BiSON: Billion-Scale Oblivious Nearest-Neighbor Search in Milliseconds

Sankha Das†
Georgia Institute of Technology
Atlanta, USA
sdas435@gatech.edu

Rohan Ravi†
IIT Kanpur
Kanpur, India
rohanra@cse.iitk.ac.in

Nishanth Chandran
Microsoft Research India
Bangalore, India
nichandr@microsoft.com

Divya Gupta
Microsoft Research India
Bangalore, India
divya.gupta@microsoft.com

***Abstract*—Semantic search over vector databases is a fundamental problem in both theory and practice, with large-scale systems relying on approximate nearest-neighbor (ANN) algorithms to retrieve semantically similar results efficiently. Achieving this capability securely while keeping both data and queries hidden remains a major challenge. Existing secure semantic search systems incur high latency and fail to scale to realistic database sizes.**

We present BiSON , the first secure nearest-neighbor search protocol capable of supporting billion-scale encrypted vector databases. Even at this scale, BiSON answers queries in mere milliseconds and maintains search accuracy comparable to state-of-the-art insecure ANN algorithms, demonstrating that secure semantic search can be both private and truly high-performance. Compared to Compass, the prior state-of-the-art system, BiSON reduces communication up to $28\times$, improves end-to-end latency by up to $23.5\times$, and scales to datasets that are two orders of magnitude larger.

A central contribution of BiSON is a new disk-compatible Oblivious RAM (ORAM) architecture that enables seamless scaling to billion-point datasets without compromising latency or privacy. Together, these innovations make BiSON the first practical and scalable solution for secure semantic search at cloud scale.

***Index Terms*—Semantic Search, Approximate nearest neighbors, Secure Computation, ORAM**

I. Introduction

The rapid adoption of cloud platforms such as Azure [1], AWS [2], and Google Cloud [3] has led organizations and individuals alike to outsource large volumes of data for durability, scalability, and ease of access. Personal cloud-storage systems such as OneDrive [4], iCloud [5], and Google Drive [6] have further popularized this paradigm. While outsourcing eliminates the cost and complexity of managing local storage, it raises two fundamental requirements: *(i) confidentiality*: even a fully compromised cloud provider must not learn information about the stored data or future queries and accesses made to the data, and *(ii) efficiency*: the data owner (or a weak client) must be able to retrieve the required information with minimal latency.

These requirements are increasingly critical in modern AI workflows. Consider an AI agent running locally on a user's device that augments its reasoning using Retrieval-Augmented Generation (RAG) [7]. The user's private document collection is stored encrypted on the cloud, and the agent repeatedly issues *semantic queries* to fetch the relevant records. It is well established that semantic search significantly outperforms lexical search approaches; consequently, contemporary RAG systems rely on semantic retrieval over the data. Now, the latency of these semantic queries directly impacts the responsiveness of the agent. Moreover, the content of the queries may itself reveal sensitive information about the user's intent or private data. Ensuring fast, accurate, and *privacy-preserving* retrieval is therefore essential for practical deployment of such personalized AI systems.

At the scale of organizations, the problem becomes even more challenging. A cloud provider may simultaneously serve semantic search queries for *thousands of clients*, each maintaining large encrypted vector indices. Storing all client indices in memory is infeasible; instead, the server must operate over vector databases that collectively span billions of embeddings. Any practical secure semantic search solution must therefore: (1) provide strong privacy for both data and queries, (2) maintain accuracy competitive with state-of-the-art Approximate Nearest Neighbor (ANN) methods, and (3) support *disk-resident, billion-scale* vector databases.

A. Secure Semantic Search: Limitations

A large body of work, including searchable symmetric encryption (SSE) systems [8]–[18] and privacy-preserving search protocols [19]–[23], studies secure retrieval of encrypted data. However, these solutions either leak query or database information, rely on strong non-collusion assumptions, achieve poor semantic accuracy, or incur cryptographic overheads that make them impractical for low-latency workloads.

Semantic search engines today rely heavily on *approximate nearest-neighbor* (ANN) algorithms such as HNSW [24], DiskANN [25], NSG [26], and SpaNN [27], which map documents and queries to high-dimensional embeddings using models such as BERT [28]. ANN methods achieve state-of-the-art accuracy with millisecond latency and power many production systems including Bing [29] and Elasticsearch [30].

However, adapting ANN search to a *secure* setting remains a central open challenge. The recent system Compass [31] provides the strongest known privacy guarantees by combining HNSW with cryptographic protection layers. Compass achieves high accuracy on standard benchmarks such as SIFT,

†Work done while at Microsoft Research India

LAION, TripClick and MS MARCO, but it inherits a crucial limitation from HNSW: *HNSW is an in-memory algorithm and does not scale to billion-point datasets.* Under secure deployment, this limitation becomes prohibitive. Because Compass must repeatedly retrieve node embeddings and adjacency lists from a remote server, its in-memory design results in high bandwidth costs, large round-trip latencies, and ultimately, scalability bottlenecks. Our evaluation confirms that secure protocols built atop in-memory ANN structures fail to support vector databases at billion scale. These observations lead to the natural question:

Can we build secure semantic search that achieves strong privacy, high search accuracy, and low latency while scaling to billion-point encrypted databases?

B. Our Contributions

In this work, we answer the above question in the affirmative. We introduce BiSON , the first secure semantic search protocol that matches the accuracy of state-of-the-art ANN systems while scaling to *billion-point, disk-resident* encrypted databases. At a high level, cleartext ANN algorithms operate by traversing a graph index constructed over the datapoints to locate the nearest neighbors of a query. In the secure setting, where the index resides on the server, the client retrieves nodes from the graph iteratively, decrypts them locally, and computes their distances to the query. To preserve data and query privacy, however, the sequence of nodes accessed during the traversal must remain hidden from the server. In other words, the server must not learn which nodes in the graph were touched during the search. This requirement is achieved through the use of Oblivious RAM (ORAM) [32], which ensures that any two access sequences of the same length are computationally indistinguishable. As a result, ORAM becomes the fundamental tool that enables privacy-preserving graph traversal in secure ANN systems.

In our work, we build upon DiskANN [25] and a new disk-compatible ORAM architecture, BiSON and advance the state-of-the-art in secure semantic search along several dimensions:

- **A scalable and accurate secure semantic search protocol.** We design the first secure semantic search protocol that operates efficiently over billion-scale encrypted vector databases. Our protocol builds on the DiskANN search algorithm, the leading cleartext method for billion-point ANN search, and integrates it with Ring ORAM [33] to preserve both data and query privacy while maintaining accuracy competitive with cleartext ANN baselines.
- **Motivate the use of DiskANN.** We highlight the inherent scalability limitations of in-memory ANN structures in secure settings (such as the work of Compass [31]). By contrasting HNSW with DiskANN, we justify our choice of DiskANN as the cleartext foundation for secure, disk-resident, large-scale vector search.
- **A disk-compatible Ring ORAM design.** We develop a Ring ORAM implementation optimized for SSD-backed storage at the server. This enables efficient retrieval of ORAM buckets directly from disk and overcomes the memory bottlenecks of prior in-memory ORAM implementations, allowing BiSON to scale seamlessly to billion-point datasets.
- **Implementation and large-scale evaluation.** We implement the full BiSON protocol[1] and evaluate its accuracy, communication, and latency. On benchmarks used by Compass, BiSON reduces user-perceived (or online) communication by $2.8-18\times$, and user-perceived latency by 2.0–$8.5\times$ in varied network settings. Even when considering end-to-end communication and latency, BiSON outperforms prior work by $4-28\times$ and 2.5–$23.5\times$ respectively. We demonstrate BiSON 's scalability by evaluating on four additional datasets containing 10M, 50M, 100M, and 1B vectors, supporting databases that are up to $113\times$ larger than those handled by prior secure systems.

Taken together, these contributions establish BiSON as the first practical, accurate, and scalable solution for secure semantic search at cloud scale. While our protocol's current implementation only supports semi-honest servers, it can be easily extended to a malicious server model using standard authentication techniques (similar to Compass) incurring only minimal additional overhead.

C. Other Related Work

Encrypted semantic search over private databases can be broadly categorized into three settings:

1) a client queries a **public database** hosted by a server;
2) a client queries a **server-owned private database**;
3) a client outsources **its own private database** to an untrusted server and later queries it.

The privacy goals differ across these settings. In the first, only the client's query must remain hidden. In the second, both the query (from the server) and the database (from the client) must be protected. In the third, both the query and the database must remain hidden from the server. BiSON operates in this third setting. We now review prior work across these settings.

In the first setting, Tiptoe [21] proposes a protocol based on private information retrieval (PIR), inheriting its substantial computational and communication overheads. To mitigate these costs, Tiptoe employs clustering; however, this leads to a noticeable drop in accuracy compared to state-of-the-art cleartext semantic search. More recently, Zhou *et al.*'s Pacmann [22] improves efficiency using preprocessing-based PIR [34], [35]. Pacmann reduces online communication to $O(\sqrt{n}\log n)$, but requires a preprocessing phase with linear communication. Since these works assume the database is public, they are not applicable to our setting, where the database itself must remain private from the server.

In the second setting, SANNS [23] achieves strong privacy guarantees but incurs high computational overhead due to its

[1]Code available at: https://github.com/r-dott/bison

reliance on heavyweight cryptographic primitives, including distributed ORAM, garbled circuits (GC), and lattice-based techniques. Panther [36] reduces communication costs relative to SANNS, but still relies on a combination of PIR, homomorphic encryption, and GC, and thus inherits similar computational bottlenecks. Preco [20] also operates in this setting. It uses lightweight techniques such as batch PIR and locality-sensitive hashing, but additionally assumes two non-colluding servers to provide query privacy. Furthermore, it leaks limited information about the database to the client with each query and therefore does not guarantee full database privacy.

The closest work to BiSON is Compass by Zhu *et al.* [31], which operates in the third setting. Compass achieves strong query privacy (no leakage) for approximate nearest neighbor (ANN) search over encrypted, outsourced databases. However, it does not scale beyond a few million records, limiting its applicability to large, cloud-scale deployments. In contrast, BiSON addresses this limitation by leveraging a more scalable ANN index (DiskANN [25]) along with a novel disk-resident ORAM design.

D. Organization

The remainder of the paper is organized as follows. Section II introduces notation and reviews the necessary technical background. Section III presents a high-level overview of our protocol and outlines its key differences from Compass. Section IV provides a detailed description of BiSON along with a sketch of its privacy proof. Section V describes our implementation and experimental evaluation, comparing the performance of BiSON against Compass. We provide our concluding thoughts in Section VI.

II. Preliminaries

We set notation, present background relevant to approximate nearest-neighbor search as well as describe the various cryptographic primitives we make use of.

A. Notation

The set of natural numbers $\{1, \ldots, n\}$ is denoted by $[n]$. Vectors are typeset as bold lowercase letters (eg. $\mathbf{x}$). $G = (V, E)$ denotes an undirected graph with a set of vertices/nodes V and edges E. The set of neighbors of a node $x \in V$ is represented by $\mathsf{nbrs}(x) \subset V$. $\mathsf{Enc}_k(x)$ denotes the encryption of x under a private-key cryptosystem with the key k. $||$ denotes the concatenation operation.

B. Approximate Nearest-Neighbor Search

Consider a database $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of datapoints (e.g., documents, images) embedded in a high-dimensional vector space equipped with a distance metric δ (say, ℓ_2). A *nearest-neighbor search* for a query vector $\mathbf{q}$ returns the datapoint closest to $\mathbf{q}$ under δ, i.e.,

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{D}} \delta(\mathbf{q}, \mathbf{x}).$$

This problem naturally generalizes to *k-nearest neighbors search* (k-NNS), which retrieves the k closest datapoints to the query. For the large databases common in modern information-retrieval systems, exact k-NNS becomes computationally prohibitive. This motivates the use of *approximate k-nearest neighbors search* (ANN), which significantly accelerates retrieval by allowing a small deviation in the ordering of the returned results. The quality of an ANN method on a set of queries Q is typically measured using metrics such as recall and mean reciprocal rank (MRR).

ANN algorithms [24]–[26], [37]–[42] achieve high throughput by constructing an *index*, a data structure that organizes datapoints according to their geometric relationships in the embedding space. The index guides the algorithm to examine only datapoints that are likely to be close to the query, thereby avoiding a full scan of the database. A prominent class of indices is based on *proximity graphs*, where each datapoint corresponds to a node and edges connect vectors that are close in the embedding space [24]–[27]. Each node stores its vector embedding and the adjacency list of its neighbors. During search, the algorithm begins at a chosen entry point and traverses the graph through a series of hops, progressively moving toward nodes whose embeddings are closer to the query. The index and traversal strategy together determine the tradeoff between search latency and accuracy. Well-known examples include HNSW [24] and NSG [26]; however, these methods typically scale only to databases of a few million datapoints, as the graph index quickly grows to be too large to fit in memory.

Algorithm 1 DiskANN Search [25]

Require: Graph G with start node s, query x_q, result size k, search list size $L \geq k$
Ensure: Result set $\mathcal{L}$ containing k-approximate nearest neighbors, and set $\mathcal{V}$ of visited nodes

```
Initialize sets L ← {s} and V ← ∅
while L \ V ≠ ∅ do
    Let p* ← arg min_{p∈L\V} ||x_p − x_q||
    Update L ← L ∪ nbrs(p*) and V ← V ∪ {p*}
    if |L| > L then
        Update L to retain closest L points to x_q
    end if
end while
return Closest k points from L and V
```

C. DiskANN

DiskANN [25] is a graph-based ANN search algorithm designed to scale to datasets containing billions of datapoints. Its scalability stems from storing the graph index on disk rather than in memory. We briefly describe the DiskANN search procedure here. At a high level, the algorithm performs a graph traversal in which only the information associated with the currently explored node is loaded into memory. Formally, the index is a connected graph $G = (V, E)$ with $|V| = n$ and maximum degree R.

Algorithm 1 outlines the DiskANN search procedure. A priority queue $\mathcal{L}$ maintains nodes ordered by their distance

to the query; it is initialized with only the start node. A set $\mathcal{V}$ tracks nodes that have already been explored. The compressed embedding for every datapoint in the database as well as for the query are stored in memory; these compressed vectors are considerably smaller than the full embeddings stored on disk.

In each iteration, the algorithm dequeues from $\mathcal{L}$ the node p^* closest to the query. It then reads from disk the full embedding of p^* along with the identifiers of its neighbors. Using the compressed embeddings, the algorithm computes the distances from the query to each neighbor of p^* and inserts these neighbors into $\mathcal{L}$. The node p^* is added to the explored set $\mathcal{V}$ and is never revisited. The priority queue $\mathcal{L}$ is then pruned to retain only the L nodes closest to the query. This process continues until all nodes in $\mathcal{L}$ have been explored. At this point, the complete embeddings of all nodes remaining in $\mathcal{L}$ have been fetched from disk, and the k closest among them (under the full embeddings) are returned as the search result.

While the index is constructed using full embeddings, distance evaluations during search rely on compressed embeddings to ensure that all necessary embeddings fit in memory. Full embeddings are used only at the end to rank the final candidates, as compressed embeddings are lossy and can otherwise degrade search accuracy.

DiskANN attains high search throughput by carefully balancing the graph degree and diameter. Intuitively, this design ensures that the nearest node to the query can be reached from the start node within $O(\log n)$ hops. Reducing the number of hops is critical for performance because each hop requires reading a node from disk, incurring the associated read latency. DiskANN further improves efficiency through parallel disk reads: instead of dequeuing a single node per iteration, it retrieves the closest W nodes from $\mathcal{L}$, thereby reducing the number of disk round-trips per explored node. The parameter W, known as the *beamwidth*, is a tunable hyperparameter. As we will see in the context of our secure search protocol, the number of search iterations corresponds to the number of rounds of interaction between the client and server, while the product of the number of iterations and the beamwidth determines the communication cost. Thus, high search performance requires striking an appropriate balance between the number of iterations and the beamwidth.

D. Oblivious RAM and Ring ORAM

Oblivious RAM (ORAM) [32], [43] enables a client to access elements stored in encrypted remote memory without revealing its access patterns to the server. Thus, the server remains oblivious to which memory locations the client reads or writes. A trivial way to achieve this is for the client to read or write every block on each access, but this incurs linear overhead in the RAM size. A line of work [33], [44] has since developed ORAM constructions with logarithmic communication overhead while maintaining practical client-side storage requirements.

Ren *et al.* [33] proposed a tree-based ORAM protocol, *Ring ORAM*, which achieves online bandwidth independent of the RAM size N while requiring only $O(\log N)$ blocks of client storage. Consequently, Ring ORAM is well-suited for ANN search over large databases. We briefly review its design. The server stores N data blocks, each of size B, organized as a binary tree. Each tree node (called a *bucket*) holds Z data blocks and S dummy blocks. Each leaf $\ell \in [L]$ corresponds to a unique path from the root to that leaf, where L is the number of leaves. We use the terms *leaf* and *path* interchangeably. Every data block is conceptually mapped to a particular root-to-leaf path. The client maintains this mapping in a *position map*, which it updates during the protocol to conceal access patterns from the server. The client also stores a bounded-size *stash* that temporarily caches recently accessed blocks. The Ring ORAM invariant guarantees that every data block is always stored either in one of the server buckets or in the client stash. The client further tracks metadata for each block to support online reads.

Ring ORAM implements read and write operations using three abstract procedures: ReadPath , Evict , and EarlyReshuffle . The procedure ReadPath (ℓ, b) takes as input a path identifier ℓ and a block identifier b, and reads exactly one block from each bucket along the path. The block read from a bucket may be either a (real) data block or a dummy block; to hide the distinction between the two types, Ring ORAM enforces two conditions: (1) the $(Z + S)$ blocks in each bucket are stored in a random order, and (2) no block is read more than once (after being read, the block is invalidated). Reading all $\log L$ blocks from the path would incur $O(\log N)$ online bandwidth, so Ring ORAM reduces this to constant overhead by XOR-ing all blocks read from the path and returning only the XOR result to the client. Because data and dummy blocks are suitably encrypted, the client can always recover the single data block of interest from this XOR-ed value. If the access is a write, the client locally updates the block's contents. As part of ReadPath , the client also remaps the accessed block to a fresh, randomly selected leaf, updating the position map accordingly.

Blocks retrieved through ReadPath are placed temporarily in the stash, which is periodically flushed by evicting blocks to a designated path via the Evict procedure. This process also writes back any blocks modified during earlier write operations. The EarlyReshuffle procedure replenishes a bucket with $(Z + S)$ fresh data and dummy blocks after it has been read S times, ensuring that the bucket never exhausts its supply of valid dummy blocks. Both read and write operations use the same three abstract procedures; a read is simply a write in which the block contents remain unchanged. We refer the reader to [33] for further details.

III. Technical Overview

In this section, we formalize our secure search problem and provide a high-level overview of our protocol, BiSON .

A. Problem Formalization

We consider a client $\mathcal{C}$ and a server $\mathcal{S}$. The client holds a private document database

$$\mathsf{DocDB} = \{d_1, \ldots, d_n\},$$

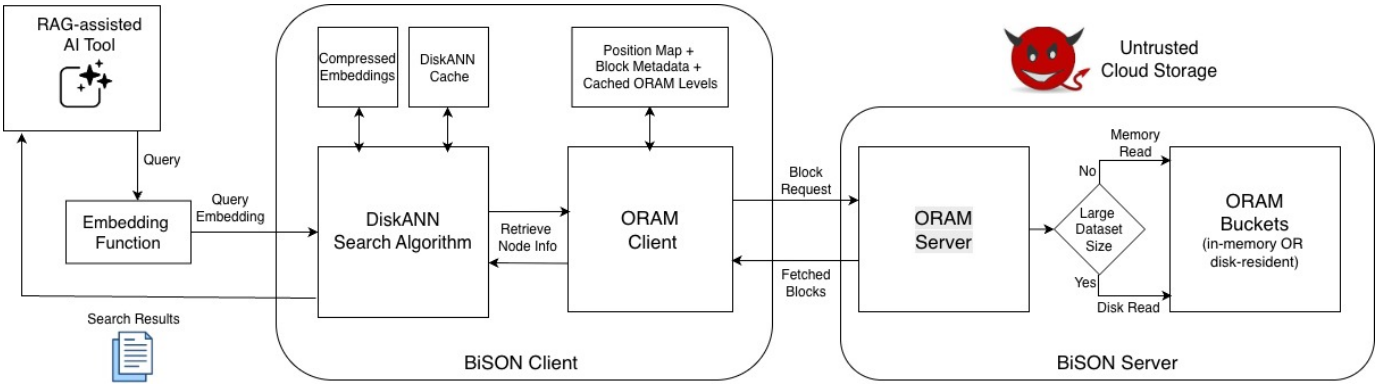

Fig. 1. BiSON System Architecture and Query Pipeline

which it encrypts and uploads to the server during a setup phase. During the online query phase, given a query query, the client wishes to retrieve from the server the k documents in DocDB that are semantically closest to query. Our secure search system aims to satisfy the following goals:

- **Accuracy:** The retrieved documents should maximize search accuracy under a chosen metric (e.g., MRR, Recall).
- **Privacy:** After the protocol, the server should learn no additional information about either the client's data DocDB or query beyond what it already may know.
- **Efficiency:** The system should return the desired documents with minimal end-to-end latency.

We note that other privacy considerations—such as hiding the client's query frequency or timing, or concealing metadata about the database (e.g., its size or number of documents)—are outside the scope of this work.

Threat Model. We assume that the server is an *honest-but-curious* (semi-honest) party: it follows the protocol correctly but may attempt to infer additional information from the messages it receives. As in prior work, our techniques can be extended to achieve security against a fully malicious server with negligible additional overhead. The server may maintain state across interactions. Finally, we assume that communication occurs over authenticated and encrypted channels, and that adversaries cannot corrupt or tamper with transmitted data.

B. Protocol Overview

Our secure search protocol, BiSON is built on top of the DiskANN algorithm. The client first fixes a high-dimensional embedding space and an associated distance metric. Using an embedding function, it maps each document in the database to a vector in this space. The client then constructs an index $G = (V, E)$ over these vectors using DiskANN's index-construction algorithm, *Vamana*. Each document in the database corresponds to a unique node in V, identified by its document identifier id $\in [n]$. Henceforth, we use the terms *document* and *node* interchangeably. The client also selects a designated start node in the index.

Next, the client and server jointly initialize an instance of Ring ORAM, with the client acting as the ORAM client and the server as the ORAM server. For each node of G, the client encrypts and stores in a separate ORAM data block the node identifier, its full embedding, and the identifiers of its neighbors. The client sends all ORAM buckets (containing both real and dummy blocks) to the server, while locally maintaining the position map, block metadata[2], and a stash. The client also stores all compressed embeddings in memory; these are assumed to be sufficiently small in comparison to the full embeddings. At the end of this one-time *Setup* phase, the client no longer retains the documents or the index structure, both of which now reside entirely on the server.

We now describe the *online* phase of BiSON during which the client issues a query. BiSON 's architecture and a typical query pipeline is described in Figure 1. The end-application (say, a RAG agent) embeds the query into the above high-dimensional space and computes its compressed embedding. This embedding is then passed to BiSON 's client, which retrieves the requested documents. Recall that the DiskANN search algorithm proceeds iteratively by dequeuing the W (beamwidth) closest nodes to the query, based on compressed embeddings, from the priority queue and exploring their neighbors. In our protocol, this corresponds to the ORAM client issuing a read request for the data blocks associated with these dequeued nodes. The ORAM server executes Ring ORAM's ReadPath operation and returns the corresponding encrypted data blocks. Note that, depending on the dataset size, BiSON 's server decides whether to use an in-memory ORAM server or a disk-assisted ORAM server. On receiving the requested blocks from the server, the ORAM client decrypts them and passes the node information to the DiskANN search algorithm. DiskANN's search algorithm extracts the neighbor identifiers, inserts the neighbors into the priority queue, and proceeds to the subsequent search iteration. At the end of the search, the retrieved documents are returned to the end-application. Implicitly, the ORAM client also performs the necessary Ring ORAM bookkeeping operations, such as updating the position

[2]Caching the block metadata at the client saves an extra round trip during an ORAM read.

map and reshuffling buckets, at the end of each iteration.

A key technical novelty of our work is the design of a disk-resident ORAM that allows BiSON to scale to billion-record databases. A common bottleneck in existing ORAM protocols, including RingORAM, is large server-side memory requirement to hold encrypted ORAM data blocks. The absolute size of ORAM buckets for large datasets exceeds typical server memory capacity. For example, supporting search on the SIFT1B [45] requires the ORAM server to reserve ≈ 3.3 TB RAM, far exceeding the memory available on standard machines. Existing works, including Compass, assume all ORAM buckets reside in-memory at the server, thus thwarting them from scaling to cloud-scale databases typical of RAG-pipelines.

We leverage two key principles in our design. First, we replace sequentially reading buckets (proportional in number to the height of the ORAM tree) in each ORAM read-request with a batched read of all required buckets. Since the server knows the specific buckets lying on the requested path in the tree, it can issue parallel disk-reads for all these buckets in a batch. This greatly reduces I/O latency as compared to sequential reads; the latter requires the server to wait until the current bucket is completely fetched from the disk before issuing a read for the next bucket on the path. Secondly, all disk reads and writes are sector-aligned which ensures that each access cleanly spans a complete physical sector of the disk. This obviates the additional overhead of handling partial-sector reads due to costly read-modify-write cycles. All in all, our disk-resident ORAM allows the server to store ORAM buckets on an SSD storage while minimising the I/O overhead for fetching the buckets *on-demand*.

We highlight another benefit of our disk-resident ORAM. Practical deployments, including but not limited to RAG applications, involve a multi-client setting served by a single server. Such settings require the server to separately store encrypted databases for each client, thus aggravating the memory requirement for handling concurrent queries from multiple clients. Clearly, relying on an in-memory ORAM server is infeasible for deployments typical of practical usecases and thus warrants a similar disk-resident ORAM design sketched above. While we do not implement concurrent queries in our current implementation, we leave this open for future work.

Finally, an important distinction from cleartext DiskANN is that the number of search iterations in BiSON must be fixed to a constant T to ensure query privacy. Otherwise, the server might infer information about the query from the number of iterations taken by the DiskANN algorithm. Thus, if the cleartext DiskANN search would terminate in $S < T$ iterations (i.e., all nodes in $\mathcal{L}$ have already been explored), the client issues $(T-S)$ additional *fake* ORAM reads. By ORAM security, the server cannot distinguish fake reads from genuine ones.

C. Performance Metrics

In each search iteration of BiSON , the client and server engage in one round of interaction, during which W data blocks are communicated. Thus, the number of search iterations T determines the *round complexity*, while the total communication cost is $T \cdot W$ blocks.

We define the *user-perceived latency* of a query as the elapsed wall-clock time from the start of the online phase until the search results are returned to the user. After returning the results, the client and server execute Ring ORAM's Evict operation as a cleanup step to prepare for future queries. This technique of delayed evictions introduced in Compass [31], reduces the user-perceived latency by not performing any evictions during the search iterations and only performing an eviction after the query results are returned.

We define the sum of the user-perceived latency and the time required to complete this Evict operation as the *end-to-end latency* of the online phase. While minimizing the user-perceived latency is the primary efficiency objective of BiSON , we show that we also obtain much better end-to-end latency than prior work. In Section V, we compare these performance metrics with those of Compass.

D. Comparison with Compass

Our choice of DiskANN as the underlying cleartext algorithm for BiSON is motivated by several efficiency benefits that DiskANN offers over HNSW based Compass [31]. In this section, we compare DiskANN with Compass and its cleartext algorithm HNSW, thereby motivating the design decisions behind our protocol.

HNSW is an ANN search algorithm designed explicitly for *in-memory* search. It assumes that all information associated with every node, including the full embedding and the identifiers of its neighbors, fits entirely in memory. In each search iteration, HNSW computes the distance from the query to *all* neighbors of the node currently closest to the query. Under the in-memory assumption, these distances can be computed directly using full embeddings.

However, this assumption does not hold for Compass, where each node's complete embedding and neighbor information reside on a remote server. Although Compass builds upon HNSW's search logic, computing distances to all neighbors would require fetching every neighbor's full embedding from remote storage. Doing so verbatim would impose prohibitive online bandwidth costs. To mitigate this, Compass retrieves the complete embeddings of only efn neighbors (where $efn <$ degree) of the closest node - an optimization termed *directional neighbor filtering*. Additionally, Compass parallelizes the fetches for $efspec$ closest nodes, similar to DiskANN's beam search. Thus, in each iteration, Compass fetches $efn \cdot efspec$ nodes from remote storage (instead of degree $\cdot\, efspec$).

A critical difference arises in how neighbors are prioritized. Compass (like HNSW) orders neighbors using distances computed from the *full* embeddings. DiskANN, and consequently BiSON , depart from this design: neighbors are ordered solely using *compressed* embeddings. As a result, DiskANN does not need to fetch full embeddings of all neighbors during search. Instead, in each iteration, it fetches only the complete

embeddings of the closest W nodes (where W is semantically equivalent to $efspec$) along with their neighbor identifiers. Thus, unlike Compass, BiSON avoids the multiplicative efn factor in bandwidth. Full embeddings are used only once, at the end of the search, to rank the final candidate set. This is feasible because the complete embeddings of all explored nodes have already been fetched in earlier iterations. As shown in Section V, DiskANN's significantly lower online bandwidth leads to reduced user-perceived latency for BiSON compared to Compass.

DiskANN and BiSON also benefit from requiring fewer round-trips to disk. Each search iteration fetches node information from storage, so the number of iterations directly determines the number of round-trips and therefore the round-trip latency. Our experiments show that BiSON achieves search accuracy comparable to Compass($\geq 90\%$ Recall@10), but with *fewer* iterations. This improvement stems from the shallow structure of Vamana-built graphs, whose diameters are significantly smaller than those of HNSW graphs. Consequently, starting from the designated entry point, DiskANN traverses fewer nodes before reaching those closest to the query. For BiSON , the reduced number of server round-trips translates directly to lower user-perceived latency.

Finally, similar to how Compass caches the initial HNSW layers of the graph, BiSON also caches the first few BFS (Breadth First Search) levels from the entry point of the graph in client memory. Thus, the initial 2 - 3 search iterations proceed entirely on the client side without requiring any round-trips to the server. Reducing the number of server accesses greatly improves user-perceived latency, especially in slower network environments. We emphasize that this local cache is distinct from Ring ORAM's stash; the two serve completely different purposes and arise from independent design requirements.

Taken together, these considerations justify our choice of DiskANN as the cleartext foundation for BiSON . Both the theoretical differences between DiskANN and Compass (HNSW) and our experimental results demonstrate that building BiSON on top of DiskANN is a deliberate and effective design decision aimed at maximizing search efficiency.

IV. Secure Semantic Search with BiSON

We now describe our secure semantic search protocol, BiSON in detail. We present the asymptotic costs of our protocol and establish that our protocol preserves privacy as specified in Section III-A.

A. Protocol

Our protocol proceeds in two phases: a one-time query-independent Setup phase and an online Query phase. We elaborate on both phases below.

1) Phase I: Setup: Setup is a one-time protocol executed between the client and the server before the queries are known. The client fixes a high-dimensional embedding space $\mathcal{E} \subset \mathbb{R}^m$ and a distance metric $\delta : \mathcal{E} \times \mathcal{E} \to \mathbb{R}$. It then maps each document $d \in \mathsf{DocDB}$ to a vector $\mathbf{x} \in \mathcal{E}$ using a fixed embedding function $f : \mathsf{Dom}(\mathsf{DocDB}) \to \mathcal{E}$. Denote $\mathcal{D} = \{\mathbf{x} = f(d)\}_{d \in \mathsf{DocDB}}$. We assume that f preserves the semantic similarity between the documents.

The client then constructs an index G on top of $\mathcal{D}$ using DiskANN's index-building algorithm called Vamana. The client caches nodes in the first c layers spreading out from the start node in G in $\mathsf{cachedNodes}$, indexed by node ids. In addition, it computes a compressed embedding $PQ(\mathbf{x})$ for each $\mathbf{x} \in \mathcal{D}$ using Product Quantization [46]. Denote $\mathsf{CompEmb}(\mathcal{D}) = \{PQ(\mathbf{x})\}_{\mathbf{x} \in \mathcal{D}}$. For each document $d_i \in \mathsf{DocDB}$, the client constructs $\mathsf{ct}_i = \mathsf{Enc}_k(i||\mathbf{x}_i||\mathsf{nbrs}(i))$ and instantiates Ring ORAM with the encrypted data blocks $\{\mathsf{ct}_i\}_{i \in [|\mathcal{D}|]}$. We remark that the bulk of the data blocks are occupied by the vector embeddings for each document. Observe that all the steps so far are done locally at the client's end.

Finally, the client communicates the ORAM buckets to the server, which concludes the setup phase. At the end of the setup phase, the client holds in memory cached nodes $\mathsf{cachedNodes}$, $\mathsf{CompEmb}(\mathcal{D})$, ORAM stash, position map and blocks' metadata.

2) Phase II: Query: For each query, the client and the server engage in the online query phase of our search protocol. Since we already gave an overview of this phase in Section III-B, we present only the formal specification of the query phase here in terms of pseudocode.

Algorithm 2 describes the query phase of our protocol in terms of steps executed at the client's end. The client $\mathcal{C}$ executes T search iterations of the DiskANN search algorithm. The choice of T differs from dataset to dataset. In lines 3-7, the client issues fake read requests to the server when the search algorithm has already converged on the k nearest neighbors for the query. This is necessary for ensuring query privacy as the total number of read requests issued to the server must be fixed across queries, which in our case is the number of search iterations T.

In lines 14-18, the data blocks corresponding to the nearest W nodes to the query are fetched. If the nodes are cached by the client, the corresponding datablocks are fetched from $\mathsf{cachedNodes}$. Otherwise, they are fetched from the ORAM server (analogous to a disk read in cleartext DiskANN). For the sake of readability, all operations related to $\mathsf{ReadPath}$ (as discussed in Section II-D) are abstracted as a function $\mathsf{ORAM.FetchBlocks}$ called by the client, which outputs the data blocks corresponding to each of the requested nodes. As in the cleartext DiskANN search, the priority queue $\mathcal{L}$ is ordered by distance of the nodes to the query vector, computed using the compressed embeddings. This approximated distance on the compressed embeddings is computed using an abstract function $PQ.dist$ which takes in two compressed embeddings as input.

We note here that after returning the search results to the user (line 30), the client and the server execute the Evict protocol of Ring ORAM. Compass introduces this technique, termed as *multi-hop lazy evictions* [31], where no evictions are performed during search iterations(hops), and are executed

Algorithm 2 BiSON Query Phase

Require: Client $\mathcal{C}$ holds start node s of index G, compressed embeddings $\mathsf{CompEmb}(\mathcal{D})$, query embedding $\mathbf{q}$, query compressed embedding $PQ(\mathbf{q})$, beam width W, result size k, search list size $L \geq k$, search iterations T

Ensure: Client outputs k-approximate nearest neighbors from $\mathcal{L}$, and set $\mathcal{V}$ of visited nodes

$\mathcal{L} \leftarrow \{s\}$, $\mathcal{V} \leftarrow \emptyset$
for $t \in [T]$ **do**
  if $\mathcal{L} \setminus \mathcal{V} = \emptyset$ **then**
    $E \leftarrow \{p_i \xleftarrow{\$} [n]\}_{i \in [W]}$
    $\{\mathsf{block}_p\}_{p \in E} \leftarrow \mathsf{ORAM.FetchBlocks}(E)$
    continue
  end if
  $E \leftarrow \Phi$
  for $i \in [W]$ **do**
    $p^* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} d(\mathbf{x}_p, \mathbf{q})$
    $E \leftarrow E \cup \{p^*\}$
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
  end for
  if $E \subseteq \mathsf{cachedNodes}$ **then**
    $\{\mathsf{block}_p\}_{p \in E} \leftarrow \mathsf{cachedNodes}[E]$
  else
    $\{\mathsf{block}_p\}_{p \in E} \leftarrow \mathsf{ORAM.FetchBlocks}(E)$
  end if
  for $p \in E$ **do**
    $(p \parallel \mathbf{x}_p \parallel \mathsf{nbrs}(p)) \leftarrow \mathsf{block}_p$
    for $r \in \mathsf{nbrs}(p)$ **do**
      $pqdist_r \leftarrow PQ.dist(PQ(\mathbf{x}_r), PQ(\mathbf{q}))$
      $\mathcal{L} \leftarrow \mathcal{L} \cup \{r\}$ based on $pqdist_r$
    end for
  end for
  if $|\mathcal{L}| > L$ **then**
    Update $\mathcal{L}$ to retain closest L points to $\mathbf{q}$
  end if
end for
return Closest k points from $\mathcal{L}$ and $\mathcal{V}$

only after the query response is returned. This is advantageous as the time taken for eviction does not affect the user's perceived latency since it has already retrieved the results. Similarly, the bandwidth consumed during the eviction step does not impact the bandwidth usage of the search, since all eviction related communication occurs only after the search terminates.

B. Disk-compatible ORAM

We presented the motivation and benefits of using our disk-resident ORAM in Section III-B. Here, we elaborate further on its design.

Our disk-compatible Ring ORAM design functions by allowing the server to store the ORAM buckets on an SSD storage device, without incurring huge I/O overhead when compared against in-memory buckets. To achieve disk-efficient access, our implementation is designed with two key principles in mind: leveraging asynchronous batched I/O to exploit SSD parallelism, and performing sector-aligned accesses to minimize SSD controller overhead.

We now detail how the use of **asynchronous batched I/O** significantly improves the efficiency of disk accesses at the ORAM server. During each search iteration(round-trip) for a given query, the client requests W(beamwidth) nodes (or blocks) from the ORAM server. To return a single block to the client, the Ring ORAM server has to read one block from each of the buckets on the path from the root bucket to the leaf. Thus, this translates fetching W blocks into $W \times H$ disk reads at the server, where H is the height of the ORAM tree. If the client enables tree-top caching and caches the top t levels of the ORAM locally, fetching W blocks incurs $W \times (H - t)$ disk reads at the server. A naive implementation would be to perform blocking disk reads sequentially for each of the $W \times (H - t)$ disk locations, stalling the server until each individual read completes. This serialization does not utilize SSD parallelism and incurs huge I/O latency.

However, since the server can precompute all locations on the disk to read from, before issuing *any* I/O operations, we batch these reads using the Linux `liburing` library [47]. Rather than waiting for individual reads to complete, `liburing` enables submission of all read requests as a single batch to the kernel's I/O submission queue. The kernel then schedules these operations in parallel, allowing the SSD to process multiple reads concurrently. The server subsequently collects data blocks from all completed reads once the batch finishes.

To validate the effectiveness of our implementation, we measured the online server time per query on the SIFT100M dataset using values $W = 8$, $H = 22$, $t = 6$ (giving $W \times (H - t) = 128$ disk locations), for 6 search round trips. Since the majority component of online server time is the cost of performing disk I/O($\approx 92\%$) in the latter two settings, these measurements directly capture the overhead introduced by disk accesses. We evaluate across three different server configurations: (i) in-memory ORAM, where all the data blocks reside in the server's memory, (ii) a naive implementation with sequential disk accesses, where the server stores ORAM buckets on disk and performs I/O operations sequentially using blocking reads/writes, and (iii) our asynchronous batched I/O implementation described above.

The in-memory ORAM server takes 0.3 ms per query. The naive implementation of a disk-equipped ORAM server that issues sequential reads requires approximately 110 ms per query. In contrast, our batched-read implementation lowers this time to roughly 10 ms. Therefore, by exploiting SSD parallelism, we achieve $\approx 11\times$ speedup, substantially narrowing the gap between disk-based ORAM and the in-memory implementation of an ORAM server.

To maximize SSD efficiency, all disk reads and writes rely on **sector-aligned accesses**. Aligning our I/O operations to SSD sector boundaries means each request cleanly spans whole physical sectors, avoiding any need for the device to

perform partial-sector handling. In contrast, unaligned writes force the SSD into a costly read-modify-write(RMW) cycle, where it must read the existing sector, modify the intended portion with the new data, and rewrite the whole sector. This causes extra latency and internal overhead. Consequently, by issuing sector-aligned I/O, we reduce the overhead during ordinary reads and significantly improve performance for write-heavy steps like eviction and reshuffling. Sector aligned I/O is also used in disk-based cleartext search like DiskANN [25].

C. Privacy

We now argue that our secure semantic search protocol achieves query privacy and database privacy.

Query privacy during the setup phase is easy to see, since the only information that the server knows at this point is the encrypted ORAM blocks that are built independent of the queries. Therefore, the setup phase of our protocol does not reveal any information about the subsequent queries to the server. At the end of the setup phase, only encrypted data blocks are stored on the server; therefore, database privacy is preserved during setup.

Throughout the online phase, the server only sees the memory locations that it needs to read in order to return the requested data blocks. From standard ORAM security, these memory locations look random to the server and hence it does not learn anything about the client's requested data blocks. Formally, the execution traces of the ORAM server on two distinct memory access sequences is computationally indistinguishable to an adversary that corrupts the server. Hence, the read requests made by the client in each iteration do not reveal which nodes are accessed in the course of the search algorithm, thereby guaranteeing privacy of the client's query. Moreover, since all data is only decrypted at the client's end, the server only reads encrypted data blocks throughout the online phase. From security of standard symmetric-key cryptography, database privacy is preserved during the online phase as well. Taken together, the above arguments prove privacy of our end-to-end secure semantic search protocol.

V. Evaluation

A. Implementation and Experimental Setup

We implemented BiSON in C++ on top of the DiskANN codebase [48]. We integrate DiskANN search with the Ring ORAM implementation in Compass's open-source repository [49] to obtain our protocol's implementation. We confirmed functional equivalence between BiSON and cleartext DiskANN search by verifying equality of the sequences of index nodes traversed for each query in the two cases. Our code leverages DiskANN's support for *single-instruction multiple data* (SIMD) instructions and uses 4 threads for ORAM execution. Besides code for the search protocol, we implemented Python scripts to search for optimal search hyperparameters.

We run BiSON between two Azure machines running Ubuntu 24.04.3 connected via ethernet. We instantiate our SSD storage using Linux LVM (Logical Volume Manager) by combining two local NVMe SSDs into a single 3.6 TB logical volume. We benchmark BiSON 's performance in the LAN setting by using the available network bandwidth of 10 Gbps and ping latency (RTT) of 0.3 ms. The WAN setting is emulated by configuring the network bandwidth as 400 Mbps and RTT as 80 ms using Linux's `tc` tool.

TABLE I
Dataset Specifications

Dataset	#docs	dim.	#queries
LAION	100,000	512	1,000
SIFT1M	1,000,000	128	10,000
TripClick	1,523,871	768	1,175
MS MARCO	8,841,823	768	6,980
SIFT10M	10,000,000	128	10,000
SIFT50M	48,863,636	128	10,000
SIFT100M	100,000,000	128	10,000
SIFT1B	1,000,000,000	128	10,000

B. Datasets and Baseline

We benchmark BiSON against Compass on 4 datasets, namely LAION [50], SIFT1M [45], TripClick [51] and MS MARCO [52]. In addition, we also test BiSON 's accuracy and performance on 4 larger datasets, namely SIFT10M, SIFT50M, SIFT100M and SIFT1B (also known as *BigANN* dataset); these datasets were not covered by Compass in their experiments. Details of each dataset are tabulated in Table I.

Datasets LAION, TripClick and MS MARCO were downloaded using Compass's repository. Except for SIFT50M, the SIFTX datasets were procured by extracting the first X datapoints from the SIFT BigANN dataset [45], respectively, where X represents the number of datapoints in the dataset. SIFT50M, was procured by extracting each datapoint from SIFT BigANN dataset with a probability 0.05. We remark that these datasets already existed in their vector embedding form and we did not explicitly run an embedding function.

Our baseline is Compass's open-source implementation [49]. We run their implementation on the same experimental setup as ours and report accuracy and performance metrics for comparison.

C. Search Hyperparameters

We build Vamana indices for all datasets with graph degree $R = 128$, candidate list size $L = 125$, and pruning parameter $\alpha = 1.2$. For datasets whose index construction fits within available memory (all except SIFT1B), we use one-shot Vamana construction. However, SIFT1B index construction requires approximately 1.1 TB of RAM, exceeding our machine's capacity. To address this limitation, we use merged-Vamana construction[3], with number of clusters $k = 13$ for SIFT1B. We refer readers to the DiskANN paper [25] for details on index construction techniques.

BiSON 's search algorithm uses search list size $L = 125$ for all datasets. Other than that, the search algorithm relies on

[3]Merged Vamana construction partitions the dataset into smaller clusters that each fit in memory, constructs indices on each cluster and merges them into a single graph.

TABLE II
SEARCH HYPERPARAMETERS: PQ DENOTES THE SIZE OF COMPRESSED EMBEDDINGS IN BYTES. W DENOTES BEAMWIDTH.

		LAN		WAN	
Dataset	PQ	Iterations	W	Iterations	W
LAION	64	5	12	5	12
SIFT1M	16	7	13	5	47
TripClick	64	7	12	5	45
MS MARCO	64	7	20	6	37
SIFT10M	32	7	9	6	18
SIFT50M	32	9	7	6	48
SIFT100M	32	9	8	7	19
SIFT1B	32	9	14	8	23

three key hyperparameters: Search Iterations (T), Beamwidth (W), and the size (in bytes) of compressed embeddings (PQ). The values of T, W and PQ for each dataset are tabulated in Table II. The best hyperparameters used for LAN and WAN are chosen based on observed latencies; as one can observe, we prefer lower search iterations in the WAN setting as the round trip time adds significant latency.

We evaluate search accuracy using two standard metrics: Recall@10 and MRR@10. The definitions of these metrics along with their values for each dataset is provided in Appendix A. Similar to Compass, we choose our search hyperparameters to achieve atleast 90% Recall@10 across datasets.

D. Performance

We compare performance of BiSON with that of Compass in terms of the following metrics (per query):

- **User-Perceived Latency:** Wall-clock time from query submission until the client receives the search results. This time includes all network latency and local computation time at both client and the server.
- **End-to-End Latency:** Total time including the user-perceived time plus the additional time for ORAM eviction step after the query response.
- **User-Perceived Communication:** Total bytes exchanged between client and server until the client receives the search results.
- **Total Communication:** Total bytes exchanged between client and server, including user-perceived communication and from additional interaction during ORAM evictions after the query response.

In Table III, the columns pertaining to User-Perceived Latency (Communication) are abbreviated as User-per. Latency (Comm.).

1) In-Memory Performance: Since Compass only supports deployment where all ORAM buckets reside in-memory at the server, we restrict our comparisons with Compass to the *in-memory* server configuration (i.e, a server which does not make reads/writes to a disk throughout the protocol execution). We evaluate BiSON and Compass in the LAN and WAN settings described in Section V-A and tabulate our comparison results between the two protocols in Table III.

The careful selection of search hyperparameters for BiSON (listed in table II) was targeted at achieving high search accuracy with minimal beamwidth and search iterations. Optimizing search iterations directly translates to a reduction in the number of required round-trips; when coupled with reduced beamwidth, their product results in reduced user-perceived communication.

Moreover, for each dataset, the first few BFS levels of the graph from the entry point are cached at the client side. This allows initial search iterations to run locally and decreases the number of round trips required by BiSON . In particular, we cache 2 levels for LAION, and 3 levels for all other datasets.

In the LAN setting, BiSON 's search converges in $4-5$ less round-trips as compared to Compass. Furthermore, BiSON incurs $3-18\times$ less user-perceived communication, and $5-28\times$ less total communication. As seen in Table III, BiSON achieves a speedup of $2-8.5\times$ and $3-23.5\times$ over Compass in terms of user-perceived latency and end-to-end latency.

In the WAN setting, BiSON requires $5-7$ less round-trips than Compass and incurs $2.8-12\times$ (resp. $4-14\times$) less user-perceived (resp. total) communication. BiSON is $2.1-4.6\times$ and $2.5-8.6\times$ faster than Compass in terms of user-perceived latency and end-to-end latency, respectively.

In addition to the datasets used in Compass, we demonstrate the scalability of BiSON to datasets over $10\times$ larger than those used in Compass in the in-memory server configuration. We tabulate our search performance for datasets of size 10 million, 50 million, 100 million in Table IV.

The number of round trips remains relatively stable (3 to 6), even as the dataset size increases. This is attributable to the shallow structure of Vamana-built indices. The smaller diameter compared to HNSW graphs enables BiSON to converge to nodes closer to the query with fewer hops. As previously discussed in Section III-D, this reduced traversal lowers the number of server round trips, and consequently, user-perceived latency. As a result, despite the increase in dataset size, latency remains practical as user-perceived latency stays ≤ 7 ms in the LAN setting and ≤ 450 ms in the WAN setting.

User-perceived and total communication also remain low, partly due to fewer server round trips. It is important to note that communication values (user-perceived and total communication) for SIFTX datasets are lesser compared to those of datasets like MS MARCO or TripClick. This is because the block size (size of response from the server for a single ORAM.FetchBlocks() operation) depends on the dimensions of the full embeddings: MS MARCO (and TripClick) has 768 dimensional embeddings, compared to SIFTX having 128 dimensions.

2) On-Disk Performance: We now demonstrate the performance of our disk-compatible ORAM (described in Section IV-B) implementation, wherein the server stores ORAM buckets on the disk. We evaluate our implementation on datasets of size 100 million (SIFT100M) and 1 billion (SIFT1B) and tabulate our search performance in Table V.

When comparing the latency results for SIFT100M dataset between in-memory ORAM server (Table IV) and disk-

TABLE III
IN MEMORY SERVER: COMMUNICATION(IN MB) AND LATENCY(IN MS) COMPARISON FOR SMALL DATASETS ($\leq$ 10M RECORDS)

LAN Setting												
Dataset	**User-per. Comm.**		**Total Comm.**		**Round Trips**		**User-per. Latency**			**End-to-End Latency**		
	BiSON	Compass	BiSON	Compass	BiSON	Compass	BiSON	Compass	Impr.	BiSON	Compass	Impr.
LAION	0.10	0.39	2.63	13.1	3	8	5	10	2.0×	8	24	3.0×
SIFT1M	0.11	0.34	1.41	11.2	4	8	6	13	2.2×	9	40	4.4×
TripClick	0.21	3.80	4.63	129.7	4	9	10	85	8.5×	16	376	23.5×
MS MARCO	0.30	5.07	11.78	221.3	4	9	20	122	6.1×	40	610	15.3×
WAN Setting												
Dataset	**User-per. Comm.**		**Total Comm.**		**Round Trips**		**User-per. Latency**			**End-to-End Latency**		
	BiSON	Compass	BiSON	Compass	BiSON	Compass	BiSON	Compass	Impr.	BiSON	Compass	Impr.
LAION	0.10	0.39	2.63	13.1	3	8	290	615	2.1×	389	981	2.5×
SIFT1M	0.12	0.34	2.71	11.2	2	8	225	593	2.6×	328	951	2.9×
TripClick	0.35	3.80	9.18	129.7	2	9	215	997	4.6×	477	4093	8.6×
MS MARCO	0.41	5.07	17.63	221.3	3	9	313	1145	3.6×	782	6424	8.2×

TABLE IV
IN MEMORY SERVER: COMMUNICATION(IN MB) AND LATENCY(IN MS) FOR LARGER DATASETS (10 M – 100M). RT DENOTES THE NUMBER OF SEARCH ROUND TRIPS(EXCLUDING EVICTION). E2E STANDS FOR END-TO-END.

LAN Setting					
Dataset	**Communication**		**RT**	**Latency**	
	User-per.	Total		User-per.	E2E
SIFT10M	0.04	1.72	4	5	10
SIFT50M	0.06	2.00	6	6	11
SIFT100M	0.08	2.15	6	7	13
WAN Setting					
Dataset	**Communication**		**RT**	**Latency**	
	User-per.	Total		User-per.	E2E
SIFT10M	0.16	1.84	3	347	644
SIFT50M	0.19	7.94	3	450	1187
SIFT100M	0.10	4.23	4	440	1032

TABLE V
ON-DISK SERVER: COMMUNICATION(MB) AND LATENCY(MS). RT DENOTES THE NUMBER OF SEARCH ROUND TRIPS(EXCLUDING EVICTION). E2E STANDS FOR END-TO-END.

LAN Setting					
Dataset	**Communication**		**RT**	**Latency**	
	User-per.	Total		User-per.	E2E
SIFT100M	0.08	2.15	6	26	44
SIFT1B	0.11	3.48	6	32	57
WAN Setting					
Dataset	**Communication**		**RT**	**Latency**	
	User-per.	Total		User-per.	E2E
SIFT100M	0.10	4.24	4	470	1071
SIFT1B	0.15	5.32	5	535	680

equipped ORAM server (Table V), we observe a modest increase in user-perceived latency in both LAN and WAN settings. This increase stems primarily from the overhead of disk I/O operations relative to in-memory accesses. Importantly, this overhead remains minimal (7 ms $\rightarrow$ 26 ms in the LAN setting and 440 ms $\rightarrow$ 470 ms in the WAN setting), thus demonstrating that the transition from in-memory to on-disk ORAM storage preserves practical query response time. Interestingly, even for the largest dataset of size 1 billion records, our user-perceived latency is a few milliseconds in the LAN setting and just around half-a-second in WAN setting.

E. Storage

We use $Z = 32$ (number of real data blocks) and $S = 64$ (number of dummy blocks), same as Compass, when building our ORAM buckets. This introduces significant blowup in the server-side storage, as the ORAM server must store at least twice as many dummy blocks as real blocks per bucket. Consequently, server-side storage exhibits roughly a $3\times$ blowup compared to storing the cleartext index.

On the client side, all storage reside in client memory, and consists of three main components: compressed embeddings (Hints), ORAM-specific states (ORAM) and DiskANN graph cache (Graph). This is similar to the client-side storage breakdown in Compass. In Table VI we tabulate the size of each client-side storage component and the total client storage. Our choice for the value of PQ (size of compressed embeddings in bytes) is tabulated in Table II. The size of compressed embeddings grows linearly with the number of dataset records, as each vector requires its corresponding compressed embedding. The ORAM-specific storage component comprises of ORAM-related states such as the block metadata, position map, stash and top-t cached layers of the ORAM tree (t is a dataset-specific hyperparameter). In DiskANN graph cache, the client caches the first $2-3$ levels of the graph, expanding out from the start node in a breadth-first fashion. The cache helps reduce server round-trips as search iterations for the nodes present in the cache do not require fetching data from the ORAM server.

We also report the total client-side storage as a percentage of the dataset size, where dataset size is the total size of full vector embeddings for all records in the dataset.

F. Limitations

Client storage and scalability. BiSON incurs a modest client-side storage overhead, approximately 2% higher than Compass, primarily due to the position map in the underlying RingORAM construction. This overhead can be reduced by employing more space-efficient techniques such as hierarchical position maps [33], at the cost of increased query latency. Overall, BiSON strikes a practical balance between client storage and latency, enabling the first secure semantic search system that simultaneously provides full query and data pri-

TABLE VI
CLIENT-SIDE STORAGE BREAKDOWN FOR ALL DATASETS

Dataset	Hints MB	ORAM MB	Graph MB	Total MB	% Full Data
LAION	6	3	0.3	9	4.8%
SIFT1M	15	18	13	46	9.3%
TripClick	93	36	54	184	4.1%
MS MARCO	539	232	55	827	3.2%
SIFT10M	305	232	15	553	11.2%
SIFT50M	1491	956	15	2463	10.2%
SIFT100M	3051	1919	16	4987	10.2%
SIFT1B	30517	16233	159	46910	9.1%

vacy, millisecond-scale performance, and scalability to large datasets.

Security model. Our current implementation assumes a semi-honest server and does not protect a client from a malicious server. We expect such guarantees to be readily incorporated into BiSON using authenticated ORAM techniques similar to those employed in Compass.

VI. CONCLUSION

In this paper, we propose a new semantic search protocol, BiSON that scales to encrypted databases consisting of billions of records. Our protocol is based on the DiskANN search algorithm and preserves both query and dataset privacy using the Ring ORAM protocol. We validated the suitability of DiskANN as the underlying cleartext search algorithm for a secure client-server search setting by comparing it to the state-of-the-art Compass search algorithm. Our experimental results show that BiSON outperforms Compass by up to $8.5\times$ on all benchmarks in terms of query latency and achieves up to $23\times$ reduction in end-to-end latency (while reducing communication by up to $30\times$). Furthermore, we demonstrate scalability of our protocol by evaluating its performance on datasets of up to 1 billion records, an improvement in scale that we attribute to our disk-compatible ORAM implementation. The above improvements jointly establish BiSON as the first secure semantic search protocol that scales to cloud-scale datasets while maintaining high search accuracy and millisecond search latency. Our secure search protocol based on DiskANN can be seamlessly integrated with any efficient ORAM protocol, improvements in which will further improve the efficiency of secure semantic search.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments to improve the paper. We also thank Yash Kunwar, Tanmay Rajore and Sandhya Saravanan for helping out with the experiments.

REFERENCES

[1] "Azure customer stories." [Online]. Available: https://azure.microsoft.com/en-us/resources/customer-stories
[2] "Aws customers." [Online]. Available: https://aws.amazon.com/s3/customers/
[3] "Google cloud customers." [Online]. Available: https://cloud.google.com/customers
[4] "Onedrive." [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage
[5] "icloud." [Online]. Available: https://www.icloud.com/
[6] "Google drive." [Online]. Available: https://workspace.google.com/products/drive/
[7] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
[8] F. Olumofin and I. Goldberg, "Privacy-preserving queries over relational databases," in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, ser. PETS'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 75–92.
[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 79–88.
[10] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "Seal: attack mitigation for encrypted databases via adjustable leakage," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.
[11] C. Örencik and E. Savaş, "An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking," *Distrib. Parallel Databases*, vol. 32, no. 1, p. 119–160, Mar. 2014. [Online]. Available: https://doi.org/10.1007/s10619-013-7123-9
[12] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica, "{DORY}: An encrypted search system with distributed trust," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1101–1119.
[13] D. Agun, J. Shao, S. Ji, S. Tessaro, and T. Yang, "Privacy and efficiency tradeoffs for multiword top k search with linear additive rank scoring," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 1725–1734. [Online]. Available: https://doi.org/10.1145/3178876.3186084
[14] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," *Cryptology ePrint Archive*, 2014.
[15] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, Aug. 2014. [Online]. Available: https://doi.org/10.1145/2636328
[16] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE symposium on security and privacy. S&P 2000*. IEEE, 2000, pp. 44–55.
[17] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1053–1067.
[18] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Annual International Cryptology Conference*. Springer, 2018, pp. 371–406.
[19] D. Aritomo, C. Watanabe, M. Matsubara, and A. Morishima, "A privacy-preserving similarity search scheme over encrypted word embeddings," in *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, 2019, pp. 403–412.
[20] S. Servan-Schreiber, S. Langowski, and S. Devadas, "Private approximate nearest neighbor search with sublinear communication," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 911–929.
[21] A. Henzinger, E. Dauterman, H. Corrigan-Gibbs, and N. Zeldovich, "Private web search with tiptoe," in *Proceedings of the 29th symposium on operating systems principles*, 2023, pp. 396–416.
[22] M. Zhou, E. Shi, and G. Fanti, "Pacmann: Efficient private approximate nearest neighbor search," *Cryptology ePrint Archive*, 2024.
[23] H. Chen, I. Chillotti, Y. Dong, O. Poburinnaya, I. Razenshteyn, and M. S. Riazi, "{SANNS}: Scaling up secure approximate {k-Nearest} neighbors search," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2111–2128.
[24] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.
[25] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor

search on a single node," *Advances in neural information processing Systems*, vol. 32, 2019.

[26] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proc. VLDB Endow.*, vol. 12, no. 5, p. 461–474, Jan. 2019. [Online]. Available: https://doi.org/10.14778/3303753.3303754

[27] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: highly-efficient billion-scale approximate nearest neighbor search," in *Proceedings of the 35th International Conference on Neural Information Processing Systems*, ser. NIPS '21. Red Hook, NY, USA: Curran Associates Inc., 2021.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[29] "Bing search." [Online]. Available: https://www.microsoft.com/en-us/bing

[30] "Elasticsearch." [Online]. Available: https://www.elastic.co/

[31] J. Zhu, L. Patel, M. Zaharia, and R. A. Popa, "Compass: Encrypted semantic search with high accuracy," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 915–938.

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

[33] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 415–430. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-ling

[34] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 44–75.

[35] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: extremely simple, single-server pir with sublinear server computation," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4296–4314.

[36] J. Li, Z. Huang, M. Zhang, J. Liu, C. Hong, T. Wei, and W. Chen, "Panther: Private approximate nearest neighbor search in the single server setting," *Cryptology ePrint Archive*, 2024.

[37] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2011, pp. 861–864.

[38] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, p. 117–122, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327494

[39] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, ser. STOC '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 537–546. [Online]. Available: https://doi.org/10.1145/1374376.1374452

[40] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[41] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, p. 509–517, Sep. 1975. [Online]. Available: https://doi.org/10.1145/361002.361007

[42] A. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, 1997, pp. 21–29.

[43] R. Ostrovsky, "Efficient computation on oblivious rams," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 514–523.

[44] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," vol. 65, no. 4, 2018. [Online]. Available: https://doi.org/10.1145/3177872

[45] "Sift bigann dataset," http://corpus-texmex.irisa.fr/.

[46] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[47] J. Axboe, "liburing library for io_uring," 2022. [Online]. Available: https://github.com/axboe/liburing

[48] H. V. Simhadri, R. Krishnaswamy, G. Srinivasa, S. J. Subramanya, A. Antonijevic, D. Pryce, D. Kaczynski, S. Williams, S. Gollapudi, V. Sivashankar, N. Karia, A. Singh, S. Jaiswal, N. Mahapatro, P. Adams, B. Tower, and Y. Patel.

[49] "Compass repository." [Online]. Available: https://github.com/Clive2312/compass

[50] C. Schuhmann, R. Vencu, R. Beaumont, R. Kaczmarczyk, C. Mullis, A. Katta, T. Coombes, J. Jitsev, and A. Komatsuzaki, "Laion-400m: Open dataset of clip-filtered 400 million image-text pairs," *arXiv preprint arXiv:2111.02114*, 2021.

[51] N. Rekabsaz, O. Lesota, M. Schedl, J. Brassey, and C. Eickhoff, "Tripclick: the log files of a large health web search engine," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 2507–2513.

[52] N. Craswell, B. Mitra, E. Yilmaz, D. Campos, and J. Lin, "Ms marco: Benchmarking ranking models in the large-data regime," in *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, 2021, pp. 1566–1576.

APPENDIX A
SEARCH ACCURACY

We evaluate search accuracy using two standard metrics: **Recall@10**, which is the fraction of true 10-nearest neighbors that appear in the retrieved list, and **MRR@10** (Mean reciprocal rank at 10) which is equal to the inverse of the rank of the first relevant result in the retrieved set. Higher value of MRR indicates relevant results appear earlier in the retrieved list.

For the search hyperparameters shown in Table II, we tabulate in Table VII the accuracy of BiSON in terms of MRR@10 and Recall@10 for each dataset. For TripClick and MS MARCO, we use cosine similarity as our distance function, while for other datasets we use L2 Euclidean distance. Similar to Compass [31], computation for TripClick and MS MARCO was done using human annotated relevance judgements; for other datasets brute-force nearest neighbor results were used. We selected hyperarameters to ensure that Recall@10 stays $\geq 90\%$ across datasets. The accuracy acheived by BiSON is on par with cleartext DiskANN search. This is because we confirmed functional equivalence between the two systems, i.e. for each query the traversed sequence of nodes is same. Hence, we do not compromise on search accuracy despite the addition of our privacy guarantees.

TABLE VII
SEARCH ACCURACY (%). MRR@10 FOR BRUTE-FORCE SEARCH RESULTS FOR TRIPCLICK AND MS MARCO ARE 26.9% AND 27.1% RESPECTIVELY.

	LAN		WAN	
Dataset	Recall@10	MRR@10	Recall@10	MRR@10
LAION	92.66	99.22	92.66	99.22
SIFT1M	90.24	97.91	90.00	94.42
TripClick	90.18	25.81	90.15	26.12
MS MARCO	91.43	25.68	91.00	25.46
SIFT10M	90.63	96.64	90.69	95.07
SIFT50M	90.86	96.56	90.09	92.65
SIFT100M	90.45	96.23	90.50	92.13
SIFT1B	90.28	95.53	90.33	95.10