Compass: Encrypted Semantic Search with High Accuracy

Jinhao Zhu
UC Berkeley
jinhao.zhu@berkeley.edu

Liana Patel
Stanford University
lianapat@stanford.edu

Matei Zaharia
UC Berkeley
matei@berkeley.edu

Raluca Ada Popa
UC Berkeley
raluca.popa@berkeley.edu

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. Compass also enables privacy-preserving RAG where both the RAG database and the query are protected.

Compass's search index contributes a novel way to traverse the search graph in Hierarchical Navigable Small Worlds (HNSW), a top performing vector nearest neighbor search, using 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 few seconds and is orders of magnitude faster than a baseline for encrypted embeddings search.

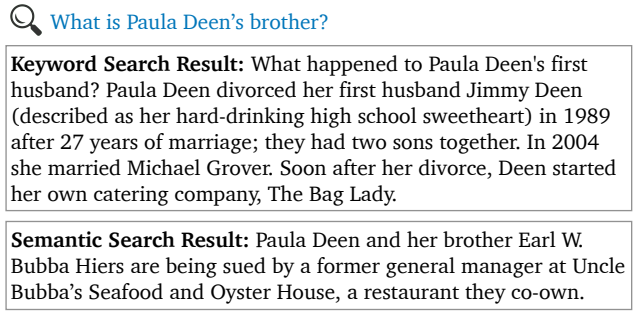

Figure 1. Example of top-1 search results with the keyword search using TF-IDF (top) versus semantic search in Compass (bottom) on the MS MARCO dataset. Keyword search looks up each keyword individually and intersects them, but the term "brother" by itself is generic and produces too many results. In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

1 Introduction

An increasing number of user data systems have adopted end-to-end encryption because of its strong security properties: the server, even if compromised, has access to only encrypted data; for an encrypted data item, only the users with legitimate access to it have the key to decrypt this data item on their local device. Examples of such systems include WhatsApp, iCloud Backups, Telegram, Signal, and PreVeil [2, 19, 59, 68, 74]. In this setting, there has been a decades-long rich line of work on encrypted search [15, 17, 20–22, 28, 30, 38, 39, 51, 53, 65, 67, 70, 72, 78]: enabling the server to search on the encrypted data without learning the data or the search query. Despite much progress, two challenges remain: reduced security and low search accuracy.

(1) Security: The desiderata is to protect the data, query and query results – including search access patterns (*which* data items are accessed) – against a *fully* compromised server.

1. *Leaky practical search.* A long line of work enables the server to learn some information about the query or data [11, 12, 15, 32, 39, 43, 53, 70, 72], such as search access patterns, to provide fast search. However, leakage-abuse attacks [10, 34, 40, 47, 57, 58, 80] can reconstruct a significant amount of data or the query from this leakage.
2. *Search with partial server trust.* A line of work assumes trusted hardware enclaves at the server [5, 51, 73], but hardware enclaves suffer from a wide range of side channels [9, 41, 79], including some that can entirely subvert remote attestation and their security [76]. Another line of work [17, 18, 67] considers the logical server is formed from 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, 46], and an attacker can still compromise both of these servers.

(2) Accuracy: Most of the works above have a significantly lower search accuracy than state-of-the-art plaintext search systems. The reasons are that they implement a *lexical* search which is less accurate than *semantic* search used by state-of-the-art search systems today. For example, some works above [4, 15, 17, 38, 39, 51, 70, 75, 78] implement an inverted index that maps a keyword to a list of documents. This data structure answers accurately single-keyword searches, but a lot of users today search for more complex queries, such as expressions or questions. State-of-the-art search systems such as Bing, and Elasticsearch [3, 50] implement semantic search. Semantic search is more accurate because it understands the intent and contextual meaning of the search query. It further extends search capabilities beyond text, to various unstructured data types such as audio, images, and videos.

In this work, through our system ***Compass***[1], we show that one can design an efficient search system over encrypted

[1]A compass helps us search for a destination despite low visibility, and one navigates using a compass similarly to how our system navigates in semantic space. It is also an acronym for **C**ryptographic **O**blivious **M**aliciously **P**rotected and **A**ccurate **S**emantic **S**earch.

data that achieves (1) the security desiderata, and (2) a highly-accurate semantic search. Namely, Compass does not leak access patterns at the server, and protects against a fully untrusted logical server. Compass additionally provides integrity guarantees for the search results against a malicious server compromise as discussed in §4.8.

Private RAG. This setup enables Compass to excel not only in traditional document search but also in Private Retrieval Augmented Generation (RAG) systems [44]. RAG is a technique used to enhance the responses of generative AI models by incorporating pertinent information retrieved from external sources, often stored in a vector database. The relevance of this information is determined by the similarity between the embedding vectors of the query (or prompt) and the vectors in the database. Unlike public RAG systems that draw from publicly available data, 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 sectors like healthcare or finance. With Compass, individuals or organizations can seamlessly and securely retrieve relevant data from outsourced, encrypted databases, ensuring both accuracy and confidentiality during the process.

In Compass, search and insert time are both theoretically (poly)logarithmic in expectation[2] and empirically efficient as we show in §6. Importantly, Compass's search quality is on par with state-of-the-art unencrypted search systems. Figure 1 shows an example of a search result from Compass versus from a keyword search system like in prior encrypted search work.

Our goal is ambitious: to execute, over encrypted data, a powerful *state-of-the-art search index for embeddings – HNSW [49].* Embeddings are numeric representations of information that underpin modern machine learning, including generative AI. HNSW is a top-performing high-accuracy index for vector similarity search, used in modern plaintext search [56]. The goal here is to support HNSW securely and efficiently on encrypted data without reducing its accuracy. However, HNSW performs a complex walk in embedding space that would be too inefficient to perform with fully homomorphic encryption (FHE) [29] or GarbledRAM [27].

Searching over encrypted embeddings is a nascient 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 garbledRAM), or have weak security ([6, 81] leak query information and Preco [67] assumes partial server trust), as we elaborate in §7.2.

In Compass, we observe that Oblivious RAM (ORAM) [54, 71] provides some particularly attractive properties for searching over encrypted embeddings: not only it protects access patterns from an untrusted ORAM server, but it supports sublinear search and *any embedding distance metric* enabling a wealth of state-of-the-art embeddings; the reason for the latter is that the ORAM controller locally decides the next step to take in an index walk, thus being able to compute locally distances in embedding space. Fig. 2 shows an overview of Compass's architecture leveraging ORAM.

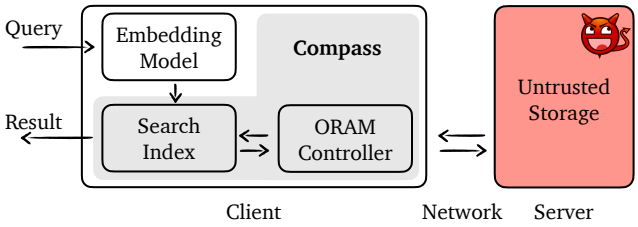

Figure 2. Compass Architecture

However, the fundamental challenge in supporting HNSW with ORAM is that HNSW assumes that the index and embeddings are stored in local memory and is optimized for performing a *many-hop walk locally.* Employing ORAM here transforms every step into a roundtrip to the server over the wide area network. Namely, a strawman solution to implement the search index is to make an ORAM request to the server to fetch every node visited by HNSW's search algorithm. HNSW is a multi-layer graph structure with fewer nodes and edges on the top layers and more nodes and edges on the bottom layers. Searches start from the top layer, follow links within each layer to find the locally nearest neighbor to the query, and then continue to the layer below. Unfortunately, this strawman is highly inefficient because it has:

- *High bandwidth consumption.* At every node, HNSW requires fetching each neighbor's embedding vector and computing its distance to the query vector. This means that to evaluate a single node in the walk, we have to perform tens to a hundred of ORAM requests.
- *A high number of roundtrips, resulting in high latency.* A walk in HNSW typically visits tens of nodes, each having tens to hundreds of neighbors that also are fetched, resulting in thousands of network round trips to achieve good recall on realistic 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. Our search index introduces 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 chunk of compressed 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

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

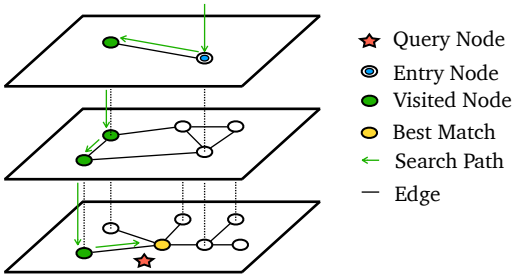

Figure 3. HNSW Graph Data Structure

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 3 popular datasets of different sizes and dimensions. Our techniques above provide up to 12× speedup over the strawman implementation of HNSW on top of ORAM above, and make encrypted embedding search with high accuracy practical for the first time. Our results indicate that Compass not only significantly outperforms baselines but also matches the accuracy of the state-of-the-art plaintext search algorithm across all datasets. Additionally, Compass achieves user-perceived latencies ranging from 0.15 to 2.21 seconds on a Local Area Network and from 0.9 to 10.3 seconds on a Wide Area Network, even for only a single-threaded server in the cloud.

Limitations. Compass makes multiple roundtrips to the ORAM server for each search (e.g., an average of 10–16 in our evaluation), unlike a vanilla unencrypted search which makes only one. While §6 shows that there are a wide range of settings in which the user-perceived latency is reasonable, 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 [49], a state-of-the-art graph-based *Approximate-Nearest-Neighbor (ANN)* method that has empirically demonstrated strong search performance [7, 69]. 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-level graph index that has bounded node-degrees.

2.1.1 HNSW search algorithm The HNSW search algorithm follows a simple iterative greedy search procedure. As shown in Figure 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 level of the HNSW graph. The search procedure iterates over each HNSW level, performing a greedy search, before dropping down to the level below to continue its search. On each upper level, 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 level below. Finally, once the bottom-most level is reached, the algorithm once again perform the greedy search, but increases the size of the dynamic candidate list to *efSearch* and greedily chooses K nodes, rather than a single node. The parameter *efSearch* is typically larger than K and controls the trade-off between the quality and efficiency of the search, with higher values of *efSearch* corresponding to higher quality at the cost of higher search latency, or reduced query throughput. In practice, one can choose the *efSearch* parameter for an application by empirically generating the accuracy-efficiency trade-off curve over a benchmark dataset and choosing the *efSearch* value that meets the application's target accuracy at minimal search latency.

2.2 Path ORAM

First introduced by Goldreich and Ostrovsky, Oblivious RAM (ORAM) [31, 54] allows a client to access encrypted data on remote storage without leaking actual access patterns. Path ORAM [71] 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 types of 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 blocks that have real data 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 who are storing their private documents on the server and are performing searches on it. Each client owns a separate set of documents stored on the server. Compass maintains a separate search index for each user.

Compass's API is as follows:

- INIT($\mathcal{D}$): Given a set of documents $\mathcal{D}$, initiate the index.
- SEARCH(Q, K) $\rightarrow$ LIST(DOCID, SCORE): Given Q, the embedding of the query, find the top K relevant documents and return a list containing the id of these documents with relevance scores.
- INSERT(F): Given a document F, generate the embedding set $\mathcal{E}$ and store both the encrypted document and its embeddings in the cloud. Depending on the size of the document and the model's context window, $\mathcal{E}$ may contain one or multiple embeddings.
- DELETE(F): Given a document F, remove the encrypted form of the document as well as the embeddings from the cloud.

When initializing the system, sequentially writing each block into ORAM is very inefficient. Instead, we can use the bulk loading protocol introduced in BULKOR [45]. We focus on enabling a user to search on their own data, and data sharing is not a focus of our system.

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 server machines.

The server is untrusted and can be maliciously compromised. Some prior works weaken the threat model by assuming that the server comprises of multiple logical servers where at least one is honest [17, 64, 67, 77]: we do not make this assumption, namely, the entire server is untrusted. We also do not rely on uncompromised hardware modules or hardware trust assumptions at the server unlike TEE-based systems [26, 51]. This means that the server in Compass can deviate from the protocol arbitrarily including launching a replay attack. Compass ensures that the server cannot learn the user query, whether this query is the same as a previous query ("query access patterns"), the length of the query, the document id-s returned by the query as well as their number, as well as which data items are touched by the search process. Compass does not hide the type of operations—whether a user is performing a search, insert, or delete—on the remote data. The server also cannot learn the data contents from the search index or modify the search results without detection by the client.

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 how large the data of each user is (e.g. how many documents or 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 the plethora of attacks [10, 34, 40, 47, 57, 58, 80] on encrypted search that leverage access patterns or size of results. Like much prior work [51], Compass does not protect against denial of service attacks from the server. The server may drop the requests from clients at any time or even delete 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 [66, 71], we provide an indistinguishability-based security definition. In our security game, there's a challenger 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, which means it will follow the protocol and the adversary is malicious, which means it 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 client challenger.

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 speculating set *efspec*, the size of the directional filter *efn* in our modified HNSW search.

The security game is as follows:

1. The challenger 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'.

The adversary $\mathcal{A}$ wins the game if the challenger did not abort and one of the following two conditions are met:

1. $b' = b$,
2. the sequence of queries $(q_{i,b}, r_i)$ on the initial dataset D_b is an incorrect execution of plaintext Compass's search algorithm with param.

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

Table 1. Summary of Notation

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.

4 Compass's Search Index

In this section, we describe Compass's index. For simplicity, we primarily focus on the search function since the insert algorithm in HNSW closely resembles that of the search function, and the techniques described in this section are applicable to both.

4.1 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.

4.2 Data Layout

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

The client holds three types of data:

- *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 index's layer count denoted by L_h, and the HNSW degree limit represented by M. If client-side caching is enabled, the client also maintains the graph for the upper layers and the embeddings of nodes within these layers.
- *Quantized hints* 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 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,
- the list of identifiers of the neighbors of node x also called the *neighbor list of x*, and
- a hash used for integrity check (§4.8).

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 many 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. 4a, 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 each of the following iterations, the node that is closest to the query in the candidate list will be processed as the candidate node. This implies that the neighbors of the candidate nodes are visited and inserted into the candidate list. In Fig. 4a, 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 located on the same instance. However, in our setting, the client has to retrieve neighbors' embeddings and neighbor list from the remote server through ORAM requests, making it expensive because of multiple costly network round trips as well as an extensive amount of data transferred. For instance, during the final layer search of HNSW, the number of required round trips is determined by the *efSearch*, and the degree of each node is bounded by M. Consequently, to complete a search in the last layer, it takes *efSearch* round trips communication and retrieves roughly in total of $efSearch * M$ nodes' embeddings and neighbor lists. Empirically, M is 32, and *efSearch* ranges from tens to hundreds for sufficient accuracy, resulting in an inefficient solution with a high user-perceived latency and high bandwidth consumption.

4.4 Towards Compass's search index

To understand our approach, it helps to first consider two attempts to reduce the number of sequential ORAM requests

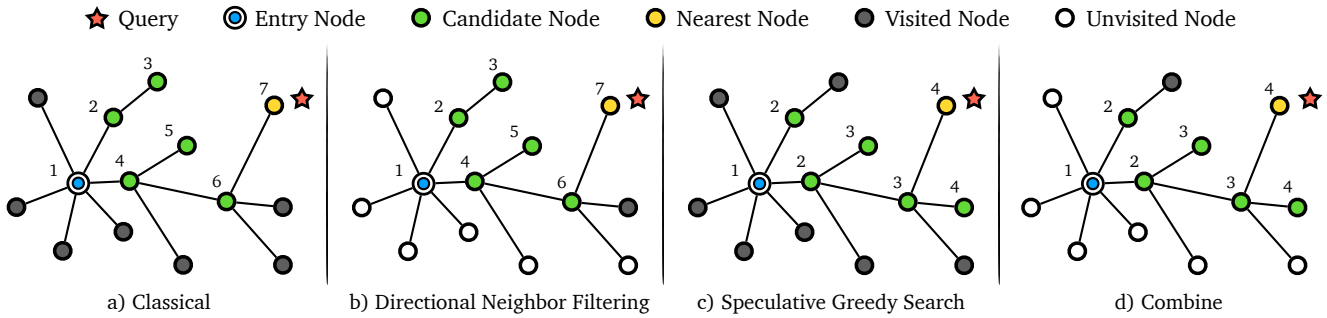

Figure 4. HNSW graph traversal followed by our ORAM-friendly traversal techniques. The search aims to identify the Nearest Node to the Query. The distance here is Euclidean distance. The numbers indicate iterations in the traversal, and which nodes are *processed* at each iteration. We say that a node is processed, if its neighbors are visited.

in the strawman above. The first idea is to include the embeddings 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 size of each ORAM block.

Instead, we propose a novel way to traverse the HNSW 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 a variety of compression algorithms, like PCA [37] or Product Quantization (PQ) [36]. 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* in the ORAM 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. PQ [36] 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 modest 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 quantized neighbors to the query, 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. 4b. 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 visits

only the top *efn* closest neighbors from the server obtaining their full coordinates and adding them to the candidate list. The client fetches all of these neighbors simultaneously in a batched ORAM request (§4.7). As compared to Fig. 4a, we can see in Fig. 4b 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 [42] 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 [55] 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 *currently-needed* nodes, 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' information within one request. Once we get the response from the remote server, the candidate list is updated by evaluating each node's neighborhood in the speculative set.

In Fig. 4c, we show an example of speculative greedy search. Here 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 [35] 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. 4d shows the final example of the search process when we compose both directional neighbor filtering and speculative greedy search. Compared to Fig. 4a, the algorithm in Fig. 4d 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. 4d, the client needs to invoke multiple ORAM requests per iteration to fetch the information of each neighbor. A natural solution is to use batching. Batching in ORAM has already been proposed as a mechanism for saving bandwidth and roundtrips in a scenario when multiple requests arrive at the same time[66, 78, 82]. 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, blocks are evicted from the stash at a predetermined rate. Path ORAM specifically schedules evictions after every path access to keep the stash size within reasonable limits. 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, it performs the stash eviction. While [66, 78, 82] 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 of all the ORAM requests, in fact the sequence depends on the private query and nodes, and performs eviction only at the end of all iterations. This data dependency could be problematic for security, but we show that our ORAM accesses remain randomly distributed. The key enabler is that we keep a list of visited nodes to ensure that we never request the same node from the ORAM before we process the eviction.

This approach assumes that users will tolerate a temporary increase in memory usage during the query process. For example, on the MS MARCO dataset, the memory increase is roughly 100MB. The advantages of multi-hop lazy eviction are substantial. First, it enables us to use smaller bucket sizes while maintaining a reasonable stash size. Further, the number of overflow blocks in the stash is reduced between two queries. This reduction is achieved by evicting multiple

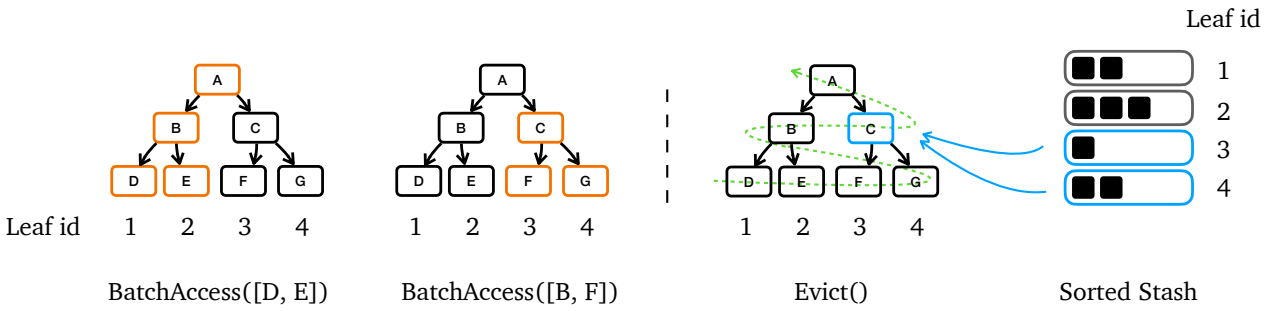

Figure 5. Example of batch access and eviction in Compass's HNSW tailored-ORAM

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. Lastly, multi-hop lazy eviction offers lower user-perceived latency, as the eviction processes can be done after returning the query results.

A side effect of multi-hop lazy eviction is the stash size grows larger and 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.

Aside from the stash, we maintain two additional data structures on the client side: vt_l and vt_p. The former tracks the history of visited paths and the latter tracks 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, as shown from line 5 to line 12. If so, a new path is selected randomly 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. Therefore, from line 13 to line 18, we only read the buckets that have not been fetched in the prior requests within the same epoch. The last step in the algorithm is to update the stash and perform the corresponding read or write operations on the stash.

In Fig. 5 we give an example of ORAM operations involved in one search request, two batch accesses followed by the eviction. At the beginning, the stash is empty. 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 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 simply employ existing work in the Path ORAM literature [63, 71] to construct a Merkle Tree on top of the ORAM Tree. The client stores the root of the Merkle tree for their data, 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

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

Alg. 1 shows the search algorithm Compass used in the final layer of the HNSW graph. The main difference compared to the vanilla HNSW is between line 4 and line 14. Here, the number of search iterations, n_s, is determined by $\lceil ef/efspec \rceil$, as we extract the top *efspec* nodes from the candidate list as the speculative set 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 neighbors of the nodes in the speculative set, ranking them based on their distance to the query, which is estimated using quantized coordinates, or local hints. The

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

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

```
1  visited ← ep // set of visited nodes
2  C ← ep // set of candidates
3  W ← ep // set of found nearest neighbors
4  n_s ← ⌈ef/efspec⌉
5  for step ← 0...n_s do
6      B ← extract top efspec nearest nodes from C to q
7      N ← ∅ // set of univisited neighbors
8      foreach b ∈ B do
9          foreach e ∈ neighbors(b) do
10             if e ∉ visited then
11                 N ← N ∪ e
12     k ← efspec * efn
13     N' ← get top k nearest nodes from N
        based on local hint
       // fetch full coordinates and neighbor
          list from ORAM, padded to k if |N'| < k
14     oram.batch_access(N', k)
15     foreach n ∈ N' do
16         visited ← visited ∪ n
17         W ← W ∪ n
18         C ← C ∪ n
19         if |W| > ef then
20             remove furthest element to q from W
21 return W
```

top k neighbors of the current speculative set are selected, and a batch ORAM request is issued to retrieve their full coordinates and neighbor lists. Here, k is set to $efspec * efn$. If the total number of neighbors in the current speculative set is less than k, the batch request is padded to k for security. With the full coordinates of the selected neighbors, we insert each one into the set of found nearest neighbors, W, as well as the candidate set, C. The set W is dynamic, removing the furthest element when its size exceeds ef. After completing n_s iterations, W is returned as the set of nearest neighbors to the query q.

The algorithm only requires n_s 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 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 into the speculative set in future steps. A trade-off we made 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

Because of space limitation, we provide a proof sketch covering the salient points in our proof, leaving a full proof to our extended version. The integrity guarantees of our protocol come directly from the Merkle tree on top of ORAM, which has already been proposed and proved [25, 63, 71]. 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 [78, 82]. The distinction between Compass and these works is that Compass's ORAM requests follow a multi-hop pattern. In Compass, each query in Compass involves a fixed amount of batch ORAM accesses. During batched ORAM accesses, the paths visited 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 ≈ 7k lines of C++ code. We implement our search algorithm based on the HNSW implementation of Faiss [23]. We constructed the HNSW search index and applied Product Quantization to the dataset, also using Faiss. We use AES-256-CBC to encrypt the ORAM block and SHA-256 for hashing, both via OpenSSL's EVP interface [1].

Level-wise Caching. To save the cost of frequently visited nodes in the upper layers of the HNSW graph, we cache all but the last layers of the HNSW graph and the embeddings of nodes assigned to these layers locally on the client. We show the client side overhead of level-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

6.1.1 Environment We evaluate our experiments on the Google Cloud Platform with one n2-standard-8 instance as

the client and one n2-highmem-64 instance as the server. The client has 8 vCPUs and 32 GB memory. The server has 64 vCPUs and 512 GB memory. We select these instances mainly to have enough memory for experiments. All experiments are done using a *single* thread. Both instances are in the same region and we use Linux Traffic Control (TC) to simulate different network settings. Similar to [13, 67], 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.

6.1.2 Datasets We evaluate our system on three datasets:

MS MARCO [8] is a large-scale dataset created from real Bing search queries. It consists of 8,841,823 passages and 6980 queries in the validation set. We generate the 768-dimensional embedding vectors for MS MARCO using a pre-trained model, `msmarco-distilbert-dot-v5`, from Sentence-Bert [61].

TripClick [62] is a dataset containing real click logs obtained from a health web search engine. It consists of 1523871 passages and 1175 queries in the validation set of head split. Relevance scores for each query are assigned using document click-through-rates. We use the same model above to generate TripClick's embeddings.

SIFT1M [36] is a widely-used dataset in ANN search literature. It consists of 1M base vectors and 10K queries. Each vector is a SIFT (Scale-Invariant Feature Transform) descriptor [48] extracted from images, with a dimension of 128.

6.1.3 Parameters We use the default construction parameters suggested by the Faiss library, where $M = 32$ and $efConstruction = 40$. For SIFT1M, each embedding vector is divided into 8 subvectors for PQ. For TripClick and MS MARCO, which have higher-dimensional embeddings, we split each vector into 32 subvectors. The search parameters, detailed in Tab. 2, are configured to achieve a Recall@10 of at least 0.9.

	MS MARCO	TripClick	SIFT1M
efSearch	224	192	32
efspec	16	16	4
efn	12	12	8

Table 2. Search parameters for each dataset.

6.2 Baselines

We compare our system with two baselines.

Inverted Index with ORAM. Our first baseline is inspired from [51, 78]. 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 [60], 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, we pad the query length to 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 [52], this can be worse than streaming the entire database. Therefore, we adopt the idea from [51] 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 [78], we also implemented this baseline with batch ORAM access and our better stash eviction algorithm. We evaluate this baseline on TripClick and MSMARCO, as this baseline only supports text-based search. We refer to this baseline as Inv-ORAM.

Homomorphic Encryption with clustering. The second baseline is built on embeddings with homomorphic encryption. To avoid linear communication, we use clustering, which is inspired by Tiptoe [33]. Each dataset of size N is first clustered into $\sqrt{N}$ clusters using k-means. 80% of the documents are assigned to a single cluster. The rest of 20% of documents, which are closer to the 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 refer to this baseline as HE-Cluster.

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). A Reciprocal Rank is the inverse of the rank of the first relevant item in the search result.

As Fig. 6 shows, Compass achieves significantly better accuracy and lower latency compared to the baselines. We compare the search performance of Compass with that of baselines under two network configurations. For Compass, we report user-perceived latency, as the eviction can be done after we return the search result to the user. For the Inv-ORAM baseline, we create 4 variations, in which the size of each keyword's truncated document list is set to be 10, 100, 1000, and 10000. The expectation here is with a larger size, we achieve better search quality but worse latency. In each one of the datasets, the dark blue dashed line shows the accuracy of a brute-force search over the embeddings, which indicates the highest achievable accuracy with the current embedding model. For MS MARCO and TripClick, we show the plaintext search quality of TF-IDF in a light blue dashed line, and BM25 [14, 62], a widely used exact matching model, in an orange dashed line.

Overall, Compass achieves significantly better accuracy and lower latency compared to the baselines.

The search quality of Compass outperforms both baselines and matches the accuracy of the brute-force search in all

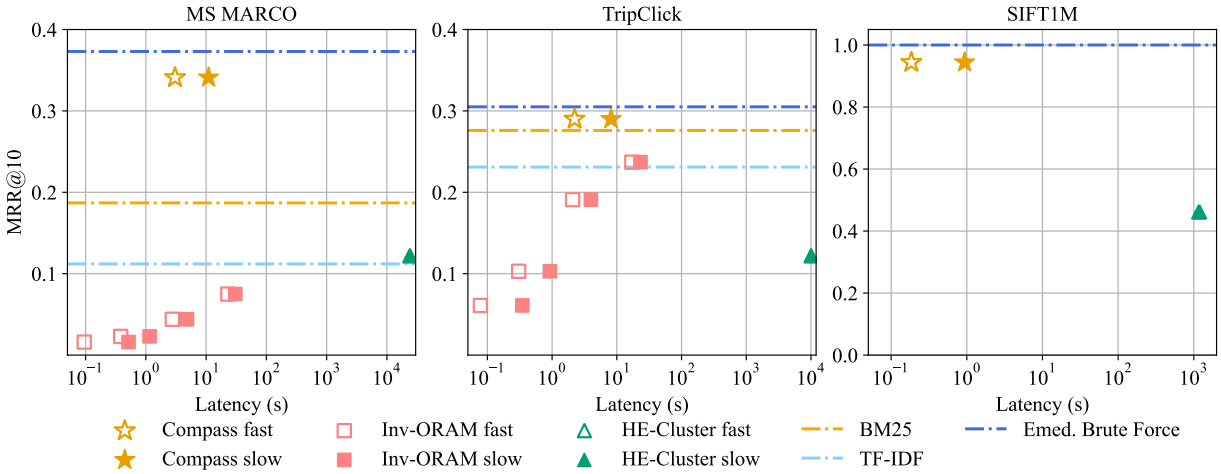

Figure 6. Comparison of Compass's search performance with Baselines. The results of the HE-Cluster under different networks are overlapped as the main bottleneck is the computation on the server. The search latency for HE-Cluster on MS MARCO is extrapolated due to memory limitation. (See estimated memory consumption in §6.3.3)

three datasets. On the MS MARCO dataset, the accuracy of the clustering baseline aligns closely with the search quality of the inverted index baseline. However, on the TripClick dataset, the inverted index baseline's accuracy is twice that of the clustering baseline. We attribute this difference to the health-related nature of the TripClick content, where sequence matching is particularly effective for identifying symptom and disease-related terms.

Compared to the HE-Cluster baseline, the search latency of Compass is orders of magnitude faster. This is because, while the clustering technique reduces communication costs to sublinear, the computation cost on the server remains linear relative to the dataset size. Granted that this latency can be greatly improved by leveraging parallelization with more CPUs or instances, the cost is very expensive and it's infeasible to scale to multiple users. When the size of the truncated document list is smaller than 1000, Inv-ORAM is able to achieve a faster search response than Compass. However, the search quality under these settings is significantly lower than plaintext TF-IDF, not to mention Compass. When the truncated list size is 1000, the search quality of Inv-ORAM becomes closer or even better than plaintext TF-IDF, while their search latency is 3-10 × slower than Compass.

	MS MARCO		**TripClick**		**SIFT1M**	
Network	*fast*	*slow*	*fast*	*slow*	*fast*	*slow*
Client Compute	0.023	0.020	0.017	0.017	0.003	0.003
ORAM Access	1.227	10.152	0.885	7.528	0.058	0.930
Perceived Lat.	**1.250**	**10.175**	**0.902**	**7.545**	**0.061**	**0.933**
ORAM Evict	1.506	9.702	1.119	7.121	0.046	0.257
Full Lat.	2.756	19.877	2.021	14.676	0.107	1.190

Table 3. Breakdown of Compass's search latency (s)

6.3.1 Latency breakdown In Tab. 3, we report the breakdown of Compass's search latency on all three datasets. The primary overhead comes from ORAM-related operations: access and eviction. The time spent on client-side HNSW graph traversal, excluding ORAM accesses, is negligible compared to ORAM operations. Thanks to our lazy eviction technique, the user-perceived latency, as highlighted in the table, is up to twice as fast as the full latency. For SIFT1M, we observe that ORAM access time exceeds eviction time in the *slow* network setting. This is because multiple rounds of access are required during a search, while eviction only involves a single round, making the network round trip the bottleneck. For other two datasets, however, the time spent on access and eviction is relatively similar. This is primarily due to two factors. First, these datasets have embeddings with six times the dimensionality of SIFT1M, making network bandwidth the dominant factor in latency. Second, these datasets require a larger ef, which leads to more blocks in the stash needing to be evicted during the eviction process, resulting in more time spent finding available paths for each block. Under *fast* network, the user-perceived latency of all three datasets is around or less than 1 second. Under *slow* network, the user-perceived latency increases to more than 10 seconds for MS MARCO. This is slightly higher than what a user usually expects. However, we argue that MS MARCO is a large, web-scale dataset, significantly larger than a user would typically have. Additionally, the embedding dimensionality, another factor contributing to the higher latency, can be reduced by using a smaller model or applying dimension reduction techniques.

6.3.2 Communication Tab. 4 presents the per-query communication costs, specifically the amount of data transferred between the server and client and the number of network round trips. For Compass, we present the communication

	MS MARCO		TripClick		SIFT1M	
	Comm.	RT	Comm.	RT	Comm.	RT
Compass SH.	586MB	16	422MB	14	14.3MB	10
Compass Mal.	587MB	16	423MB	14	14.4MB	10
Inv-ORAM	252MB	1	197MB	1	-	-
HE-Cluster	604MB*	1	835MB	1	530MB	1

Table 4. Comparison of communication cost per query. SH. represents semi-honest and Mal. represents Malicious. * indicates the result is extrapolated.

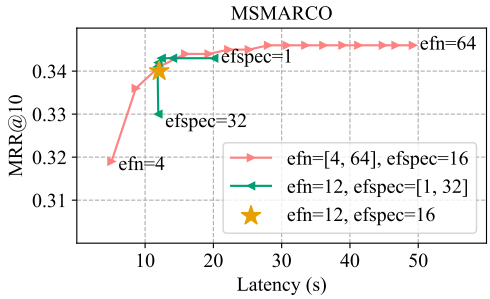

Figure 7. Ablations on Compass techniques

cost with and without integrity checks, corresponding to malicious and semi-honest servers respectively. For Inv-ORAM, we set the size of the truncated document list to 1000. As demonstrated in the previous section, this setting maintains reasonable search quality and provides a meaningful benchmark for comparison. Compass achieves less communication bandwidth than the HE-Cluster baseline, especially on the SIFT1M dataset. Compass requires slightly more communication bandwidth than the Inv-ORAM baseline with truncation. A notable distinction in communication is the number of network round trips. Both baselines are single-round protocols, whereas Compass requires multiple rounds of interactions. We justify the increased round trips for two reasons. First, Compass targets matching plaintext search quality, with the additional round trip overhead amounting to only about one second in the slower network settings. Moreover, when faced with higher network latencies than those in our tests, the number of round trips could be reduced at the expense of a slight accuracy drop. Second, our speculative greedy search algorithm ensures that the increase in round trips is minimal relative to the growth of dataset size. For example, Compass only requires two extra round trips in MS MARCO compared to TripClick, while the size of MS MARCO is 6x larger.

6.3.3 Memory Consumption We report the breakdown of memory consumption of Compass in Tab. 5. 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 1.5-2.6 × 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. On the client side, the memory overhead consists of three parts, upper layers of 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 upper layers of the graph are cached locally on the client. This part of memory overhead can be 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 example, caching the last two layers for MSMARCO reduces the client-side graph to roughly 27MB at the cost of two extra round trips for searching on the second last layer. Overall, Compass only requires approximately 1GB of client memory for a web-scale dataset such as MS MARCO. For smaller datasets that are closer to what users would have, the memory consumption of Compass will be in the tens or hundreds of megabytes.

6.3.4 Insert & Delete We now briefly discuss how Compass compares to two baselines on insert and delete operations. It's unclear how to securely insert or delete a document in the HE-Cluster baseline 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 due to padding. In this case, the insert latency of Inv-ORAM resembles the search latency in Fig. 6. 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, which then will incur the same cost as the search. Insertion, however, is cheaper as the default candidate list size required for insertion is only 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 21.5 seconds.

6.4 Ablation Studies

In Fig. 7 we evaluate the effectiveness of the two techniques of our search on the MS MARCO dataset under the slower network configuration. Specifically, we perform two sets of experiments and compare their search quality and latency with a default setting we used in the prior sections, where the directional filter size is 12 and the speculation size is 16.

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

	Embed. Size	Compass Server		Compass Client				HE-Cluster		Inv-ORAM	
		SH.	Mal.	Hints	PosMap	Graph	Total	Server	Client	Server	Client
MS MARCO	25.32GB	39.19GB	39.31GB	271MB	35MB	884MB	1.12GB	1000GB*	9MB*	10GB	810MB
TripClick	4.37GB	9.80GB	9.82GB	47MB	6MB	145MB	198MB	415 GB	4MB	10GB	606MB
SIFT1M	0.48GB	1.15GB	1.17GB	8MB	4MB	20MB	32MB	49.8GB	0.5MB	-	-

Table 5. Comparison of memory consumption. * indicates the result is extrapolated.

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.

7 Related Work

7.1 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 provide the high-accuracy of 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, 39, 43, 53, 70, 72], which can be exploited by leakage-abuse attacks [10, 34, 40, 47, 57, 58, 80].
- *Assuming trusted hardware enclaves* [5, 51, 73] that can be exploited by a wide range of side-channel attacks [9, 41, 76, 79].
- Assuming that some parts of a logical server (e.g. non-colluding servers, at least one trusted server in a set of servers) [17, 18, 67].

7.2 Encrypted search systems over embeddings

Combining embeddings with cryptographic search is a nascient 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 performance 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 results in reduced search quality as we discuss in §6.

Other works attempt to build sublinear indices over embeddings, but sacrifice security or efficiency. In some works [6, 81], 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 [67], Riazi et al. [64], and Wu et al. [77] 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.

References

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

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

[3] 2024. Semantic search. 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. 2021. Coeus: A system for oblivious document ranking and retrieval. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 672–690.

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

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

[7] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (Jan. 2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[8] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. 2016. 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. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings*

of the 22nd ACM SIGSAC conference on computer and communications security. 668–679.

[11] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2014. 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. 2013. 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*. Springer, 353–373.

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

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

[15] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*. 79–88.

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

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

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

[19] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horvárth, and Tibor Jager. 2023. 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. 2019. Dynamic searchable encryption with small client storage. *Cryptology ePrint Archive* (2019).

[21] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. 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*. Springer, 371–406.

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

[23] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

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

[25] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2015. 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*. 103–116.

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

[27] S. Garg, S. Lu, and R. Ostrovsky. 2015. Black-Box Garbled RAM. In *IEEE Annual Symposium on Foundations of Computer Science,*.

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

[29] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph. D. Dissertation.

[30] Eu-Jin Goh. 2003. Secure indexes. *Cryptology ePrint Archive* (2003).

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

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

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

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

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

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

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

[38] Seny Kamara and Charalampos Papamanthou. 2013. 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*. Springer, 258–274.

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

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

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

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

[43] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. 2014. Mimesis Aegis: A Mimicry Privacy {Shield–A}{System's} Approach to Data Privacy on Public Cloud. In *23rd usenix security symposium (USENIX Security 14)*. 33–48.

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

[45] Xiang Li, Yunqian Luo, and Mingyu Gao. 2024. BULKOR: Enabling Bulk Loading for Path ORAM. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 103–103.

[46] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Popa. 2023. The Deployment Dilemma: Merits Challenges of Deploying MPC. https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html

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

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

[49] Yu A. Malkov and D. A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. http://arxiv.org/abs/1603.09320 arXiv:1603.09320 [cs].

[50] Microsoft. [n. d.]. The science behind semantic search: How AI from Bing is powering Azure Cognitive Search.

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

[52] Muhammad Naveed. 2015. The fallacy of composition of oblivious ram and searchable encryption. *Cryptology ePrint Archive* (2015).

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

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

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

[56] Pinecone. [n. d.]. Vector Search: Hierarchical Navigable Small Worlds. https://www.pinecone.io/learn/series/faiss/hnsw/.

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

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

[59] preveil [n. d.]. PreVeil: Encrypted Email and File Sharing. https://www.preveil.com/.

[60] Juan Ramos. [n. d.]. Using TF-IDF to Determine Word Relevance in Document Queries. ([n. d.]).

[61] Nils Reimers and Iryna Gurevych. 2019. 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. http://arxiv.org/abs/1908.10084

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

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

[64] M. Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. 2019. Sub-Linear Privacy-Preserving Near-Neighbor Search. Cryptology ePrint Archive, Paper 2019/1222.

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

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

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

[68] Signal Messenger [n. d.]. Signal Messenger. https://signal.org/.

[69] 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. 2022. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. http://arxiv.org/abs/2205.03763 arXiv:2205.03763 [cs].

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

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

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

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

[74] Telegram Messenger [n. d.]. Telegram Messenger. https://telegram.org/.

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

[76] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice.

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

[78] Zhiqiang Wu and Rui Li. 2023. OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption.. In *NDSS*.

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

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

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

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