Compass: Encrypted Semantic Search with High Accuracy

Jinhao Zhu
UC Berkeley

Liana Patel
Stanford University

Matei Zaharia
UC Berkeley

Raluca Ada Popa
UC Berkeley

Abstract

We introduce Compass, a semantic search system over encrypted data that offers high accuracy, comparable to state-of-the-art plaintext search algorithms while protecting data, queries and search results from a fully compromised server. Additionally, Compass enables privacy-preserving RAG where both the RAG database and the query are protected.

Compass contributes a novel way to traverse the Hierarchical Navigable Small Worlds (HNSW) graph, a top-performing nearest neighbor search index, over Oblivious RAM, a cryptographic primitive with strong security guarantees. Our techniques, Directional Neighbor Filtering, Speculative Greedy Search, and HNSW-tailored Path ORAM ensure that Compass achieves user-perceived latencies of a few seconds and is orders of magnitude faster than baselines for encrypted embeddings search.

1 Introduction

An increasing number of user data systems have adopted end-to-end encryption because of its strong security properties. Examples of such systems include WhatsApp, iCloud Backups, Telegram, Signal, and PreVeil [2, 19, 64, 75, 82]. In this setting, there has been a decades-long rich line of work on encrypted search [15, 17, 20–22, 27, 29, 40, 41, 54, 57, 71, 74, 77, 80, 86]: enabling the server to search on the encrypted data without learning the data or the query. Despite much progress, two challenges remain: reduced security and low search accuracy.

Security: The desiderata is to protect the data, query and query results – including search access patterns – against a *fully* compromised server. ***i) Leaky practical search.*** A long line of work enables the server to learn some information about the query or data [11, 12, 15, 32, 41, 45, 57, 77, 80], such as search access patterns, to provide fast search. However, leakage-abuse attacks [10, 35, 42, 49, 62, 63, 88] can reconstruct a significant amount of data or the query from this leakage. ***ii) Search with partial server trust.*** A line of work assumes trusted hardware enclaves at the server [5, 54, 81], but hardware enclaves suffer from a wide range of side channels [9, 43, 87], including some that can entirely subvert remote attestation and their security [84]. Another line of work [17, 18, 74] considers the logical server consists of two or more server trust domains, where at least one of them is trusted. However, it has been shown to be very difficult to find and deploy such different trust domains [16, 48], and attackers can still compromise both of these servers.

Who is Paula Deen's brother?

Keyword Search Result: What happened to Paula Deen's first husband? Paula Deen divorced her first husband Jimmy Deen (described as her hard-drinking high school sweetheart) in 1989 after 27 years of marriage; they had two sons together. In 2004 she married Michael Grover. Soon after her divorce, Deen started her own catering company, The Bag Lady.

Semantic Search Result: Paula Deen and her brother Earl W. Bubba Hiers are being sued by a former general manager at Uncle Bubba's Seafood and Oyster House, a restaurant they co-own.

Figure 1: Example of top-1 search results with the keyword search using TF-IDF (top) versus semantic search (bottom) on the MS MARCO dataset. The keyword search looks up each keyword individually and intersects them, but the term "brother" by itself is generic and produces too many results. In contrast, semantic search understands that the user is interested in Paula's brother.

Accuracy: Most of the works mentioned above have significantly lower search accuracy compared to state-of-the-art plaintext search systems. This is because they implement *lexical* search, which is less accurate than the *semantic* search used in modern systems. For example, some works above [4, 15, 17, 40, 41, 54, 77, 83, 86] implement an inverted index that maps a keyword to a list of documents. While effective for single-keyword searches, this method struggles with complex queries like expressions or questions. State-of-the-art search systems such as Bing, and Elasticsearch [3, 53] implement semantic search. Semantic search converts both the query and the content into vector embeddings in a high-

dimensional space. This approach allows the search algorithm to compare the meanings of texts based on their proximity in this space, rather than just matching exact keywords. semantic search is more accurate as it understands the intent and contextual meaning of the search query. It also extends search capabilities to unstructured data types such as audio, images, and videos. Fig. 1 shows an example of a search result from a semantic search versus from a keyword search.

In this work, through our system ***Compass*** we show that one can design an efficient search system over encrypted data that achieves the best of both worlds — namely, a highly accurate semantic search without leaking user data or queries while ensuring search integrity against malicious servers (§4.8). In Compass, search and insert time are both theoretically (poly)logarithmic in expectation[1] and empirically efficient as we show in the evaluation (§6). Compass's search quality is on par with state-of-the-art unencrypted search systems. Besides, Compass can be applied to Private *Retrieval Augmented Generation (RAG)* systems [46], as discussed in §7.

Searching over encrypted embeddings is a nascent line of work. The few prior proposals are either inefficient (for each query, HERS [24] performs a linear scan with FHE over the entire data, and SANNS [13] combines heavyweight tools like FHE, distributed ORAM, and Garbled Circuit), or have weak security ([6, 89] leak query information and Preco [74] assumes partial server trust), as we elaborate in §7.

Our goal is ambitious: to execute a powerful *state-of-the-art* search index for embeddings, Hierarchical Navigable Small World (HNSW) [52], over encrypted data. HNSW is a top-performing index used in modern plaintext semantic search [61]. At a high level, HNSW is a multi-layer graph with fewer nodes and edges on the upper layers and more nodes and edges on the bottom layers. A search starts from the top layer, follows links within each layer to find the locally nearest neighbor to the query, and then continues to the layer below. The goal here is to support HNSW securely and efficiently on encrypted data without reducing its accuracy. However, HNSW performs a complex traversal in embedding space that would be inefficient to perform with fully homomorphic encryption (FHE) [28] or GarbledRAM [51].

Instead, we build Compass based on Oblivious RAM [58], specifically Path ORAM [79]. Beyond access pattern protection, Path ORAM offers additional attractive properties for searching over encrypted embeddings. First, on the practical side, the constant has significantly improved since ORAM was first proposed, and the communication is sublinear. Second, with ORAM, the client performs distance computations locally and decides the next step in an index walk. Unlike FHE, this allows us to support any embedding distance metric, enabling a wealth of state-of-the-art embeddings. Besides, as noted in Path ORAM, it's straightforward to support integrity against a malicious server. Fig. 2 shows an overview

[1] under the assumption that the HNSW graphs emulate a Delaunay graph

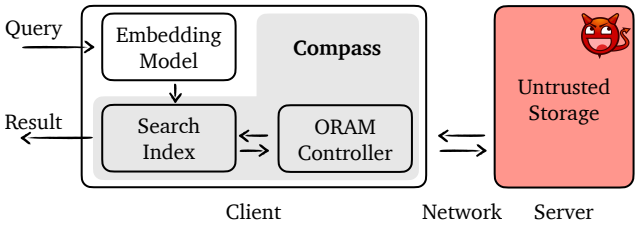

Figure 2: Compass Architecture

of Compass's architecture.

The fundamental challenge in supporting HNSW with ORAM is that HNSW assumes that the index and embeddings are stored in local memory and it is optimized for performing a *multi-hop* walk locally. A strawman solution is to make an ORAM request to the server for every node visited by HNSW's search algorithm. Unfortunately, this strawman is highly inefficient because it has:

- *High bandwidth consumption:* At every candidate node, HNSW fetches each neighbor's embedding and computes its distance to the query vector. This means that to evaluate a single node in the walk, we have to perform tens to hundreds of ORAM requests.
- *High number of roundtrips:* A search in the HNSW graph typically visits tens to hundreds of nodes, each having tens to hundreds of neighbors that also are fetched. Empirically, it requires thousands of network round trips to achieve good recall on real datasets.

To address this challenge, we develop a new search index that *co-designs* an alternative HNSW-like graph walk and an ORAM backend for efficient semantic search. Specifically, we introduce three techniques. To reduce bandwidth consumption, we introduce *Directional Neighbor Filtering*. The key idea here is, at every node in the graph walk, to only fetch the embeddings of a subset of neighbors that are in the same "direction" in the embedding space as the query vector. This subset is determined based on a small chunk of data, which is orders of magnitude smaller than the original dataset, cached locally on the client side. Second, we introduce *Speculative Greedy Search*, which speculatively fetches additional nodes that are likely to be visited in the future to reduce the number of network round trips and user-facing latency of the graph walk, while maintaining privacy. Finally, we introduce *HNSW-tailored Path ORAM* that integrates the search index walk into a white-box "rearrangement" of the Path ORAM protocol, further improving the communication overhead and significantly reducing user-perceived latency.

Evaluation summary: We implement and evaluate Compass across 4 popular datasets of different sizes and dimensions. Our techniques above deliver up to 125× speedup over the strawman implementation of HNSW on top of ORAM and make encrypted embedding search with high accuracy practical for the first time. Our results show that Compass

not only significantly outperforms baselines but also matches the accuracy of the state-of-the-art plaintext search algorithm across all datasets. For user-scale datasets, Compass achieves user-perceived latencies of 0.05 to 0.09 seconds on same-region networks and 0.77 to 1.03 seconds on cross-region networks. For a web-scale dataset like MS MARCO, Compass responds to search queries in just 13 seconds over a slow cross-region network setting.

Limitations: To match the plaintext search quality, Compass makes multiple roundtrips to the ORAM server for each search (e.g., an average of 8 to 16 in our evaluation), unlike a vanilla unencrypted search which makes only one. While §6 shows that the user-perceived latency is reasonable for a single search, this might be concerning for systems that need to perform a chain of searches for one user-facing operation.

2 Background

2.1 Hierarchical Navigable Small Worlds

In this section, we provide a brief overview of HNSW [52], a state-of-the-art graph-based *Approximate-Nearest-Neighbor (ANN)* method that has empirically demonstrated strong search performance [7, 76]. HNSW creates an index over a vector dataset by constructing a proximity graph G(V, E). This proximity graph represents each vector in the dataset as a vertex in the graph, and connects vertices by edges. HNSW, in particular, creates a hierarchical, multi-layer graph index that has bounded node-degrees.

The HNSW search algorithm follows a simple iterative greedy search procedure. As shown in Fig. 3, the search begins from a pre-defined entry point. This entry point is chosen during the index construction and is a node on the uppermost layer of the HNSW graph. The search procedure iterates over each layer, performing a greedy search, before dropping down to the layer below to continue its search. On each upper layer, the greedy search uses a dynamic candidate list of size one and greedily chooses a single node, which then becomes the entry point to the next layer below. Finally, once the bottom-most layer is reached, the algorithm once again perform the greedy search, but increases the size of the dynamic candidate list to ef and greedily chooses K nodes, rather than a single node. The parameter ef is typically larger than K and controls the trade-off between the quality and efficiency, with higher values of ef corresponding to higher quality at the cost of higher search latency. In practice, one can choose the ef parameter by empirically generating the accuracy-efficiency trade-off curve over a benchmark dataset and choosing the ef value that meets the application's target accuracy at minimal search latency.

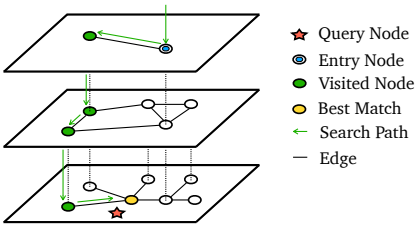

Figure 3: HNSW Graph Data Structure

2.2 Path ORAM

First introduced by Goldreich and Ostrovsky, Oblivious RAM (ORAM) [30, 58] allows a client to access encrypted data on remote storage without leaking actual access patterns. Path ORAM [79] is a Tree-based ORAM scheme that has been widely adopted due to its simplicity and practicability.

Path ORAM organizes the server-side storage as a binary tree with height L. Each node in the tree is called a bucket, which contains Z blocks. Z is also referred to as bucket size. A block contains either real data or a dummy. The client holds two data structures: position map and stash. The position map maintains a mapping of each block to a path that this block is assigned to. The stash serves as a local buffer that temporarily stores the blocks to be evicted.

In Path ORAM, each block is uniformly randomly assigned to one of 2^L paths in the tree. The invariant in Path ORAM is that, if a block is assigned to path l in the position map, then this block is either in one of the buckets on the path l or in the stash. Path ORAM provides a unified operation, Access for both read and write requests. The Access of a block b consists of following steps:

1. Read the path l of this block from the server.
2. Insert non-dummy blocks on this path into the stash and update the content of the target block if it's a write.
3. Randomly assign the target block to a new path l'.
4. Greedily evict the blocks in the stash to the path l and write the path back to the server.

3 System Overview

Our system setup consists of a logical server and multiple clients. Each client owns a separate set of documents and a separate index, both stored on the server. We focus on enabling a user to search on their own data, and data sharing is not a focus of our system.

Compass's API is as follows:

- INIT($\mathcal{D}$): Given a set of documents $\mathcal{D}$, initiate the index.
- SEARCH(Q, K) $\rightarrow$ LIST(ID, SCORE): Given Q, the embedding of the query, find the top K relevant documents and return a list of documents' ids with relevance scores.
- INSERT($\mathcal{F}$): Given a document $\mathcal{F}$, generate the embedding set $\mathcal{E}$ and store both the encrypted document and its embed-

dings on the server. The size of $\mathcal{E}$ depends on the document length and the model's context window.

- DELETE($\mathcal{F}$): Given a document $\mathcal{F}$, remove the document and the embeddings from the cloud.

When initializing the system, sequentially writing each block into ORAM is very inefficient. Instead, one can use the bulk loading protocol introduced in BULKOR [47].

3.1 Threat Model & Security Guarantees

Our system is based on a two-party model: a client and a server. By server, we refer to a logical server that may consist of multiple physical machines. The server is untrusted and can be maliciously compromised. Unlike some prior works that weaken the threat model by assuming multiple logical servers with at least one being honest [17, 70, 74, 85], we assume the entire server is untrusted. We also do not rely on uncompromised hardware modules or hardware trust assumptions at the server, like TEE-based systems [26, 54]. This means that the server in Compass can deviate from the protocol arbitrarily. Compass ensures that the server cannot learn any information about the user query, such as the query embedding, the IDs and lengths of the returned documents, or whether the query matches a previous one ("query access patterns"). Moreover, the server cannot learn the data contents from the search index or modify the search results without detection by the client.

Compass does not hide the type of operations—whether a user is performing a search, insert, or delete—on the remote data. We do not protect against timing side-channel attacks resulting from the duration of execution at the client, or from the server observing the timing of each user request. We consider parameters used in Compass algorithm as public information and do not protect them. The server can tell roughly how large the data of each user is (e.g. how many GBs each user has) but it cannot learn about the size of the result. (Of course, these can be protected through padding in time and space at a performance cost.) This strong model means that Compass protects against a plethora of attacks [10, 35, 42, 49, 62, 63, 88] on encrypted search that leverage access patterns or size of results. Like much prior work [54], Compass does not protect against DoS attacks from the server. The server may drop the requests from clients at any time or even delete the client's data. We assume it is the server's business interest to maintain a good availability to attract more customers.

The client, who owns the outsourced data, is trusted with its own documents. That is, the client may search and read the contents of these documents. However, the client cannot access or perform a search on the documents of other clients, even if the client colludes with the server.

3.1.1 Security Definition

Like prior work [72, 79], we provide an indistinguishability-based security definition. In our security game, there's a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ who acts as the client and the server of Compass respectively. According to the threat model, the challenger is honest and the adversary is malicious, which means $\mathcal{A}$ can deviate from the protocol arbitrarily. The attacker wins the security game if it can either learn partial information about the query or data or modify the search results without detection by the challenger (client).

1. The challenger $\mathcal{C}$ chooses a uniformly random bit b.
2. The adversary $\mathcal{A}$ chooses two equally large datasets D_0 and D_1, as well as one set of public parameters param. The challenger initializes the system with D_b and param.
3. The adversary iterates as follows. At step i:
   (a) The adaptive adversary $\mathcal{A}$ chooses a pair of requests $q_{i,0}$ and $q_{i,1}$. A request can be a search, insert, or delete.
   (b) When the challenger receives a pair of requests, it first checks whether the type matches. If they are of different types, the challenger aborts. Otherwise, the challenger interacts with the adversary according to the Compass's protocol for request $q_{i,b}$. As part of this execution, the client protocol verifies the integrity of the data from the server. If any verification fails, the challenger aborts.
   (c) The challenger returns the request result r_i to the user.
4. The adversary $\mathcal{A}$ outputs a guess b'.

Figure 4: Security game for Compass

Let param be the set of public parameters in our system. This includes HNSW parameters M and ef, as well as the number of cached layers, the size of the directional filter *efn* (§4.5), the size of the speculating set *efspec* (§4.6) in our modified HNSW search. We define the security game in Fig. 4. The adversary $\mathcal{A}$ wins the game if the challenger $\mathcal{C}$ does not abort and one of the following two conditions are met: ***1)*** $b' = b$; ***2)*** the sequence of queries $(q_{i,b}, r_i)$ is an incorrect execution of plaintext Compass's search algorithm with param on D_b.

Theorem 1. *Assuming a collision-resistant hash function and an IND-CCA2 secure encryption schemes, for any probabilistic polynomial time stateful adversary $\mathcal{A}$, $\mathcal{A}$'s chance of winning in the above security game instantiated with Compass's protocol is: less than half plus negligible for winning condition (1) and negligible for winning condition (2).*

Due to space constraints, we show a proof sketch in §4.10 and leave the full proof in App. C

4 Compass's Search Index

In this section, we describe Compass's index. We primarily focus on the search since the insert algorithm in HNSW closely

Symbol	Description
L_H	number of layers of HNSW
ef	size of dynamic candidate list in HNSW search
M	degree bound of traversed nodes in HNSW search
n	number of search iterations in HNSW search
efn	size of directional filter
$efspec$	size of speculation set
Z	bucket size of ORAM

Table 1: Summary of Notation

resembles that of the search, and the techniques described in this section apply to both. We show the summary of notation in Tab. 1.

4.1 Data Layout

Before we dive into the details of each technique, we first present the data layout in Compass. The client holds:

- *ORAM-related data:* a stash that serves as temporary storage for blocks to be evicted and a position map that maintains the mapping of each block to its assigned path.
- *HNSW-related data:* the client manages essential metadata for HNSW search, such as the dimension of the embeddings, the number of layers L_H, and the HNSW degree bound M. If client-side caching (see §5) is enabled, the client also maintains the graph for the upper layers and the embeddings of nodes within these layers.
- *Quantized hints* that are a part of our Directional Neighbor Filtering technique, which we explain later in §4.5.

The server holds the ORAM tree in the Path ORAM construction. Each node (bucket) of the tree consists of Z blocks. In our search index, each ORAM block corresponds to one node in the HNSW graph. The block of node x contains the embedding of node x and the list of identifiers of the neighbors of node x also called the *neighbor list of* x. Each bucket has a hash used for integrity check (§4.8).

4.2 Workflow overview

Fig. 2 shows the architecture and workflow of our system. Similar to plaintext search over embeddings, the query from the user is first transformed into fixed-dimensional, dense vector using a pre-trained embedding model. This query vector is then used to perform search over the embedded document corpus. Our search framework builds on HNSW and provides security guarantees by storing vector embeddings and the graph index encrypted on a remote server, accessed by the ORAM controller. In our setting, the client carries out all computational tasks while the server only acts as a remote storage. During the search process, the client interacts with the server to retrieve the index and embeddings.

4.3 An initial attempt

We now present a natural way of traversing HNSW using ORAM by the Compass client: the Compass client follows HNSW's greedy search algorithm to find the nearest node in the graph by making an ORAM request to fetch every node that is visited by this algorithm. This solution is inefficient, consuming a large amount of bandwidth and resulting in many round trips that affect user-perceived latency. The reason is that HNSW is designed for a local search with multi-hops, but with ORAM, every step becomes a non-trivial server request. It nevertheless establishes the foundation and terminology for our subsequent improvements in the rest of this section.

This HNSW greedy traversal is illustrated in Fig. 5a, which we now explain. Starting from the entry node (1), the algorithm fetches all the neighbors of the entry node and inserts them into the *candidate list*, namely it visits them. We say that the algorithm *visits* a node if it retrieves the ORAM block of the node containing its embedding and the list of neighbor identifiers. In subsequent iterations, the node that is closest to the query in the candidate list will be processed as the candidate node. Then the unvisited neighbors of the candidate nodes are visited and inserted into the candidate list. In Fig. 5a, it takes 7 search iterations to find the nearest node, with every node in the graph being visited during the search. This algorithm works well when the computation and storage are colocated. However, in our setting, the client has to retrieve neighbors' embeddings and neighbor lists from the remote server through ORAM requests, making it inefficient because of multiple costly network round trips as well as an extensive amount of data transferred. In each layer, the number of required round trips depends on ef, and each node's degree is bounded by M. Thus, completing the search in one layer requires $ef * M$ rounds of communication to retrieve in total of $ef * M$ nodes' embeddings and neighbor lists. Empirically, in the final layer of HNSW, M is 32, and ef ranges from tens to hundreds for sufficient accuracy, causing high search latency and bandwidth usage.

4.4 Towards Compass's search index

To understand our approach, it helps to first consider an attempt to reduce the number of sequential ORAM requests in the strawman above. The idea is to include the embeddings and neighbor lists of the neighbors in the neighbor list of a node within the ORAM block of that node. This will halve the number of ORAM requests and roundtrips as compared to the strawman. However, this strategy will also cause $M\times$ storage overhead on the server side because the embedding of each node will be stored in the ORAM block of that node and in the ORAM block of its neighbors. Moreover, the increase in storage does not result in substantial bandwidth savings due to the larger ORAM block size.

Instead, we propose a novel way to traverse the HNSW

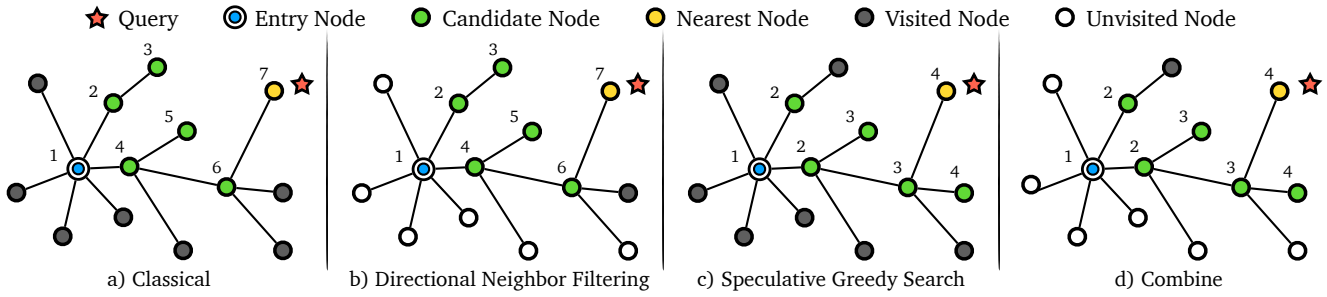

Figure 5: HNSW graph traversal followed by our ORAM-friendly traversal techniques. The search aims to identify the Nearest Node to the Query. The distance here is Euclidean distance. The numbers indicate iterations in the traversal, and which nodes are *processed* at each iteration. We say that a node is processed, if its neighbors are visited.

graph designed for running on top of ORAM. Our traversal approach significantly reduces roundtrips and bandwidth usage via three techniques. The first two techniques make black-box usage of ORAM; like in the attempt above, they try to guess which nodes will be needed in the search, in order to fetch them selectively or to prefetch them, but it does so in a more effective way than above: *Directional Neighbor Filtering* (§4.5) reduces the network bandwidth overhead and *Speculative Greedy Search* (§4.6) reduces the number of network round trips required. Our third technique, *HNSW-tailored ORAM* (§4.7), makes white-box use of Path ORAM to leverage the memory access characteristic in graph search to further reduce bandwidth and computation overhead.

4.5 Directional Neighbor Filtering

The key idea behind our Directional Neighbor Filtering technique is to consider only a subset of a node's neighbors during traversal because those closer to the query point are more likely to contribute to the final result; namely, to embed a sense of "direction". This algorithm disregards neighbors farther from the query point than nodes in the candidate set.

One possibility to implement this idea is to store with every node, not only its own embedding but also the list of *compressed* embeddings of its neighbors. The intuition is to determine which compressed neighbors are closer to the query, and only fetch those from the ORAM, reducing bandwidth consumption. One can use compression techniques like PCA [39]. However, since a node does not have a large number of neighbors, we found that the compression is not effective in reducing the size of each ORAM block because there are not enough neighbors to amortize the space taken by the compression state (e.g. the transformation matrix).

Instead, we compress together *all the nodes* into a data structure we call *Quantized Hints*. The client stores the Quantized Hints, a map from node id to quantized embedding of this node. Product Quantization (PQ) [37] fits well here: it is a widely-used quantization method that is highly effective at compressing a large number of high-dimensional vectors. For example, we achieve 98% compression on various datasets (see §6.3.3) with a codebook size equivalent to 128 full coordinates, thus requiring a minimal amount of client storage.

However, if we use PQ directly (namely, construct the index directly on quantized embeddings), there is a big accuracy drop because the closest quantized neighbor to the query is often not the closest full-coordinates neighbor.

Instead, our idea is to use the quantized nodes merely as "directional hints": for each node, we identify the top *efn* closest neighbors to the query based on the hint, and then fetch their *full* coordinates (the ground-truth) from the server to identify the next node to process. The intuition is that the closest *efn* quantized neighbors are very likely to contain the closest (full-coordinates) neighbor. We show in §6.4 that, using this technique, the decrease in search accuracy is negligible. Since we only fetch *efn* out of M neighbors, this technique saves significant bandwidth, also shown in §6.4.

For clarity, we include an example of using directional neighbor filtering during the search in Fig. 5b. For simplicity, assume that we only fetch the top 2 closest neighbors in each iteration. First, for each query, the Compass client computes its quantized embedding. Then, during each iteration in the traversal, let node A be the currently closest node to the query in the candidate list. The client iterates through A's neighbor ids looking up their quantized embedding in Quantized Hints, and computing each neighbor's distance from the query in the quantized space. The client then only fetches the top $efn = 2$ closest neighbors from the server in a batched ORAM request (§4.7), and adds them to the candidate list. As compared to Fig. 5a, we can see in Fig. 5b that some of the neighbors of the nodes processed at (1), (4) and (6), which are not among the top 2 closest nodes to the query, are not visited. We thus avoid visiting 6 nodes in the graph, which approximately halves the bandwidth consumption.

4.6 Speculative Greedy Search

In computer architecture, speculative execution [44] refers to an optimization technique in which the processor makes

guesses of potentially useful instructions and executes them in advance to improve performance. Our speculative greedy search shares the same intuition with speculative execution.

Inspired by beam search [59] from NLP, each time the Compass client performs a round trip to the ORAM server to retrieve nodes that will be visited next by HNSW's search, the client also speculatively fetches additional nodes. This implicitly requires batching the access of multiple ORAM blocks into one request, which is introduced in §4.7. These additional nodes are not yet needed by the HNSW search algorithm but are likely to be visited later during the search. Thus, by pre-emptively retrieving these *likely-needed* nodes alongside the *currently-needed* node, we can reduce the number of future network round trips to ORAM during search. But how do we identify these nodes? Fortunately, the candidate list used in the greedy search algorithm serves as a good basis for speculation. The candidate list is sorted by the distance to the query. In our speculative greedy search algorithm, we fetch the first *efspec* nodes's neighbors within one request. Once we get the response from the remote server, the candidate list is updated by evaluating the neighborhood of each node in the speculative set.

In Fig. 5c, we show an example of speculative greedy search. In each iteration, the client extracts two candidates from the candidate list and visits their neighbors simultaneously. In this toy example, Compass reduces the number of iterations from 7 to 4. We show in §6 that implementing speculative greedy search on real datasets significantly reduces the search steps required while maintaining the same accuracy.

DiskANN [36] uses a similar speculative approach in a different setting in which they try to optimize the interaction between memory and SSD. In DiskANN, they fetch a small amount of extra nodes to balance the cost of computing and SSD bandwidth. However, in Compass, the cost model of fetching data from the disk and fetching data from ORAM over the network is different. In our setting, a key observation is that optimizing the number of round trips is much more important than optimizing bandwidth because the round trips directly affect user-perceived search latency and the bandwidth consumption is already in a reasonable ballpark. Therefore, depending on the dataset, for Compass, the size of the speculation set *efspec* can grow as large as 16 to achieve good overall performance.

Fig. 5d shows the final example of the search process when we compose both directional neighbor filtering and speculative greedy search. Compared to Fig. 5a, the algorithm in Fig. 5d achieves the same search results with three fewer network round trips and six fewer node retrievals.

4.7 HNSW-tailored Path ORAM

Compass rearranges how ORAM requests are performed, in a way designed to reduce the cost of an end-to-end HNSW traversal. The reader should recall the Path ORAM background in §2.2. In Fig. 5d, the client needs to invoke multiple ORAM requests per iteration to fetch the information of each neighbor. A natural solution is batching, which has already been proposed as a mechanism for saving roundtrips in a scenario when multiple requests arrive at the same time [72, 86, 90]. This reduces the number of network round trips required from the number of neighbors to only one. Another benefit of batching requests is savings on network bandwidth. In tree-based ORAM, any two paths have overlapping buckets, at least the root bucket. Accessing multiple paths in a batch allows us to transfer each bucket only once.

In Path ORAM, evictions are scheduled after every path access to keep the stash within a reasonable size. However, when batching multiple path accesses into one request, it's not feasible to perform evictions for each path individually; instead, evictions are managed on a per-batch basis. In our system, we introduce *multi-hop lazy eviction*, a white-box modification of the Path ORAM algorithm: the Compass client performs a sequence of ORAM request batches, and only at the end of the query, it performs the stash eviction.

The advantages of multi-hop lazy eviction are substantial. First, it enables us to use smaller bucket sizes while maintaining a reasonable stash size, as the number of overflow blocks in the stash is reduced between two queries. This reduction is achieved by evicting multiple paths simultaneously, which increases the probability of overlaps between newly assigned paths and those due for eviction. Such overlaps make it more likely for a block in the stash to find an available bucket during the eviction process. Throughout the query, blocks fetched from the server are cached locally, allowing us to save bandwidth not only for overlapping paths within the same batch but also across the entire query. Besides, multi-hop lazy eviction offers lower user-perceived latency, as the eviction processes can be done after returning the query results.

With this approach, users will tolerate a temporary increase in memory usage during the query process. However, according to our analysis (App. B), the stash size remains reasonable. For example, on the largest MS MARCO dataset, the peak stash size is roughly 110MB. Another side effect of potentially large stash size is that it becomes inefficient to search for a candidate block for a specific bucket through a linear scan of the stash. Therefore, we sort the stash structure in which blocks are sorted according to their assigned path index. The key idea here is that, when evicting blocks into a bucket, we can recompute the range of path indices such that blocks assigned to one of these paths are candidates to be filled in the bucket. Indeed, this will increase the complexity of inserting a block into the stash to $O(\log N)$, where N is the size of the stash. However, it reduce the cost of the search for Z candidate blocks for a bucket from $O(ZN)$ to $O(\log N)$, as candidate blocks are already grouped during sorting.

While [72, 86, 90] already batches ORAM requests, all the requests were prepared together followed by an ORAM eviction. In our case, the client does not know the sequence

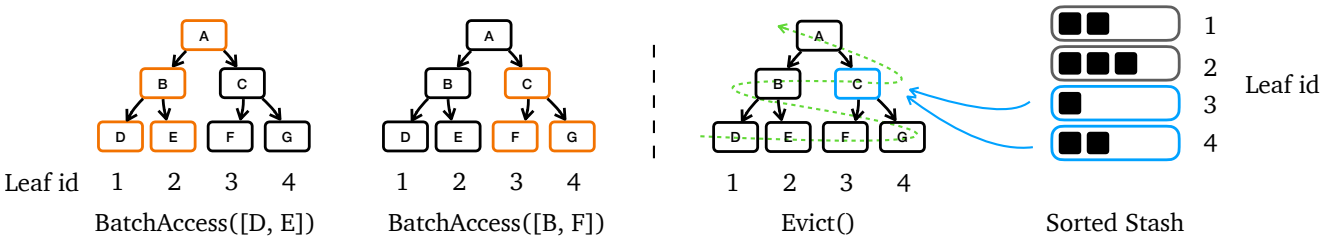

Figure 6: Example of batch access and eviction in Compass's HNSW tailored-ORAM

of all the ORAM requests, in fact, the sequence depends on the private query, and performs eviction only at the end of all iterations. This data dependency could be problematic for security. To address this, we keep a set of visited paths to ensure we never request the same path from the server more than once before we process the eviction.

Specifically, we maintain two additional data structures on the client side that track the history of visited paths and the history of visited buckets. These two structures are reset after each search query. For each requested block, we first determine whether the designated path has been visited before within the same query. If so, a new path is randomly selected from those not yet visited. This ensures that the total number of requested paths from the server matches the number of blocks in the batch access input. Next, once we decide on the paths to fetch from the server, we compute the positions of buckets residing on these paths. Here we don't fetch a bucket twice within a query, as the bucket is already cached locally.

In Fig. 6 we give an example of ORAM operations involved in one search request, consisting of two batch accesses followed by one eviction. In the first request, path 1 and path 2 are requested from the server. When the second request arrives, as block B is cached in the last request, we randomly sampled an unvisited path. Therefore, path 3 and path 4 are selected and requested from the server. After these two requests, we perform the eviction. The green dashed arrow indicates the sequence of eviction. When it comes to eviction for bucket C, we first determine which paths C belongs to, in this case, paths 3 and 4, and fill C with blocks assigned to these two paths in the sorted stash.

4.8 Malicious server protection

A malicious server can alter ciphertexts, rearrange the ORAM buckets layout, perform replay attacks, or answer queries incorrectly in other ways. Protecting integrity and freshness in our setting is easy: we employ existing work in the Path ORAM literature [69, 79] to construct a Merkle Tree on top of the ORAM Tree. The client stores the root of the Merkle tree and verifies every response from the server against this Merkle root. Since this is a well-understood solution, we do not provide further details.

4.9 Putting it All Together

Algorithm 1: SEARCH($q, ep, ef, efspec, efn$)

Input: query q, entry point ep,
size of speculation set $efspec$,
size of directional filter efn,
number of nearest to q elements to return ef
Output: ef closest neighbors to q

```
1  V ← ep // set of visited nodes
2  C ← ep // set of candidates
3  W ← ep // set of found nearest neighbors
4  n ← ⌈ef/efspec⌉
5  for step ← 0...n do
6      E1 ← extract top efspec nearest nodes from C to q
7      E2 ← ∅
8      foreach e1 ∈ E1 do
9          foreach e2 ∈ neighbors(e1) do
10             if e2 ∉ V then
11                 E2 ← E2 ∪ e2
12     t ← efspec * efn
       // Based on quantized hints
13     E3 ← extract top t nearest nodes from E2
14     oram.batch_access(E3, t)
15     foreach e ∈ E3 do
16         V ← V ∪ e
17         C ← C ∪ e
18         W ← W ∪ e
19         if |W| > ef then
20             remove furthest element from W to q
21 return W
```

With the techniques above, we present the concrete algorithm used in Compass. In HNSW, the insertion process closely mirrors the search, but with a different ef value, typically set to 40. A standard method for deleting a node in HNSW involves searching for the node and marking it as deleted. Therefore, we mainly focus on search in this section.

Alg. 1 shows the search algorithm Compass used in the final layer of the HNSW graph. The main difference from the vanilla HNSW is between line 4 and line 14. The num-

ber of search iterations, n, is determined by $\lceil ef/efspec \rceil$, as we extract the top *efspec* nodes from the candidate list as the speculative set, E_1, in each iteration. In the first iteration, there is only one candidate, the entry point, so only one node is extracted from the candidate set. We then traverse the unvisited neighbors of the nodes in E_1 (referred to as E_2), ranking them based on their distance to the query, estimated using quantized local hints. The top t nodes from E_2 are selected as E_3, and a batch ORAM request is issued to retrieve their full coordinates and neighbor lists. The value of t is set to $efspec * efn$, as we want *efn* neighbors per candidate. If the total number of nodes in E_3 is less than t, the batch ORAM request is padded to t for security. Once the full coordinates are retrieved, we insert each node in E_3 into the set of visited nodes V, the set of found nearest neighbors W, and the candidate set C. The set W is dynamic, as the furthest element will be removed when its size exceeds ef. After completing n iterations, W is returned and the first K nodes in W are the final search result of q. For search in upper layers, the ef is set to 1 and therefore we only apply directional neighbor filtering.

The algorithm requires n rounds of communication to the server. As stated in the prior section, each ORAM block contains a node's full coordinates and neighbor list. The fetched full coordinates are immediately used in line 20 to determine the furthest node. However, the neighbor list will not necessarily be used if the node is not selected as a candidate in future steps. A trade-off here is that we over-fetch this part of the data to avoid an extra round of communication when the full coordinates and the neighbor list are fetched separately.

4.10 Security proof sketch

Due to space limit, we provide a proof sketch covering the salient points in our proof, leaving a full proof to App. C. The integrity guarantee of our protocol comes directly from the Merkle tree on top of ORAM, which has already been proposed and proved [25, 69, 79]. Therefore, the attacker must respond correctly to all requests, with their only capability being to learn which items are accessed on the server during these requests. Prior works have already demonstrated that batching ORAM requests doesn't reduce security [86, 90]. The distinction between Compass and these works is that Compass's ORAM requests follow a multi-hop pattern. Each query in Compass involves a fixed amount of batch ORAM accesses. During batched ORAM accesses, the already visited paths are replaced with randomly selected unvisited paths, and the total number of paths is padded to a constant number. Therefore, if two queries are of the same type, they will request the same number of randomly selected paths from the server. In this case, the access pattern in Compass can be regarded as equivalent to that of a single, larger batch ORAM request containing an identical number of paths. (For a small dataset, in which the number of paths requested in a query exceeds the total number of paths available in the ORAM tree on the server, Compass thus streams the entire database from the server, shuffles it locally, and then streams it back.) Hence, Compass does not reveal any information about the access pattern, similar to traditional Path ORAM.

5 Implementation

We implement Compass in ≈ 5k lines of C++ code. We use Faiss [23] library for HNSW construction and Product Quantization. We use AES-256-CBC to encrypt the ORAM block and SHA-256 for hashing, via OpenSSL's EVP [1].

Layer-wise Caching: To save the cost of frequently visited nodes in the upper layers, we cache all but the last two layers of the HNSW graph and the embeddings of nodes in these layers locally on the client. We set the number of search steps in the second last layer to be 1. Due to the space limit, we explain the choice of the number of cached layers and search steps in App. A. We show the client side overhead of layer-wise caching in §6.3.3.

6 Evaluation

In this section, we aim to answer: What is the overhead of Compass, and how does it compare to prior encrypted search schemes? What is the search accuracy of Compass, and how does it compare to state-of-the-art plaintext search systems?

6.1 Experimental Setup

We evaluate our experiments on the Google Cloud Platform with one n2-standard-8 instance (8 vCPUs and 32 GB memory) as the client and one n2-highmem-64 instance (64 vCPUs and 512 GB memory) as the server. Both instances are in the same region and we use Linux Traffic Control (TC) to simulate different network settings. Similar to [13, 74], we have two sets of network configurations to simulate the network conditions within a region (*fast*) and across two regions (*slow*). Specifically, we set the network with 3Gbps bandwidth and 1ms round-trip latency for *fast* network, and 400Mbps bandwidth and 80ms latency for *slow* network.

We evaluate our system on four datasets:

MS MARCO [8] is a large-scale dataset created from real Bing search queries. It consists of 8,841,823 passages and 6,980 queries. We generate the 768-dimensional embedding vectors for MS MARCO using a pre-trained model, `msmarco-distilbert-dot-v5`, from Sentence-Bert [67].

TripClick [68] is a dataset containing real click logs from a health web search engine. It consists of 1,523,871 passages and 1,175 queries. Relevance scores for each query are assigned using document click-through rates. We use the same model above to generate TripClick's embeddings.

SIFT1M [37] is a widely-used dataset in ANN search literature, containing 1M base vectors and 10K queries. Each

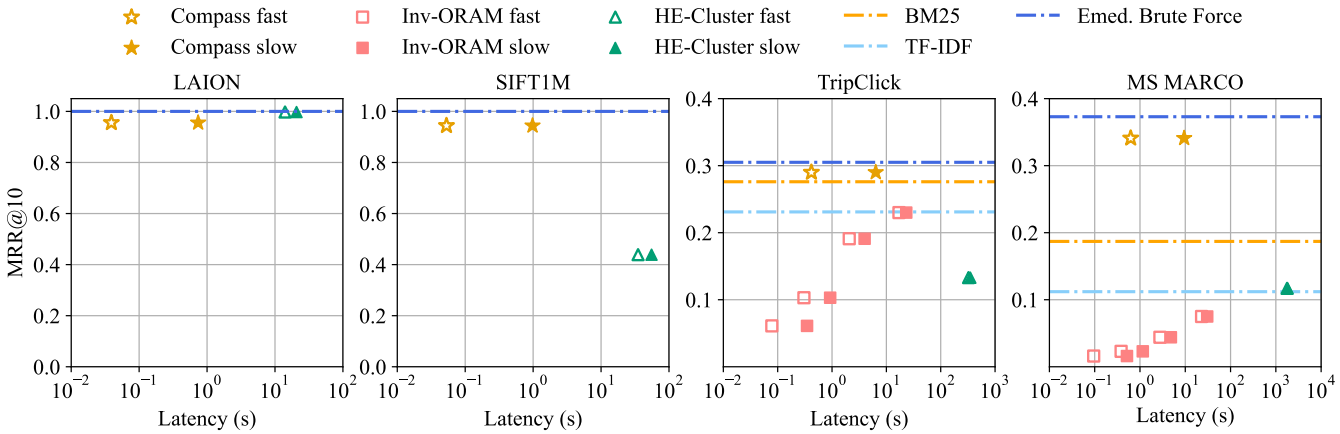

Figure 7: Comparison of Compass's search performance with Baselines. For TripClick, the results of the HE-Cluster under two networks are overlapped as the main bottleneck is the computation on the server. The search latency for HE-Cluster on MS MARCO is extrapolated due to memory limitation. (See estimated memory consumption in §6.3.3)

vector is a SIFT (Scale-Invariant Feature Transform) descriptor [50] extracted from images, with a dimension of 128.

LAION [73] is a large-scale dataset consisting of 512-dimensional embeddings of 400M image-text pairs generated by CLIP [65]. In the evaluation, we create a subset, containing 100K vectors to simulate the size of a real user's cloud storage.

We build the index with the default construction parameters suggested by Faiss, with $M = 32$ and $efConstruction = 40$. For SIFT1M, each embedding vector is divided into 8 subvectors for PQ, while for other higher-dimensional datasets, we split into 32 subvectors. The search parameters, shown in Tab. 2, are set to achieve a Recall@10 of at least 0.9.

	MS MARCO	TripClick	SIFT1M	LAION
ef	224	192	32	12
efspec	16	16	4	2
efn	12	12	12	12

Table 2: Search parameters for each dataset.

6.2 Baselines

We compare our system with two baselines.

Inv-ORAM: Our first baseline is inspired from [54, 86]. It is built on an inverted index stored inside ORAM. The inverted index maintains a mapping from the keyword to the documents that contain the keyword. We compute the relevance score of a keyword in a document using TF-IDF [66], a widely used algorithm in text search systems. Each $(keyword, document, score)$ pair is stored inside an ORAM block. To avoid the leakage of query length and keyword frequency. We pad each query to the length of the longest query, specifically, 12 for TripClick and 16 for MS MARCO. To hide keyword frequency, ideally, we should fetch the maximum number of documents a keyword maps to. However, as pointed out by [56], this can be worse than streaming the entire database. Therefore, we adopt the idea from [54] and truncate the document list for each keyword to a fixed size. To make the truncation meaningful, this list is sorted by the relevance score between the document and the keyword during index construction. To simulate the search performance of OBI [86], we also implemented this baseline with batch ORAM access and our better stash eviction algorithm. We evaluate this baseline on TripClick and MS MARCO, as this baseline only supports text-based search.

HE-Cluster: Our second baseline uses homomorphic encryption. Inspired by Tiptoe [33], we apply clustering to avoid linear communication. Each dataset of size N is clustered into roughly $\sqrt{N}$ clusters using k-means. 80% of the documents are assigned to a single cluster, while the remaining 20%, located near cluster boundaries, are assigned to two clusters. To avoid the leakage of cluster size, we pad the number of documents in each cluster to the maximum. We parallelize the server-side computation of this baseline across 64 threads.

6.3 Search Performance

We measure the search performance in two dimensions: search quality and latency. For search quality, we use the Mean Reciprocal Rank at 10 (MRR@10), where Reciprocal Rank is the inverse of the rank of the first relevant item in the results. For Compass, we report user-perceived latency (excluding eviction) under the semi-honest setting. Latency under the malicious setting will be shown in §6.3.1.

For the Inv-ORAM baseline, we create 4 variations, setting each keyword's truncated document list size to 10, 100, 1000, and 10,000. The tradeoff here is a larger size for better search quality but worse latency.

In Fig. 7, we present the search performance of Compass with that of baselines under two network configurations. The

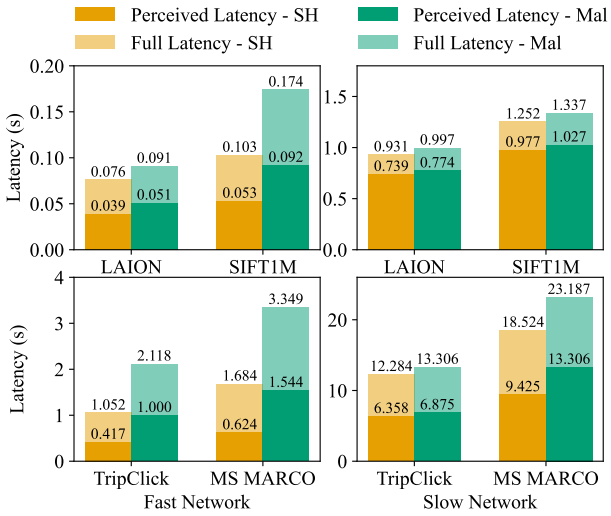

Figure 8: Breakdown of Compass's search latency. *SH* represents semi-honest and *Mal* represents malicious.

dark blue dashed line represents the accuracy of a brute-force search over the embeddings, indicating the highest achievable accuracy with the current embedding model. The light blue dashed line shows the plaintext search quality of TF-IDF, while the orange dashed line represents BM25 [14, 68], a widely used ranking algorithm. Overall, Compass outperforms both baselines in search quality, matching the accuracy of brute-force embedding search across all four datasets.

For Inv-ORAM, the accuracy is limited by the search algorithm and the size of the truncated list. While a smaller truncated list allows Inv-ORAM to achieve lower latency than Compass, its search quality is significantly worse. To reach a similar accuracy to TF-IDF, Inv-ORAM requires a much larger truncated list, exceeding 10,000 on MS MARCO, which significantly increases latency.

On the MS MARCO dataset, the HE-Cluster's accuracy closely matches the search quality of plaintext TF-IDF. However, on TripClick, the HE-Cluster's accuracy is only half that of TF-IDF. This discrepancy is likely due to the health-related content of TripClick, where text sequence matching is particularly effective for identifying health-related terms. Compared to HE-Cluster, Compass is orders of magnitude faster in search latency. While clustering reduces communication costs to sublinear, the server's computation cost remains linear. Although latency can be further improved with parallelization using more CPUs and instances, this approach is expensive and difficult to scale for multiple users.

We also evaluate the throughput of Compass by increasing the number of clients. On the user-scale dataset LAION, we achieve 34.8 queries per second on the *fast* network and 4.3 queries per second on the *slow* network.

6.3.1 Latency breakdown

In Fig. 8, we present the latency breakdown of Compass across all datasets under various network settings and security assumptions. We report both user-perceived latency and full latency. The user-perceived latency is primarily due to ORAM access operations, while the impact of distance computation and graph traversal, excluding ORAM accesses, is minimal—taking less than 0.02 seconds even on the largest dataset, MS MARCO. The full latency includes additional time spent on ORAM eviction. With our lazy eviction technique, the user-perceived latency, as highlighted in the figure, saves 1.2 - 2.2× compared to full latency.

In the top right of Fig. 8, for smaller datasets like LAION and SIFT1M on the *slow* network, the eviction time is faster than ORAM access. This is because graph traversal involves multiple rounds of ORAM accesses, while eviction in Compass only requires a single round and in this setting, the network round trip is the bottleneck. In the bottom left of Fig. 8, for larger datasets like TripClick and MS MARCO on the *fast* network, the eviction takes slightly longer. This is due to the larger ef required by these datasets, resulting in more blocks in the stash that need to be evicted, which increases the time spent finding available paths for each block. In the other two cases, the bottleneck is on communication bandwidth. Since the number of bytes transferred between the server and the client is the same between access and eviction, the perceived latency is about half of the full latency. The cost of supporting malicious security is more significant on the *fast* network, increasing as the size of the dataset grows. Overall, under the *fast* network, the user-perceived latency across all four datasets is around or less than 1 second. However, under the *slow* network, the user-perceived latency exceeds 5 seconds for TripClick and MS MARCO, which may be impractical for real-world applications. One key contributing factor is the size of the dataset. MS MARCO is a web-scale dataset with 8 million entries, significantly larger than the cloud storage a user would typically have. For smaller datasets like LAION and SIFT1M, the search latency remains under 1 second on *slow* network. Another factor is the embedding dimensionality. Both datasets have 768-dimensional embeddings which increase pressure on the network bandwidth. This can be optimized by using a model with a smaller embedding dimension or by applying dimension reduction techniques.

6.3.2 Communication

Tab. 3 presents the per-query communication costs, namely the amount of data transferred between the server and client, and the number of network round trips. For Inv-ORAM, we fix the truncated document list size at 10,000 for reasonable search quality. For Compass, the additional cost of transferring hashes in the malicious setting is negligible. The communication cost for LAION and SIFT1M in Compass is similar, despite SIFT1M having 10× more vectors. This

	Compass			HE-Cluster		Inv-ORAM	
	SH	Mal	RT		RT		RT
LAION	19.37	19.40	8	225.5	1	-	-
SIFT1M	19.27	19.54	10	737.2	1	-	-
TripClick	380.2	381.3	14	929.2	1	197.1	1
MS MARCO	562.4	564.5	16	2262.5*	1	252.1	1

Table 3: Comparison of communication cost (MB) and round trips per query. *SH* represents semi-honest and *Mal* represents malicious. * indicates the result is extrapolated.

is because LAION's embedding dimension is 4x larger than SIFT1M. Compass achieves less communication cost than the HE-Cluster baseline. This is partially because the communication is (ploy-)logarithmic in Compass and $O(\sqrt{N})$ in HE-Cluster. The FHE scheme used in the HE-Cluster also produces larger ciphertexts. Compared to Inv-ORAM, Compass requires roughly 2× communication bandwidth.

The main difference is the network round trips. Both baselines are single-round protocols, whereas Compass requires multiple rounds of interactions. We justify this for two reasons: First, Compass targets matching plaintext search quality, with the additional round trip overhead adding only about one second in the slower network settings. Additionally, when dealing with a higher network latency, the number of round trips could be reduced by tuning the parameter or at the cost of minor accuracy loss. Second, our speculative greedy search algorithm ensures a minimal increase in round trips relative to dataset size growth. For example, Compass adds only two extra round trips in MS MARCO compared to TripClick, despite MS MARCO containing 6× more embeddings.

6.3.3 Memory Consumption

We report the breakdown of memory consumption in Tab. 4. In the second column, we show the size of plaintext embeddings as a reference. Similar to the communication section, we report the server-side memory consumption under both malicious and semi-honest settings. On the server side, we require 2.2-3.1× memory compared to the plaintext embedding. The main overhead here comes from the Path ORAM, as a certain amount of dummy blocks are necessary for a good eviction rate. This part of the overhead can be further reduced by moving the ORAM tree from memory to disk. On the client side, the memory overhead consists of three parts, cached layers in the HNSW graph, quantized hints, and the position map. The size of quantized hints is only 1% of the size of the original embeddings. In our experiments, all but the last two layers of the graph are cached locally on the client. This part of memory overhead can be further reduced by caching fewer layers on the client side and paying several extra round trips to retrieve required nodes in un-cached layers from the server. For a web-scale dataset such as MS MARCO, Compass only requires approximately 334MB of client memory For smaller datasets that are closer to what users would have, the memory consumption will be less than a hundred megabytes.

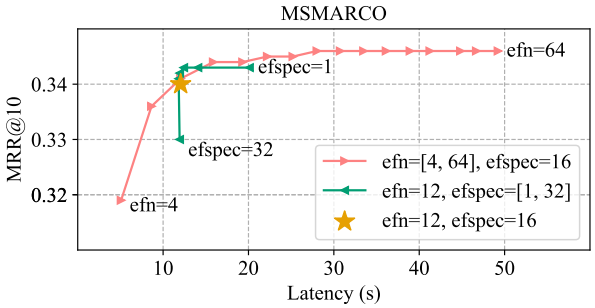

Figure 9: Ablation study on Directional Neighbor Filtering and Speculative Greedy Search

HE-Cluster requires significantly larger server-side storage because of larger HE ciphertexts. It requires more than 1TB of memory on MS MARCO and it doesn't fit into our server instance. Therefore, we extrapolate the results. Inv-ORAM requires less server-side memory but larger client-side memory for the position map and keyword document mapping.

6.3.4 Insert & Delete

We now briefly discuss how Compass compares to two baselines on insert and delete. For HE-Cluster, it's nontrivial to securely insert or delete a document without streaming the whole dataset. Otherwise, the server may learn which cluster this document belongs to. The truncation technique used for search in Inv-ORAM can be similarly applied to insert, but not delete, as the client has to fetch every keyword's document list to perform complete deletion, which incurs significant overhead. The insert latency of Inv-ORAM is similar to the search latency in Fig. 7. As mentioned in the prior section, deletion in the HNSW graph can be achieved by first searching the node and marking this node as deleted, incurring the same cost as a search. Insertion, on the other hand, is similar to search but with the default candidate list size set to 40. For example, MS MARCO's search candidate list size is 224. Compass insertion latency on MS MARCO dataset over *slow* network is 5.6 seconds, while the search takes 18.5 seconds.

6.4 Ablation Studies

We evaluate the effectiveness of our techniques on the MS MARCO dataset under the slower network configuration. Specifically, we perform three sets of experiments and compare their search quality and latency with the default setting we used in the prior sections. Fig. 9 shows the result for the first two experiments.

The first set of experiments fixes the size of the speculation set to 16 and varies the directional filter size from 4 to 64.

	Embed. Size	Compass Server		Compass Client				HE-Cluster		Inv-ORAM	
		SH.	Mal.	Hints	PosMap	Graph	Total	Server	Client	Server	Client
	GB	GB	GB	MB	MB	MB	MB	GB	MB	GB	MB
LAION	0.19	0.42	0.43	3.6	0.4	0.2	4.2	9.1	0.8	-	-
SIFT1M	0.48	1.16	1.17	7.8	3.9	0.8	12.4	16.1	0.7	-	-
TripClick	4.37	9.80	9.83	47.3	6	5	58.3	249	4.5	10	606
MS MARCO	25.33	78.37	78.62	270.6	34.8	28.4	333.8	1364*	11.2*	10	810

Table 4: Comparison of memory consumption. * indicates the result is extrapolated.

When directional filtering is completely removed ($efn = 64$), we observe roughly 5× slower search latency compared to the default setting. This is because a smaller filter size can help us remove more irrelevant neighbors and therefore save bandwidth. However, the filter size cannot be too small. When we decrease the filter size to 4, it shows a significant accuracy drop compared to the default setting.

The second set of experiments fixes the directional filter size and varies the size of the speculation set from 1 to 32. When the speculation size is 1, it means the speculative greedy search is removed during the search. We observe a 2× search latency when the speculative greedy search is completely removed ($efspec = 1$). This is because a larger speculation size can effectively reduce the network roundtrips. Similarly, if the speculation size is too large, the search quality will drop and the search latency doesn't improve as network bandwidth becomes the bottleneck.

In the third experiment, we replace our HNSW-tailored ORAM with the vanilla Path ORAM. In this case, the search latency increases by 20× to 209 seconds. This is mainly due to the increase in network round trips, from 16 to 2500.

7 Discussion & Related Work

Private RAG: Besides traditional document search, Compass can also be applied to Private Retrieval Augmented Generation (RAG) systems [46]. RAG is a technique used to enhance the responses of generative AI models by incorporating relevant information retrieved from external sources, often stored in a vector database. The relevance of this information is determined by the similarity between the embedding of the query (or prompt) and the vectors in the database. Unlike public RAG, private RAG systems are designed to retrieve and generate content based on sensitive, internal databases specific to an organization or individual. These databases could include personal cloud drives, workspaces, or proprietary information in healthcare or finance. With Compass, individuals or organizations can securely retrieve relevant data from outsourced, encrypted databases. Together with the prompt, this part of the data can be fed into a model ran locally or jointly with a server with recent secure inference techniques [31, 34, 38, 55, 60, 78].

Lexical encrypted search: Searching on encrypted data has been a rich and fruitful line of work. Most of the work in encrypted search so far has focused on keyword/lexical search. As discussed in §1, these works do not support the state-of-the-art semantic search like Compass does. Furthermore, to achieve high efficiency, many works reduce security in the following ways, whereas Compass does not make these compromises:

- Leaking access patterns [11, 12, 15, 32, 41, 45, 57, 77, 80], which can be exploited by leakage-abuse attacks [10, 35, 42, 49, 62, 63, 88].
- Assuming trusted hardwares [5, 54, 81] that can be exploited by a wide range of side-channel attacks [9, 43, 84, 87].
- Assuming non-colluding servers or at least one trusted server [17, 18, 74].

Encrypted semantic search systems Encrypted search over embeddings is a nascent line of work. HERS [24] uses fully homomorphic encryption (HE) to perform a linear search at the server over embeddings, which results in a high overhead. Tiptoe [33] performs a more efficient linear scan over embeddings by crucially relying on the data at the server being public / not encrypted, so Tiptoe does not provide a solution for searching over encrypted data (nor for malicious security, integrity protection, and sublinear search) in contrast to Compass. Furthermore, Tiptoe's clustering technique reduces search quality as discussed in §6.

Other works attempt to build sublinear indices over embeddings, but sacrifice security or efficiency. In some works [6, 89], the index traversal is not privacy-preserving and leaks the query information if the server has some knowledge about the plaintext. SANNS [13]'s performance overhead is high because it combines multiple heavy-weight cryptographic tools such as lattice-based homomorphic encryption, distributed oblivious RAM, and garbled circuits.

Preco [74], Riazi et al. [70], and Wu et al. [85] weaken security by relying on a two-server model that are not both compromised. Preco mentions the potential of removing the non-colluding assumption by adopting a single-server PIR scheme, but estimates the resulting performance to become orders of magnitude slower.

8 Conclusion

Compass is a search system over encrypted embedding data that achieves 1) comparable accuracy to the state-of-the-art

search algorithm in plaintext, 2) strong security guarantee against malicious attackers, and 3) practical user-perceived latency at low server operation costs. Compass achieves these properties through a novel search index that co-designs the traversal of the HNSW graph on top of Oblivious RAM via three techniques.

9 Ethics and Open Science

The primary goal of this work is to enable efficient and accurate semantic search over encrypted personal data. We treat encryption schemes and embedding models as black boxes. Misuse of encryption or vulnerabilities (e.g., side-channel attacks) could compromise data confidentiality. The embeddings used in this work are generated by language models. Although encryption protects the data, biases in the models may introduce ethical concerns related to fairness and discrimination in search results. However, these issues in the embedding model are orthogonal to Compass's contribution because Compass can support a wide range of embedding models. Of the four datasets used, three (LAION, SIFT1M, and MS MARCO) are publicly available, while the TripClick dataset is accessible for non-commercial research purposes. In this work, we only report search accuracy and latency, without revealing any content or aggregate results of the original dataset.

We plan to open-source and include the link of the implementation in the final version of the paper.

References

[1] OpenSSL EVP, 2016. https://wiki.openssl.org/index.php/EVP.

[2] iCloud keychain security overview, 2021. https://support.apple.com/guide/security/icloud-keychain-security-overview-sec1c89c6f3b.

[3] Semantic search, 2024. https://www.elastic.co/guide/en/elasticsearch/reference/current/semantic-search.html.

[4] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Coeus: A system for oblivious document ranking and retrieval. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 672–690, 2021.

[5] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[6] Daisuke Aritomo, Chiemi Watanabe, Masaki Matsubara, and Atsuyuki 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*, pages 403–412, 2019.

[7] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, January 2020.

[8] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. Ms marco: A human generated machine reading comprehension dataset. *arXiv preprint arXiv:1611.09268*, 2016.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679, 2015.

[11] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *Cryptology ePrint Archive*, 2014.

[12] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373. Springer, 2013.

[13] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. {SANNS}: Scaling up secure approximate {k-Nearest} neighbors search. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2111–2128, 2020.

[14] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, Daniel Campos, and Jimmy 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*, pages 1566–1576, 2021.

[15] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings*

of the 13th ACM conference on Computer and communications security, pages 79–88, 2006.

[16] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. Reflections on trusting distributed trust. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 38–45, 2022.

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

[18] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2450–2468. IEEE, 2022.

[19] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horvárth, and Tibor Jager. Security analysis of the whatsapp end-to-end encrypted backup protocol. *Cryptology ePrint Archive*, 2023.

[20] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. *Cryptology ePrint Archive*, 2019.

[21] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*, pages 371–406. Springer, 2018.

[22] Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1053–1067, 2017.

[23] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[24] Joshua J Engelsma, Anil K Jain, and Vishnu Naresh Boddeti. Hers: Homomorphically encrypted representation search. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, 4(3):349–360, 2022.

[25] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–116, 2015.

[26] Benny Fuhry, HA Jayanth Jain, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 438–450. IEEE, 2021.

[27] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*, pages 563–592. Springer, 2016.

[28] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, 2009.

[29] Eu-Jin Goh. Secure indexes. *Cryptology ePrint Archive*, 2003.

[30] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[31] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. *Advances in neural information processing systems*, 35:15718–15731, 2022.

[32] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1028–1039, 2014.

[33] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 396–416, 2023.

[34] Xiaoyang Hou, Jian Liu, Jingyu Li, Yuhan Li, Wen-jie Lu, Cheng Hong, and Kui Ren. Ciphergpt: Secure two-party gpt inference. *Cryptology ePrint Archive*, 2023.

[35] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Ndss*, volume 20, page 12. Citeseer, 2012.

[36] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[37] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[38] Wen jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. BumbleBee: Secure two-party inference framework for large transformers. Cryptology ePrint Archive, Paper 2023/1678, 2023.

[39] Ian T Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[40] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*, pages 258–274. Springer, 2013.

[41] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976, 2012.

[42] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[44] Butler W Lampson. Lazy and speculative execution in computer systems. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 1–2, 2008.

[45] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis aegis: A mimicry privacy {Shield–A}{System's} approach to data privacy on public cloud. In *23rd usenix security symposium (USENIX Security 14)*, pages 33–48, 2014.

[46] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[47] Xiang Li, Yunqian Luo, and Mingyu Gao. Bulkor: Enabling bulk loading for path oram. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 103–103. IEEE Computer Society, 2024.

[48] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Popa. The deployment dilemma: Merits & challenges of deploying mpc, 2023.

[49] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[50] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

[51] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 719–734. Springer, 2013.

[52] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, August 2018. arXiv:1603.09320 [cs].

[53] Microsoft. The science behind semantic search: How ai from bing is powering azure cognitive search, 2021. https://learn.microsoft.com/en-us/azure/search/semantic-search-overview.

[54] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE symposium on security and privacy (SP)*, pages 279–296. IEEE, 2018.

[55] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[56] Muhammad Naveed. The fallacy of composition of oblivious ram and searchable encryption. *Cryptology ePrint Archive*, 2015.

[57] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy*, pages 639–654. IEEE, 2014.

[58] Rafail Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523, 1990.

[59] Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

[60] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. Cryptology ePrint Archive, Paper 2023/1893, 2023.

[61] Pinecone. Vector search: Hierarchical navigable small worlds. https://www.pinecone.io/learn/series/faiss/hnsw/.

[62] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 354–369. IEEE, 2020.

[63] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1341–1352, 2016.

[64] Preveil: Encrypted email and file sharing. https://www.preveil.com/.

[65] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[66] Juan Ramos. Using TF-IDF to Determine Word Relevance in Document Queries.

[67] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[68] Navid Rekabsaz, Oleg Lesota, Markus Schedl, Jon Brassey, and Carsten 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*, pages 2507–2513, 2021.

[69] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Marten Van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2013.

[70] M. Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. Sub-linear privacy-preserving near-neighbor search. Cryptology ePrint Archive, Paper 2019/1222, 2019.

[71] Panagiotis Rizomiliotis and Stefanos Gritzalis. Oram based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, pages 65–76, 2015.

[72] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 198–217. IEEE, 2016.

[73] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs, 2021.

[74] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 911–929. IEEE, 2022.

[75] Signal Messenger. https://signal.org/.

[76] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search, May 2022. arXiv:2205.03763 [cs].

[77] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE symposium on security and privacy. S&P 2000*, pages 44–55. IEEE, 2000.

[78] Wenting Zheng Srinivasan, PMRL Akshayaram, and Popa Raluca Ada. Delphi: A cryptographic inference service for neural networks. In *Proc. 29th USENIX secur. symp*, volume 3, 2019.

[79] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[80] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. *Cryptology ePrint Archive*, 2013.

[81] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment*, 14(6):1019–1032, 2021.

[82] Telegram Messenger. https://telegram.org/.

[83] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management: 7th VLDB Workshop, SDM 2010, Singapore, September 17, 2010. Proceedings 7*, pages 87–100. Springer, 2010.

[84] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice, 2020.

[85] W. Wu, U. Parampalli, J. Liu, and M. Xian. Privacy preserving k-nearest neighbor classification over encrypted database in outsourced cloud environments. In *World Wide Web*, 2019.

[86] Zhiqiang Wu and Rui Li. Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In *NDSS*, 2023.

[87] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[88] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of {File-Injection} attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, 2016.

[89] Qian Zhou, Hua Dai, Yuanlong Liu, Geng Yang, Xun Yi, and Zheng Hu. A novel semantic-aware search scheme based on bci-tree index over encrypted cloud data. *World Wide Web*, 26(5):3055–3079, 2023.

[90] Jingchen Zhu, Guangyu Sun, Xian Zhang, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: Batching oram requests to remove redundant memory accesses. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2279–2292, 2019.

A Layer-wise Caching

Layer	4	3	2	1	0
LAION	-	1.915	2.66	2.826	12
SIFT1M	1	1.99	2.63	2.58	32
TripClick	1	2.04	1.82	1.64	192
MS MARCO	1.83	2.68	2.44	2.42	224

Table 5: Average number of search steps in each layer. Layer 0 is the bottom layer and therefore the number of search steps is defined by ef

In HNSW, the upper-layer search uses a dynamic candidate list of size one, making it non-trivial to apply the Speculative Greedy Search technique directly. As a result, from the server's perspective, the memory footprint for upper-layer searches differs from that of the bottom layer. For security, it is necessary to set a separate bound for search steps in the upper layers when not all upper layers are cached locally.

To figure out how many search steps in each layer are sufficient, we perform the plaintext HNSW search on the entire query set and collect the average number of search steps in each layer, as reported in Tab. 5. Surprisingly, unlike the bottom layer, the upper-layer search patterns are similar across all datasets, with an average of 1 to 3 steps per layer. Based on this observation, we empirically found that a single search step in the second last layer is enough for search quality when we cache all but the last two layers.

B ORAM Stash Size Analysis

In this section, we present an empirical analysis of the impact of Compass's ORAM modification on stash size. In Fig. 10 and Fig. 11, we report the distribution of stash usage before and after eviction. The orange line represents the median and the green dashed line represents the average. We collect the data for each dataset by searching over the entire query set.

Fig. 10 reports the upper bound of the stash usage during a search. For MS MARCO, the stash contains up to 35,000 blocks in the worst case, approximately 110 MB. For smaller datasets like SIFT1M and LAION, the worst-case stash size is under 4 MB.

In Fig. 11, the stash size after eviction is mostly zero for LAION, TripClick, and MS MARCO. For SIFT1M, there are more data points of non-zero stash size, with peak usage reaching up to 45 blocks. However, the average stash size after the eviction remains close to zero for SIFT1M.

C Security Proof

In this section, we provide a formal treatment of the security of Compass.

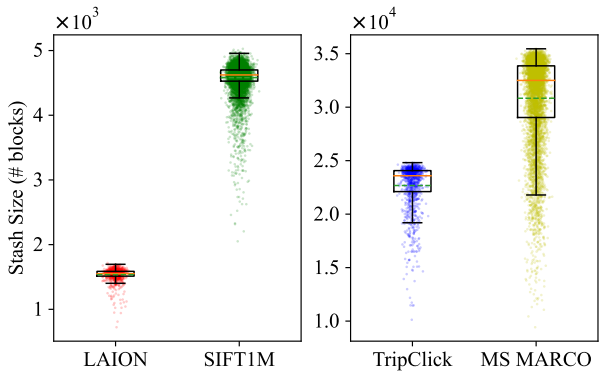

Figure 10: Stash size before the eviction

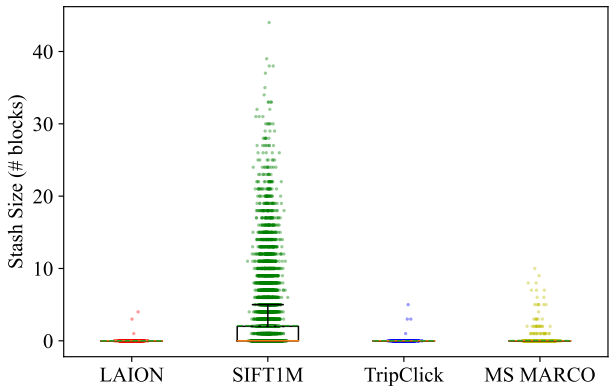

Figure 11: Stash size after the eviction

Our proof strategy is to consider two security games that are the original security game with only one of the adversary winning conditions and prove that an adversary cannot fulfill each condition separately more than the threshold allowed. Proving these two statements suffices to prove our theorem in our setting.

In this proof, as we empirically demonstrated in App. B, we assume that the main memory is enough for the stash during the execution of the protocol. We leave the formal analysis of Compass's stash size for future work. For simplicity, our proof focuses on Compass's protocol within a single layer.

C.1 Condition 1

We first prove that the probability of the adversary winning in condition 1 is less than half plus negligible. Our proof strategy is by showing that what the adversary observes when $b = 0$ is computationally indistinguishable from what they observe when $b = 1$.

We start by defining what the adversary can observe during the game. During the security game, the challenger executes a sequence of queries:

$$Q_b = (q_{1,b}, q_{2,b}, ..., q_{i,b}, ...) \quad (1)$$

Let $O_{i,b}$ be the sequence of ORAM operations for query $q_{i,b}$. According to Compass's protocol, each query consists of a sequence of batch access operations and an eviction operation at the end. Let m be the number of batch access operations in a query and we have

$$O_{i,b} = \mathsf{op}_{\mathsf{access}}(B_1^{i,b}, s_1^{i,b}), ..., \mathsf{op}_{\mathsf{access}}(B_m^{i,b}, s_m^{i,b}), \mathsf{op}_{\mathsf{evict}}, \quad (2)$$

where $\mathsf{op}_{\mathsf{access}}(B_j^{i,b}, s_j^{i,b})$ represents the jth batch access operation of a set of blocks $B_j^{i,b}$ of size $s_j^{i,b}$. Notice that, unlike classical Path ORAM, each batch access here reads multiple paths and doesn't write back. $\mathsf{op}_{\mathsf{evict}}$ represents the eviction operation. All the paths being read before are assigned with newly evicted blocks from the stash, encrypted, and written back to the server.

Let $P_j^{i,b}$ be the set of paths read during $\mathsf{op}_{\mathsf{access}}(B_j^{i,b}, s_j^{i,b})$, $P_w^{i,b}$ be the set of paths written back to the server during $\mathsf{op}_{\mathsf{evict}}$, and $\mathcal{T}_{i,b}$ be the trace (or memory footprint) the adversary observes on the server side for query $q_{i,b}$. We have

$$\mathcal{T}_{i,b} = \mathsf{t}_{\mathsf{read}}(P_1^{i,b}), ..., \mathsf{t}_{\mathsf{read}}(P_m^{i,b}), \mathsf{t}_{\mathsf{write}}(P_w^{i,b}), \quad (3)$$

where $\mathsf{t}_{\mathsf{read}}(P_j^{i,b})$ represents reads on the paths in $P_j^{i,b}$, $\mathsf{t}_{\mathsf{write}}(P_w^{i,b})$ represents writes on the paths in $P_w^{i,b}$. According to Compass's protocol, the lazy eviction evicts all the paths being read in prior accesses, therefore we have $P_w^{i,b} = P_1^{i,b} \cup ... \cup P_m^{i,b}$.

Having defined what the adversary can observe during the game, we now prove that the following lemmas hold for Compass. In the following discussion, we will assume that the number of paths accessed during a request is below the total number of paths on the server. In the case of a small dataset, if the number of requested paths exceeds the total number of paths on the server side, Compass is equivalent to the naive solution: streaming the entire database from the server, shuffling it locally, and then streaming it back.

Lemma 1. *For any pair of* $(\mathsf{op}_{\mathsf{access}}(B_j^{i,b}, s_j^{i,b}), \mathsf{t}_{\mathsf{read}}(P_j^{i,b}))$*, Compass ensures:*

- $|P_j^{i,b}| = s_j^{i,b}$*, where* $|P_j^{i,b}|$ *is the size of* $P_j^{i,b}$
- $P_j^{i,b} \cap (P_1^{i,b} \cup ... \cup P_{j-1}^{i,b}) = \varnothing$
- *paths in* $P_j^{i,b}$ *are randomly sampled from the unvisited paths in current request*

Proof. In Compass, for a batch access operation $\mathsf{op}_{\mathsf{access}}(B_j^{i,b}, s_j^{i,b})$, the client-side ORAM controller iterates over each block in $B_j^{i,b}$ to determine the corresponding path assigned to each block. If a block has been accessed previously or belongs to a path that has already been fetched in a previous access, a new path is randomly sampled from the unvisited paths. Such design ensures that the size of $P_j^{i,b}$ equals $s_j^{i,b}$ and there's no intersection between $P_j^{i,b}$ and the prior path set in the same request.

Each path in $P_j^{i,b}$ is either a path, unvisited, assigned to one of the requested blocks according to the position map or a new path randomly selected from the current set of unvisited paths. Since every block is reassigned to a completely new random path after being accessed, the third point of Lemma 1 holds. □

Lemma 2. *For any pair of* $q_{i,0}$ *and* $q_{i,1}$*, if they are of the same type, Compass ensures the corresponding traces* $\mathcal{T}_{i,0}$ *and* $\mathcal{T}_{i,1}$ *are of* ***the same structure****.*

Before we prove this lemma, we first introduce the definition of **the same structure**.

Definition 1. *Two traces are structurally the same if they have*

- *the same number of operations*
- *the same type of operation at the same positions*
- *the same number of paths operated by the operation at the same positions*

For example, consider a binary tree with 8 paths, which is labeled from 0 to 7, and the following 5 example traces:

$$\begin{aligned}
\mathcal{T}_1 &= \mathsf{t}_{\mathsf{read}}(\{1,2\}), \mathsf{t}_{\mathsf{write}}(\{1,2\}) \\
\mathcal{T}_2 &= \mathsf{t}_{\mathsf{read}}(\{1,2\}) \\
\mathcal{T}_3 &= \mathsf{t}_{\mathsf{read}}(\{1,2\}), \mathsf{t}_{\mathsf{read}}(\{3,4\}) \\
\mathcal{T}_4 &= \mathsf{t}_{\mathsf{read}}(\{3,4,5\}), \mathsf{t}_{\mathsf{write}}(\{3,4,5\}) \\
\mathcal{T}_5 &= \mathsf{t}_{\mathsf{read}}(\{3,4\}), \mathsf{t}_{\mathsf{write}}(\{3,4\})
\end{aligned} \quad (4)$$

If we take $\mathcal{T}_1$ as a reference, among $\mathcal{T}_2$ to $\mathcal{T}_5$, only $\mathcal{T}_5$ is considered to have the same structure as $\mathcal{T}_1$. $\mathcal{T}_2$ has a different length compared to $\mathcal{T}_1$. The second operation in $\mathcal{T}_3$ is a read while the second operation in $\mathcal{T}_1$ is a write. $\mathcal{T}_4$ has the same length and operation sequence as $\mathcal{T}_1$. However, each operation in $\mathcal{T}_4$ operates on 3 paths while each operation in $\mathcal{T}_1$ operates on 2 paths.

Proof. Now we start the proof for Lemma 2. Take a search request as an example. As shown in Algorithm 1, a search consists of n iterations (line 5) and each iteration contains one batch access (line 14). Therefore, for each search request, the number of operations in the trace is $n+1$, where the extra 1 comes from the final eviction. Since n is initialized with $\lceil ef/efspec \rceil$, given a fixed set of parameters param, the number of operations in the trace is fixed, and the trace always starts with n reads at the beginning and 1 write at the end. Lastly, we need to show that the number of paths being operated is the same for the same position. For

each batch access, the number of requested blocks is t (line 12 - 14), which is defined as $efspec * efn$. Based on Lemma 1, we know that the corresponding read operation operates on t paths. Similarly, the final write operation operates on $n*t = efspec * efn * \lceil ef/efspec \rceil$ paths. Therefore, Lemma 2 holds.

□

With Lemma 1 and Lemma 2, we have:

- the sequence pair $\mathcal{T}_{i,0}$ and $\mathcal{T}_{i,1}$ are structurally the same, and
- paths related to these traces are sampled uniformly at random without replacement.

The final step in our proof here is to invoke the security guarantees of the IND-CCA2 encryption scheme. For each pair of requests $q_{i,0}$ and $q_{i,1}$, the corresponding traces, $\mathcal{T}_{i,0}$ and $\mathcal{T}_{i,1}$, are computationally indistinguishable from the adversary. By the definition of our security games, if the challenger didn't abort, we know Q_0 and Q_1 are of the same length and they have the same type of requests at the same positions. Therefore, the execution trace of Q_0 is computationally indistinguishable from the execution trace of Q_1 and the probability of the adversary winning condition 1 is less than half plus negligible.

C.2 Condition 2

We then show that the probability of the adversary winning in condition 2 is negligible.

Lemma 3. *Assuming that the underlying hash function $\mathcal{H}$ is collision-resistant, then the probability of the adversary winning through condition 2 is negligible.*

Proof. In Compass's protocol, the server operates as remote storage, either sending the client buckets at requested locations or updating its local storage based on the client's messages. Excluding the scenario where the server becomes unresponsive, the only way the server can deviate from the protocol is by sending the client an altered bucket content. To ensure the integrity and freshness of the buckets that the client reads from the server, Compass uses a Merkle Tree that is built on collision-resistant hash functions. By the definition of collision resistance hash functions, the probability of finding two different inputs that produce the same hash output is negligible. It is already well-proven by prior works that a Merkle Tree built with such hash functions ensures data integrity and the probability for an adversary to change the data in a way that the root hash remains the same is negligible. Although Compass uses an HNSW graph as the underlying data structure rather than a tree, Compass ensures that each node's data, including the coordinates and neighbor list, is stored inside one of the buckets that form a binary tree on the server side. Therefore, the properties of the Merkle hash tree follow directly, which means the probability of the server successfully forging a response that passes the integrity check on the client's side is also negligible.

□