Private Nearest Neighbor Search
with Sublinear Communication and Malicious Security

Sacha Servan-Schreiber
MIT CSAIL
3s@mit.edu

Simon Langowski
MIT CSAIL
slangows@mit.edu

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

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, depending on parameters.

I. INTRODUCTION

Nearest neighbor search is used in a wide range of online applications, including recommendation engines [23, 83], reverse image search [58], image-recognition [60], earthquake detection [89], computational linguistics [65], natural-language processing [76], targeted advertising [75, 84], and numerous other areas [55, 61, 71, 75].

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).

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, 48, 75, 82, 84], biometric data [14, 39], medical records [9, 80], and DNA data analysis [24, 63, 70, 86]. 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 [68]. Both regulatory and ethical reasons dictate that such personal information (represented in the query) should be kept private from the database.

The potential privacy issues surrounding similarity search
have motivated a handful of privacy-preserving protocols [26, 52, 73, 80, 90]. 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.

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 highly inefficient to instantiate securely [87]. 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.

**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 $v_1, \ldots, v_N$ and corresponding item identifiers $ID_1, \ldots, ID_N$. A client has a query vector $q$. The client must learn only the ID(s) of the nearest neighbor(s) relative to $q$ under some similarity (or distance) metric. For client privacy, the protocol must not leak any information about $q$ to the database servers. For database privacy, the protocol must leak as little as possible on $v_1, \ldots, v_N$ 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.

**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 [56] 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 [43] 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. We note, however, that using non-colluding servers to instantiate lightweight privacy-preserving systems has proven fruitful in other areas [2, 18, 27, 32, 33, 35, 36, 38, 57, 69], including systems deployed in industry [35, 46].

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 [51]. 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 \( D \), output space \( R \), and a distance metric \( \Delta \), an LSH family is defined as:

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

\[
\begin{align*}
\text{if } \Delta(v, q) \leq R & \text{ then } \Pr[h(v) = h(q)] \geq p_1, \\
\text{if } \Delta(v, q) > cR & \text{ then } \Pr[h(v) = h(q)] \leq p_2,
\end{align*}
\]

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. [44], 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 to obtain good accuracy while keeping the query time sublinear [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 \( q \) 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 \( DB \) the database of vectors and their IDs. We let \( N \) be the total number of \( d \)-dimensional vectors in \( DB \). A vector is denoted in bold as \( v \) where the \( i \)th coordinate of \( v \) is denoted by \( v_i \). A distance metric (e.g., Euclidean distance) is denoted \( \Delta \), where threshold distances \( R \) and \( cR \) are as defined in Section II-A. We let \( F \) denote any prime-order finite field (e.g., integers mod a prime \( p \)). A secret-share of a value \( v \in F \) is denoted using bracket notation as \( [v] \). Coordinate-wise secret-shares of a vector \( v \) are denoted as \( [v] \). Variable assignment is denoted by \( x \leftarrow y \), where \( x \leftarrow S \) 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 \( q \) to output the ID of the nearest neighbor relative to \( q \) 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 neighbor farther 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 a 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 \( R = R_{max} \) 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 \( R_{max} < D_{max} \).

\[ \text{This name comes from the } \text{near neighbor search problem} [50][3]. \]
**Functionality 1: ANN Search**

*Public Parameters:* Distance metric $\Delta$, database size $N$, feature vector dimensionality $d$, maximum nearest neighbor radius $R_{\text{max}}$.

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

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

*Procedure:*
1. $v_a \leftarrow$ nearest neighbor to $q$ in $DB$ via brute-force search.
2. If $\Delta(v_a, q) > R_{\text{max}}$ 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. We require black-box public-key infrastructure (e.g., TLS [77]) to encrypt communication over the internet.

**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 ($\text{BUILD}$ and $\text{QUERY}$; 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 $\text{QUERY}$ (see [28, 29, 41]). 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 further 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 [87]). The state-of-the-art 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 [56] 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 $(R, \epsilon R, p_1, p_2)$-sensitive hash family can be “amplified” to result in an $(R, \epsilon R, p_1', p_2')$-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 $(R, \epsilon R, p_1, p_2)$-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 $(p_1, p_2, R, \epsilon R)$-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 $\delta$, 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 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. [50], which resembles radix sorting [56]. A radix sort does not perform direct comparisons, which aligns well with our goals. We recursively apply the data structure of Proposition 1 on a series of increasing neighbor radii, retrieving a set of candidates from each radius [50]. The ANN is then chosen at random from the first non-empty candidate set (see Figure 3 for an illustration of this process).

### C. LSH multi-probing

The number of tables as given by Proposition 1 can be large, but can be decreased through an LSH optimization called 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) [64, 72]. To exploit this observation, each hash table is probed on $\ell$ keys which “surround” the query. In turn, this reduces the number of hash tables ($L$) required (the original motivation of multi-probing [64]) 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 multi-probing function that outputs $\ell$ hashes for a function $h$ and query $q$ as $\text{multiprobe}(h, \ell, q) \rightarrow (\alpha_1 \ldots \alpha_\ell)$.

**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. [44]. 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 one (since only one ID is revealed), which matches the asymptotic (Section II), then roughly speaking, this simple change reduces in Section IV. We first formalize the necessary building-blocks attempt to learn more than an honest client (Section VII).

achieves close to optimal concrete database leakage per

leakage of the baseline functionality. We provide details on

one ID that collided with its query. This further reduces

the masked candidate set

hides all-but-one non-zero candidate ID from the client. From
from optimal. We eliminate (most of) the additional leakage by

Compared to the ideal leakage of

Hiding the candidate set. Compared to the ideal leakage of

O(log N+d) bits, the leakage of O(√N · 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 Ĉ, the client is able to extract at most one ID that collided with its query. This further reduces leakage from O(√N · 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

Pα is a function that evaluates to 1 on a single input α in
its domain and evaluates to 0 on all other inputs j ≠ α. 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 Pα on a given input j.

Definition 2 (Distributed Point Function (DPF) [20, 21, 43]).
Fix any positive integer n. Let F be any finite field (e.g.,
integers mod prime p), and let λ be a security parameter.
A DPF consists of two (possibly randomized) algorithms:

• Gen(1λ, α ∈ {1,...,2n}) → (kA, kB) takes as input an
index α and outputs two evaluation keys kA and kB,
• Eval(k, j) → vj ∈ F takes as input an evaluation key k and
index j ∈ {1,...,2n} and outputs a field element vj.

These algorithms must satisfy correctness and privacy:

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

Pr \[ \text{Eval}(k_A, j) + \text{Eval}(k_B, j) = 1, \text{ if } j = \alpha \]

0, otherwise. = 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, 43].

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,...,2n\} and let the corresponding values be in the field \mathbb{F}. To retrieve the value under key α in T the client generates DPF keys by running Gen(1\^\lambda, α) and sends one key to each server. Each server locally evaluates the DPF key with each item α_i for i = 1,...,N to obtain the secret-share of \text{P}_\alpha(\alpha_i) on input \alpha_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:

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

Because \text{P}_\alpha(\alpha_i) = 1 only when α = \alpha_i, the resulting sum (computed in the field \mathbb{F}) 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 [42] (analogous to oblivious transfer [74]), which guarantees that the answer consists of at most one value.
in $T$. Additionally, modern DPF \cite{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 $\mathbb{F}$), this transformation must only involve affine operations; addition and scalar multiplication of shares \cite{16,79}. The idea is to recursively compute a randomized sum, moving from left to right. For a secret-shared input vector $[v] \in \mathbb{F}^L$, let $z \in \{1, \ldots, L \}$ be the index of the first non-zero element in $v$. The randomized sums map each element $v_i$ to 0 for $i < z$ and a uniformly random element in $\mathbb{F}$ for $i > z$. Crucially, this process does not affect the first non-zero element, $v_z$. We will use this property to mask all-but-one result from a sequence of PIR query answers.

**Algorithm 1: ObliviousMasking.**

<table>
<thead>
<tr>
<th>Procedure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for $i \in {1, \ldots, L}$:</td>
</tr>
<tr>
<td>2. Output $y \in \mathbb{F}^L$.</td>
</tr>
</tbody>
</table>

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

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

1. for $i < z$, $v_i = 0$, so $y_i = 0 + r_i \sum_{j=0}^{i-1} 0 = 0 \in \mathbb{F}$,
2. for $i = z$, $y_z = v_z + r_z \sum_{j=0}^{z-1} 0 + v_z \in \mathbb{F}$,
3. for $i > z$, $y_i = v_i + r_i \sum_{j=0}^{i-1} 0 + r_i v_z + r_i \sum_{j=z+1}^{i-1} v_j \in \mathbb{F}$.

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 $\mathbb{F}$. To see why (3) holds, observe that for $v_z \neq 0$, $v_z$ is a uniformly random element of $\mathbb{F}$.

**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. 

through PIR. A naïve way to approach this problem is to issue $\ell$ separate PIR queries—one per bucket—which incurs a factor of $\ell$ processing overhead on the servers. Partial Batch Retrieval (PBR) (inspired by probabilistic batch codes \cite{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 \cite{53}. 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 $\ell$ 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? Abtractly, the fraction of retrievable elements can be modeled by the classic “balls and bins” problem \cite{66}, where $\ell$ balls are tossed into $m$ bins uniformly at random. If all $\ell$ balls fall into unique bins, then the number of full bins is $\ell$. Fewer than $\ell$ 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 $\frac{\ell}{m} \cdot (1 - e^{-\ell/m})$ of the $\ell$ 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 $\frac{m}{\ell}$. 

**New tool: Partial Batch Retrieval.** LSH multi-probing requires retrieving multiple hash table buckets (called a batch)
**Protocol 1: Private Similarity Search**

Public Parameters: LSH family $\mathcal{H}$, number of hash tables $L$, and LSH functions $h_1, \ldots, h_L$ sampled i.i.d. from $\mathcal{H}_i$.

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

Client Input: query vector $q$.

**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 $\alpha_{i,d} \leftarrow h_i(v_d)$ 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: $\alpha_1, \ldots, \alpha_\ell \leftarrow$ multiprobe $(h_i, \ell, q)$.
   1.2: Choose one hash key for each PBR partition, and fill in 0 for empty partitions (Section V-A). For $j = 1 \ldots m$, let $\alpha^{(i,j)}$ be the key for the $j$th partition of the $i$th table.
   1.3: $(k_A^{(i,j)}, k_B^{(i,j)}) \leftarrow$ DPF.Gen $(1^\lambda, \alpha^{(i,j)})$.

2: $k_A \leftarrow (k_A^{(1,1)}, \ldots, k_A^{(L,m)})$ and $k_B \leftarrow (k_B^{(1,1)}, \ldots, k_B^{(L,m)})$.
3: Send $k_A$ and $k_B$ to servers $A$ and $B$, respectively.

**Step 2** (on each server)
1: 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: $|D_{i,m+j}| \leftarrow \sum_{\alpha \in D_{i,j}} T_i(\alpha) \cdot$ DPF.Eval $(k^{(i,j)}, \alpha)$.

3: $[\hat{C}] \leftarrow (|D_1|, \ldots, |D_L|m)$.
4: $[\hat{C}] \leftarrow$ OBLIVIOUSMASKING([C], rand). // Algorithm 1.

**Step 3** (on the client)
1: Receive $[\hat{C}]_A$ and $[\hat{C}]_B$ from servers $A$ and $B$, respectively.
2: $(\tilde{D}_1, \ldots, \tilde{D}_{L,m}) \leftarrow [\hat{C}]_A + [\hat{C}]_B$. // Recover candidates
3: Output $\tilde{D}_i \neq 0$ for smallest $i$, or 0 if all $\tilde{D}_i$ are zero.

**B. Putting things together**

The full protocol is presented in Protocol 1 and uses the DPF, the oblivious masking, and probabilistic 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 (i.e., indices) 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 $q$ 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 $\ell$ partitions of each of the $L$ hash tables using the PIR technique described in Section V-A. The result is a vector $C$ 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 $C$ and obtains the masked secret-shared vector $\hat{C}$ as output. Each server sends its share of $\hat{C}$ to the client in response.

**Step 3.** From the received secret-shares $(\hat{C}_A, \hat{C}_B)$, the client recovers the vector $\hat{C}$ (in cleartext) by computing $\hat{C} = \hat{C}_A + \hat{C}_B$. The transformation applied by the servers in Step 2 ensures that the client learns at most one non-zero candidate from the original $C$. Specifically, by Claim 1, only the first non-zero value of $C$ will be non-random in $\hat{C}$. 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 $F^k$ (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 analysis**

**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 $\rho < \frac{1}{2}$; see definition of $\rho$ in Proposition 1) and where we assume $c = 2$ (see LSH; Definition 1).

**Communication.** The protocol communication consists of the $L$ DPF keys, each of which retrieves one candidate from a table. The DPF key size is $O(\log N)$ [20, 21]. Each masked

---

Additive secret-share recovery follows from the correctness property of DPFs; see Definition 2.
TABLE I: Asymptotic efficiency of Protocol 1; constant and log factors suppressed. We assume $L \approx \sqrt{N}$ (see Section II-A).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Replication Factor</th>
<th>Total Partitions</th>
<th>Fraction Retrieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>1</td>
<td>1</td>
<td>$\frac{1}{3}$</td>
</tr>
<tr>
<td>SBC [53]</td>
<td>$\frac{3}{2}$ $\ell$</td>
<td>$3\log \ell$</td>
<td>1</td>
</tr>
<tr>
<td>PBC [8]</td>
<td>$3$</td>
<td>$1.5\ell$</td>
<td>$1 - 2^{-20}$</td>
</tr>
<tr>
<td>PBR</td>
<td>$1$</td>
<td>$m$</td>
<td>$\frac{m}{t} \cdot (1 - e^{-\ell/m})$</td>
</tr>
</tbody>
</table>

TABLE II: Replication and partitioning costs of existing batching schemes: Naïve (perform $\ell$ PIR queries), Subcube batch codes (SBC) [53], 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.

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 $A$ corrupting server $b$ for $b \in \{A, B\}$, Protocol 1 guarantees that $A$ 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 $\text{Sim}$ such that

$$\text{Sim}(DB) \approx_c \text{view}(DB, k_b),$$

where $\text{view}$ is the view of $A$ from an execution of Protocol 1 with the client. $\text{Sim}(DB)$ is trivially constructed by invoking the DPF simulator for each DPF key present in the vector $k_b$ provided to $A$ [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 $\ell$ 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, except now it generates a DPF key to query each of the $m$ PBR partitions.

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, \([0, \ldots, 0, 1, D, r_{z+1}, \ldots, r_L]\). 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 \(x\). The simulator must create two vectors, one for each server’s response. It does so by first constructing a vector \(C\) with the first \(z - 1\) entries all zero, ID in the \(z\)th entry, and random numbers for all remaining entries. It then secret shares this vector into two “response vectors” \(C_A\) and \(C_B\) 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 \(q\). 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 \(D_{\text{max}}, D_{\text{min}},\) and \(\Delta\) 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 \(\ell_{\infty}\)-norm and induced distance metric \(\Delta\). The \(\ell_{\infty}\)-norm is the absolute value of the maximum coordinate: \(\ell_{\infty}(x) = \max_{i} |x_i|\). The induced distance metric is \(\Delta(x, y) = \ell_{\infty}(|x - y|)\). We first prove the ideal leakage for the \(\ell_{\infty}\)-norm, as it is easier to intuit. We then show how to extend the proof to any \(\ell_p\)-norm induced metric, which includes Euclidean distance. Other common distance metrics, such as angular distance and Hamming distance, can be embedded into Euclidean space [26] [10].

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

For any \(\ell_p\)-norm, the above argument holds by considering similar shapes and their relative volumes. In the Euclidean metric \(\Delta(v, q) \leq D_{\text{max}}\) defines a ball of radius \(D_{\text{max}}\). The ideal functionality reveals that \(v\) is in a ball of radius \(R_{\text{max}}\) centered at \(q\). The ratio of the volumes of these balls is \(D_{\text{max}}^d/R_{\text{max}}^d\), making the leakage \(\log(D_{\text{max}}^d/R_{\text{max}}^d) = d \log(D_{\text{max}}/R_{\text{max}})\) bits, as in the \(\ell_{\infty}\) 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 \(\log(D_{\text{max}}/R_{\text{max}})\).

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 ideal functionality.

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

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 \(v \in DB\)—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 \(h(x) = x\). It follows that the worst-case implicit query leakage is then \(O(d)\) bits, because \(\alpha_i = v_j\) 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 \(DB\) 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 \(S_j\).

<table>
<thead>
<tr>
<th>Hash tables</th>
<th>Maximum concrete leakage factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEEP1B</td>
<td>MINIST</td>
</tr>
<tr>
<td>(L = 10)</td>
<td>1.97×</td>
</tr>
<tr>
<td>(L = 30)</td>
<td>2.79×</td>
</tr>
<tr>
<td>(L = 50)</td>
<td>3.29×</td>
</tr>
</tbody>
</table>

**TABLE III:** Concrete leakage (computed as \(0.77 \cdot \frac{R_{\text{max}}}{D_{\text{max}}}\)) 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 imprecision 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, this leakage assumes that all points collide at the smallest radius $R_{\text{min}}$, when in practice the points will collide across buckets defined by radii between $R_{\text{min}}$ and $R_{\text{max}}$.

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, 38]. Our implementation uses AES as a pseudo-random generator which exploits the AES-NI instruction for hardware-accelerated operations. We use the GMP library [40] 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 7. 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, 44, 50]. 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 9), 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 9. 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.
Communication. We provide the total communication required to query one hash table in Table IV. The communication 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 [80, 90] 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 [62], 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 [47], which capped at 3.6 GB/s. The first setting has network throughput between 40 MB/s to 2.2 GB/s, which

---

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.

<table>
<thead>
<tr>
<th>Multi-probes:</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication:</td>
<td>3 kB</td>
<td>10 kB</td>
<td>18 kB</td>
<td>87 kB</td>
<td>172 kB</td>
</tr>
</tbody>
</table>

---

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]).

---

Fig. 9: 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).
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 [81], which are over $30 \times$ 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 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 $1.3–5.65 \times$ greater on the “localhost” network. On the “fast network”, our protocol is $1.1–12.9 \times$ faster compared to SANNS. Our communication cost is a factor of $1180–3250 \times$ less in comparison to SANNS. As a result, over slower networks, e.g., average mobile network supporting 12 Mbps [81], 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 $4e–11e$ per query just for bandwidth alone. In contrast, our protocol costs up to $0.02e$ per query (a $200–550 \times$ 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. A summary of related work is provided in Table VI. Two-party computation based approaches. Indyk and Woodruff [52] investigate nearest neighbor search between two parties under the Euclidean distance metric. They show a $O(\sqrt{N})$ 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.

By assuming data-dependent clusters, SANNS is able to forgo a linear scan over the database when evaluating oblivious distance comparisons. While their implementation is efficient over high-bandwidth connections ($40$MB/s to $7$GB/s bandwidth), such throughput is often too high to be practical on real-world connections (especially on mobile networks, which can be $30 \times$ slower [81]). Moreover, SANNS requires the data to be clusterable to reduce communication. Our protocol, in contrast, does not make any assumptions on the input data to achieve sublinear communication. We show in our evaluation (Section VIII) that our protocol requires between $484–1511 \times$ less communication compared to SANNS, and microseconds of processing on the client.

Qi and Atallah [73] 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 [73] do not provide an implementation or any concrete efficiency estimates for their protocol.

Fully-homomorphic encryption based approaches. Shaul et al. [80] 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 [90] implement a secure nearest neighbor classifier using (threshold) fully-homomorphic encryption with applications to collaborative learning and nearest neighbor

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Comm.</th>
<th>Comp.</th>
<th>Rounds</th>
<th>Tools</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFR20 [80]</td>
<td>$\log N$</td>
<td>$N$</td>
<td>1</td>
<td>FHE</td>
<td>$\bullet \circ \circ \circ$</td>
</tr>
<tr>
<td>SANNS [26]</td>
<td>$N/k$</td>
<td>$N$</td>
<td>2PC</td>
<td>$\bullet \circ \circ$</td>
<td></td>
</tr>
<tr>
<td>IW06 [52]</td>
<td>$\sqrt{N}$</td>
<td>$N$</td>
<td>2PC</td>
<td>$\bullet \circ \circ$</td>
<td></td>
</tr>
<tr>
<td>QA08 [73]</td>
<td>$N$</td>
<td>$N$</td>
<td>2PC</td>
<td>$\bullet \circ \circ$</td>
<td></td>
</tr>
<tr>
<td>ZS21 [90]</td>
<td>$\log N$</td>
<td>$N^2$</td>
<td>1</td>
<td>FHE</td>
<td>$\bullet \circ \circ \circ$</td>
</tr>
</tbody>
</table>

1. Uses fully-homomorphic encryption to evaluate the entire computation; takes several hours of computation to evaluate on very small datasets.
2. Theoretical work; no implementation or concrete runtime estimates provided.
3. Uses fully-homomorphic encryption to evaluate the entire computation; takes over 20 minutes to evaluate on very small datasets ($N \approx 1000$).
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. [78] 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, 78]. 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.

Riazi et al. [78] propose a protocol based on a locality-sensitive hash transformation (similar to [11]) to achieve similarity search with slightly less leakage compared to revealing locality sensitive hashes directly. Their approach provides a trade-off between privacy and accuracy, leaking information about the client’s query and the database simultaneously. They make use of two servers to instantiate a garbled circuit for the purpose of securely evaluating locality-sensitive hashes without revealing the description of the hash function.

Dauterman et al. [35] develop a two-server encrypted search system called Dory for reverse-index queries to a database (i.e., exact keyword matching). Their goal is solving metadataprivate exact keyword search where only privacy for the client is considered. We believe that our techniques can be directly applied to extend Dory to support fuzzy-keyword reverse-index search.

**Client-outsourced data.** Other systems for privacy-preserving nearest neighbor search (e.g., [9, 37, 49, 59, 85]) focus on a setting where a client outsources the data to a remote database in a way that it can later privately query for nearest neighbors. As such, they do not consider malicious clients nor are they interested in preserving the privacy of item attributes in the database. We consider this setting as orthogonal.

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).

References


APPENDIX

A. LSH-based ANN data structure

We describe the LSH-based data structure for approximate nearest neighbor search of Gionis et al. [44] in Figure 10.

---

**LSH-based ANN search data structure**

\[
\text{BUILD}(\mathcal{DB}, \mathcal{H}, L) \rightarrow (T_1, \ldots, T_L)\]

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

1. Sample \(L\) LSH functions \(h_1, \ldots, h_L\) from \(\mathcal{H}\).
2. Use \(h_i\) to build \(T_i\) by hashing each vector \(v_1, \ldots, v_N\).
3. Output \(L\) hash tables \(T_1, \ldots, T_L\).

\[
\text{QUERY}(T_1, \ldots, T_L, h_1, \ldots, h_L, q) \rightarrow \text{ID}\]

takes as input a query vector \(q\). \(L\) hash tables, and LSH functions.

1. Compute bucket key \(\alpha \leftarrow h_q(q)\) and retrieve the corresponding bucket \(B_\alpha\) in \(T_i\) under key \(\alpha\) (if non-empty):\(^{\dagger}\)
2. Set \(C := B_1 \cup \ldots \cup B_L\).
3. Find \(j\) such that \(v_j \in C\) and \(\Delta(v_j, cR) \leq cR\) via brute-force distance comparisons.
4. If no such \(j\) exists, output 0; else output \(j\).

\(^{\dagger}\)Note that by the properties of LSH, the query will collide with probability proportional to the relative distance from other vectors.

---

**Fig. 10:** ANN search data structure based on locality-sensitive hashing of Gionis et al. [44].

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 12. We
use a normal distribution computed over the resulting distances as an approximation (orange line in Figure 12), and choose the \( R_i \) along the quantiles of the normal distribution. Figure 12 is shown for \( L = 10 \). This ensures that the number of expected candidates colliding at each \( R_i \) is approximately the same (in Figure 12, 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 12 corresponds to the \( R_i \) for an LSH family for the \( i \)th hash table, where \( R_1 > D_{\text{min}} \) and \( R_{10} = R_{\text{max}} \).

**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 [54]. 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 [64]. 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 closest 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 \( \{ x \mid h(x) = h(y) \} \) for a fixed \( y \) becomes bounded. Specifically, there exists a bounding distance \( b \) such that

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

For a false-positive rate of \( \delta \), 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, 44, 51] described in Figure 10. This data structure achieves asymptotic space and query time \( O(N^{1+\rho}) \) and \( O(\delta^\rho) \), respectively [6].

Now consider the same data structure but where \( \text{QUERY} \) is modified to return a random element \( \in C \) (if \( C \neq \emptyset \)) instead 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 \( C \), when the database size is \( N \), is bounded by

\[
\frac{N \cdot p_2^k}{p_1^k + N \cdot p_2^k} < \frac{N \cdot p_2^k}{p_1^k},
\]

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

\[
k \geq \left\lfloor \log(N) + \log(1/\delta) \over \log(1/p_2) + \log(p_1) \right\rfloor
\]

we get that (1) is bounded by \( \delta \), 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 \( k \geq \left\lfloor \log(N) \over \log(1/p_2) \right\rfloor \) [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

\[
L = \left\lfloor p_1^{-k} \right\rfloor = N^{1+\rho} \cdot \delta. \frac{1+\rho}{\rho}
\]

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, \( \rho \) is much smaller (see Section VIII). For Euclidean distance, Andoni and Indyk [4] show \( \rho = {1 \over 4} = {1 \over 2} \) with \( c = 2 \). Andoni et al. [5] also show a similar result for angular distance. For
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 more precise approximation compared to Functionality 1. We answer this in Claim 5. We show that the Leech lattice LSH (Appendix C), leaks a more precise approximation compared to Functionality 1. Intuitively, this comes from the client learning which radix bucket the nearest neighbor to the query is located in. To analyze this digest corresponds to a full feature vector, leading to extra leakage comes from the client learning which radix bucket the nearest neighbor to the query is located in. To analyze this extra leakage comes from the client learning which radix bucket the nearest neighbor to the query is located in. To analyze 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 \( \frac{V_B}{77} \), 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 \( \frac{5}{12} \) [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 \( \frac{1}{\sqrt{V_B}} \approx 1.29 \).

The inverse of this quantity is 0.77, which proves the lemma. For the \((1.2, 1.8, 0.00515, 0.000771)\)-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 \( R_{\text{min}} \) and \( R_{\text{max}} \) as defined in both Figure 4 and Functionality 1, where \( D_{\text{min}} < R_{\text{min}} < R_{\text{max}} < D_{\text{max}} \). Protocol 1, when instantiated with the Leech-lattice based LSH of Andoni and Indyk [4] leaks at most a multiplicative factor of \( 0.77 \cdot \frac{R_{\text{max}}}{R_{\text{min}}} \) more information compared to Functionality 1.
Proof. We recall the argument in the proof of Claim 3. Fix any $v_j \in DB$ 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 $v_j$ 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 ideal oracle $O(DB, R_i, \cdot)$, which given a query $q$, outputs the ID of any vector in $DB$ within distance $R_i$ of $q$, if such a vector exists. Observe that Functionality 1 is modeled by $O(DB, R_{\max}, \cdot)$.

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 $\alpha_i$ 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 $\alpha_1 = h_1(q)$ is less than or equal to the information revealed by $O(DB, R_{1}, q)$. Indeed, as was shown in Lemma 1, $\alpha_i$ is $0.77$ less precise compared to $O(DB, R_{i}, q)$, for any $i$.

Therefore, we have that the ratio in precision between $\alpha_1$ and $O(DB, R_{\max}, \cdot)$, that is, the ratio between the precision of the smallest radix bucket and the ideal functionality, is bounded by $0.77$ times the ratio of precision between the ideal oracles $O(DB, R_{\min}, \cdot)$ and $O(DB, R_{\max}, \cdot)$, respectively. Because the latter ratio is simply $\frac{R_{\max}}{R_{\min}}$, we get that Protocol 1 leaks at most a multiplicative factor of $0.77 \cdot \frac{R_{\max}}{R_{\min}}$ more compared to the ideal leakage of Functionality 1, derived in Theorem 1.

We empirically compare the concrete leakage of Protocol 1 to the ideal 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 $0.77 \cdot \frac{R_{\max}}{R_{\min}}$ more than the leakage of $k$ queries to Functionality 1.

Proof. We only need to consider the relative leakage between $k$ successive queries to $O(DB, R_{\min}, \cdot)$ and $O(DB, R_{\max}, \cdot)$, 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 ideal 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 $v_j$; see proof of Claim 3) each time, which is captured in our concrete leakage bound. □