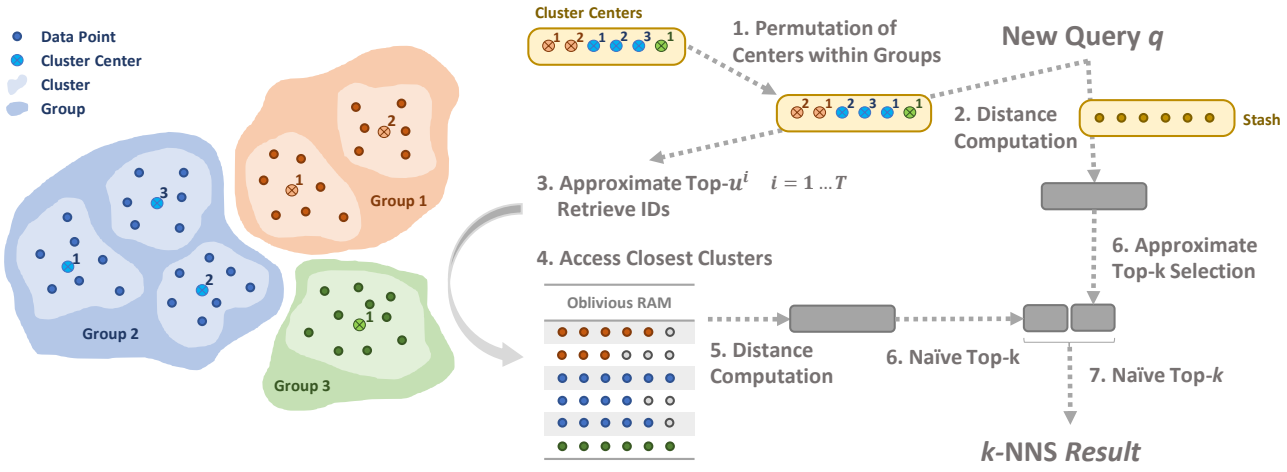

Figure 2: Visualization of SANNS clustering-based algorithm.

Algorithm 4 Plaintext clustering-based algorithm

function CLUSTERINGKNNS($\mathbf{q}$, C^i_j, $\mathbf{c}^i_j$, S, ID)
  # The algorithm uses hyperparameters in Figure 1
  Randomly permute the cluster centers in each group and all points in stash
  for $i \leftarrow 1,\ldots,T$ **do**
    for $j \leftarrow 1,\ldots,k^i_c$ **do**
      $d^i_j \leftarrow \|\mathbf{q}-\mathbf{c}^i_j\|^2$
      $d^i_j \leftarrow \lfloor \frac{d^i_j}{2^{r_c}} \rfloor$
    end for
    $(v_1, ind^i_1),\ldots,(v_{u^i}, ind^i_{u^i}) \leftarrow$
      $\leftarrow$ APPROXTOPK$((d^i_1,1),\ldots,(d^i_{k^i_c},k^i_c),u^i,l^i)$
  end for
  $C \leftarrow \bigcup_{1\le i\le T} \bigcup_{1\le j\le u_i} C^i_{ind^i_j}$
  for $\mathbf{p} \in C \cup S$ **do**
    $d_{\mathbf{p}} \leftarrow \|\mathbf{q}-\mathbf{p}\|^2$
    $d_{\mathbf{p}} \leftarrow \lfloor \frac{d_{\mathbf{p}}}{2^{r_{\mathbf{p}}}} \rfloor$
  end for
  $(a_1,\widetilde{\mathrm{ID}}_1),\ldots,(a_{k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{k_{\mathrm{nn}}}) \leftarrow$
    $\leftarrow$ NAIVETOPK$(\{(d_{\mathbf{p}},\mathrm{ID}(\mathbf{p}))\}_{\mathbf{p}\in C},k_{\mathrm{nn}})$
  $(a_{k_{\mathrm{nn}}+1},\widetilde{\mathrm{ID}}_{k_{\mathrm{nn}}+1}),\ldots,(a_{2k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{2k}) \leftarrow$
    $\leftarrow$ APPROXTOPK$(\{(d_{\mathbf{p}},\mathrm{ID}(\mathbf{p}))\}_{\mathbf{p}\in S},k_{\mathrm{nn}},l_s)$
  $(v_1,\widehat{\mathrm{ID}}_1),\ldots,(v_{k_{\mathrm{nn}}},\widehat{\mathrm{ID}}_{k_{\mathrm{nn}}})) \leftarrow$
    $\leftarrow$ NAIVETOPK$((a_1,\widetilde{\mathrm{ID}}_1),\ldots,(a_{2k_{\mathrm{nn}}},\widetilde{\mathrm{ID}}_{2k_{\mathrm{nn}}}),k_{\mathrm{nn}})$
  return $\widehat{\mathrm{ID}}_1,\ldots,\widehat{\mathrm{ID}}_{k_{\mathrm{nn}}}$
end function

expensive. Therefore, we garble a *naïve* circuit for exact computation of top-k for the retrieved points. Figure 2 visualizes SANNS clustering-based algorithm.

Figure 1 lists the hyperparameters used by our algorithms. See Figure 5 and Figure 6 for the values that we use for various datasets. Our plaintext algorithms are presented in Algorithm 3 and Algorithm 4.

4 Secure Protocols for k-NNS

Here we describe our secure protocols for k-NNS. For the security proofs, see Appendix D. The formal specifications of the protocols are given in Figure 6 and Figure 7. On a high level, our secure protocols implement plaintext algorithms 3 and 4, which is color-coded for reader's convenience: we implemented the blue parts using AHE, yellow parts using garbled circuit, and red parts using DORAM. These primitives are connected using secret shares, and we perform share conversions (between arithmetic and Boolean) as needed.

4.1 Ideal Functionalities for Subroutines

Here we define three ideal functionalities $\mathcal{F}_{\mathrm{TOPk}}$, $\mathcal{F}_{\mathrm{aTOPk}}$, and $\mathcal{F}_{\mathrm{DROM}}$ used in our protocol. We securely implement the first two using garbled circuits, and the third using Floram [27].

Parameters: array size m, modulus t, truncation bit size r, output size k, bit-length of ID b_{pid}
Extra parameter: returnVal $\in \{false, true\}$ (if set to true, return secret shares of (value, ID) pairs instead of just ID.)

- On input A_c and $idlist_c$ from the client, store A_c.
- On input A_s, $idlist_s$ from the server, store A_s and $idlist$.
- When both inputs are received, compute $A = (A_s + A_c) \mod t = (a_1,\ldots,a_n)$ and set $a'_i = [a_i/2^r]$, $idlist = idlist_c \oplus idlist_s$. Then, let $(b,c) = \overline{\mathrm{MIN}}^k_n(a'_1,a'_2,\ldots,a'_n,idlist,k)$. Sample an array w of size k with random entries in $\{0,1\}^{b_{\mathrm{pid}}}$, output $c \oplus w$ to the client, and w to the server. If returnVal is true, sample a random array s of size k in $\mathbb{Z}_{2^t}$, output $b-s$ to client and s to the server.

Figure 3: Ideal functionality $\mathcal{F}_{\mathrm{TOPk}}$

Parameters: array size m, modulus t, truncation bit size r, output size k, bin size l, ID bit length b_{pid}.
Extra parameter: returnVal $\in \{false, true\}$ (if set to true, return (value, ID) instead of just ID.)

- On input $A_c \in \mathbb{Z}_t^m$ from the client, store A_c.
- On input $A_s \in \mathbb{Z}_t^m$ and *idlist* from the server, store A_s and *idlist*.
- When both inputs are received, compute $A = A_s + A_c \mod t = (a_1, \ldots, a_n)$. and set $a'_i = [a_i/2^r]$. Let $(b,c) = \text{APPROXTOPK}(a'_1, \ldots, a'_n, idlist, k, l)$. Sample an array w of size k with random entries in $\{0,1\}^{b_{\text{pid}}}$. Output $c \oplus w$ to the client, and w to the server. If returnVal is true, sample a random array s of size k, output $b - s$ to client and s to the server.

Figure 4: Ideal functionality $\mathcal{F}_{\text{aTOPk}}$

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.

Figure 5: Ideal functionality $\mathcal{F}_{\text{DROM}}$

4.2 Distance Computation via AHE

We use the BFV scheme [33] to compute distances. Compared to [40], which uses BFV for matrix-vector multiplications, our approach avoids expensive ciphertext *rotations*. Also, we used the coefficient encoding and a plaintext space modulo a power of two instead of a prime. This allows us to later avoid a costly addition modulo p inside a garbled circuit.

More precisely, SIMD for BFV requires plaintext modulus to be prime $p \equiv 1 \mod 2N$. However, it turns out our distance computation protocol only requires multiplication between *scalars* and vectors. Therefore we can drop the requirement and perform computation modulo powers of two without losing efficiency. Recall that plaintext space of the BFV scheme is $R_t := \mathbb{Z}_t[x]/(x^N + 1)$. The client encodes each coordinate in to a constant polynomial $f_i = \mathbf{q}[i]$. Assume the server points are $\mathbf{p}_1, \ldots, \mathbf{p}_N$ for simplicity. It encodes these points into d plaintexts, each encoding one coordinate of all points, resulting in $g_i = \sum_j \mathbf{p}_{j+1}[i] x^j$. Note that $\sum_{i=1}^d f_i g_i = \sum_{j=1}^N \langle \mathbf{q}, \mathbf{p}_j \rangle x^{j-1}$. The client sends encryption of f_i. Then the server computes an encryption $h(x) = \sum_i f_i g_i$, masks $h(x)$ with a random polynomial and sends back to the client, so they hold secret shares of $\langle \mathbf{q}, \mathbf{p}_j \rangle$ modulo t. Then, secret shares of Euclidean distances modulo t can be reconstructed via local operations.

Note that we need to slightly modify the above routine when computing distances of points retrieved from DORAM. Since the server does not know these points in the clear, we let client and server secret share the points and their squared Euclidean norms.

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

1. Client calls AHE.Keygen to get sk; server randomly permutes its points. They both quantize their points into $\mathbf{q}', \mathbf{p}'_i \in \mathbb{Z}_{2^{b_c}}^d$.
2. Client sends $c_i = \text{AHE.Enc}(sk, \mathbf{q}'[i])$ for $1 \le i \le d$ to the server.
3. 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 $\mathbf{r} \in \mathbb{Z}_t^n$ and computes for $1 \le k \le \lceil n/N \rceil$
$$f_k = \sum_{i=1}^d \text{AHE.CMult}(c_i, \mathbf{p}_{ik}) + \mathbf{r}[kN : (k+1)N]$$
.
4. Server sends f_k to client who decrypts them to $\mathbf{s} \in \mathbb{Z}_t^n$.
5. Client sends $-2\mathbf{s} + ||\mathbf{q}'||^2 \cdot (1, 1, \ldots, 1)$ to $\mathcal{F}_{\text{aTOPk}}$, server sends *idlist* and $(-2r_i + ||\mathbf{p}'_i||^2)_i$ to $\mathcal{F}_{\text{aTOPk}}$, with parameters $(k, l_s, false)$. They output $[\mathbf{id}]_c$, $[\mathbf{id}]_s \in \{0,1\}^{b_{\text{pid}}}$. Server sends $[\mathbf{id}]_s$ to client, who outputs $\mathbf{id} = [\mathbf{id}]_c \oplus [\mathbf{id}]_s$.

Figure 6: SANNS linear-scan protocol $\Pi_{\text{ANN}_{\text{ls}}}$.

4.3 Point Retrievals via DORAM

We briefly explain the functionality of Floram and refer the reader to the original paper [27] 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) [34] 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 $u^A_j \oplus u^B_j = 1$ iff $j = i$, where i is the address we are retrieving. Then, party A computes $\bigoplus_j u^A_j \cdot \overline{DB}[i]$ and, likewise, party B computes $\bigoplus_j u^B_j \cdot \overline{DB}[i]$. 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 PRFs and XOR to remove the masks.[3]

We implemented Floram with a few optimizations described below.

Precomputing OT To run FSS, the 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. We use Beaver OT precomputation protocol [10] 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.

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

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_c^i, 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; server also inputs a list of n IDs *idlist*, and all cluster centers $\mathbf{c}_j^i$.

1. Client calls AHE.Keygen to get sk.
2. Client and server quantize their points and the cluster centers.
3. Server sends all clusters with one block per cluster, and each point accompanied by its ID and squared norm, to $\mathcal{F}_{\text{DROM}}$.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 6 to compute secret shares of the vector $(||\mathbf{q}-\mathbf{c}_j^i||_2^2)_j$.
   - Client and server send their shares to $\mathcal{F}_{\text{aTOPk}}$ with $k = u^i$, $l = l^i$ and returnVal = false, when server inputs the default $idlist = \{0,1,\ldots,k_c^i - 1\}$. They obtain secret shares of indices $j_1^i,\ldots,j_{u_i}^i$.
6. Client and server input secret shares of all cluster indices $\{(i,j_c^i) : i \in [1,T], c \in [1,u^i]\}$ obtained in step 5 into $\mathcal{F}_{\text{DROM}}$.Read, to retrieve Boolean secret shares of tuples $(\mathbf{p}, \text{ID}(\mathbf{p}), ||\mathbf{p}||^2)$ of all points in the corresponding clusters. They convert $\mathbf{p}$ and $||\mathbf{p}||^2$ to arithmetic secret shares using e.g. the B2A algorithm in [23].
7. Client and server use line 3-6 in Figure 6 to get secret shares of a distance vector for all points determined in step 6. Then, they input their shares of points and IDs to $\mathcal{F}_{\text{TOPk}}$ with returnVal = true, and output secret shares of a list of tuples $(d_i^{Cluster}, \text{ID}_i^{Cluster})_{i=1}^k$.
8. For the stash S, client and server use line 3-6 in Figure 6 to obtain the secret shared distance vector. Then, they input their shares (while server also inputs IDs of stash points and client input a zero array for its ID shares) to $\mathcal{F}_{\text{aTOPk}}$ with parameters $(k, l_s, true)$, and output shares of $(d_i^{Stash}, \text{ID}_i^{Stash})_{i=1}^k$.
9. Each party inputs the union of shares of (point, ID) pairs obtained from steps 7-8 to $\mathcal{F}_{\text{TOPk}}$ with returnVal=false, and outputs secret shares of k IDs. Server sends its secret shares of IDs to the client, who outputs the final list of IDs.

Figure 7: SANNS clustering-based protocol $\Pi_{\text{ANN}_{\text{cl}}}$.

Kreyvium as PRF Floram implemented PRF using AES. While computing AES is fast in plaintext due to Intel AES instructions, it requires many gates to be evaluated within a garbled circuit. We propose a more efficient solution based on Kreyvium [20] which requires significantly fewer number of AND gates (see Appendix B for various related trade-offs). Evaluating Kreyvium 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 In Floram, accessing the database at k different locations requires $k \log_2 n$ number of interactions. In our case, these memory accesses are non-adaptive, hence we can fuse these accesses and reduce the number of rounds to $\log_2 n$ which has significant effect in practice.

4.4 Top-k Selection via Garbled Circuit

We implement secure top-k selection using garbled circuit while we made some further optimizations to improve the performance. First, we truncate distances by simply discarding some lower order bits, which allows us to reduce the circuit size significantly (see Section 3.2). The second optimization comes from the implementation side. Recall that existing MPC frameworks such as ABY [23] require storing the entire circuit explicitly with accompanying bloated data structures. However, our top-k circuit is highly structured, 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 can be essentially independent of n, which makes them much more cache-efficient. To accomplish this, we developed our own garbled circuit implementation with most of the standard optimizations [11,12,41,71][4], which allows us to save more than an order of magnitude in both time and memory usage compared to ABY.

5 Implementation and Performance Results

5.1 Environment

We perform the evaluation on two Azure `F72s_v2` instances (with 72 *virtual* cores equivalent to that of Intel Xeon Platinum 8168 and 144 GB of RAM each). We have two sets of experiments: for *fast* and *slow* networks. For the former we use two instances from the "West US 2" availability zone (latency 0.5 ms, throughput from 500 MB/s to 7 GB/s depending on the number of simultaneous network connections), while for the latter we run on instances hosted in "West US 2" and "East US" (latency 34 ms, throughput from 40 MB/s to 2.2 GB/s). We use g++ 7.3.0, Ubuntu 18.04, SEAL 2.3.1 [52] and libOTe [59] for the OT phase (in the single-thread mode due to unstable behavior when run in several threads). For networking, we use ZeroMQ. We implement balanced clustering as described in Section 3.3 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 2.5).

5.2 Datasets

We evaluate SANNS algorithms as well as baselines on four datasets: SIFT ($n = 1\,000\,000$, $d = 128$) is a standard dataset of image descriptors [48] that can be used to compute similarity between images; Deep1B ($n = 1\,000\,000\,000$, $d = 96$) is also a dataset of image descriptors [8], which is built from feature vectors obtained by passing images through a deep neural network (for more details see the original paper [8]), Amazon ($n = 2^{20}$, $d = 50$) is dataset of reviews [51], where feature vectors are obtained using word embeddings. We conduct the

[4] For oblivious transfer, we use libOTe [59]

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. For Amazon, we take 2^{20} Amazon reviews of the CDs and Vinyls category, and create a vector embedding for each review by processing GloVe word embeddings [56] as in [5]. SIFT comes with 10 000 sample queries which are used for evaluation; for Deep1B-1M, Deep1B-10M and Amazon, a sample of 10 000 data points from the dataset are used 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 all four datasets have been extensively used in nearest neighbors benchmarks and information retrieval tasks. In particular, SIFT is a part of ANN Benchmarks [7], where a large array of NNS algorithms has been thoroughly evaluated. Deep1B has been used for evaluation of NNS algorithms in, e.g., [8, 39, 49]. Various subsets of the Amazon dataset have been used to evaluate the accuracy and the efficiency of k-NN classifiers in, e.g., [28, 44].

5.3 Parameters

Accuracy In our experiments, we require the algorithms to return $k_{\text{nn}} = 10$ nearest neighbors and measure accuracy as the average portion of correctly returned points over the set of queries ("10-NN accuracy"). Our algorithms achieve 10-NN accuracy at least 0.9 (9 out of 10 points are correct on average), which is a level of accuracy considered to be acceptable in practice (see, e.g., [43, 45]).

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 values of all the coordinates. For Amazon we do the same but with 9 bits. For these datasets, quantization barely affects the 10-NN accuracy compared to using the true floating point coordinates.

Cluster size balancing As noted in Section 3.3, 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, Amazon and Deep1B-1M, the balancing algorithm reduced the maximum cluster size by factors of 4.95×, 3.67×, 3.36× and 3.31×, respectively.

Parameter choices We initialized the BFV scheme with parameters $N = 2^{13}$, $t = 2^{24}$ for Amazon and $t = 2^{23}$ for the other datasets, 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.[5] The LWE estimator[6] by Albrecht et al. [2] suggests 141 bits of computational security.

Here is how we set the hyperparameters for our algorithms. See Figure 1 for the full list of hyperparameters, below we list the ones that affect the performance:

- Both algorithms depend on n, d, k_{nn}, which depend on the dataset and our requirements;
- The 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 $1 \le i \le T$.

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 during hyperparameter search phase (this is due to the complexity of garbling a circuit depending heavily on the number of AND gates due to the Free-XOR optimization [41]). Moreover, for simplicity we neglect the FSS part of ORAM, since it does not affect the performance much. Overall, we search for the hyperparameters that yield 10-NN accuracy at least 0.9 minimizing the total number of AND-gates. In Figure 5 and Figure 6 of Appendix A, we summarize the parameters we use for both algorithms on each dataset.

5.4 SANNS End-to-End Evaluation

Single-thread We run SANNS on the above mentioned four datasets using two algorithms (linear scan and clustering) over fast and slow networks in a single-thread mode, summarizing results in Table 1. We measure per-client preprocessing of Floram separately and split the query measurements into the OT phase, distance computation, approximate top-k selection and ORAM retrievals. For each of the components, we report communication and average running time for fast and slow networks. We make several observations:

- On all the datasets, clustering-based algorithm is much faster than linear scan: up to 12× over the fast network and up to 8.2× over the slow network.
- For the clustering algorithm, per-client preprocessing is very efficient. In fact, even if there is a *single* query per client, clustering algorithm with preprocessing is faster than the linear scan.
- In terms of communication, distance computation part is negligible, and the bottleneck is formed by the top-k selection and ORAM (which are fairly balanced).
- As a result, when we move from fast to slow network, the time for distance computation stays essentially the same, while the time for top-k and ORAM goes up dramatically. This makes our new circuit for approximate top-k selection and optimizations to Floram absolutely crucial for the overall efficiency.

Multi-thread In Table 2 we summarize how the performance

[5] We refer the reader to [40] for details on the noise flooding technique

[6] We used commit 3019847 from https://bitbucket.org/malb/lwe-estimator

of SANNS depends on the number of threads. We only measure the query time excluding the OT phase, since libOTe is unstable when used from several threads. We observe that the speed-ups obtained this way are significant (up to 8.4× for the linear scan and up to 7.1× for clustering), though they are far from being linear in the number of threads. We attribute it to both of our algorithms being mostly memory- and network-bound. Overall, the multi-thread mode yields query time under 6 *seconds* (taking the single-threaded OT phase into account) for our biggest dataset that consists of *ten million* 96-*dimensional vectors.*

5.5 Microbenchmarks

As we discussed in the Introduction, all the prior work that has security guarantees similar to SANNS implements linear scan. Thus, in order to provide a detailed comparison, we compare our approaches in terms of distance computation and top-k against the ones used in the prior work.

Top-k selection We evaluate the new protocol for the approximate top-k selection via garbling the circuit designed in Section 3.1 and compare it with the naïve circuit obtained by a direct implementation of Algorithm 1. The latter was used in some of the prior work on the secure k-NNS [6, 61, 64]. We assume the parties start with arithmetic secret shares of $n = 1\,000\,000$ 24-bit integers. We evaluate both of the above approaches for $k \in \{1, 5, 10, 20, 50, 100\}$. For the approximate selection, we set the number of bins l such that on average we return $(1-\delta)\cdot k$ entries correctly for $\delta \in \{0.01, 0.02, 0.05, 0.1\}$, using the formula from the proof of Theorem 1. For each setting, we report average running time over slow and fast networks as well as the total communication. Table 4 summarizes our experiments. As expected, the performance of the approximate circuit is essentially independent of k, whereas the performance of the naïve circuit scales linearly as k increases. Even if we allow the error of only $\delta = 0.01$ (which for $k = 100$ means we return a *single* wrong number), the performance improves by a factor up to 25 on the fast network and up to 37 on the slow network.

The works [62, 63] used fully-homomorphic encryption (FHE) for the top-k selection. However, even if we use TFHE [22], which is by far the most efficient FHE approach for highly-nonlinear operations, it will still be several orders of magnitude slower than garbled circuits, since TFHE requires several milliseconds per gate, whereas GC requires less than a microsecond.

Distance Computation The most efficient way to compute n Euclidean distances securely, besides using the BFV scheme, is arithmetic MPC [23] based on oblivious transfer (one other alternative used in many prior works [9, 29–31, 60] is Paillier AHE scheme, which is known to be much less suitable for the task due to the absence of SIMD capabilities [40]). Let us compare BFV scheme used in SANNS with the OT-based distance computation from [23] with an optimization from [53]. The latter allows to compute n l-bit distances between d-dimensional vectors ($l = 24$ for Amazon, $l = 23$ for all the other datasets), using $ndl(l+1)/256$ OTs of 128-bit strings. We perform those OTs using libOTe for each of our datasets and measure time (over fast and slow networks) as well as communication. The results are summarized in Table 3. As expected, the communication required by OT-based multiplication is much larger than for AHE (by a factor up to 127×). As a result, for the slow network, OT-based multiplication is noticeably slower, by a factor up to 7.5×; for the fast network, OT-based approach is no more than 4% faster than AHE.

5.6 End-to-End Comparison with Prior Work

We have shown that individual components used by SANNS are extremely competitive compared to the ones proposed by the prior work. Here, we provide the end-to-end performance results on the largest dataset we evaluated SANNS on: Deep1B-10M. For the fast network, our linear scan requires 395 seconds per query (taking the OT phase into account), and clustering requires 31 seconds; for the slow network, it is 1720 and 194 seconds, respectively (see Table 1).

One issue with a fair comparison with the prior work is that they are done before the recent MPC and HE optimizations became available. Based on the benchmarks in the previous section, one can definitively conclude that the fastest protocol from the prior work is from [23]. Namely, we compute distances using OT with the optimization from [53], and perform the top-k selection using garbled circuit with the naïve circuit in Algorithm 1. To estimate the running time of this protocol, we use Table 3 for distances and we run a separate experiment for naïve top-k for $n = 10^7$ and $k = 10$. This gives us *the lower bound* on the running time of 578 seconds on the fast network and 6040 seconds on the slow network, and the lower bound of 240 GB on the communication.

Overall, this indicates that our linear scan obtains a speed-up of 1.46× on the fast network and 3.51× on the slow network. The clustering algorithm yields the **speed-up of** 18.5× **on the fast network and** 31.0× **on the slow network**. The improvement in communication is 4.1× for the linear scan and 39× for the clustering algorithm.

Note that these numbers are based on the lower bounds for the runtime of prior work and several parts of the computation and communication of their end-to-end solution are not included in this comparison. In particular, just computing distances using the original implementation from [23] on SIFT dataset takes 620 seconds in the fast network, *more than* 32× *higher compared against our assumed lower bound of* 19.1 *seconds in Table 3*. When scaling their implementation to ten million points, the system runs out of memory (more than 144 GB of RAM is needed). In conclusion, the speed-up numbers we reported reflect running the best prior algorithm using our

	Algorithm	Per-client Preprocessing	OT Phase	Query			
				Total	Distances	Top-k	ORAM
SIFT	Linear scan	None	1.83 s / 21.6 s 894 MB	33.3 s / 139 s 4.51 GB	19.8 s / 25.6 s 98.7 MB	13.5 s / 111 s 4.41 GB	None
	Clustering	12.6 s / 24.7 s 484 MB	0.346 s / 4.34 s 156 MB	8.06 s / 59.7 s 1.77 GB	2.21 s / 3.67 s 56.7 MB	1.96 s / 18.0 s 645 MB	3.85 s / 38.1 s 1.06 GB
Deep 1B-1M	Linear scan	None	1.85 s / 20.6 s 894 MB	28.4 s / 133 s 4.50 GB	14.9 s / 20.6 s 86.1 MB	13.5 s / 112 s 4.41 GB	None
	Clustering	11.0 s / 20.6 s 407 MB	0.323 s / 4.09 s 144 MB	6.95 s / 47.8 s 1.58 GB	1.66 s / 3.13 s 44.1 MB	1.93 s / 16.6 s 620 MB	3.37 s / 27.9 s 920 MB
Deep 1B-10M	Linear scan	None	20.0 s / 232 s 9.78 GB	375 s / 1490 s 47.9 GB	202 s / 201 s 518 MB	173 s / 1280 s 47.4 GB	None
	Clustering	86.0 s / 167 s 3.71 GB	1.04 s / 13.4 s 541 MB	30.1 s / 181 s 5.53 GB	6.27 s / 10.2 s 59.4 MB	7.22 s / 61.0 s 2.35 GB	16.5 s / 107 s 3.12 GB
Amazon	Linear scan	None	1.99 s / 23.3 s 960 MB	22.9 s / 133 s 4.85 GB	8.27 s / 14.0 s 70.0 MB	14.6 s / 118 s 4.78 GB	None
	Clustering	7.27 s / 13.4 s 247 MB	0.273 s / 3.17 s 108 MB	4.54 s / 35.2 s 1.12 GB	0.68 s / 2.31 s 24.4 MB	1.64 s / 13.8 s 528 MB	2.22 s / 18.8 s 617 MB

Table 1: Evaluation of SANNS in a single-thread mode. Preprocessing is done once per client, OT phase is done once per query. In each cell, timings are given for fast and slow networks, respectively.

	Algorithm	Threads								Speed-up
		1	2	4	8	16	32	64	72	
SIFT	Linear scan	33.3 s 139 s	23.2 s 76.4 s	13.4 s 46.9 s	8.04 s 32.5 s	4.78 s 25.7 s	4.25 s 22.1 s	**3.96 s** **20.9 s**	4.14 s 21.3 s	8.4 6.7
	Clustering	8.06 s 59.7 s	4.84 s 35.2 s	3.16 s 23.6 s	2.18 s 24.4 s	1.65 s 20.1 s	1.55 s 14.2 s	**1.44 s** **11.1 s**	1.47 s 12.1 s	5.6 5.4
Deep 1B-1M	Linear scan	28.4 s 133 s	19.9 s 75.5 s	11.4 s 44.5 s	7.39 s 31.9 s	4.53 s 24.5 s	**3.94 s** 22.0 s	**3.94 s** 22.5 s	4.05 s **21.1 s**	7.2 6.3
	Clustering	6.95 s 47.8 s	4.20 s 28.5 s	2.62 s 22.0 s	2.03 s 23.0 s	1.52 s 18.4 s	1.43 s 14.7 s	**1.37 s** **11.0 s**	1.39 s 11.5 s	5.1 4.3
Deep 1B-10M	Linear scan	375 s 1490 s	234 s 800 s	118 s 480 s	81.8 s 343 s	65.8 s 266 s	55.0 s 231 s	**53.1 s** **214 s**	58.5 s* 216 s*	7.1 7.0
	Clustering	30.1 s 181 s	18.0 s 97.5 s	10.8 s 60.0 s	7.21 s 54.5 s	4.85 s 48.1 s	4.58 s 37.2 s	**4.23 s** 30.3 s	4.25 s **28.4 s**	7.1 6.4
Amazon	Linear scan	22.9 s 133 s	15.4 s 73.1 s	10.1 s 46.1 s	6.66 s 33.8 s	4.14 s 26.2 s	3.73 s 24.1 s	3.78 s 22.0 s	**3.64 s** **21.7 s**	6.3 6.1
	Clustering	4.54 s 35.2 s	2.66 s 21.4 s	1.87 s 14.9 s	1.40 s 16.8 s	1.17 s 14.2 s	1.15 s 11.5 s	**1.12 s** 10.8 s	1.16 s **9.19 s**	4.1 3.8

Table 2: Evaluation of SANNS query algorithms in the multi-thread mode. Each cell contains timings for fast and slow networks. Optimal settings are marked in bold. For the numbers marked with an asterisk, we take the *median* of the running times over several runs, since with small probability (approximately $20-30\%$), memory swapping starts due to exhaustion of all the available RAM, which affects the running times dramatically (by a factor of $\approx 2\times$).

	SIFT	Deep1B-1M	Deep1B-10M	Amazon
AHE	19.8 s / 25.6 s 98.7 MB	14.9 s / 20.6 s 56.7 MB	202 s / 201 s 518 MB	8.27 s / 14.0 s 70 MB
OT-based (lower bound)	19.1 s / 181 s 8.83 GB	14.5 s / 153 s 6.62 GB	204 s / 1510 s 66.2 GB	8.59 s / 88.7 s 3.93 GB

Table 3: Comparison of AHE- and OT-based approach for computing distances. Each cell has two timings: for the fast and the slow networks.

k	Exact	Approximate				Speed-up
		$\delta = 0.01$	$\delta = 0.02$	$\delta = 0.05$	$\delta = 0.1$	
1	11.1 s / 93.9 s 3.48 GB	N/A	N/A	N/A	N/A	N/A
5	22.4 s / 249 s 9.62 GB	10.5 s / 90.6 s 3.48 GB	10.6 s / 88.8 s 3.48 GB	10.5 s / 94.5 s 3.48 GB	10.7 s / 90.6 s 3.48 GB	2.1 / 2.7
10	36.1 s / 448 s 17.3 GB	10.7 s / 86.9 s 3.48 GB	10.6 s / 91.2 s 3.48 GB	11.0 s / 89.6 s 3.48 GB	11.0 s / 91.3 s 3.48 GB	3.4 / 5.2
20	67.8 s / 821 s 32.7 GB	10.6 s / 95.2 s 3.50 GB	10.7 s / 94.0 s 3.49 GB	10.8 s / 92.9 s 3.48 GB	10.6 s / 93.8 s 3.48 GB	6.4 / 8.6
50	153 s / 2100 s 78.7 GB	11.1 s / 99.2 s 3.66 GB	10.6 s / 97.4 s 3.57 GB	10.5 s / 94.5 s 3.51 GB	10.5 s / 94.1 s 3.49 GB	14 / 21
100	301 s / 4130 s 156 GB	12.0 s / 113 s 4.22 GB	12.0 s / 98.3 s 3.85 GB	10.8 s / 96.0 s 3.62 GB	11.2 s / 98.6 s 3.55 GB	25 / 37

Table 4: Comparison of the exact and the approximate top-k selection protocols (selecting from one million values). Each cell has two timings: for the fast and the slow networks. We report the speed-ups for fast and slow networks between the approximate algorithm with error rate $\delta = 0.01$ and the exact algorithm.

new optimized implementation, which leads to a more fair comparison (SANNS speed-up is significantly higher if the *original* implementations of prior works are considered).

6 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 AHE, 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 is secure in the semi-honest model, but it would be interesting to extend our protocols to protect against malicious adversaries which can deviate from the protocol.
- One possible future direction is 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.
- It is important to study to what extent k-NNS queries leak information about the dataset and how much approximation in the answers adds to this leakage. For instance, the client may try to locate individual points in a dataset by asking several queries that are perturbations of each other and checking if the point of interest ends up in the answer. For *low-dimensional* datasets there are known strong recovery attacks [42], but for the high-dimensional case—which is the focus of this paper—the possibility of such attacks remains open. Besides attacks, an interesting research direction is how to restrict the client (in the number of k-NNS queries or the degree of adaptivity) so to minimize the dataset leakage.

That being said, let us state a few simple observations about additional leakage that can happen due to approximation in the results. There are two sources of approximation: approximate top-k selection and clustering-based k-NNS algorithm. For the sake of simplicity, let us discuss the effects of these components separately. For the former, one can show that the probability that the element with rank $l > k$ is included in the output is exponentially small in $l - k$. For the latter, we can notice the following. First, we never leak more than the union of the sets of points closest to the query in the clusters whose centers are closest to the query. Second, if the dataset is clusterable (i.e., can be partitioned into clusters with pairwise distances being significantly larger than the diameters of the clusters) and queries are close to clusters, then the clustering based k-NNS algorithm is exact and there is no additional leakage due to approximation.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and helpful comments. This work was partially done while all the authors visited Microsoft Research Redmond.

The second-named author has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the FWO under an Odysseus project GOH9718N and by the CyberSecurity Research Flanders with reference number VR20192203. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ERC or FWO.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9. ACM, 1983.

[2] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[3] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, pages 430–454, 2015.

[4] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, pages 1225–1233, 2015.

[5] S. Arora, Y. Liang, and T. Ma. A simple but tough-to-beat baseline for sentence embeddings. In *International Conference on Learning Representations*, pages 43–52, 2017.

[6] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin. Privacy-preserving search of similar patients in genomic data. *Proceedings on Privacy Enhancing Technologies*, 2018(4):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*, pages 34–49. Springer, 2017.

[8] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE CVPR*, pages 2055–2063, 2016.

[9] 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 *ACM MM& Sec*, pages 231–240, 2010.

[10] D. Beaver. Precomputing oblivious transfer. In *CRYPTO*, pages 97–109. Springer, 1995.

[11] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, volume 90, pages 503–513, 1990.

[12] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*, pages 478–492. IEEE, 2013.

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

[14] D. J. Bernstein. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*, pages 84–97. 2008.

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

[16] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[17] 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 *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[18] J. Boyar and R. Peralta. A small depth-16 circuit for the AES s-box. In *SEC*, pages 287–298, 2012.

[19] C. D. Cannière and B. Preneel. Trivium. In *New Stream Cipher Designs - The eSTREAM Finalists*, pages 244–266. 2008.

[20] 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 *FSE*, pages 313–333, 2016.

[21] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE TPDS*, 25(1):222–233, 2013.

[22] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, pages 3–33. Springer, 2016.

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

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

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

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

[27] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535. ACM, 2017.

[28] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner. Scalable nearest neighbor search for optimal transport. *arXiv preprint arXiv:1910.04126*, 2019.

[29] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*, pages 664–675. IEEE, 2014.

[30] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *PETS*, pages 235–253. Springer, 2009.

[31] D. Evans, Y. Huang, J. Katz, and L. Malka. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.

[32] D. Evans, V. Kolesnikov, M. Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[33] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[34] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658. Springer, 2014.

[35] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.

[36] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *ACM STOC*, pages 684–693, 2014.

[37] G. N. Gopal and M. P. Singh. Secure similarity based document retrieval system in cloud. In *ICDSE*, pages 154–159. IEEE, 2012.

[38] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

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

[40] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security*, 2018.

[41] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.

[42] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S& P*, pages 1033–1050, 2019.

[43] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th international conference on computer vision*, pages 2130–2137, 2009.

[44] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger. From word embeddings to document distances. In *ICML*, pages 957–966, 2015.

[45] H. Li, W. Liu, and H. Ji. Two-stage hashing for fast document retrieval. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 495–500, 2014.

[46] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[47] S. Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[48] D. G. Lowe et al. Object recognition from local scale-invariant features. In *ICCV*, volume 99, pages 1150–1157, 1999.

[49] 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.

[50] C. M. Manoj and G. K. Mrs Sandhia. Privacy preserving similarity based file retrieval through blind storage.

[51] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*, pages 43–52. ACM, 2015.

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

[53] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S& P*, pages 19–38, 2017.

[54] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[55] H. Pang, J. Shen, and R. Krishnan. Privacy-preserving similarity-based text retrieval. *TOIT*, 10(1):4, 2010.

[56] J. Pennington, R. Socher, and C. D. Manning. Glove. In *EMNLP*, pages 1532–1543, 2014.

[57] I. Razenshteyn. *High-dimensional similarity search and sketching: algorithms and hardness*. PhD thesis, Massachusetts Institute of Technology, 2017.

[58] M. S. Riazi, B. Chen, A. Shrivastava, D. Wallach, and F. Koushanfar. Sub-linear privacy-preserving near-neighbor search. *arXiv preprint arXiv:1612.01835*, 2016.

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

[60] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*, pages 229–244. Springer, 2009.

[61] P. Schoppmann, A. Gascón, and B. Balle. Private nearest neighbors classification in federated databases. *IACR Cryptology ePrint Archive*, 2018:289, 2018.

[62] 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.

[63] H. Shaul, D. Feldman, and D. Rus. Secure *k*-ish nearest neighbors classifier. *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 *DAC*, pages 1–6. IEEE, 2015.

[65] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In *ASIA CCS*, pages 71–82. ACM, 2013.

[66] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[67] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, pages 850–861. ACM, 2015.

[68] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202. ACM, 2014.

[69] W. Wu, U. Parampalli, J. Liu, and M. Xian. Privacy preserving *k*-nearest neighbor classification over encrypted database in outsourced cloud environments. *World Wide Web*, 22(1):101–123, 2019.

[70] A. C.-C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science*, pages 162–167. IEEE, 1986.

[71] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, pages 220–250. Springer, 2015.

[72] 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 *S&P*, pages 218–234. IEEE, 2016.

A Chosen Hyperparameters in Clustering-Based Algorithm

In Table 5 and Table 6, we summarize the parameters we use for both of our algorithms on each of the datasets.

	Linear scan				Clustering			
Params	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon
l_s	8334	8334	83	8739	262	210	423	84
b_c	8	8	8	9	8	8	8	9
r_p	8	8	9	7	8	8	8	6

Table 5: (Near-)optimal hyperparameters that are used both by linear scan and the clustering-based algorithm.

Params	SIFT	Deep 1B-1M	Deep 1B-10M	Amazon
T	4	5	6	5
k_c^i	50810 25603 9968 4227	44830 25867 11795 5607 2611	209727 107417 39132 14424 5796 2394	41293 24143 9708 3516 1156
m	20	22	48	25
u^i	50 31 19 13	46 31 19 13 7	88 46 25 13 7 7	37 37 22 10 7
s	31412	25150	50649	8228
l^i	458 270 178 84	458 270 178 84 84	924 458 178 93 84 84	364 364 178 84 84
r_c	5	5	5	4
α	0.56	0.56	0.56	0.56

Table 6: (Near-)optimal hyperparameters that are specific to the clustering-based algorithm.

B Stream Ciphers as PRF

In the original Floram construction [25–27], the PRF and the PRG used in the read-only process are chosen by the authors to be AES-128. The implementations of AES are highly optimized, with less than 5000 non-free gates per block [18]. As

an alternative to AES, the authors also propose the streams Salsa20 [14] and its variant Chacha20 [13]. 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). Some of the most promising constructions are the block cipher LowMC [3] and the stream cipher Kreyvium [20] (variant of Trivium [19]). In particular Kreyvium is flexible in terms of input and output size, since there is no fixed block size to respect, and its evaluation is very efficient in terms of AND gates per output bit of stream. The advantage in using Kreyvium starts showing when the size of the inputs starts growing. In Table 7 we estimate the number of AND gates that are executed by the different ciphers for 3 dataset sizes. We compute 2 PRFs per input, so the actual number of AND gates in Table 7 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 7: 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 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 I is:
$\mathrm{E}[|I|] = l \cdot \Pr[U_i \text{ contains at least one of the top-}k \text{ elements}]$
$= l \cdot \left(1 - \left(1 - \frac{1}{l}\right)^k\right)$, 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,

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

where the second step uses the Taylor series of $\ln x$, the third step uses the Taylor series of e^x and the fourth step uses the inequality $e^{-x} \leq 1 - x + \frac{x^2}{2}$, which holds for small $x > 0$.

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 [24]. 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:
$\Pr[\text{all top-}k \text{ elements end up into different bins}]$
$= \left(1 - \frac{1}{l}\right) \cdot \left(1 - \frac{2}{l}\right) \cdot \ldots \cdot \left(1 - \frac{k-1}{l}\right)$
$= \left(1 - \frac{\delta}{k^2}\right) \cdot \left(1 - \frac{2\delta}{k^2}\right) \cdot \ldots \cdot \left(1 - \frac{(k-1)\cdot\delta}{k^2}\right)$
$= \exp\left(\ln\left(1 - \frac{\delta}{k^2}\right) + \ln\left(1 - \frac{2\delta}{k^2}\right) + \ldots + \ln\left(1 - \frac{(k-1)\cdot\delta}{k^2}\right)\right)$
$= \exp\left(-\frac{\delta(1+2+\ldots+(k-1))}{k^2} + O\left(\frac{1}{k}\right)\right)$
$= e^{-\delta/2} + O\left(\frac{1}{k}\right) \geq 1 - \frac{\delta}{2} + O\left(\frac{1}{k}\right),$
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:
$\Pr[c_1 = j_1 \wedge c_2 = j_2 \wedge \ldots \wedge c_k = j_k]$
$= \Pr[c_1 = j_1] \cdot \Pr[c_2 = j_2 \mid c_1 = j_1] \cdot \ldots$
$\cdot \Pr[c_k = j_k \mid c_1 = j_1 \wedge \ldots \wedge c_{k-1} = j_{k-1}]$.
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 condi-

tion on t equalities, then the probability is equal to $\frac{n}{l(n-t)}$. This implies the required monotonicity result. □

D Security Proofs

We prove simulation-based security for our protocols for approximate k-NNS. First, we recall the definition (see e.g. [46]) of two party computation and simulation-based security for semi-honest adversaries.

Definition 1. *A two-party functionality is a possibly randomized function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, that is, for every pair of inputs $x, y \in \{0,1\}^n$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$. The first party (with input x) obtains $f_1(x,y)$ and the second party (with input y) obtains $f_2(x,y)$.*

Let π be a protocol computing the functionality f. The *view* of the i-th party during an execution of π on (x,y) and security parameter λ is denoted by $\text{View}_{\pi,i}(x,y,\lambda)$ and equals the party i's input, 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*
$(S_1(1^\lambda, x, f_1(x,y)), f(x,y)) \approx (\text{View}_{\pi,1}(x,y,\lambda), f(x,y))$ *and* $(S_2(1^\lambda, y, f_2(x,y)), f(x,y)) \approx (\text{View}_{\pi,2}(x,y,\lambda), f(x,y))$. *Here $\approx$ means computational indistinguishability.*

D.1 Ideal Functionalities

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 $\mathcal{F}_{\text{ANN}_{\text{cl}}}$ (for clustering) and $\mathcal{F}_{\text{ANN}_{\text{ls}}}$ (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 $DB = [(\mathbf{p}_i, \text{ID}_i)]_{i=1}^n$. Note that points are truncated to b_c bits.
- Output: returns output of Algorithm 3 to client.

Figure 8: Ideal functionality $\mathcal{F}_{\text{ANN}_{\text{ls}}}$

Parameters: number of elements n, dimension d, bits of precision b_c, and clustering-based hyperparameters T, k_c^i, m, u^i, s, l^i, l_s, b_c, r_c and r_p.
- Input: client inputs a query $\mathbf{q} \in \mathbb{R}^d$ and server inputs database $DB = [(\mathbf{p}_i, \text{ID}_i)]_{i=1}^n$. The points are truncated to b_c bits.
- Output: returns output of Algorithm 4 to client.

Figure 9: Ideal functionality $\mathcal{F}_{\text{ANN}_{\text{cl}}}$

D.2 Proofs

Theorem 3. *Assuming the hardness of the decision-RLWE problem, our linear scan protocol $\Pi_{\text{ANN}_{\text{ls}}}$ securely implements the functionality $\mathcal{F}_{\text{ANN}_{\text{ls}}}$ in the $\mathcal{F}_{\text{aTOPk}}$ 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 $\mathcal{F}_{\text{aTOPk}}$ 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 $1 \le i \le d$. Based on the RLWE assumption, these views are indistinguishable. Finally, the simulator generates random vector $R = (r_1, \ldots, r_n)$ and sends that to the server. □

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

Proof. Again correctness is easy to verify. We first describe 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 $\mathcal{F}_{\text{DROM}}$.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 $\mathcal{F}_{\text{aTOPk}}$ and gets back secret shares of indices, namely $[i_1], \ldots, [i_u]$. Then, it feeds these indices shares to $\mathcal{F}_{\text{DROM}}$.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 $m \cdot u_{\text{all}} + s$ of (point, ID) pairs from the client, it samples $m \cdot u_{\text{all}} + s$ random pairs of values and send the first $m \cdot u_{\text{all}}$ values to $\mathcal{F}_{\text{TOPk}}$ and the last s values to $\mathcal{F}_{\text{aTOPk}}$. 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 uniform, based on the RLWE assumption. □