Private Approximate Nearest Neighbor Search with Sublinear Communication

Sacha Servan-Schreiber
MIT CSAIL

Simon Langowski
MIT CSAIL

Srinivas Devadas
MIT CSAIL

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833702

***Abstract*—Nearest neighbor search is a fundamental building-block for a wide range of applications. A privacy-preserving protocol for nearest neighbor search involves a set of clients who send queries to a remote database. Each client retrieves the nearest neighbor(s) to its query in the database *without* revealing any information about the query. To ensure database privacy, clients must learn as little as possible beyond the query answer, even if behaving maliciously by deviating from protocol.**

Existing protocols for private nearest neighbor search require heavy cryptographic tools, resulting in high computational and bandwidth overheads. In this paper, we present the first *lightweight* protocol for private nearest neighbor search. Our protocol is instantiated using two non-colluding servers, each holding a replica of the database. Our design supports an arbitrary number of clients simultaneously querying the database through the two servers. Each query consists of a single round of communication between the client and the two servers. No communication is required between the servers to answer queries.

If at least one of the servers is non-colluding, we ensure that (1) no information is revealed on the client's query, (2) the total communication between the client and the servers is sublinear in the database size, and (3) each query answer only leaks a small and bounded amount of information about the database to the client, even if the client is malicious.

We implement our protocol and report its performance on real-world data. Our construction requires between 10 and 20 seconds of query latency over large databases of 10M feature vectors. Client overhead remained under 10 ms of processing time per query and less than 10 MB of communication.

I. Introduction

Nearest neighbor search is used in a wide range of online applications, including recommendation engines [23, 81], reverse image search [56], image-recognition [57], earthquake detection [86], computational linguistics [62], natural-language processing [73], targeted advertising [72, 82], and numerous other areas [53, 58, 68, 72].

In these settings, a server has a database of high-dimensional feature vectors associated with items. Clients send query vectors to the server to obtain the set of items (a.k.a. neighbors) that have similar vectors relative to the issued query. Typically, the client only obtains the identifiers (IDs) of the neighbors rather than the feature vectors themselves, as the server may wish to keep the feature vectors private. The IDs of the feature vectors can be documents, songs, or webpages, and therefore all the client requires as output for correct functionality.

For a concrete example, consider a music recommendation engine such as Spotify. The Spotify server holds a database of song feature vectors. Each feature vector can be seen as a concise representation of song attributes—e.g., genre, popularity, user ratings—encoded in a high-dimensional vector space. A Spotify user has a vector of features (the query) representing their musical interests. The goal is to recommend songs the user may find interesting, which should have similar features. This is done using nearest neighbor search to find the ID (e.g., the song) of a vector similar to the query. The client learns the recommended song *without* learning the potentially proprietary feature vector associated with it (beyond what can be implicitly inferred through similarity with the query).

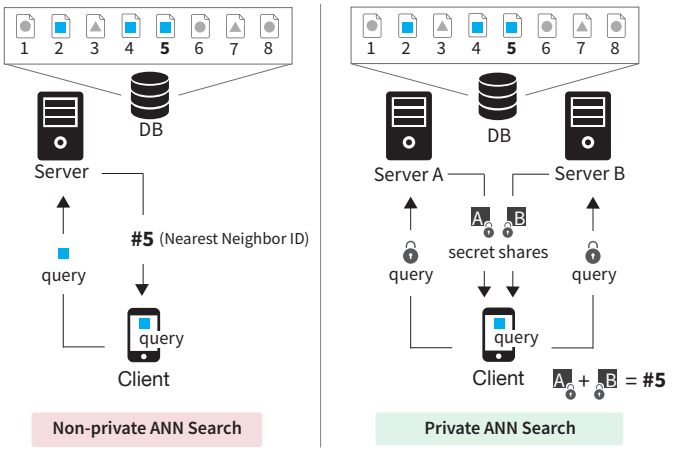

Fig. 1: Overview of approximate nearest neighbor search and the privacy-preserving protocol considered in this paper. In the private setting, a client with a query (blue square) interacts with a remote database via two non-colluding servers (Servers A and B). The client combines the responses from both servers to obtain the approximate nearest neighbor ID (in this case the ANN ID $= 5$) without revealing the query to the servers.

In the above example, the Spotify database learns the client's features. It is not difficult to see that in applications that involve more sensitive user data, revealing these features can easily violate user privacy. Such applications include targeted advertising [13, 47, 72, 80, 82], biometric data [14, 38], medical records [9, 78], and DNA data analysis [24, 60, 67, 83]. These applications construct queries from highly personal user information. For example, in the medical setting, a person's medical history, demographics, and even DNA can be compiled into a query. The resulting neighbors can consist of other people who have similar symptoms, gene sequences, or health conditions [65]. Both regulatory and ethical reasons dictate that such personal information (represented in the query) should be kept private from the database.

DOI 10.1109/SP46214.2022.00074

The potential privacy issues surrounding similarity search have motivated a handful of privacy-preserving protocols [26, 50, 70, 78, 87]. However, existing protocols are highly inefficient. Prior work either makes use of *heavy* cryptographic tools (e.g., two-party computation and fully-homomorphic encryption) or fails to provide strong privacy guarantees for users (e.g., leak information on the query to the server). See the overview of related work in Section IX.

Additionally, existing protocols do not consider *malicious*[1] clients that may attempt to abuse the system to learn more information about the database. The proposed solutions leave open the problem of designing a concretely efficient protocol for private nearest neighbor search, especially in settings where parties may act maliciously by deviating from protocol.

The contribution of this paper is the design and implementation of a lightweight protocol for private Approximate Nearest Neighbor (ANN) search. Specifically, by *lightweight* we require: (1) minimal communication and computation overhead on the client, and (2) reasonable computational overheads on the database. This is in contrast to prior work which either requires *gigabytes* of communication to instantiate a multi-party computation protocol or uses fully-homomorphic encryption to perform the computation resulting in excessive computational overhead on the database (see related work in Section IX).

Our protocol provides strong security guarantees for both the client and the database. We ensure that our protocol is concretely efficient and requires little communication between the client and the database servers (and no communication between servers). We achieve this without compromising on privacy for the client—nothing is leaked on the client's query. For database privacy, our construction requires some extra database leakage compared to the (minimal) *baseline leakage* which only reveals the ANN to the client. The small additional leakage allows us to eschew oblivious comparisons, which are inefficient to instantiate [84]. However, we are careful to quantify the extra leakage relative to the baseline. In our analysis, we show that the leakage is at most a constant factor worse than the baseline. We empirically show that this constant is at most $16\times$ the baseline leakage on real-world datasets, under worst-case parameters (see Section VII-B).

Private ANN search. We operate in the following model (see Figure 1 for a simplified illustration). Fix a database $\mathcal{DB}$ containing a set of N feature vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}$ and corresponding item identifiers $\mathsf{ID}_1, \ldots, \mathsf{ID}_N$. A client has a query vector $\boldsymbol{q}$. The client must learn only the ID(s) of the nearest neighbor(s) relative to $\boldsymbol{q}$ under some similarity (or distance) metric. For client privacy, the protocol must not leak any information about $\boldsymbol{q}$ to the database servers. For database privacy, the protocol must leak as little as possible on $\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}$ to the client. Observe that perfect database privacy (i.e., no database leakage) is unattainable because the client *must* learn at least one ID corresponding to a neighboring vector in the database, which indicates that the vector associated with the ID is similar to the query. We therefore focus on minimizing extra leakage of the database to the client beyond this baseline.

[1]Existing work requires secure function evaluation between the client and server, which can be "upgraded" to malicious-security at the cost of computationally expensive transformations based on zero-knowledge proofs [44]. The use of fully-homomorphic encryption is another alternative to provide malicious security but comes at a high efficiency cost.

Our approach. We start by redesigning the standard locality-sensitive-hashing based data structure for ANN search with the goal of avoiding oblivious comparisons (the efficiency bottleneck of prior work). We achieve this by replacing brute-force comparisons with a radix-sort [54] inspired approach for extracting the nearest neighbor, *without* sacrificing accuracy. We then show how to query this new data structure through a novel privacy-preserving protocol. Our protocol uses distributed point functions [42] as an existing building block for private information retrieval. We then apply a new tool we call partial batch retrieval (a spin on batch-PIR [8]; see Section V-A) to reduce the server processing overhead when answering queries. Finally, to provide database privacy, we use a new technique we call oblivious masking (Section V-A), which hides all-but-one neighbor ID from the query answer.

We show that our protocol is (1) *accurate* through a theoretical analysis and empirical evaluation on real-world data, (2) *private* by analyzing the security properties of our protocol with respect to client and database privacy, and (3) *efficient* in terms of concrete server-side computation and client-server communication. See Sections VII and VIII for analytical and empirical results, respectively.

Contributions. In summary, this paper makes the following four contributions:

1) design of a single-round protocol for privacy-preserving ANN search, achieving sublinear communication and concrete query processing efficiency,
2) leakage analysis with quantifiable database privacy, which we show asymptotically matches the optimal leakage,
3) security against malicious clients that may deviate from protocol in an attempt to abuse leakage, and
4) an open-source implementation [1] which we evaluate on real-world data with millions of feature vectors.

Limitations. Our protocol has greater database leakage compared to the baseline leakage required for correct functionality. We show that the database leakage is asymptotically optimal, but concretely a small factor worse on real data; see empirical analysis in Section VII. Additionally, in contrast to prior work, our threat model assumes two non-colluding servers. In the full version of this paper [76, Appendix E], we sketch how our techniques also apply to the single-server setting (albeit at a concrete efficiency cost). For now, however, the non-colluding assumption enables lightweight privacy-preserving systems [2, 18, 27, 32, 33, 35–37, 55, 66], including systems deployed in industry [35, 45].

II. Background: Nearest Neighbor Search

We begin by describing the standard (non-private) approach to approximate nearest neighbor search based on locality-sensitive hashing. Even outside of a privacy-preserving context,

nearest neighbor search in higher dimensions ($d \geq 10$) requires tolerating approximate results to achieve efficient solutions [49]. In Section IV, we transform the ideas from non-private ANN search into a private protocol instantiated between a client and two servers holding replicas of the database.

A. Locality-sensitive hashing

The approximate nearest neighbor search problem is solved using hashing-based techniques that probabilistically group similar feature vectors together (see survey of Andoni et al. [6]). Approximate solutions based on Locality-Sensitive Hashing (LSH) provide tunable accuracy guarantees and only require examining a small fraction of feature vectors in the database to find the approximate nearest neighbor(s).

LSH families are defined over a distance metric (such as Euclidean distance) and have the property that vectors close to each other in space hash to the same value with good probability. Formally, for a vector space $\mathcal{D}$, output space $\mathcal{R}$, and a distance metric Δ, an LSH family is defined as:

Definition 1 (Locality-Sensitive Hash (LSH))**.** A family of hash functions $\mathcal{H} := \{\mathsf{h} : \mathcal{D} \to \mathcal{R}\}$ is (R, cR, p_1, p_2)-sensitive for distance metric Δ if for any pair of vectors $\boldsymbol{v}, \boldsymbol{q} \in \mathcal{D}$,

$$\textbf{if } \Delta(\boldsymbol{v}, \boldsymbol{q}) \leq R \textbf{ then } \Pr[\mathsf{h}(\boldsymbol{v}) = \mathsf{h}(\boldsymbol{q})] \geq p_1,$$
$$\textbf{if } \Delta(\boldsymbol{v}, \boldsymbol{q}) \geq cR \textbf{ then } \Pr[\mathsf{h}(\boldsymbol{v}) = \mathsf{h}(\boldsymbol{q})] \leq p_2,$$

where $R < cR$ and $p_1 > p_2$.

Remark 1. Note that an LSH family is usually combined with a universal hash function to map to a fixed output size [34]. Without loss of generality, we will assume that the output of the LSH function is mapped by a universal hash.

LSH for nearest neighbor search. In this work we adapt the data structure of Gionis et al. [43], which is the standard way of solving the ANN search problem using LSH [6]. The data structure consists of two algorithms: BUILD and QUERY (described in Appendix A for completeness). At a high level, BUILD hashes each vector into a hash table using a locality-sensitive hash function. QUERY performs a lookup in the hash table and returns the nearest-neighbor in the colliding bucket. This process is repeated L times to increase accuracy. Because the probability that a nearest neighbor collided with the query in a subset of hash tables can be made arbitrarily high (by tuning parameters), BUILD and QUERY ensure that the nearest neighbor is found with high probability (see Figure 2). In practice, one must query $L \approx \sqrt{N}$ tables (N is the database size) to obtain good accuracy and sublinear query time [6].

III. OVERVIEW

We adapt the standard LSH-based data structure described in Section II into a privacy-preserving protocol between a client with query vector $\boldsymbol{q}$ and two servers with access to replicas of the database. We begin by describing the baseline functionality of private ANN search.

Notation. We denote by $\mathcal{DB}$ the database of vectors and their IDs. We let N be the total number of d-dimensional vectors in

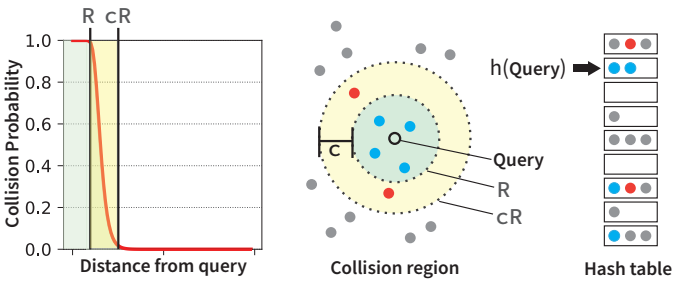

Fig. 2: Visualization of the nearest neighbor search problem. **Left:** collision probability of a LSH function as the distance between the query and a point grows larger. **Center:** representation of the collision radius centered at the query for a collection of points in the database. The blue points within distance R have a high probability of colliding with the query. The orange points within distance R and cR from the query have a lower probability of colliding with the query. The approximation factor $c > 1$ determines the quality of the results; typically $c = 2$ in practical applications. **Right:** The query is likely to collide with buckets containing the near neighbors (blue points) when using a LSH function h to construct the hash table.

$\mathcal{DB}$. A vector is denoted in bold as $\boldsymbol{v}$ where the ith coordinate of $\boldsymbol{v}$ is denoted by $\boldsymbol{v}_i$. A distance metric (e.g., Euclidean distance) is denoted Δ, where threshold distances R and cR are as defined in Section II-A. We let $\mathbb{F}$ denote any prime-order finite field (e.g., integers mod a prime p). A secret-share of a value $v \in \mathbb{F}$ is denoted using bracket notation as $[v]$. Coordinate-wise secret-shares of a vector $\boldsymbol{v}$ are denoted as $[\boldsymbol{v}]$. Variable assignment is denoted by $x \leftarrow y$, where $x \xleftarrow{R} S$ denotes a random sample from S.

A. Baseline functionality

The baseline ANN search functionality is described in Functionality 1. The functionality takes as input the public parameters and query $\boldsymbol{q}$ to output the ID of the nearest neighbor to the client. The servers obtain no output. Without loss of generality, we assume the ID of the ANN is the *index* of the ANN for some canonical ordering of the feature vectors in the database. We let ID $= 0$ when no nearest neighbor exists.

Restricting the problem. Note that an LSH-based algorithm will only return an answer that is within distance cR of the query. We formalize this by assuming that the baseline functionality outputs the nearest neighbor that is also a *near*[2] (distance less than cR away from the query) neighbor. While it is possible to imagine contrived databases where the nearest neighbor is *not* also a near neighbor, most practical instances of the problem impose this additional restriction (returning no neighbor if the nearest neighbor is beyond a threshold distance from the query) because points beyond some threshold are effectively unrelated to the query. To this end, the baseline functionality is defined to reveal the ID of the nearest neighbor (if one exists) within a fixed distance $R = R_{\mathsf{max}}$ from the query. Following Bayer *et al.* [17], we define two quantities D_{max} and D_{min} to be the maximum and minimum distance between any two points, respectively. Because the distance between any two vectors is at most D_{max}, it suffices to have $R_{\mathsf{max}} < D_{\mathsf{max}}$.

[2]This name comes from the *near* neighbor search problem [3, 48].

Functionality 1: ANN Search

Public Parameters: Distance metric Δ, database size N, feature vector dimensionality d, maximum nearest neighbor radius R_{max}.

Client Inputs: query $\boldsymbol{q} \in \mathbb{R}^d$.

Server Inputs: Database $\mathcal{DB} := \{\boldsymbol{v_1}, \ldots, \boldsymbol{v_N} \mid \boldsymbol{v_i} \in \mathbb{R}^d\}$.

Procedure:

1: $\boldsymbol{v_a} \leftarrow$ nearest neighbor to $\boldsymbol{q}$ in $\mathcal{DB}$ via brute-force search.

2: **if** $\Delta(\boldsymbol{v_a}, \boldsymbol{q}) > R_{\mathsf{max}}$ **then** output 0 to the client and $\perp$ to the servers.

3: **else** output a to the client and $\perp$ to the servers.

B. Threat model and security guarantees

Our protocol is instantiated with two non-colluding servers and an arbitrary number of clients. Clients query the servers to obtain the ANN ID from a remote database replicated on both servers. We do not require any communication between servers when answering queries.

Threat model.

- No client is trusted by either server. Clients may deviate from protocol, collude with other clients, or otherwise behave maliciously to learn more about the database.
- No server is trusted by any client. One or both servers may deviate from protocol in an attempt to obtain information on a client's query or the resulting nearest neighbor.

Assumptions. Our core assumption, required for client privacy, is that the two servers do not collude with each other. For database privacy, we assume that neither server shares the database with a client. We also require black-box public-key infrastructure (e.g., TLS [74]) to encrypt communication between the clients and the servers.

Guarantees. Under the above threat model and assumptions, the protocol provides the following guarantees.

Correctness. If the client and servers both follow protocol, then the client obtains the ID of the ANN with respect to its query. The result is guaranteed to have the same approximation accuracy of standard, non-private data structures for ANN search, and has tunable accuracy guarantees.

Client privacy. If the servers do not collude, then neither server learns any information on the client's query, even if one or both servers arbitrarily deviate from protocol.

Bounded leakage. Each query answer is guaranteed to leak a small (and tightly bounded) amount of information over the ideal functionality, even if the query is maliciously generated by the client. We provide a precise definition and in-depth analysis of this leakage in Section VII.

IV. Main Ideas

To introduce privacy, as required for the client *and* the database, we make several changes to the standard LSH data structure (Build and Query; described in Section II). A simple strawman protocol with *client* privacy (but no database privacy) can be realized by applying well-known techniques in Private Information Retrieval (PIR) to privately obtain the answer to Query (see [28, 29, 40]). PIR allows a client to privately retrieve a specified object from a remote database without revealing which object was retrieved, which naturally generalizes to retrieving buckets from a hash table [29]. While PIR solves the client privacy problem, it provides no database privacy. The client learns $\sqrt{N}$ (recall that $L \approx \sqrt{N}$ to provide good accuracy) feature vectors from the database *per query*.

The challenge with database privacy. The primary challenge in reducing database leakage comes from preventing the client from learning extra vectors in the candidate set. This is non-trivial to do given that the standard approaches to removing false-positive candidates (vectors farther than cR from the query) require some form of direct distance comparisons. In the private setting, these become *oblivious* comparisons (a comparison between secret-shared values), which in turn require *heavy* cryptographic techniques (e.g., garbled circuits [84]). The state-of-the-art approach for privacy-preserving ANN search (SANNS [26]) prunes candidates by using an expensive two-party computation, which requires several gigabytes of communication between the client and database server.

The insight that we exploit to overcome this challenge is that LSH can itself be used to accomplish the same goal of pruning false-positives. By carefully tuning the LSH parameters, we can eliminate a large number of false-positives from the ANN candidates. We then apply a trick inspired by radix sorting [54] to extract the *nearest* neighbor, fully removing the need for direct comparisons between vectors. Our new data structure is slightly less efficient when viewed from an *algorithmic* perspective (i.e., when not considering privacy). However, this is not a problem for us given that oblivious comparisons are the primary bottleneck in a privacy-preserving setting. We elaborate on this observation in the next section.

A. Reframing the problem

LSH-based ANN search is typically optimized to minimize the number of hash tables (L) and the size of the candidate set for each query. Removing false-positives via brute-force comparisons is relatively "cheap" from a computational standpoint while hash table lookups are relatively expensive. Therefore, LSH-based ANN search is typically tuned to retrieve as many (reasonable) candidates as possible from each table. The extra candidates are then pruned via brute-force comparisons.

The privacy-preserving setting requires different priorities. First, note that it is not possible to perform only one lookup per hash table. To preserve privacy, *all* hash table buckets must be "touched" by the database server(s) to avoid revealing information on the client's query. This is the lower bound on private information retrieval [15, 28]: if it is not met, then the servers learn that the client's hash does *not* correspond to any untouched bucket. This is exactly why existing solutions for privacy-preserving ANN search require $O(N)$ communication between the client and server, or alternatively, the use of fully-homomorphic encryption (which requires linear server work).

As such, we cannot hope to have sublinear (in N) work for the servers when answering client queries.

With this in mind, we observe that the optimal LSH parameters in the non-private (a.k.a. algorithmic) setting might not in fact be optimal for privacy-preserving setting. We therefore approach the problem from a different angle by re-designing and re-tuning the ANN search data structure of Section II to limit the use of comparisons between vectors.

B. Eliminating oblivious comparisons

One idea to remove comparisons is to prevent values that will be pruned from being added to the candidate set in the first place. More precisely, by tuning the parameters of the ANN search data structure, and the LSH functions it uses, we can bound the probability of false-positives in the candidate set to any $0 < \delta < 1$. A standard result in LSH theory is that a (R, cR, p_1, p_2)-sensitive hash family can be "amplified" to result in an (R, cR, p_1', p_2')-sensitive hash family, where $p_1' = p_1^k$ and $p_2' = p_2^k$. This is done by simply concatenating the outputs of k independent (R, cR, p_1, p_2)-sensitive hash functions. Amplification is generally used to reduce the size of the candidate set [6], but we can take this approach to its extreme. For a larger value of k, fewer collisions will occur. However, when considering the collisions that do occur, they are more likely to be true positives. As a result, with a sufficiently large k, we can significantly reduce false positives and compensate for the smaller p_1' by increasing the number of tables L. We capture this idea in Proposition 1.

Proposition 1. Fix failure probability $\delta > 0$, and any (p_1, p_2, R, cR)-sensitive hash family where $\rho = \frac{\log(p_1)}{\log(p_2)} < \frac{1}{2}$. There exists a data structure solving the cR-approximate near neighbor problem in $O(N^{\rho'})$ time and $O(N^{1+\rho'})$ space, where $\rho < \rho' < 1$. This data structure returns the true cR-approximate nearest neighbor with probability $1-\delta$, without requiring brute-force distance comparisons between vectors.

Proof. See Appendix D. ∎

Bounded false-positives (in the worst case). The consequence of Proposition 1 is that while we can bound false-positives to any δ, this comes with the cost of increasing the number of hash tables L, since $L = \lceil p_1^{-k} \rceil$. Because k is a function of the LSH sensitivity, we need to ensure that the difference between p_1 and p_2 is sufficiently large to result in reasonable values of k and L. We describe such LSH families in Appendix C. In our evaluation (Section VIII), we show that on real data, we can have false positive probability less than 0.05 with $k = 2$. This guarantees that all collisions are within cR from the query (i.e., all collisions are *near* neighbors), with high probability. Additionally, in Section IV-C, we describe a trick called LSH multi-probing which amortizes the number of hash tables required to retrieve L candidates.

Finding the nearest neighbor. We are now left with the problem of finding the *nearest* neighbor within the set of all cR-neighbors. Our idea for doing so is based on a bucketing technique of Ahle *et al.* [48], which resembles radix sorting [54]. A radix sort does not perform direct comparisons, which aligns well with our goals. We repeatedly apply the data structure of Proposition 1 on a series of increasing neighbor radii, retrieving a set of candidates from each radius [48]. The ANN is then chosen at random from the first non-empty candidate set (see Figure 3 for an illustration of this process).

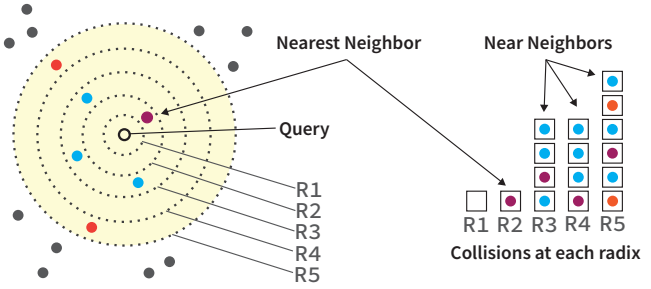

Fig. 3: Illustration of using multiple radii (R) to search for the nearest neighbor. **Left:** Each of the dotted regions represents a different hash function radius. **Right:** The candidate result with the smallest R_i is the nearest neighbor, in this case the bucket corresponding to R_2.

Comparison-free ANN search data structure

BUILD$(\mathcal{DB}, \mathcal{H}_1, \ldots, \mathcal{H}_L) \to (T_1, \ldots, T_L, \mathsf{h}_1, \ldots, \mathsf{h}_L)$:
takes as input a set of N vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}$ and L hash families $\mathcal{H}_i$ corresponding to a radix $R_L \le R_i \le R_L$.

1: Sample random h_i from $\mathcal{H}_i$, for $i \in \{1, \ldots, L\}$.
2: Use h_i to build T_i by hashing each vector $\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}$.
3: Output the similarity search data structure consisting of L hash tables $T_1, \ldots, T_L$ and LSH functions $\mathsf{h}_1, \ldots, \mathsf{h}_L$.

QUERY$(T_1, \ldots, T_L, \mathsf{h}_1, \ldots, \mathsf{h}_L, \boldsymbol{q}) \to \mathsf{ID}$; as in Figure 9.

1: Compute $(\alpha_{i,1} \ldots \alpha_{i,\ell}) \leftarrow \mathsf{multiprobe}(\mathsf{h}_i, \ell, \boldsymbol{q})$. and retrieve buckets $\mathcal{B}_{\alpha_{i,j}}$ from T_i.
2: $x \leftarrow$ min i with nonempty $\mathcal{B}$, or 0 if no such i exists.
3: **if** $x \neq 0$ **then** output any γ s.t. $\boldsymbol{v_\gamma} \in \mathcal{B}_{\alpha_x}$; **else** output 0.

Fig. 4: ANN search data structure with no direct comparisons.

C. LSH multi-probing

The number of tables as given by Proposition 1 can become very large due to the amplification. However, we the number of tables can be decreased through an optimization called LSH multi-probing. The idea behind LSH multi-probing is the following: if the bucket to which the query hashes to in a table is empty, then it is likely that "adjacent" buckets in the table contain a collision (due to the locality property) [61, 69]. To exploit this observation, each hash table is probed on ℓ keys which "surround" the query. In turn, this reduces the number of hash tables required (the original motivation of multi-probing [61]) to find a non-empty candidate. An additional appealing property of multi-probing is that it allows us to apply batching techniques when processing queries which heavily amortizes processing time for the servers (see Section V-A). We define the multiprobing function that outputs ℓ hashes for a function h and query $\boldsymbol{q}$ as $\mathsf{multiprobe}(\mathsf{h}, \ell, \boldsymbol{q}) \to (\alpha_1 \ldots \alpha_\ell)$.

D. The comparison-free ANN data structure

Our new ANN data structure (presented in Figure 4) merges the above ideas to eliminate the brute-force step present in the LSH-based data structure of Gionis et al. [43]. The client can simply select any element from the first non-empty candidate set. This data structure can be further adapted to suppress database leakage, as we explain in the next section.

E. Database privacy and suppressing leakage

In this section we explain how the ideas of Section IV-B are useful to suppress database leakage. Our approach is a combination of three changes applied to QUERY in Figure 4. We recall that a simple strawman protocol achieving client privacy can be constructed by having the client privately retrieve colliding buckets through PIR [29].

Capping buckets. Our first observation is that we only need to retrieve one element from the data structure of Figure 4. As such, we can limit each hash bucket to only contain *one* vector without affecting the success of the protocol. Any non-empty bucket will remain non-empty. This will ensure that no information is revealed (to either party) by the size of the bucket returned to the client.

Hiding the vectors. Because the client selects the first non-zero ID from the candidate set using the data structure described in Figure 4, it does not need access to the raw vectors. As such, we can modify each hash table (BUILD; Figure 4) to only store the IDs of each vector. The client can still query the hash tables using PIR but now only obtains a candidate set of IDs (absent the vectors). If each vector in the database is d dimensional and the number of hash tables is $L = \sqrt{N}$ (Section II), then roughly speaking, this simple change reduces leakage from $O(\sqrt{N} \cdot \log N \cdot d)$ bits to $O(\sqrt{N} \cdot \log N + d)$ bits (each ID is at least $\log N$ bits and $O(d)$ bits are leaked *implicitly* by the inference that the neighbor features are similar to the query; see Section VII for more details).

Hiding the candidate set. Compared to the ideal leakage of $O(\log N + d)$ bits, the leakage of $O(\sqrt{N} \cdot \log N + d)$ bits is far from optimal. We eliminate (most of) the additional leakage by designing a special "oblivious masking" transformation which hides all-but-one non-zero candidate ID from the client. From the masked candidate set $\tilde{\mathcal{C}}$, the client is able to extract *at most* one ID that collided with its query. This further reduces leakage from $O(\sqrt{N} \cdot \log N + d)$ bits down to $O(\log N + d)$ bits (since only one ID is revealed), which matches the asymptotic leakage of the baseline functionality. We provide details on the oblivious masking transformation in Section V-A, and a formal leakage analysis in Section VII.

With these three leakage-suppressing steps, our protocol achieves close to optimal concrete database leakage per query. Importantly, the leakage guarantees hold in the face of malicious clients that may deviate from protocol in an attempt to learn more than an honest client (Section VII).

V. PROTOCOL

We now describe the details of the high-level ideas covered in Section IV. We first formalize the necessary building-blocks in Section V-A (distributed point functions and our oblivious masking technique) and then present the full ANN search protocol in Section V-B.

A. Building blocks

Existing tool: Distributed Point Functions. A *point function* P_α is a function that evaluates to 1 on a single input α in its domain and evaluates to 0 on all other inputs $j \neq \alpha$. A *distributed* point function (Definition 2) is a point function that is encoded into a pair of keys which are used to obtain a secret-shared evaluation of P_α on a given input j.

Definition 2 (Distributed Point Function (DPF) [20, 21, 42])**.** Fix any positive integer n. Let $\mathbb{F}_p$ be a finite field (e.g., integers mod prime p), and let λ be a security parameter. A DPF consists of two (possibly randomized) algorithms:

- $\mathsf{Gen}(1^\lambda, \alpha \in \{1, \ldots, 2^n\}) \to (k_A, k_B)$ takes as input an index α and outputs two evaluation keys k_A and k_B,
- $\mathsf{Eval}(k, j) \to v_j \in \mathbb{F}_p$ takes as input an evaluation key k and index $j \in \{1, \ldots, 2^n\}$ and outputs a field element v_j.

These algorithms must satisfy correctness and privacy:

Correctness. A DPF is *correct* if for all pairs of keys generated according to Gen,

$$\Pr\left[\ \mathsf{Eval}(k_A, j) + \mathsf{Eval}(k_B, j) = \begin{cases} 1, & \text{if } j = \alpha \\ 0, & \text{otherwise.} \end{cases}\ \right] = 1.$$

Privacy. A DPF is *private* if each individual evaluation key output by Gen is pseudorandom (i.e., reveals nothing about α to a computationally bounded adversary). Formally, this means that there exists an efficient simulator Sim that can generate an indistinguishable view for each generated DPF key, without knowledge of the input α [20, 42].

Application: Private Information Retrieval. DPFs form the basis for efficient two-server private information retrieval [20, 21, 28, 29]. Consider a key-value table T with N keys, replicated on two servers. Let each key in T be in the set $\{1, \ldots, 2^n\}$ and let the corresponding values be in the field $\mathbb{F}_p$. To retrieve the value under key α in T the client generates DPF keys by running $\mathsf{Gen}(1^\lambda, \alpha)$ and sends one key to each server. Each server locally evaluates the DPF key with each item α_i for $i = 1, \ldots, N$ to obtain the secret-share of $P_\alpha(\alpha_i)$ on input α_i. Each server then locally multiplies the resulting secret-share by the associated value $T(\alpha_i)$, and outputs the sum of all N component-wise products. That is, each server computes:

$$\sum_{j=1}^{N} \Big(T(\alpha_j) \cdot \mathsf{Eval}(k, \alpha_j)\Big) = T(\alpha_i) \cdot [1] + \sum_{j=1, j\neq i}^{N} \Big(T(\alpha_j) \cdot [0]\Big).$$

Because $P_\alpha(\alpha_i) = 1$ only when $\alpha = \alpha_i$, the resulting sum (computed in the field $\mathbb{F}_p$) is a secret-share of $T(\alpha_i)$.

Summing the shares received from each server, the client recovers the desired entry in T. Observe that DPFs achieve *symmetric* PIR [41] (analogous to oblivious transfer [71]), which guarantees that the answer consists of at most one value in T. Additionally, modern DPF [20, 21] constructions achieve key size of $n(\lambda+2)$ bits (logarithmic in the evaluation domain). That is, privately querying a hash table with n-bit hash keys requires only $n(\lambda+2)$ bits of communication and $O(Nn)$ work on the server; where N is the number of non-empty buckets.

New tool: Oblivious Masking. The core of our leakage suppression technique (described at a high level in Section IV-E) hinges on the ability to reveal only the first non-zero value in a secret-shared vector. Because the vector is secret-shared (in some prime order field $\mathbb{F}_p$ where $p \gg N$), this transformation must only involve affine operations: addition and scalar multiplication of shares [16, 77]. The idea is to recursively compute a randomized sum, moving from left to right. For a secret-shared input vector $[\boldsymbol{v}] \in \mathbb{F}_p^L$, let $z \in \{1, \ldots, L\}$ be the index of the first non-zero element in $\boldsymbol{v}$. The randomized sums map each element $\boldsymbol{v}_i$ to 0 for $i < z$ and a uniformly random element in $\mathbb{F}_p$ for $i > z$. Crucially, this process does not affect the first non-zero element, $\boldsymbol{v}_z$. We will use this property to mask all-but-one result from a sequence of PIR query answers.

Algorithm 1: ObliviousMasking

Input: Secret-shared vector $[\boldsymbol{v}] \in \mathbb{F}_p^L$ and randomness rand.

Output: Secret-shared vector $[\boldsymbol{y}] = ([y_1], \ldots, [y_L]) \in \mathbb{F}_p^L$.

Procedure:

1: **for** $i \in \{1, \ldots, L\}$**:**

1.1: Sample $r_i \leftarrow$ rand.

1.2: Set $[\boldsymbol{y}_i] \leftarrow [\boldsymbol{v}_i] + r_i \cdot \left(\sum_{j=0}^{i-1} [\boldsymbol{v}_j]\right)$.

2: Output $\boldsymbol{y} \in \mathbb{F}_p^L$.

Claim 1. Let $\boldsymbol{v} \in \mathbb{F}_p^L$ be any vector and let $z \in \{1, \ldots, L\}$ be the first non-zero element of $\boldsymbol{v}$. Let $[\boldsymbol{v}]$ be an additive secret-sharing of $\boldsymbol{v}$. Algorithm 1 outputs a secret-shared vector $[\boldsymbol{y}]$ such that $\boldsymbol{y}_i = \boldsymbol{v}_i$ for $i \leq z$ and a uniformly random element of $\mathbb{F}_p$ for $i > z$.

Proof. The proof follows by examining the three possible cases for each value in $\boldsymbol{y}$ as a function of $\boldsymbol{v}$.

1) for $i < z$, $\boldsymbol{v}_i = 0$, so $\boldsymbol{y}_i = 0 + r_i \sum_{j=0}^{i-1} 0 = 0 \in \mathbb{F}_p$,
2) for $i = z$, $\boldsymbol{y}_z = \boldsymbol{v}_z + r_z \sum_{j=0}^{z-1} 0 = \boldsymbol{v}_z \in \mathbb{F}_p$,
3) for $i > z$, $\boldsymbol{y}_i = \boldsymbol{v}_i + r_i \sum_{j=0}^{z-1} 0 + r_i \boldsymbol{v}_z + r_i \sum_{j=z+1}^{i-1} \boldsymbol{v}_j \in \mathbb{F}_p$.

Case (1) ensures that all zeroes remain zeroes. Case (2) ensures that the first non-zero element is mapped to itself. Case (3) ensures that all subsequent elements are uniformly random in $\mathbb{F}_p$. To see why (3) holds, observe that $\boldsymbol{v}_z \neq 0$, so $r_i \cdot \boldsymbol{v}_z$ is a uniformly random element of $\mathbb{F}_p$ given r_i is uniformly random. It then follows that the sum is uniformly random in $\mathbb{F}_p$. Finally, correctness of the computation over secret-shares follows from all operations performed above being linear (additions and scalar multiplications) over the input secret-share of $\boldsymbol{v}$ [16]. ■

New tool: Partial Batch Retrieval. LSH multi-probing requires retrieving multiple hash table buckets (called a *batch*) through PIR. A naïve way to approach this problem is to issue ℓ separate PIR queries—one per bucket—which incurs a factor of ℓ processing overhead on the servers. Partial Batch Retrieval (PBR) (inspired by probabilistic batch codes [7, 8]) makes it possible for the client to efficiently retrieve multiple values *without* any increase in server-side processing time. PBR is inspired by, but distinct from, batch codes [51]. With PBR, only a fraction of requested values are returned, which is fine for our probabilistic setting but may not be in others. We compare PBR with batch codes (and their probabilistic variants) in Section VI.

The main idea. A simple PBR scheme can be realized by dividing the hash table keys into $m \geq \ell$ partitions at random. If all ℓ multi-probes fall into a unique partition, then it suffices for the client to issue m PIR queries, with each query retrieving one bucket from each partition. The total server processing time to compute PIR answers remains the same. To see this, observe that for a hash table of N hash keys, each partition only contains N/m hash keys to be summed in the DPF. The total work to answer all m PIR queries, each computed over a set of N/m keys, is then amortized to $O(N)$. However, the success of this PBR scheme hinges on hash keys falling into unique partitions. How many hash keys can we expect to retrieve in practice? Abstractly, the fraction of retrievable elements can be modeled by the classic "balls and bins" problem [63], where ℓ balls are tossed into m bins uniformly at random. If all ℓ balls fall into unique bins, then the number of full bins is ℓ. Fewer than ℓ full bins corresponds to a collision in a bin (i.e., partition) and only one element in each partition can be retrieved. Let X_i be the indicator random variable where $X_i = 1$ if a bin is full. Then,

$$\Pr[X_i = 1] = 1 - \left(1 - \frac{1}{m}\right)^{\ell} > 1 - e^{-\ell/m},$$

which implies that $\frac{m}{\ell} \cdot (1 - e^{-\ell/m})$ of the ℓ hash keys are simultaneously "retrievable" from the $m \geq \ell$ partitions. In the case that $m = \ell$, we can expect to retrieve approximately 63% of the required buckets. With $m > \ell$, we can increase the probability of retrieving all required buckets at the cost of also increasing communication by a factor of $\frac{m}{\ell}$.

Application: Private multi-probing. The client first computes many hash indexes using multi-probing, and keeps one for each PBR partition region. For each partition region, it sends a DPF key for the hash index (which maps to 0 for no collision), and the results from all partitions and tables are arranged and masked with oblivious masking. Our main observation is that failing to retrieve specific probes (because they collided in the same PBR partition) is not a total failure. It is equivalent to not choosing that particular multi-probe, which could have already

happened due to the probabilistic nature of LSH. Because each multi-probe hash key is uniformly distributed from universal hashing (see Remark 1), each hash key is equally likely to be selected, making it possible to directly apply our PBR scheme.

Protocol 1: Private Approximate Nearest Neighbor Search

Public Parameters: LSH families $(\mathcal{H}_1, \ldots, \mathcal{H}_L)$, number of hash tables L, and LSH functions $\mathsf{h}_1, \ldots, \mathsf{h}_L$ as in Figure 4.

Server Input: database of vectors $(\boldsymbol{v_1}, \ldots, \boldsymbol{v_N})$.

Client Input: query vector $\boldsymbol{q}$.

Setup (one-time server-side pre-processing)

1: **for** $i \in \{1, \ldots, L\}$**:**

1.1: Construct hash table T_i by storing d in bucket with key $\alpha_{i,d} \leftarrow \mathsf{h}_i(\boldsymbol{v_d})$ for all $d \in N$.

1.2: Truncate each bucket in T_i to have at most one value (as described Section IV-E).

2: Agree on common randomness source (e.g., PRG seed) rand.

Step 1 (on the client)

1: **for** $i \in \{1, \ldots, L\}$**:**

1.1: $\alpha_1 \ldots \alpha_\ell \leftarrow \mathsf{multiprobe}\,(\mathsf{h}_i, \ell, \boldsymbol{q})$.

1.2: Choose one hash key for each PBR partition, and fill in 0 for empty partitions (See Section V-A). For $j = 1 \ldots m$, let $\alpha^{(i,j)}$ be the key for the jth partition of the ith table.

1.3: $\left(k_A^{(i,j)}, k_B^{(i,j)}\right) \leftarrow \mathsf{DPF.Gen}\left(1^\lambda, \alpha^{(i,j)}\right)$.

2: $\boldsymbol{k}_A \leftarrow \left(k_A^{(1,1)}, \ldots, k_A^{(L,m)}\right)$ and $\boldsymbol{k}_B \leftarrow \left(k_B^{(1,1)}, \ldots, k_B^{(L,m)}\right)$.

3: Send $\boldsymbol{k}_A$ and $\boldsymbol{k}_B$ to servers A and B, respectively.

Step 2 (on each server)

1: Parse $\boldsymbol{k} = \left(k^{(1,1)}, \ldots, k^{(L,m)}\right)$. `// DPF query keys.`

2: **for** $i \in \{1, \ldots, L\}$ **and** $j \in \{1, \ldots, m\}$**:**

2.1: $D_{i,j} \leftarrow$ set of bucket keys in hash table T_i, partition j.

2.2: $[\mathsf{ID}_{i \cdot m + j}] \leftarrow \sum_{\alpha \in D_{i,j}} T_i(\alpha) \cdot \mathsf{DPF.Eval}\left(k^{(i,j)}, \alpha\right)$.

3: $[\mathcal{C}] \leftarrow ([\mathsf{ID}_1], \ldots, [\mathsf{ID}_{L \cdot m}])$.

4: $[\tilde{\mathcal{C}}] \leftarrow \textsc{ObliviousMasking}([\mathcal{C}], \mathsf{rand})$. `// Algorithm 1.`

Step 3 (on the client)

1: Receive $[\tilde{\mathcal{C}}]_A$ and $[\tilde{\mathcal{C}}]_B$ from servers A and B, respectively.

2: $\left(\tilde{\mathsf{ID}}_1, \ldots, \tilde{\mathsf{ID}}_{L \cdot m}\right) \leftarrow [\tilde{\mathcal{C}}]_A + [\tilde{\mathcal{C}}]_B$. `// Recover candidates`

3: Output $\tilde{\mathsf{ID}}_i \neq 0$ for smallest i, or 0 if all $\tilde{\mathsf{ID}}_i$ are zero.

B. Putting things together

The full protocol is presented in Protocol 1 and uses the DPF, the oblivious masking, and probablistic batch retrieval building-blocks described in Section V-A. We briefly describe each step of the protocol.

Setup. The public parameters consist of the number of hash tables (L) and a list of L randomly sampled hash functions, in accordance with the data structure of Figure 4 and Proposition 1. The servers construct L hash tables using the hash functions from the public parameters. Only the IDs of the input vectors are stored in the hash table; the vectors are discarded (see Section IV-E).

Step 1. The client hashes its query vector $\boldsymbol{q}$ using the LSH functions in the public parameters and multiprobing. The client keeps one hash key at random that falls into each PBR partition, so that it has exactly one key for each. Each resulting hash is used as the input index to DPF.Gen to generate a DPF key. The client distributes the generated keys to the servers.

Step 2. Each server uses the $L \cdot m$ DPF keys it receives from the client to retrieve a secret-share of a bucket in each of the ℓ partitions of each of the L hash tables using the PIR technique described in Section V-A. The result is a vector $\mathcal{C}$ of secret-shared buckets containing either a candidate ID (or zero if the bucket was empty). Each server applies the oblivious masking transformation (Algorithm 1) to $\mathcal{C}$ and obtains the masked secret-shared vector $\tilde{\mathcal{C}}$ as output. Each server sends its share of $\tilde{\mathcal{C}}$ to the client in response.

Step 3. From the received secret-shares $(\tilde{\mathcal{C}}_A, \tilde{\mathcal{C}}_B)$, the client recovers the vector $\tilde{\mathcal{C}}$ (in cleartext) by computing $\tilde{\mathcal{C}} = \tilde{\mathcal{C}}_A + \tilde{\mathcal{C}}_B$.[3] The transformation applied by the servers in Step 2 ensures that the client learns at most *one* non-zero candidate from the original $\mathcal{C}$. Specifically, by Claim 1, only the first non-zero value of $\mathcal{C}$ will be non-random in $\tilde{\mathcal{C}}$. The first non-zero value, if present, is output to the client.

C. Querying for k-nearest neighbors

To extend the construction to a k-ANN problem, that returns the top-k nearest neighbors, we use the following simple idea. For each non-empty bucket in Figure 4, the servers *precompute* the k-nearest neighbors to the point that hashed to each bucket with a standard ANN data structure. Then, the servers place a *vector* consisting of the k-nearest neighbor IDs in the table bucket. This extension can be viewed as instantiating the protocol with a vector of IDs rather than a single ID. From this vantage point, it is easy to see that the oblivious masking technique still hides all-but-one non-zero k-vector in the final result. Specifically, the oblivious masking is now defined over the vector field $\mathbb{F}_p^k$ (with addition and scalar multiplication defined elementwise in the natural way). The pre-computation incurs only a small overhead of k when computing the data structure and is thus acceptable. An alternative strategy to pre-computing the k-nearest neighbors when building the data structure is to cap each bucket to *at most* k IDs. However, such an approach would not guarantee k results are returned per query and thus may be undesirable for certain use cases.

VI. Efficiency analysys

Asymptotic efficiency analysis. We analyze the communication and computation costs individually (summarized in Table I). To derive our asymptotic guarantees, we follow the analysis of Andoni and Indyk [4] for Euclidean distance (which has $\rho < \frac{1}{c^2}$; see definition of ρ in Proposition 1) and where we assume $c = 2$ (see LSH; Definition 1).

[3] Additive secret-share recovery follows from the correctness property of DPFs; see Definition 2.

Communication	Server Work	Client Work	Rounds
$\sqrt{N}$	N	$\sqrt{N}$	1

TABLE I: Asymptotic efficiency of Protocol 1; constant and log factors suppressed. We assume $L \approx \sqrt{N}$ (see Section II-A).

Communication. The protocol communication consists of the L DPF keys, each of which retrieves one candidate from a table. The DPF key size is $n = O(\log N)$ [20, 21]. Each masked ID returned by the protocol is of size $\log N$. Therefore, when $L \approx \sqrt{N}$, the total communication is $O(\sqrt{N} \log N)$.

Computation. For each server, computing the answer to the PIR query requires evaluating the DPF on $O(N)$ hash keys per hash table. Therefore, the total server work is naïvely $O(L \cdot N)$. However, note that each PBR partition only needs to be evaluated on a portion of the hash keys. As such, if we can set $m = \ell = L/a$, for some constant a, then the total amortized server work is $O(N)$. Oblivious masking, which is performed over the candidate set, requires $O(\sqrt{N})$ additional work per server. For the client, the computational cost is proportional to the query response size, since it has to receive and recover the candidate set and find the first non-zero element to return as the answer.

Storage. The storage overhead on the client is simply the query $\boldsymbol{q}$ which requires $O(d)$ bits (assuming the feature vector consists of constant entries). On the server, the storage overhead is $O(L \cdot N \log N)$ for storing all hash tables. In particular, the storage overhead is on-par with the requirements of non-private ANN data structures [6, 49].

PBR efficiency. It is natural to ask how PBRs compare to batch codes in terms of efficiency. The main difference is with respect to the guarantees provided by both tools. Batch codes (and their probabilistic variants [7, 8]) aim to guarantee retrieval of *all* ℓ elements. Doing so comes at an efficiency cost: batch codes *replicate* data which increases both processing time and bandwidth. With PBRs, we avoid the need for replicating data, keeping the total *processing* cost fixed while only modestly increasing bandwidth (see Table II for a comparison to batch codes). However, this comes at a different cost: PBRs only guarantee *partial* retrieval of the ℓ elements. We find that for certain applications, such as Protocol 1, partial retrieval in favor of decreased processing time is a desirable trade-off, given the already probabilistic guarantees of LSH.

VII. Security and Leakage Analysis

In this section, we analyze the security of Protocol 1 with respect to client privacy and server leakage. While client privacy is conceptually simple and follows directly from the privacy property of DPFs (Definition 2), the leakage analysis of the database is more involved.

A. Client privacy

Claim 2 (Query Privacy)**.** For all Probabilistic Polynomial Time (PPT) adversaries $\mathcal{A}$ corrupting server b for $b \in \{\mathrm{A}, \mathrm{B}\}$,

Batch-PIR Scheme	Replication Factor	Total Partitions	Fraction Retrieved
Naïve	1	1	$\frac{1}{\ell}$
SBC [51]	$(\frac{3}{2})^{\ell}$	$3^{\log \ell}$	1
PBC [8]	3	1.5ℓ	$> 1 - 2^{-20}$
PBR	1	m	$\frac{m}{\ell} \cdot (1 - e^{-\ell/m})$

TABLE II: Replication and partitioning costs of existing batching schemes: Naïve (perform ℓ PIR queries), Subcube batch codes (SBC) [51], Probabilistic batch codes (PBC) [8]. Replication increases server-side processing by the same factor but influences communication by a sublinear factor. Communication increases linearly with the number of partitions for all schemes.

Protocol 1 guarantees that $\mathcal{A}$ learns no information on the client's query, even when deviating from protocol.

Proof. The proof follows from a simulation-based indistinguishability argument. We say that the protocol is *query private* if there exists a PPT simulator Sim such that

$$\mathsf{Sim}(\mathcal{DB}) \approx_c \mathsf{view}(\mathcal{DB}, \boldsymbol{k}_b),$$

where view is the view of $\mathcal{A}$ from an execution of Protocol 1 with the client. $\mathsf{Sim}(\mathcal{DB})$ is trivially constructed by invoking the DPF simulator for each DPF key present in the vector $\boldsymbol{k}_b$ provided to $\mathcal{A}$ [22]. The client does not send any other information to the servers apart from the DPF keys which are used to query the hash tables. Therefore, query privacy depends only on the privacy property of DPFs and follows immediately. We now briefly argue why the use of PBR does not alter the above simulation. When retrieving buckets with the PBR scheme, the client retrieves ℓ keys from each table, where the key space is partitioned into m uniformly random subsets (see Section V-A). This results in m DPF keys sent to each server per hash table. Because the client sends a DPF key for each partition (even if no bucket is retrieved from that partition), the simulator works as before. ■

B. Database privacy

Following Carlini et al. [25], we divide our analysis of privacy for the database into two parts: *physical privacy*, which is the leakage of an algorithm beyond what is learned by the query answer itself, and *functional privacy*, which is the leakage from the query answer itself. To quote Carlini et al. [25]: *"we emphasize that these two notions of privacy are incomparable and complementary."* For the purpose of this analysis we will consider the query answer to consist of the vector that is output by oblivious masking, $[0, \ldots, 0, \mathsf{ID}, r_{z+1}, \ldots, r_L]$. The DPF-based symmetric-PIR technique combined with oblivious masking gives us physical privacy. For functional privacy we have to compare the leakage of this query answer to the leakage of the answer in the baseline Functionality 1.

Physical privacy. The client's view of the database can be efficiently simulated given the query answer, consisting of the ID and index z. The simulator must create two vectors, one for each server's response. It does so by first constructing a vector $\mathcal{C}$ with the first $z-1$ entries all zero, ID in the zth entry,

and random numbers for all remaining entries. It then secret shares this vector into two "response vectors" $\mathcal{C}_A$ and $\mathcal{C}_B$ to construct a statistically indistinguishable view for the client.

Functional privacy. We now turn to formalizing the database leakage incurred from the answer to a client's query. We start by quantifying the *baseline leakage*, given that the client must obtain information as output: the ANN ID relative to the query $\boldsymbol{q}$. In our setting, the client learns strictly more than just the ID, which we must quantify as additional leakage.

Theorem 1 (Quantifying Baseline Leakage)**.** Fix quantities D_{max}, D_{min}, and Δ as defined in Section III-A. The baseline leakage for an instance of approximate nearest neighbor search as instantiated in Functionality 1 is captured by $O(d + \log N)$ bits of information per query, where d is the intrinsic dimensionality of the vector space and N is the size of the database.

Proof. We start by considering the ℓ_∞-norm and induced distance metric Δ. The ℓ_∞-norm is the absolute value of the maximum coordinate: $\ell_\infty||x|| = \max_{\boldsymbol{x}_i} |\boldsymbol{x}_i|$. The induced distance metric is $\Delta(\boldsymbol{x}, \boldsymbol{y}) = \ell_\infty||\boldsymbol{x} - \boldsymbol{y}||$. We first prove the baseline leakage for the ℓ_∞-norm, as it is easier to intuit. We then show how to extend the proof to any ℓ_p-norm induced metric, which includes Euclidean distance. Other common distance metrics, such as angular distance and Hamming distance, can be embedded into Euclidean space [10, 26].

Recall that for any query $\boldsymbol{q}$ that returns an ID corresponding to vector $\boldsymbol{v}$, we have that $\Delta(\boldsymbol{v}, \boldsymbol{q}) \leq R_{\mathsf{max}} < D_{\mathsf{max}}$. The baseline functionality implicitly reveals that there exists a point $\boldsymbol{v}$ such that $\Delta(\boldsymbol{v}, \boldsymbol{q}) \leq R_{\mathsf{max}}$. In the ℓ_∞ metric, this implies that $\boldsymbol{v}$ is within a cube of side length $2R_{\mathsf{max}}$ centered at $\boldsymbol{q}$. The output implicitly reveals that $\boldsymbol{v}$ is in this cube. Considering each coordinate of $\boldsymbol{v}$, the number of possible values each coordinate can take diminishes from $2D_{\mathsf{max}}$ to $2R_{\mathsf{max}}$. As a result, the information revealed is $\log(D_{\mathsf{max}}/R_{\mathsf{max}})$ bits. With the ℓ_∞ metric, this argument can be repeated individually for each coordinate, making the total leakage $d \log(D_{\mathsf{max}}/R_{\mathsf{max}})$ bits.

For any ℓ_p-norm, the above argument holds by considering similar shapes and their relative volumes. In the Euclidean metric $\Delta(\boldsymbol{v}, \boldsymbol{q}) \leq D_{\mathsf{max}}$ defines a ball of radius D_{max}. The baseline functionality reveals that $\boldsymbol{v}$ is in a ball of radius R_{max} centered at $\boldsymbol{q}$. The ratio of the volumes of these balls is $D^d_{\mathsf{max}}/R^d_{\mathsf{max}}$, making the leakage $\log(D^d_{\mathsf{max}}/R^d_{\mathsf{max}}) = d \log(D_{\mathsf{max}}/R_{\mathsf{max}})$ bits, as in the ℓ_∞ metric. ■

Intuitively, the extra leakage in our protocol corresponds to the fact that the query vector serves as an approximation for the feature vector of the nearest neighbor, by definition of the problem. However, as we show in Theorem 1, the precision of this approximation is limited to $\log(D_{\mathsf{max}}/R_{\mathsf{max}})$.

We now formalize the leakage per query in Protocol 1 and show that it is no worse than $O(d+\log N)$ bits, which matches the leakage of the baseline functionality up to a constant factor.

Claim 3 (Asymptotic Leakage of Protocol 1)**.** Let L be the number of hash tables used to instantiate Protocol 1. Then, the leakage of Protocol 1 is $O(d + \log N)$ bits, matching the baseline functionality.

Proof. Consider the leakage of an infinite number of queries answered through Protocol 1. Fix any $\boldsymbol{v_j} \in \mathcal{DB}$. The total information revealed on $\boldsymbol{v_j}$ is never more than the set $S_j = \{\alpha_1, \ldots, \alpha_L \mid \alpha_i = \mathsf{h}_i(\boldsymbol{v_j})\}$. That is, the set of all L LSH digests of $\boldsymbol{v_j}$. The maximum information revealed is thus the set S_j for all N feature vectors $\boldsymbol{v_j} \in \mathcal{DB}$.

For an individual query, the oblivious masking transformation (Section V-A) guarantees that at most one element of S—a hash corresponding to a feature vector $\boldsymbol{v} \in \mathcal{DB}$—is leaked. Consider the worst-case (least-hiding) LSH function possible. This is not even a locality-sensitive hash function but instead simply the identity function $\mathsf{h}(\boldsymbol{x}) = \boldsymbol{x}$. It follows that the worst-case implicit query leakage is then $O(d)$ bits, because $\alpha_i = \boldsymbol{v_j}$ for all $i \in \{1, \ldots, L\}$). This is in addition to the explicit $\log N$ bits revealed by the vector ID. Therefore, the total leakage is $O(d + \log N)$, which matches the leakage of the baseline functionality in Theorem 1. ■

Claim 4. A client deviating from Protocol 1 cannot learn more information on $\mathcal{DB}$ than an honest client following protocol.

Proof. Fix S_j (the set of all hashes) as defined in the proof of Claim 3. By the guarantees of the oblivious masking transformation (Claim 1), even a malicious client can only obtain one element of S_j (for some $j \in \{1, \ldots, L\}$). All other elements are uniformly random or zero. Even if the client does not follow the protocol (such as sending random hashes that don't correspond to any one query), the leakage is capped at a single element of S_j. ■

Hash tables	**Maximum concrete leakage factor**			
	DEEP1B	MNIST	GIST	SIFT
$L = 10$	1.97×	2.16×	3.25×	3.52×
$L = 30$	2.79×	3.23×	6.86×	8.24×
$L = 50$	3.29×	3.92×	11.32×	15.7×

TABLE III: Concrete leakage (computed as $0.77 \cdot \frac{R_{\mathsf{max}}}{R_{\mathsf{min}}}$) compared to baseline leakage following Claim 3, evaluated on four datasets. Leakage increase is roughly proportional to the number of tables (L) given that each table introduces more precision in the resulting answer. See Appendix B for more details. The 0.77 factor comes from the (im)precision of the LSH computed in Lemma 1 (Appendix E).

Empirical leakage calculation. In our asymptotic leakage analysis, we assume a worst-case scenario in terms of the data and the LSH instantiation. In Appendix E, we analyze the concrete leakage using a specific instantiation of LSH for Euclidean distance. We summarize the concrete leakage in Table III on the four real-world datasets we used for our evaluation in Section VIII. This represents the maximum speedup (in number of queries) for database recovery relative to the baseline leakage. For each dataset, different radii are required in order to separate near and nearest neighbors using the radix approach of Section IV-B. It is this extra precision that is captured in the leakage factor. We note that this leakage is a worst-case bound on the actual concrete leakage. Specifically,

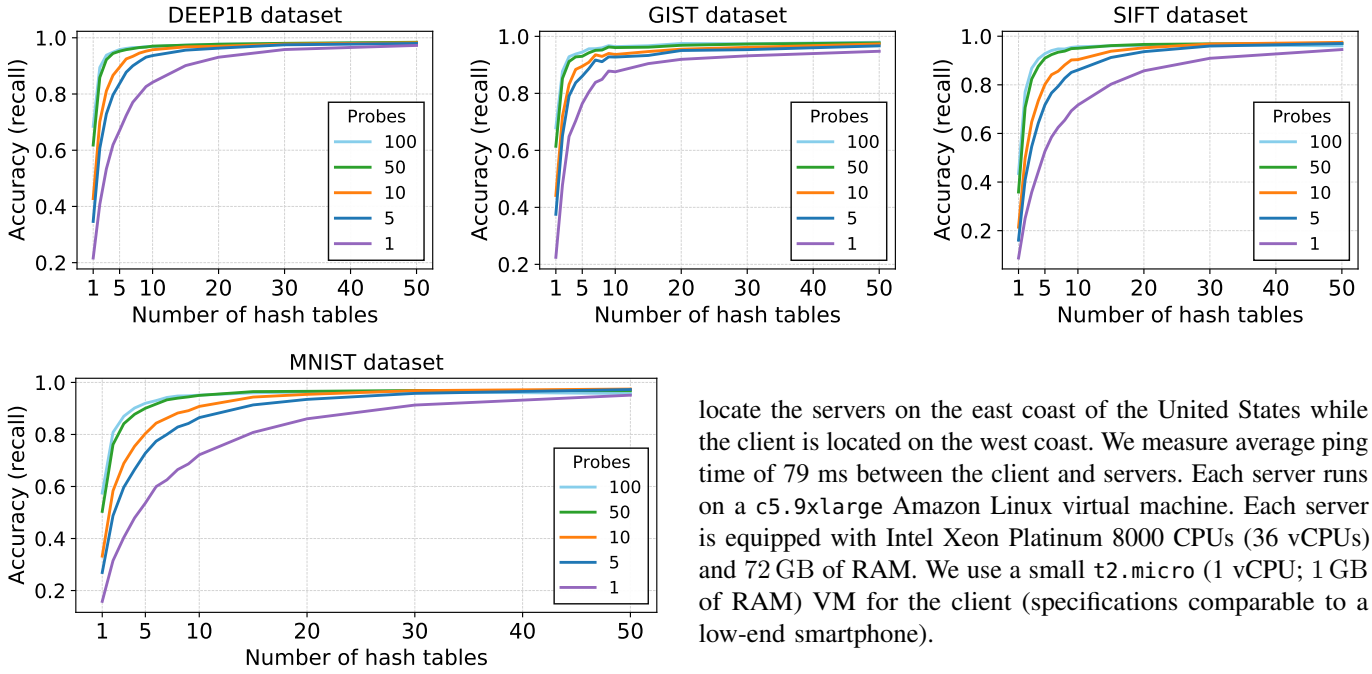

Fig. 6: Recall (fraction of queries that found an approximate nearest neighbor) for different numbers of table multi-probes using the data structure of Figure 4 and $c = 2$. Average of 10 trial runs for different numbers of probes. Probes $= 1$ corresponds to no multi-probing (only retrieving the hash of the query from the table; Section IV-C). The standard deviation is 0.002 and invisible in the plot.

this leakage assumes that *all* points collide at the smallest radius R_{min}, when in practice the points will collide across buckets defined by radii between R_{min} and R_{max}.

VIII. Empirical evaluation

We now turn to describing our implementation and empirical evaluation of Protocol 1. The goal of this section is to answer the following questions:

- What are the parameters needed to obtain high accuracy in practice using the data structure of Figure 4?
- What is the concrete performance of Protocol 1 when used for ANN search on real data?
- How does Protocol 1 compare to the state-of-the-art approach for private similarity search?

A. Implementation and environment

We implement Protocol 1 in approximately 4,000 lines of code. Our implementation is written in Go v1.16. with performance-critical components written in C. The code is open source and available online [1]. Our DPF implementation follows Boyle et al. [21] and is partially based on open-source libraries [35, 37]. Our implementation uses AES as a pseudo-random generator which exploits the AES-NI instruction for hardware-accelerated operations. We use the GMP library [39] for fast modular arithmetic.

Environment. We deploy our implementation on Amazon Elastic Cloud Compute (EC2) for our experiments. We geographically locate the servers on the east coast of the United States while the client is located on the west coast. We measure average ping time of 79 ms between the client and servers. Each server runs on a `c5.9xlarge` Amazon Linux virtual machine. Each server is equipped with Intel Xeon Platinum 8000 CPUs (36 vCPUs) and 72 GB of RAM. We use a small `t2.micro` (1 vCPU; 1 GB of RAM) VM for the client (specifications comparable to a low-end smartphone).

B. Performance evaluation

Accuracy. We report the accuracy of Protocol 1 when evaluated on real-world datasets in Figure 6. The datasets we choose form a standard for benchmarking ANN search [10]. We use $k = 2$ for the amplification factor (see Proposition 1). Accuracy is defined in terms of *recall*: what fraction of approximate nearest neighbors found are at most c-times the distance to the true nearest neighbor [3, 43, 48]. Matching the theory, increasing the number of tables *or* multi-probes increases the accuracy. For all datasets we can achieve high accuracy ($> 95\%$) without needing more than 20 tables and 50 multi-probes per table.

Parallelism. The server overhead of answering queries (which involves a linear scan over all hash table keys; see Section VI) is easily parallelizable across cores or even across different machines composing each logical server. In our runtime experiments (Figure 8), we provide results for both single core and parallelized executions (where each hash table is processed on a separate core). In our experiments, we observe a close-to-linear speedup in the degree of parallelism.

Performance. We report the end-to-end latency (as measured on the client machine) for each dataset in Figure 8. The end-to-end latency includes the server-side processing time, client-side processing, and network delay. The server processing time, per hash table, ranges from 28 ms on the MNIST dataset (60,000 items) to approximately 6.5 s on the DEEP1B dataset (10,000,000 items). The processing time is dominated by the DPF and, in practice, increases moderately with the number of multiprobes. The other steps, including ObliviousMasking, take less than 2 ms. The client processing overhead across all datasets is small: never exceeding 10 ms.

Communication. We provide the total communication required to query one hash table in Table IV. The communication

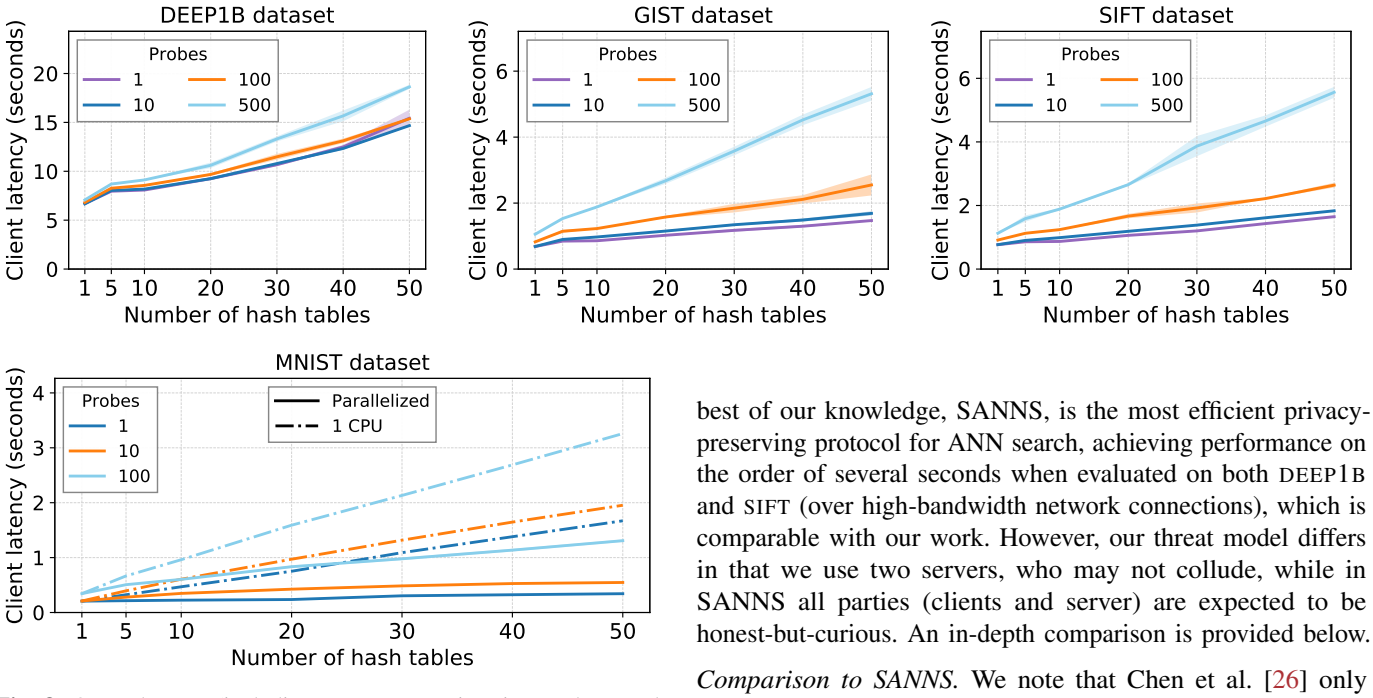

Fig. 8: Query latency (including server processing time and network delay) as a function of the number of tables and number of multi-probes performed per table. Probes do not increase computation time but do increase communication resulting in increased latency (see our PBR scheme; Section V-A). Server parallelization factor is set to equal the number of hash tables. Shaded region represents the 95% confidence interval (occasionally invisible).

Multi-probes:	**1**	**5**	**10**	**50**	**100**
Communication:	3 kB	10 kB	18 kB	87 kB	172 kB

TABLE IV: Communication between the client and both servers per hash table in terms of the number of multi-probes performed. We set $n = 40$ for the DPF domain.

overhead is determined by three factors: (1) the size of the DPF keys, (2) the number of hash tables L, and (3) the number of multi-probes performed per table (recall Section V-A). The size of each DPF key is proportional to n, where 2^n is the universal hash range (see Section V-A and Remark 1). We set $n = 30$ for MNIST, $n = 35$ for SIFT and GIST, and $n = 40$ for DEEP1B. DPF key size is logarithmic in N, and each choice of n is fixed to minimize universal hash collisions and achieve this bound. This communication is multiplied by the number of tables L and the number of probes to calculate the total communication required to perform an ANN query.

C. Comparison to related work

To the best of our knowledge, all existing works on privacy-preserving similarity search (with the exception of [78, 87] which use fully-homomorphic encryption) only consider honest-but-curious clients and servers and require many rounds of communication. While generic techniques for upgrading to active security in two-party computation exist [59], they are computationally expensive and often considered impractical. Our protocol is the first to assume fully malicious clients, which solves the challenge left open by prior approaches [26]. To the best of our knowledge, SANNS, is the most efficient privacy-preserving protocol for ANN search, achieving performance on the order of several seconds when evaluated on both DEEP1B and SIFT (over high-bandwidth network connections), which is comparable with our work. However, our threat model differs in that we use two servers, who may not collude, while in SANNS all parties (clients and server) are expected to be honest-but-curious. An in-depth comparison is provided below.

Comparison to SANNS. We note that Chen et al. [26] only evaluate their approach on the DEEP1B and SIFT datasets, and also use the smaller 10M feature vector version of DEEP1B in their evaluation. To match the evaluation of SANNS, we compare with two network settings. We note that the network configuration used by SANNS has throughput that is *faster* than what we were able to measure on `localhost` using `iperf3` [46], which capped at 3.6 GB/s. The first setting has network throughput between 40 MB/s to 2.2 GB/s, which we call the "fast" network. The second setting has network throughput between 500 MB/s to 7 GB/s, which we call the "`localhost`" network (Chen et al. [26] refer to this as the "fast" network in their evaluation). Given these network configurations, SANNS is by no means deployable over realistic network connections [79], which are over 30× slower. Because SANNS does not provide an open-source implementation, we use the query times reported in their evaluation and note that our deployment environment resembles theirs (comparable CPUs, network, and degree of parallelism applied). We report the results of our comparison in Table V. Our improvements

	Protocol 1 (this work)	**SANNS** (localhost)	**SANNS** (fast network)
SIFT			
Latency (1 CPU):	28.2 s	8.06 s (3.5× ↓)	59.7 s (2.1× ↑)
Latency (32 CPUs):	1.1 s	1.55 s (1.4× ↑)	14.2 s (12.9× ↑)
Communication:	1.5 MB	1.77 GB (1180× ↑)	
DEEP1B			
Latency (1 CPU):	170 s	30.1 s (5.65× ↓)	181 s (1.1× ↑)
Latency (32 CPUs):	6.13 s	4.58 s (1.3× ↓)	37.2 s (6.1× ↑)
Communication:	1.7 MB	5.53 GB (3250× ↑)	

TABLE V: End-to-end comparison between Protocol 1 and SANNS over a 500 MB/s to 7 GB/s network (localhost) and a fast network (40 MB/s to 2.2 GB/s). We fix $L = 20$ hash tables and 50 multi-probes per table (for $\approx 95\%$ accuracy). SANNS is network-dominated and hence parallelizes less favorably than Protocol 1 (see [26, Table 2]).

over SANNS are primarily in terms of communication costs. However, we incur a modest latency overhead on very high bandwidth networks (a setting that is highly favorable to SANNS). Our latency is between 1.3–5.65× greater on the "localhost" network. On the "fast network", our protocol is 1.1–12.9× faster compared to SANNS. Our communication cost is a factor of 1180–3250× less in comparison to SANNS. As a result, over slower networks, e.g., average mobile network supporting 12 Mbps [79], SANNS would incur latency ranging between 19 minutes (for SIFT) to an hour (for DEEP1B), just from the network delay. Over such networks, the latency of Protocol 1 is expected to be several orders of magnitude faster in comparison. Additionally, because bandwidth costs can be upwards of $0.02 per GB [12], while CPU cost is around $0.2 *per hour*, our protocol is monetarily cheaper (per query). SANNS costs 4¢–11¢ per query just for bandwidth alone. In contrast, our protocol costs up to 0.02¢ per query (a 200–550× reduction in total cost).

IX. Related work

Existing works on privacy-preserving similarity search either use heavy cryptographic tools (e.g., general secure function evaluation instantiated using two-party computation and fully-homomorphic encryption) or provide poor privacy guarantees for either the client or the database.

Two-party computation based approaches. Indyk and Woodruff [50] investigate nearest neighbor search between two parties under the Euclidean distance metric. They show a $\tilde{O}(\sqrt{N})$ communication protocol for finding an approximate *near* (as opposed to *nearest*) neighbor to a query. Their techniques rely on black-box two-party computation. This makes them only asymptotically efficient (they do not provide an implementation or any concrete efficiency estimate). However, their protocol shares some similarity to ours. Specifically, they tolerate some precisely quantified leakage, which they argue can be a suitable compromise in favor of efficiency gains.

More recently, Chen et al. [26] design and evaluate SANNS, a system for approximate nearest neighbor search that uses oblivious RAM, garbled circuits, and homomorphic encryption. Their solution combines heuristic k-means clustering techniques to reduce overhead of two-party computation of computing oblivious comparisons by a constant factor (i.e., $\frac{1}{k}$). However, they still require asymptotically linear communication, since k is typically small. They leave open the possibility of using locality-sensitive hashing to provide *provable* guarantees.

Qi and Atallah [70] present a protocol for privacy-preserving nearest neighbor search in the honest-but-curious setting with two parties. In contrast to us, they assume each party (i.e., server) has a database that is private from the other party. Queries are computed over the union of both databases. Their protocol uses secure two-party computation to compute oblivious comparisons and requires linear communication in the database size. Qi and Atallah [70] do not provide an implementation or any concrete efficiency estimates for their protocol.

Fully-homomorphic encryption based approaches. Shaul et al. [78] present a protocol based on fully-homomorphic encryption, requiring several hours of computation time to answer queries over small (1000 item) databases. While this results in both a single-round protocol and tolerates malicious clients, it is not practical for large databases. Their implementation requires between three and eight hours (parallelized across 16 cores) to compute the nearest neighbors on small datasets ranging between 1,000 and 4,000 feature vectors.

Zuber and Sirdey [87] implement a secure nearest neighbor classifier using (threshold) fully-homomorphic encryption with applications to collaborative learning and nearest neighbor search. Their approach requires over one hour of server processing time to compute a query answer over a small database of approximately 500 feature vectors. As such, their protocol is not scalable beyond databases containing a few thousand feature vectors.

Partially-private approaches. Not directly related to privacy, Aumüller et al. [11] introduce distance-sensitive hashing which they show can be beneficial to reducing information leakage between hashes. However, their security guarantees are not formally defined and their approach provides a trade-off between privacy and accuracy, leaking information about the client's query and the database simultaneously. Riazi et al. [75] likewise explore LSH as a means of trading-off privacy with accuracy, with more accurate results revealing more information on both the query and the database. They make use of two-party computation to instantiate a garbled circuit for the purpose of securely evaluating locality-sensitive hashes without revealing the description of the hash function. While this reduces some leakage, their approach still reveals information on the query and the database.

In a similar vein, Boufounos and Rane [19] develop a binary embedding (locality-sensitive hash) that preserves privacy when finding similar feature vectors in a remote database. Their technique is less general compared to [11, 75]. Boufounos and Rane [19] do not provide a formal security analysis of their nearest neighbor search protocol based on their secure embedding. Analyzing their protocol, we found that (1) some partial information on the query is inadvertently leaked to the server, and (2) the client learns the distances from its query to all near-neighbors, resulting in significant database leakage. No implementation or any concrete runtime estimates were provided for their protocol.

X. Conclusion

We presented a lightweight privacy-preserving protocol for approximate nearest neighbor search in the two-server model. Our protocol requires only one round and a few megabytes of communication between the client and servers. We do not require servers to communicate when answering queries and guarantees out-of-the-box security against malicious clients. In our evaluation (Section VIII), we show that our protocol remains practical with large databases (10M items), even on high-latency, low-bandwidth networks and lightweight clients (e.g., low-end smartphones).

XI. Acknowledgements

We thank Kyle Hogan and Zachary Newman for helpful feedback on early drafts of this paper. We would also like to thank the anonymous reviewers and our shepherd for their insightful suggestions that helped us to improve the paper.

References

[1] Source code. https://github.com/sachaservan/private-ann.

[2] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *Cryptology ePrint Archive*, 2021.

[3] Thomas D Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 239–256. SIAM, 2017.

[4] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 459–468, 2006. doi: 10.1109/FOCS.2006.49.

[5] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. *arXiv preprint arXiv:1509.02897*, 2015.

[6] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest neighbor search in high dimensions. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pages 3287–3318. World Scientific, 2018.

[7] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[8] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979. IEEE, 2018.

[9] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. *Proc. Priv. Enhancing Technol.*, 2018(4): 104–124, 2018.

[10] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *CoRR*, abs/1807.05614, 2018. URL http://arxiv.org/abs/1807.05614.

[11] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Francesco Silvestri. Distance-sensitive hashing. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 89–104, 2018.

[12] Azure. Microsoft Azure bandwidth pricing. https://azure.microsoft.com/pricing/details/bandwidth/. Accessed August 2021.

[13] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *2012 IEEE Symposium on Security and Privacy*, pages 257–271. IEEE, 2012.

[14] Mauro Barni, Tiziano Bianchi, Dario Catalano, Mario Di Raimondo, Ruggero Donida Labati, Pierluigi Failla, Dario Fiore, Riccardo Lazzeretti, Vincenzo Piuri, Fabio Scotti, et al. Privacy-preserving fingercode authentication. In *Proceedings of the 12th ACM workshop on Multimedia and security*, pages 231–240, 2010.

[15] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Annual International Cryptology Conference*, pages 55–73. Springer, 2000.

[16] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[17] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[18] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.

[19] Petros Boufounos and Shantanu Rane. Secure binary embeddings for privacy preserving nearest neighbors. In *2011 IEEE International Workshop on Information Forensics and Security*, pages 1–6. IEEE, 2011.

[20] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[21] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[22] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In *International Conference on Security and Cryptography for Networks*, pages 215–232. Springer, 2020.

[23] Rui Cai, Chao Zhang, Lei Zhang, and Wei-Ying Ma. Scalable music recommendation by search. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 1065–1074, 2007.

[24] Xia Cao, Shuai Cheng Li, Beng Chin Ooi, and Anthony KH Tung. Piers: An efficient model for similarity search in DNA sequence databases. *ACM Sigmod Record*, 33(2):39–44, 2004.

[25] Nicholas Carlini, Samuel Deng, Sanjam Garg, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Abhradeep Thakurta, and Florian Tramèr. Is private learning possible with instance encoding? In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 410–427. IEEE, 2021.

[26] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. SANNS: Scaling up secure approximate k-nearest neighbors search. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[27] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. *IACR Cryptol. ePrint Arch.*, 2020:83, 2020.

[28] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[29] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords*. Citeseer, 1997.

[30] J. Conway and N. Sloane. Soft decoding techniques for codes and lattices, including the golay code and the leech lattice. *IEEE Transactions on Information Theory*, 32(1): 41–50, 1986. doi: 10.1109/TIT.1986.1057135.

[31] J. H. Conway, R. A. Parker, and N. J. A. Sloane. The covering radius of the Leech lattice. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 380(1779):261–290, 1982. ISSN 00804630. URL http://www.jstor.org/stable/2397303.

[32] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, 2017.

[33] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[34] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[35] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020.

[36] Jack Doerner and abhi shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[37] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792, 2021.

[38] David Evans, Yan Huang, Jonathan Katz, and Lior Malka. Efficient privacy-preserving biometric identification. In *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS*, volume 68, pages 90–98, 2011.

[39] GNU foundation. GMP library. https://gmplib.org/. Accessed August 2021.

[40] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82(72-107):113, 2004.

[41] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences*, 60(3):592–629, 2000.

[42] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014.

[43] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[44] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[45] Internet Security Research Group. Prio services: Privacy-respecting application metrics. https://www.abetterinternet.org/prio/, 2021. Accessed August 2021.

[46] Vivien Gueant. iPerf—the TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/, 2021. Accessed August 2021.

[47] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical privacy in online advertising. In *USENIX conference on Networked systems design and implementation*, pages 169–182, 2011.

[48] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[49] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[50] Piotr Indyk and David Woodruff. Polylogarithmic private approximations and efficient matching. In *Theory of Cryptography Conference*, pages 245–264. Springer, 2006.

[51] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 262–271, 2004.

[52] William B Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space 26. *Contemporary mathematics*, 26, 1984.

[53] Shakira Banu Kaleel and Abdolreza Abhari. Cluster-discovery of Twitter messages for event detection and trending. *Journal of computational science*, 6:47–57, 2015.

[54] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[55] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–

892. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/kogan.

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

[57] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[58] Yury Lifshits. The homepage of nearest neighbors and similarity search. http://simsearch.yury.name/, 2008. Accessed August 2021.

[59] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 52–78. Springer, 2007.

[60] Dandan Lu, Yue Zhang, Ling Zhang, Haiyan Wang, Wanlin Weng, Li Li, and Hongmin Cai. Methods of privacy-preserving genomic sequencing data alignments. *Briefings in Bioinformatics*, 2021.

[61] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *33rd International Conference on Very Large Data Bases, VLDB 2007*, pages 950–961. Association for Computing Machinery, Inc, 2007.

[62] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[63] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[64] Rajeev Motwani, Assaf Naor, and Rina Panigrahi. Lower bounds on locality sensitive hashing. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 154–157, 2006.

[65] Muhammad Naveed, Erman Ayday, Ellen W Clayton, Jacques Fellay, Carl A Gunter, Jean-Pierre Hubaux, Bradley A Malin, and XiaoFeng Wang. Privacy in the genomic era. *ACM Computing Surveys (CSUR)*, 48(1): 1–44, 2015.

[66] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-Bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/newman.

[67] Laurent Noé and Gregory Kucherov. *YASS: Similarity search in DNA sequences*. PhD thesis, INRIA, 2003.

[68] Naoaki Okazaki and Jun'ichi Tsujii. Simple and efficient algorithm for approximate dictionary matching. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 851–859, 2010.

[69] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. *arXiv preprint cs/0510019*, 2005.

[70] Yinian Qi and Mikhail J Atallah. Efficient privacy-preserving k-nearest neighbor search. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 311–319. IEEE, 2008.

[71] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, 2005(187), 2005.

[72] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[73] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. Randomized algorithms and NLP: Using locality sensitive hash functions for high speed noun clustering. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 622–629, 2005.

[74] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3. 2018.

[75] M Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. Sub-linear privacy-preserving near-neighbor search. *arXiv preprint arXiv:1612.01835*, 2016.

[76] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. *Cryptology ePrint Archive*, 2021.

[77] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[78] Hayim Shaul, Dan Feldman, and Daniela Rus. Secure k-ish nearest neighbors classifier. *Proceedings on Privacy Enhancing Technologies*, 2020(3):42–61, 2020.

[79] Speedtest.com. Speedtest.com global index. https://www.speedtest.net/global-index, 2021. Accessed August 2021.

[80] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings Network and Distributed System Symposium*, 2010.

[81] Avery Wang et al. An industrial strength audio search algorithm. In *Ismir*, volume 2003, pages 7–13. Citeseer, 2003.

[82] Web Incubator CG. FLoC. https://github.com/WICG/floc, 2021. Accessed August 2021.

[83] Guowen Xu, Hongwei Li, Hao Ren, Xiaodong Lin, and Xuemin Sherman Shen. DNA similarity search with access control over encrypted cloud data. *IEEE Transactions on Cloud Computing*, 2020.

[84] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[85] Peter N Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. 1999.

[86] Clara E Yoon, Ossian O'Reilly, Karianne J Bergen,

and Gregory C Beroza. Earthquake detection through computationally efficient similarity search. *Science advances*, 1(11):e1501057, 2015.

[87] Martin Zuber and Renaud Sirdey. Efficient homomorphic evaluation of k-NN classifiers. *Proc. Priv. Enhancing Technol.*, 2021(2):111–129, 2021.

APPENDIX

A. LSH-based ANN data structure

We describe the LSH-based data structure for approximate nearest neighbor search of Gionis et al. [43] in Figure 9.

LSH-based ANN search data structure

BUILD$(\mathcal{DB}, \mathcal{H}, L) \to (T_1, \ldots, T_L)$ takes as input a set of N vectors $\mathcal{DB} = \{\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}\}$, LSH family $\mathcal{H}$ (Definition 1), and number of tables L. Outputs hash tables $T_1, \ldots, T_L$.

1: Sample L LSH functions $\mathsf{h}_1, \ldots, \mathsf{h}_L$ from $\mathcal{H}$.
2: Use h_i to build T_i by hashing each vector $\boldsymbol{v_1}, \ldots, \boldsymbol{v_N}$.
3: Output L hash tables $T_1, \ldots, T_L$.

QUERY$(T_1, \ldots, T_L, \mathsf{h}_1, \ldots, \mathsf{h}_L, \boldsymbol{q}) \to \mathsf{ID}$ takes as input a query vector $\boldsymbol{q}$, L hash tables, and LSH functions.

1: Compute bucket key $\alpha \leftarrow \mathsf{h}_i(\boldsymbol{q})$ and retrieve the corresponding bucket $\mathcal{B}_\alpha$ in T_i under key α (if non-empty).[a]
2: Set $\mathcal{C} := \mathcal{B}_1 \cup \cdots \cup \mathcal{B}_L$.
3: Find j such that $\boldsymbol{v_j} \in \mathcal{C}$ and $\Delta(\boldsymbol{v_j}, cR) \leq cR$ via brute-force distance comparisons.
4: **if** no such j exists, output 0; **else** output j.

[a]Note that by the properties of LSH, the query will collide with probability proportional to the relative distance from other vectors.

Fig. 9: ANN search data structure based on locality-sensitive hashing of Gionis et al. [43].

B. Determining LSH radii for radix sorting

We discuss how to determine the radii to use when instantiating the data structure of Figure 4. We sample $10,000$ points from each training set, and find the nearest neighbor among the other training set points. The results are shown in Figure 11. We use a normal distribution computed over the resulting distances as an approximation (orange line in Figure 11), and choose the R_i along the quantiles of the normal distribution. Figure 11 is shown for $L = 10$. This ensures that the number of expected candidates colliding at each R_i is approximately the same (in Figure 11, the area between the dashed lines is the same). Our intuition is that equal areas means that each table will have $\frac{1}{L}$ of the collisions, balancing the load and maximizing efficiency. Each vertical dotted line in Figure 11 corresponds to the R_i for an LSH family for the ith hash table, where $R_1 > D_{\mathsf{min}}$ and $R_{10} = R_{\mathsf{max}}$.

Implementation of the LSH. We use the Leech lattice LSH of Andoni and Indyk [4], which we describe in Appendix C for completeness. We find the closest Leech lattice point to a specified point (e.g., the query) using the decoder described by Conway and Sloane [30]. The coordinates of the lattice point are then mapped to the DPF domain using a universal hash. Since the Leech lattice is a 24-dimensional object, the first step of Andoni and Indyk [4, Appendix B] is dimensionality reduction [52]. We randomly project d/k of the coordinates onto each lattice before concatenating the hashes (we use $k = 2$; see Proposition 1). Many locality sensitive hashing algorithms have efficient methods for multi-probing [61]. For lattice based hashes, we choose the multi-probes from the set of closest lattice points, as these correspond to unique hash values. Indeed, the client can advantageously select the closer lattice points when retrieving candidates through PBR (Section V-A), letting the retrieval failures correspond to probes that are further away.

C. Efficient LSH for Euclidean distance

In this section we describe how to instantiate Proposition 1 efficiently for Euclidean distance ANN search. We first note that Proposition 1 is an upper bound and assumes worst-case data (all points are between R and cR from the query). In practice, there are LSH functions that can achieve very small p_2 without needing a large k. We take another look at Proposition 1. For a large enough $k > d$, the region of space defined by the set $\{x \mid \mathsf{h}(x) = \mathsf{h}(y)\}$ for a fixed y becomes bounded. Specifically, there exists a bounding distance b such that

$$\Pr[h(x) = h(y) \wedge ||x - y|| > b] < \delta.$$

For a false-positive rate of δ, it suffices to scale the space so that the bounding distance b is equal to cR. This scaling decreases p_1 as explained in the proof of Proposition 1. For better efficiency, we observe that it is possible to use LSH functions that *inherently* have bounded regions, such as the Leech lattice based LSH of Andoni and Indyk [4]. By using structure rather than randomness they can achieve a larger p_1 for the same p_2 as one might expect from $k \approx d$. We briefly describe a lattice-based LSH family for Euclidean distance next.

Lattice-based LSH. A simple lattice-based locality sensitive hash family for Euclidean distance is as follows. Choose an infinite set of points in $\mathbb{R}^d$, and let the hash of x be the closest point of this set. For example, if we consider the set of points with integer coordinates, we can efficiently find the hash by rounding the coordinates of x. Lattice based hashes allow us to bound the error distance on p_2; with the integer points, it is clear that $b \leq \sqrt{d}$. Furthermore, we can reduce b to b' by returning $\perp$ if the distance $||x - h(x)||$ is not less than $b'/2$. However, the set of integer points is not optimal in that the error increases greatly for the higher dimensional data we wish to use it on. For an efficient implementation, we use the Leech lattice (as in [4, Appendix B]), which has $b \leq \sqrt{2}$ for $d = 24$. See Conway et al. [31] or Conway and Sloane [30] for more information on the lattice structure.

D. Proof of Proposition 1

Consider the standard LSH-based data structure [6, 43, 49] described in Figure 9. This data structure achieves asymptotic space and query time $O(N^{1+\rho})$ and $O(N^\rho)$, respectively [6].

Now consider the same data structure but where QUERY is modified to a return a *random* element $\in \mathcal{C}$ (if $\mathcal{C} \neq \emptyset$) instead

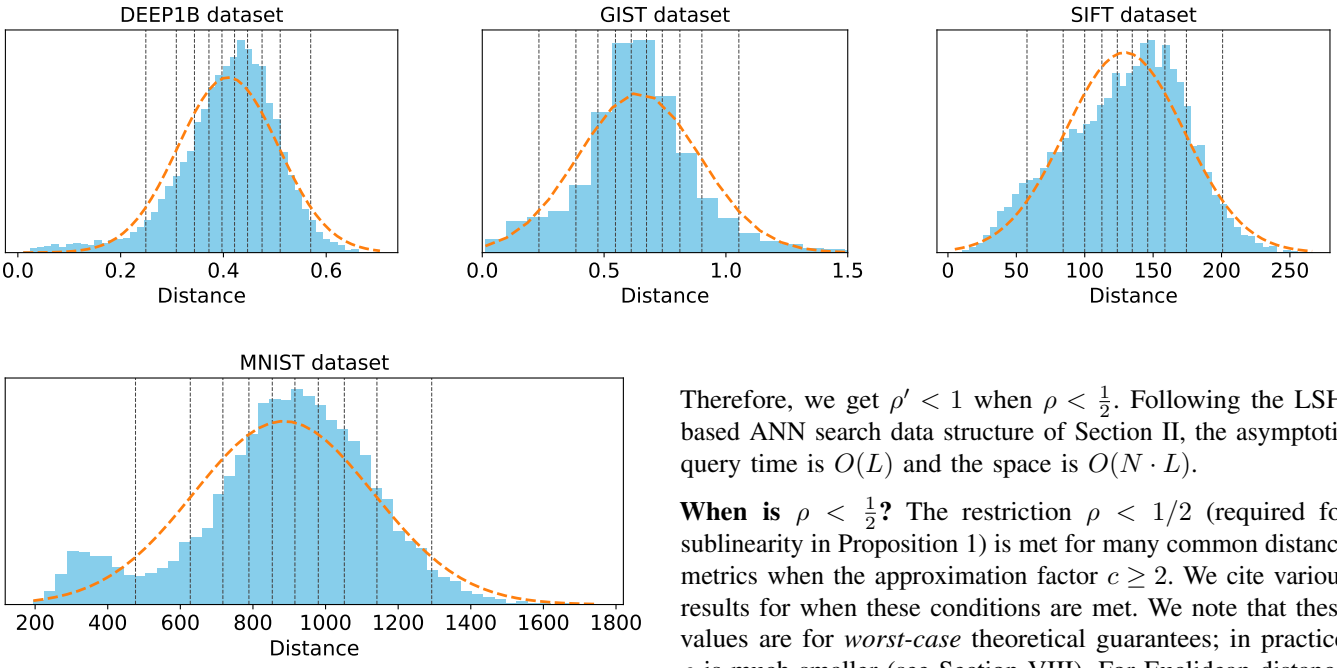

Fig. 11: Distances to the nearest neighbor over 10,000 training points in each dataset. The orange curve plots the normal approximation that informs the choice of each R_i for the radix bucketing (see Section IV-D). The vertical dashed lines represent the choice of each R_i, for $L = 10$ hash tables.

of an element that is guaranteed to be within distance cR. We need to bound the success probability of this random element being within cR of the query to $1 - \delta$. By Definition 1 (LSH), we have that $p_1 > p_2$. Therefore, there exists a "gap" between the probability of a true positive and a false positive collision, which can be amplified exponentially in k. First, observe that the probability of selecting a false-positive at random from $\mathcal{C}$, when the database size is N, is bounded by

$$\frac{N \cdot p_2^k}{p_1^k + N \cdot p_2^k} < N \cdot \frac{p_2^k}{p_1^k}, \tag{1}$$

which we further need to bound by δ. Since we have that $p_1 > p_2$ (Definition 1), it follows that $p_2/p_1 < 1$. Therefore, when

$$k \geq \left\lceil \frac{\log(N) + \log(1/\delta)}{\log(1/p_2) + \log(p_1)} \right\rceil,$$

we get that (1) is bounded by δ, which leads to success probability $1 - \delta$ of selecting a true-positive at random (when it is contained in the candidate set). We contrast this to the standard LSH-based data structures (e.g., Appendix A) where

$$k \geq \left\lceil \frac{\log(N)}{\log(1/p_2)} \right\rceil \text{ (see [6]).}$$

The new data structure results in the elimination of brute-force comparison while still preserving accuracy guarantees. Finally, to prove space and query time, we note that the expected number of tables is

$$L = \lceil p_1^{-k} \rceil = N^{\frac{\rho}{1-\rho}} \cdot \delta^{\frac{1-\rho}{\rho}}.$$

Therefore, we get $\rho' < 1$ when $\rho < \frac{1}{2}$. Following the LSH-based ANN search data structure of Section II, the asymptotic query time is $O(L)$ and the space is $O(N \cdot L)$.

When is $\rho < \frac{1}{2}$? The restriction $\rho < 1/2$ (required for sublinearity in Proposition 1) is met for many common distance metrics when the approximation factor $c \geq 2$. We cite various results for when these conditions are met. We note that these values are for *worst-case* theoretical guarantees; in practice, ρ is much smaller (see Section VIII). For Euclidean distance, Andoni and Indyk [4] show $\rho < \frac{1}{c^2} = \frac{1}{4}$ with $c = 2$. Andoni et al. [5] also show a similar result for angular distance. For any p-stable distribution with $p \in \{1, 2\}$, Datar et al. [34] show $\rho < \frac{1}{c} = \frac{1}{2}$ for $c = 2$. For the ℓ_1-norm specifically, Motwani et al. [64] show $\rho < \frac{1}{2c} = \frac{1}{4}$ with $c = 2$. Hamming distance can be embedded into Euclidean space (see Aumüller et al. [10]). Alternatively, [85, Corollary 3.10] shows $\rho < \frac{1}{c} = \frac{1}{2}$ for Hamming distance directly.

E. Bounding concrete leakage

From the proof of Claim 3, we see that the asymptotic leakage consists of at most one LSH digest, even when the client is acting maliciously (Claim 4). In the worst case, this digest corresponds to a full feature vector, leading to the asymptotic bound. However, concretely, we would like to analyze *how much worse* Protocol 1 is in comparison to Functionality 1. We answer this in Claim 5. We show that Protocol 1, when instantiated for Euclidean distance using the Leech lattice LSH (Appendix C), leaks a more precise approximation compared to Functionality 1. Intuitively, this extra leakage comes from the client learning which radix bucket the nearest neighbor to the query is located in. To analyze the concrete leakage, we must first establish some technical groundwork pertaining to the Leech-lattice LSH we use for Euclidean (and Angular) distance. We do so in Lemma 1, where we will show that a Leech-lattice LSH digest is inherently less precise than the corresponding ideal ball in Euclidean space. Specifically, we show that the ideal ball of radius R has a smaller volume compared to the region of space represented by an LSH digest.

Lemma 1. *The $(1, 2, 0.0097459, 0.0000156)$-sensitive Leech lattice LSH [4] has volume at least $\frac{V_B}{0.77}$, where V_B is the volume of the 24-dimensional unit ball.*

Proof. Our proof relies on the following fact about the Leech lattice.

> **Fact 1:** The Leech lattice is a 24 dimensional object with a sphere packing density of $\frac{\pi^{12}}{12!}$ [31], where the density is defined as the fraction of space covered by (tangent) balls centered at the lattice points.

By Fact 1, we have that the ratio of the volume of the unit ball to that of the lattice cell is given by the density. A small calculation shows that the lattice cell then has volume 1. We ask what is the minimum radius for a 24-dimensional ball such that the ball has volume 1. We find that the answer is

$$\sqrt[24]{\frac{12!}{\pi^{12}}} \approx 1.29.$$

The inverse of this quantity is 0.77, which proves the lemma. For the $(1.2, 1.8, 0.00515, 0.0000771)$-sensitive LSH [4], this factor is $\frac{1.2}{1.29} \approx 0.93$, as $c = 1.5$ is a tighter approximation.

The values of $p_1 = 0.0097459$ and $p_2 = 0.0000156$ for $c = 2$ are from [4, Table 1]. ■

Remark 2. In Lemma 1 we describe the $(1, 2, 0.0097459, 0.0000156)$-sensitive Leech lattice LSH for Euclidean space as described by Andoni and Indyk [4]. We note that this LSH can be used for any R and cR (while also preserving the same p_1 and p_2), by simply scaling the input vectors accordingly. See Andoni and Indyk [4] for a more detailed explanation.

Using Lemma 1, we can bound the concrete leakage of Protocol 1 when instantiated with the Leech-lattice LSH. This allows us to empirically bound the concrete leakage for the datasets we use in our evaluation (Section VIII). We compute a precise leakage bound in Claim 5.

Claim 5 (Concrete Leakage of Protocol 1)**.** Fix R_{min} and R_{max} as defined in both Figure 4 and Functionality 1, where $D_{\mathsf{min}} < R_{\mathsf{min}} \leq R_{\mathsf{max}} < D_{\mathsf{max}}$. Protocol 1, when instantiated with the Leech-lattice based LSH of Andoni and Indyk [4] leaks at most a multiplicative factor of $0.77 \cdot \frac{R_{\mathsf{max}}}{R_{\mathsf{min}}}$ more information compared to Functionality 1.

Proof. We recall the argument in the proof of Claim 3. Fix any $\boldsymbol{v_j} \in \mathcal{DB}$ and S_j as in the proof of Claim 3. We show that S_j is an upper bound on what can be revealed on $\boldsymbol{v_j}$ through Protocol 1. We now examine how much faster this set can be leaked with queries issued to Protocol 1 and contrast it to Functionality 1.

Define an baseline oracle $\mathcal{O}(\mathcal{DB}, R_i, \cdot)$, which given a query $\boldsymbol{q}$, outputs the ID of any vector in $\mathcal{DB}$ within distance R_i of $\boldsymbol{q}$, if such a vector exists. Observe that Functionality 1 is modeled by $\mathcal{O}(\mathcal{DB}, R_{\mathsf{max}}, \cdot)$.

Suppose that the client obtains a query answer and learns $\alpha_i \in S_j$. Note that this leakage is less than (or equal to) that of learning $\alpha_1 \in S_j$. That is, the hash of the same vector but on the smallest radius. It is easy to see that α_1 is at least as precise (contains as much information) as $\alpha_i \in S_j$, $i > 1$, due to the increasing radii of the radix buckets, as explained in Section IV-B.

Next, by Lemma 1, we have that the information revealed by $\alpha_1 = \mathsf{h}_1(\boldsymbol{q})$ is less than or equal to the information revealed by $\mathcal{O}(\mathcal{DB}, R_1, \boldsymbol{q})$. Indeed, as was shown in Lemma 1, α_i is 0.77 times less precise compared to $\mathcal{O}(\mathcal{DB}, R_i, \boldsymbol{q})$, for any i.

Therefore, we have that the ratio in precision between α_1 and $\mathcal{O}(\mathcal{DB}, R_{\mathsf{max}}, \cdot)$, that is, the ratio between the precision of the smallest radix bucket and the baseline functionality, is bounded by 0.77 times the ratio of precision between the baseline oracles $\mathcal{O}(\mathcal{DB}, R_{\mathsf{min}}, \cdot)$ and $\mathcal{O}(\mathcal{DB}, R_{\mathsf{max}}, \cdot)$, respectively. Because the latter ratio is simply $\frac{R_{\mathsf{max}}}{R_{\mathsf{min}}}$, we get that Protocol 1 leaks at most a multiplicative factor of $0.77 \cdot \frac{R_{\mathsf{max}}}{R_{\mathsf{min}}}$ more compared to the baseline leakage of Functionality 1, derived in Theorem 1. ■

We empirically compare the concrete leakage of Protocol 1 to the baseline leakage in Section VII-B on real world data. We do so by finding values for R_{min} and R_{max} as a function of L (see Appendix B for how this is done) and applying Claim 5.

Corollary 1. The leakage of k queries to Protocol 1 is bounded by a multiplicative factor of $0.77 \cdot \frac{R_{\mathsf{max}}}{R_{\mathsf{min}}}$ more than the leakage of k queries to Functionality 1.

Proof. We only need to consider the relative leakage between k successive queries to $\mathcal{O}(\mathcal{DB}, R_{\mathsf{min}}, \cdot)$ and $\mathcal{O}(\mathcal{DB}, R_{\mathsf{max}}, \cdot)$, resulting in at most a factor of k more leakage. Another attack one might consider is to learn more information across multiple queries. While the information given by k hashes for the same vector is more specific (e.g., the set of points that have both hashes is smaller), the same effect occurs with queries to the baseline oracle. If two queries spaced by some offset return the same nearest neighbor, then that neighbor must be in the intersection of the regions of radius R_{max} centered at each query. The leakage factor captures the extra precision in our case. It also might be the case that the malicious client is able to pick queries by exploiting hashes that return zeroes. However, this is bounded by obtaining a new element of S_j (the set of all LSH digests for a vector $\boldsymbol{v_j}$; see proof of Claim 3) each time, which is captured in our concrete leakage bound. ■