Private Web Search with Tiptoe

Alexandra Henzinger
MIT

Emma Dauterman
UC Berkeley

Henry Corrigan-Gibbs
MIT

Nickolai Zeldovich
MIT

Abstract. Tiptoe is a private web search engine that allows clients to search over hundreds of millions of documents, while revealing no information about their search query to the search engine's servers. Tiptoe's privacy guarantee is based on cryptography alone; it does not require hardware enclaves or non-colluding servers. Tiptoe uses semantic embeddings to reduce the problem of private full-text search to private nearest-neighbor search. Then, Tiptoe implements private nearest-neighbor search with a new, high-throughput protocol based on linearly homomorphic encryption. Running on a 45-server cluster, Tiptoe can privately search over 360 million web pages with 145 core-seconds of server compute, 56.9 MiB of client-server communication (74% of which occurs before the client enters its search query), and 2.7 seconds of end-to-end latency. Tiptoe's search works best on conceptual queries ("knee pain") and less well on exact string matches ("123 Main Street, New York"). On the MS MARCO search-quality benchmark, Tiptoe ranks the best-matching result in position 7.7 on average. This is worse than a state-of-the-art, non-private neural search algorithm (average rank: 2.3), but is close to the classical tf-idf algorithm (average rank: 6.7). Finally, Tiptoe is extensible: it also supports private text-to-image search and, with minor modifications, it can search over audio, code, and more.

1 Introduction

The first step of performing a web search today, whether using Google, Bing, DuckDuckGo, or another search engine, is to send our query to the search engine's servers. This is a privacy risk: our search queries reveal sensitive personal information to the search engine, ranging from where we are ("Tokyo weather"), to how we are doing ("Covid19 symptoms"), to what we care about ("Should I go to grad school?") [13, 55]. The search engine may inadvertently disclose this information in a data breach or intentionally resell it for profit. Even if you anonymize your IP address by accessing the search engine through Tor [36], the query itself can contain personally identifying information, and similarities across queries can link requests and deanonymize the user [99, 100].

Today's web search engines must see the user's search query because common algorithms and data structures for text search make many query-dependent lookups [10, 50, 134]. For example, the keywords in the query may determine which shard of servers processes the query, which rows of an inverted index the servers inspect, and how the servers aggregate the relevant documents. If the servers do not know the query, they cannot apply standard search techniques.

In contrast, cryptographic schemes that provide strong query privacy [28, 67] generally require the servers to scan the entire data set in response to each query [5, 9, 133]—otherwise, the servers would learn which parts of the data set were *not* relevant to the query [16]. This is challenging for Internet-scale search, as scanning every crawled web page on each query becomes very costly. Using the state-of-the-art system for private text search, Coeus [5], to search over the entire Internet would be prohibitively expensive: we conservatively estimate that, searching over a public web crawl with 360 million pages [108], a Coeus query would take more than 900 000 core-seconds and 3 GiB of traffic (see §8). For private text-to-image search, no such systems even exist.

This paper presents Tiptoe, a search engine that learns *nothing* about what its users are searching for. Tiptoe provides a strong privacy guarantee: its servers take as input a query ciphertext from the user and, using linearly homomorphic encryption [94, 95] and private information retrieval [56], compute the response ciphertext *without ever decrypting the query ciphertext*. This approach ensures privacy based only on cryptographic assumptions: Tiptoe does not require Tor, trusted hardware, or non-colluding infrastructure providers.

Inspired by non-private search engines [39, 91, 138], Tiptoe uses semantic embeddings for document selection and ranking. A semantic embedding function maps text strings (or images or other content) to vectors such that strings that are close in meaning produce vectors that are close in inner-product distance. Many pretrained embedding models have been made publicly available for off-the-shelf use [68, 81, 114]. Using embeddings, Tiptoe reduces the problem of private web search to the problem of *private nearest-neighbor search*: the client must find the document vectors that maximize the inner-product score with its query vector. With this design, Tiptoe is extensible to search over many different forms of media, including text, images, video, audio, and code; this paper demonstrates both text and text-to-image search.

To implement this search, Tiptoe introduces a new lightweight protocol for private nearest-neighbor search that

SOSP '23, October 23–26, 2023, Koblenz, Germany
ACM ISBN 979-8-4007-0229-7/23/10.
http://dx.doi.org/10.1145/3600006.3613134

allows the client to find the documents most relevant to its query. In particular, the client sends an encryption of its query vector to the Tiptoe service; the Tiptoe servers compute inner-product scores under encryption and return the encrypted results to the client. Tiptoe uses a recent lattice-based encryption scheme [56, 112] that lets the servers perform most of their computation in a client-independent preprocessing step. The servers then only need to perform a much smaller amount of per-query computation for each document.

Finally, to reduce communication cost, Tiptoe clusters documents together by topic. The client uses private-information-retrieval techniques to fetch encrypted inner-product scores for only the documents in the most relevant cluster, while hiding the identity of this cluster from the Tiptoe servers. Though this approach slightly worsens Tiptoe's search quality, it lets Tiptoe's communication costs scale sublinearly with the number of documents in the search corpus—which is crucial to operating at web scale.

We implement a prototype of Tiptoe in Go and evaluate it on a public web crawl consisting of 360 million English-language web pages [108]. When running Tiptoe on a cluster of 45 servers, Tiptoe clients execute private web search queries with 2.7 seconds of end-to-end latency, using 145 core-seconds of total server compute and 56.9 MiB of network traffic. Of this traffic, 42.2 MiB is sent *before* the client decides on its search query, leaving only 14.7 MiB on the latency-critical path. We give a detailed evaluation in §8, which includes text-to-image search over a corpus of 400 million images [119].

To evaluate the quality of Tiptoe's search results, we use the MS MARCO search-quality benchmark for end-to-end document retrieval [92]. On this benchmark, Tiptoe ranks the best result on average at position 7.7 out of 100, which is comparable to the standard tf-idf algorithm (average rank: 6.7) but worse than state-of-the-art non-private neural search engines (average rank: 2.3). While Tiptoe's search works best on conceptual queries, it performs most poorly on exact-string searches such as for phone numbers and addresses. At the same time, as Tiptoe makes black-box use of machine-learning models for information retrieval, future improvements in these techniques can directly improve Tiptoe's search quality.

2 Goals and limitations

Tiptoe is a search engine that learns *nothing* about what its users are searching for. In particular, an attacker that controls all Tiptoe servers should be able to learn *no information* about the clients' search queries (e.g., the strings typed into the search engine), even if the Tiptoe servers deviate from the prescribed protocol. We formalize this property as *query privacy*, which is essentially an adaptation of the cryptographic notion of semantic security to our setting [49]:

Definition 2.1 (Query privacy)**.** For a query string $q \in \{0,1\}^*$ and an adversarial search service $\mathcal{A}$, let $M_{\mathcal{A},q}$ be a random variable representing the messages that the search client sends to the search service when the client searches for string q. We say that a search engine provides *query privacy* if, for all pairs of strings $q_0, q_1 \in \{0,1\}^*$ and for all efficient adversaries $\mathcal{A}$, the corresponding probability distributions of $M_{\mathcal{A},q_0}$ and $M_{\mathcal{A},q_1}$ are computationally indistinguishable.

Our definition of query privacy implies that all aspects of the search engine's behavior—including the queries it receives, its memory access patterns, the precise timing of its execution, and the responses it sends back to the client—are independent of the query issued by the client, up to cryptographic assumptions. This also implies that the search engine does not learn the set of search results that it sends back to the client.

For a system to provide query privacy, the search engine's servers can never see the client's query in plaintext—otherwise, the server could easily distinguish between client queries for q_0 and for q_1. Private-search systems based on anonymizing proxies (e.g., Tor [36], mix-nets [22]) cannot provide this type of query privacy, since at some point the search system's servers must see the query in order to answer it.

As we will demonstrate, Tiptoe achieves query privacy under standard cryptographic assumptions.

Non-goals and limitations. Tiptoe hides *what* a client is searching for; Tiptoe does not hide *when* a client makes a query, or *how many* queries the client makes. Moreover, Tiptoe does not protect information about a client's web-browsing behavior *after* the client makes its query. For example, if the client browses to a URL that Tiptoe's search returns, the client's post-search HTTP/HTTPS requests could indirectly leak information about its query to a network adversary.

In the face of malicious servers, Tiptoe guarantees neither the availability of its service nor the correctness of its results. This limitation is inherent: malicious servers can decide which corpus to serve, and lie about the contents of documents.

Finally, Tiptoe's embedding-based search returns semantic matches rather than exact lexical ones. This brings with it many of the limitations of machine learning: bias, lack of interpretability, and difficulty to generalize beyond the embedding's training set [76]. Crucially, Tiptoe only relies on the embedding model for search result correctness—not privacy.

3 Tiptoe design

Tiptoe achieves query privacy by ensuring that:

- every protocol message that the client transmits is *encrypted* with a secret key known only to the client, meaning that the Tiptoe servers only ever see ciphertexts, and
- the message flow and packet sizes do not depend on the client's secret query string or on the servers' behavior.

The Tiptoe servers compute the answers to the client's queries directly on the encrypted data, without ever decrypting it.

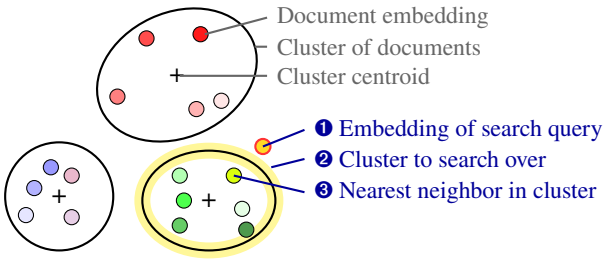

Figure 1: Tiptoe's semantic search with embeddings. Tiptoe uses embeddings to represent documents as points in a vector space. To search: ❶ the query is embedded into the same vector space, ❷ the cluster of documents nearest to the query point is identified, and ❸ the closest document to the query point within the cluster is returned.

3.1 Design ideas

The core challenge in Tiptoe lies in the tension between supporting expressive queries, hiding the query contents from the search engine, and searching over hundreds of millions of documents, all in the span of seconds. To provide privacy, the search engine's servers must, on every query, scan over a data structure whose size is linear in the number of documents searched. (Otherwise, an adversary controlling the search engine can learn which documents the client is *not* interested in.) At the same time, performance requirements rule out any expensive, per-document cryptographic computations (for example, checking for the joint appearance of encrypted keywords in documents). Tiptoe resolves this tension using the following techniques.

Embedding-based search. Tiptoe represents documents using *semantic embeddings* [68, 81, 114], a machine-learning technique that many recent non-private search systems use [39]. A semantic embedding function maps each document to a short, fixed-size vector (e.g., 768 floats for text search), such that semantically similar documents map to vectors that are close in the embedding space. The Tiptoe search protocol supports any embedding function that uses inner product or cosine similarity as its vector similarity measure; as such, Tiptoe is compatible with popular embeddings [85], including transformer models [114].

Using embeddings, Tiptoe reduces the private document-search problem to that of private nearest-neighbor search. To perform a search, the client locally embeds its query string into a vector and then needs to find the document in the server-side corpus whose embedding has the maximal inner product (i.e., dot product) with the client's query embedding. We illustrate Tiptoe's embedding-based search algorithm in Figure 1. This approach provides three key benefits:

1. Embeddings are small (less than 4% the size of the average document in our web-search corpus), which dramatically reduces the cost of the Tiptoe servers' linear scan.
2. Embeddings allow Tiptoe to natively support expressive queries—without any special machinery for keyword intersection or term-frequency analysis.
3. Embeddings exist for an array of document types (e.g., text, image, audio, and video), allowing Tiptoe to support private search over these document types.

Private nearest-neighbor search with fast linearly homomorphic encryption. To find the closest documents to its query, the client sends an *encryption* of its query embedding to the Tiptoe search engine, and the Tiptoe servers homomorphically (i.e., under encryption) compute the inner product of the query embedding with every document in the corpus. The servers return the encrypted inner-product scores to the client. The key to achieving good performance is that the document embeddings are plaintext vectors, and so the inner-product scores that the servers need to compute are a public, linear function of the client's encrypted query embedding. Therefore, we can use a *linearly* homomorphic encryption scheme (that is, an encryption scheme that supports computing only linear functions on encrypted data), which is much simpler and faster than its "fully" homomorphic counterpart [44].

In §4, we show that using a high-throughput, lattice-based encryption scheme [56] makes Tiptoe's server-side computation fast, despite touching every document. Applied directly, this encryption scheme requires the client to download and store one large (8 KiB) ciphertext for each inner-product score. We shrink the download to eight bytes per score, at the expense of requiring some per-query preprocessing that can execute *before* the client has decided on its search query (§6).

Clustering to reduce communication. Finally, Tiptoe uses clustering [25, 58, 61, 71, 106] to shrink client-server traffic. In particular, to avoid having the client download N inner-product scores on an N-document corpus, Tiptoe groups documents with similar embeddings into *clusters* of size roughly $\sqrt{N}$. The client downloads a list of $\sqrt{N}$ cluster "centroids" ahead of time and, at query time, uses this list to find the cluster closest to its query embedding. Then, the client fetches the inner-product scores for only the $\sqrt{N}$ documents in this best-matching cluster, using a cryptographic protocol that hides the cluster's identity from the servers. The servers still compute over all N documents to ensure that the protocol's privacy is not affected, but clustering drastically reduces the communication (from linear to $O(\sqrt{N})$), at some cost in search quality (see §8).

3.2 Tiptoe architecture

A Tiptoe deployment consists of data-loading batch jobs, a client, and two client-facing services—a *ranking service* and a *URL service*—that implement the core search functionality. We show an overview of Tiptoe's architecture in Figure 2. In our prototype, the batch jobs and the client-facing services run on a cluster of tens of physical machines.

Data-loading batch jobs. The Tiptoe batch jobs convert a raw corpus of documents into a set of data structures for private search. The batch jobs perform three steps:

Embed. First, the batch jobs run each document through a server-chosen embedding function to generate a fixed-size

vector representation of the document. The output of this step is one embedding vector per document. The choice of embedding function depends only on the type of document being indexed (e.g., text, image) and not on the corpus itself; our prototype uses off-the-shelf, pretrained models.

Cluster. Second, the batch jobs group the embedded document vectors into clusters of tens of thousands of documents each and compute the centroids (i.e., average embedding values) of each cluster. Since nearby embedding vectors represent documents that are close in content, the documents within each cluster are typically about related topics.

Preprocess cryptographic operations. Finally, the batch jobs compute a set of cryptographic data structures required for our private-search protocols. (These correspond to the "hint" in the SimplePIR private information retrieval scheme [56].)

Search queries with Tiptoe. Before a client can issue Tiptoe search queries, it must download the embedding function that the servers used during data loading (265 MiB for text search) along with the set of cluster centroids and associated metadata (68 MiB for a 360 million-document text corpus). To perform a private search, a Tiptoe client executes the following steps:

1. Embed query. The client embeds its query string into a fixed-size vector using the server-provided embedding function.

2. Rank documents (§4). The client then uses Tiptoe's ranking service to find the IDs of the documents that best match its query, while revealing to the Tiptoe service neither its query nor its embedded query vector. To do so, the client uses its local cache of cluster centroids to identify the cluster whose centroid is closest to its query embedding. Then, the Tiptoe client uses a new cryptographic protocol to obtain the distance between its query embedding and all of the documents in its chosen cluster, while hiding its query and its chosen cluster from the Tiptoe servers.

3. Fetch URLs (§5). Once the client has the IDs of the best matching documents, the client uses Tiptoe's URL service to privately fetch the URLs for its top few documents. The Tiptoe client uses a cryptographic private-information-retrieval protocol [28] to query the URL service for this data, while hiding which documents it is interested in.

Handling updates to the corpus. To support continuous updates to the search corpus, the Tiptoe servers can run the new or changed documents through the embedding function, assign them to a cluster, and publish the updated cluster centroids and metadata to the clients. Even if all centroids change, fetching this data (in a compressed format) requires at most 18.7 MiB of download for our 360 million-document text-search corpus.

4 Tiptoe's private ranking service

This section describes Tiptoe's ranking service, which allows the client to find the IDs of the documents that are most relevant to its query.

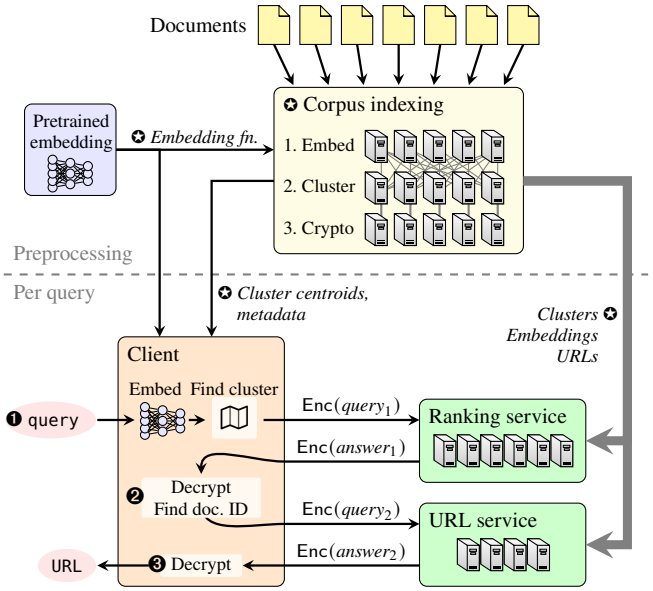

Figure 2: The Tiptoe system architecture. ✪ In a preprocessing phase, the Tiptoe batch jobs embed each document into a vector (with a pretrained embedding function), cluster the vectors, and generate the cryptographic data structures. The client and servers each store portions of the preprocessed data. ❶ For each query, the client embeds the query string into a vector, identifies the cluster nearest its query vector, and queries the ranking service to find the best documents within that cluster. ❷ Once the client knows the IDs of the best-matching documents, it queries the URL service. ❸ Finally, from the URL service's answer, the client recovers the best documents' URLs.

Tiptoe implements this ranking step using a new *private nearest-neighbor search* protocol. On a corpus of N documents with d-dimentional embedding vectors, the total communication cost required for ranking grows as $\sqrt{Nd}$, and the server-side time is roughly $2Nd$ 64-bit word operations.

An important caveat is that our protocol only allows the client to find *approximate* nearest neighbors—it does not produce exact results and provides no formal correctness guarantees. However, since Tiptoe builds on semantic embeddings which similarly do not provide formal guarantees of correctness, approximate nearest-neighbor results suffice.

4.1 Tiptoe's private nearest-neighbor protocol

At the start of the ranking step:

- the ranking service holds a list of N document-embedding vectors of dimension d, partitioned by topic into roughly $\sqrt{N}$ clusters (see §4.2), and
- the client holds the semantic embedding $\mathbf{q}$ of its query string, along with a list of the cluster centroids that it has fetched in advance and cached.

The client's goal is to find the IDs of the server-side documents whose embedding vectors are closest to its query embedding $\mathbf{q}$. For now, we think of the embedding as consisting of d integers; we discuss how to handle real-valued vectors in §4.3.

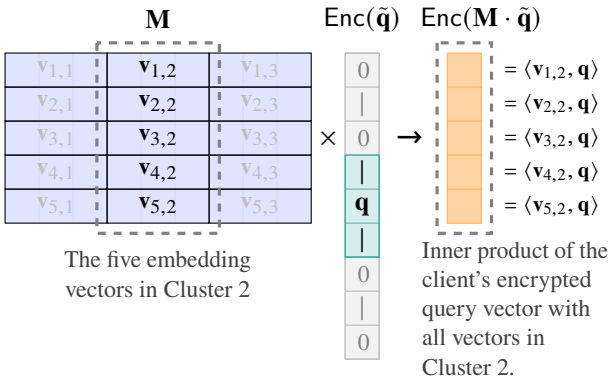

Figure 3: The matrices in our private nearest-neighbor computation, for a data set with $C = 3$ clusters of 5 vectors each. The client, in this example searching within Cluster 2, uploads an encrypted query vector $\tilde{\mathbf{q}}$ and receives the encrypted inner-product scores for all documents in Cluster 2 in response. The client then decrypts to recover the document scores.

To perform the nearest-neighbor search, the client and ranking service execute the following steps:

Step 1: Client query preparation. The client uses its locally cached set of cluster centroids to identify the index i^* of the cluster nearest to its query embedding $\mathbf{q}$.

The client then prepares a vector $\tilde{\mathbf{q}}$ that encodes its query embedding $\mathbf{q}$ and its chosen cluster index i^*. In particular, if there are C server-side clusters and the client's query vector has dimension d, the client constructs a vector $\tilde{\mathbf{q}}$ of dC integers, which is zero everywhere except that it contains the client's query $\mathbf{q}$ in the i^*-th block of d integers (Figure 3).

Then, the client encrypts this vector using a linearly homomorphic encryption scheme to get a ciphertext $\mathsf{ct} = \mathsf{Enc}(\tilde{\mathbf{q}})$. The client sends this ciphertext ct to the ranking service. Because the Tiptoe ranking service only receives a fixed-length ciphertext, it *learns nothing* about either the client's query vector $\mathbf{q}$, or about the cluster i^* it is interested in.

Step 2: Ranking-service computation. The ranking service arranges its N document-embedding vectors into a matrix $\mathbf{M}$. If there are N total document embeddings, grouped into C clusters, the ranking service arranges these vectors into a matrix $\mathbf{M}$ with C columns and N/C rows, such that for all $i \in \{1, \ldots, C\}$, column i of the matrix contains all vectors in cluster i (Figure 3). For the purposes of this protocol, we think of all clusters as being roughly the same size, and we pad the height of the matrix $\mathbf{M}$ to the size of the largest cluster. Since each embedding vector has dimension d, each element of the matrix will contain d integers.

Upon receiving a query ciphertext ct from the client, the server computes the matrix-vector product $\mathsf{ct}' = \mathbf{M} \cdot \mathsf{ct}$. Since Enc is a linearly homomorphic encryption scheme, it holds that $\mathsf{ct}' = \mathbf{M} \cdot \mathsf{Enc}(\tilde{\mathbf{q}}) = \mathsf{Enc}(\mathbf{M} \cdot \tilde{\mathbf{q}})$—meaning that the ranking service has just computed the product $\mathbf{M} \cdot \tilde{\mathbf{q}}$ *under encryption*. The server returns the resulting ciphertext ct' to the client.

Step 3: Client decryption. The client decrypts ct' to recover $\mathbf{M} \cdot \tilde{\mathbf{q}}$. Based on how we construct the ranking service's matrix $\mathbf{M}$ and the client's vector $\tilde{\mathbf{q}}$, the product $\mathbf{M} \cdot \tilde{\mathbf{q}}$ contains the $\approx \sqrt{N}$ inner-product scores for the documents in the client's chosen cluster i^* (see Figure 3). The client outputs the index of the documents with the largest inner product scores as the nearest neighbors.

We give a more detailed description of the protocol in Figure 10 of Appendix B.

4.2 Protocol analysis

Security. The server sees only the encryption of the client's augmented query vector $\tilde{\mathbf{q}}$, under a secret key known only to the client. Provided that the underlying encryption scheme is semantically secure [49], the server learns nothing about the client's query embedding $\mathbf{q}$ nor about the index i^* of its cluster of interest.

Correctness. With this scheme, the client learns the inner product of its query vector $\mathbf{q}$ with each vector in its chosen cluster i^*. This may not always return the true nearest neighbors of $\mathbf{q}$, since the true nearest neighbor may not always lie in the cluster searched by the client. (This is because the client uses its local cache of cluster centroids to determine which cluster to search.) Whenever the centroid of the cluster containing the true best match is not the closest centroid to $\mathbf{q}$, the client will instead obtain an approximate nearest neighbor.

While we do not provide guarantees on how good of an approximation this scheme will provide, our empirical evaluation suggests that it is reasonable on average (§8).

Communication cost. On a corpus of N documents with d-dimensional embeddings divided into C clusters:

- the client uploads the encryption of a vector of dimension dC,
- the ranking service applies a matrix with dN integer entries to the encrypted vector ($\frac{N}{C}$ rows, dC columns), and
- the ranking service returns an encrypted vector of dimension $\frac{N}{C}$ to the client.

Taking $C \approx \sqrt{N}$, the total communication cost scales roughly as $d\sqrt{N}$. (If the dimension d grows large, we can take $C \approx \sqrt{N/d}$ to reduce the total communication slightly to $\sqrt{dN}$.)

Performance. In this protocol, the ranking service computes a matrix-vector product between the large matrix $\mathbf{M}$ and the vector $\mathsf{Enc}(\tilde{\mathbf{q}})$, encrypted with a linearly homomorphic encryption scheme. The performance of computing on encrypted data thus determines the ranking service's throughput. In Tiptoe, we use a recent fast linearly homomorphic encryption scheme, which we describe in §6.

4.3 Implementation considerations

We now describe how we build the private ranking service that responds to private nearest-neighbor queries.

Representing real-valued embeddings. Embeddings are traditionally vectors of floating-point values [68, 114], but the linearly homomorphic encryption scheme Tiptoe uses can only compute inner products over vectors of integers modulo p. Choosing a smaller modulus p improves performance. To bridge this gap, Tiptoe represents each real number as an integer mod p using a standard fixed-precision representation, which we describe in Appendix B.1.

Scaling out to many physical machines. The server's per-query computation in Figure 10 consists of multiplying the large, corpus-dependent matrix $\mathbf{M}$ by the client's encrypted query vector $\mathsf{Enc}(\tilde{\mathbf{q}})$. For a corpus with hundreds of millions of documents, the matrix $\mathbf{M}$ can be 50 GiB or more in size. Sharding $\mathbf{M}$ across multiple servers reduces query latency and, for very large data sets, ensures that the entire matrix fits in main memory. Tiptoe shards by cluster: to shard across W worker machines, we vertically partition the matrix as $\mathbf{M} = (\mathbf{M}_1 \| \cdots \| \mathbf{M}_W)$, and we store matrix $\mathbf{M}_i$ on server i.

In our design, a front-end "coordinator" server receives the ciphertext vector $\mathsf{ct} = \mathsf{Enc}(\tilde{\mathbf{q}})$ from the client. The coordinator partitions the query vector into W chunks, $\mathsf{ct} = (\mathsf{ct}_1 \| \cdots \| \mathsf{ct}_W)$, and then, for $i \in \{1, \ldots, W\}$, ships ciphertext chunk ct_i to worker i. Worker i computes the answer $\mathbf{a}_i \leftarrow \mathbf{M}_i \cdot \mathsf{ct}_i$ and returns $\mathbf{a}_i$ to the coordinator. The coordinator computes $\mathbf{a} = \sum_{i=1}^{W} \mathbf{a}_i$, which it sends to the client.

Our implementation ships each ciphertext chunk ct_i to a single physical machine. If any machine fails during this computation, the coordinator cannot reply to the client. To improve latency and fault-tolerance at some operating cost, the coordinator could farm out each task to multiple machines.

5 Tiptoe's URL service

In this section, we describe the functionality of Tiptoe's URL service. Once the client has identified the IDs of the documents most relevant to its query, the client must fetch the metadata for these documents. In Tiptoe, this metadata is the document URL, though it could potentially also include web-page titles, summaries, or image captions. By default, the Tiptoe client fetches and outputs the metadata for the top 100 search results.

Using private information retrieval. To fetch the document metadata, Tiptoe uses an existing single-server private information retrieval [28, 67] protocol with additional optimizations (see §6). Cryptographic private-information-retrieval protocols allow a client to fetch a record from a server-held array, without revealing to the server which record it has retrieved. Tiptoe implements this step using SimplePIR [56], which has the lowest server compute cost among single-server private-information-retrieval schemes.

Under the hood, SimplePIR uses many of the same tools as the private nearest-neighbor search protocol described in §4: at a high level, the SimplePIR client builds a vector that consists of all zeros, except with a single '1' at the position of the record it would like to retrieve. The client then encrypts this vector with a linearly homomorphic encryption scheme, and uploads it to the server. The server multiplies its array of records by this encrypted vector, effectively selecting out the record that the client is trying to read, and then sends the resulting ciphertext back to the client. Exactly as in §4, the server here only ever sees and computes on fixed-length ciphertexts and touches every record in the array as part of this computation. As a result, the server learns *nothing* about the URLs that the client is retrieving.

SimplePIR serves data to the client only in relatively large chunks—roughly 40 KiB with our parameter settings. The client must always fetch at least one of these chunks from the server. Tiptoe packs as much useful information into a single chunk as possible, in the following two ways:

Compressing batches of URLs. Since the client fetches a few hundred kilobytes with each metadata query, we assemble URLs into batches and compress roughly 880 of them at a time using `zlib`, dropping any URLs that are more than 500 characters in length. By compressing many URLs at once, each URL takes only 22 bytes to represent on average.

Grouping URLs by content. We group URLs together into these batches by content. Then, if the client fetches the metadata for the best-matching document, it is likely to find the metadata for other top-matching documents within the same compressed batch. Our prototype of Tiptoe fetches only a single batch of URLs (chosen to be the one containing the best-matching document) and then outputs the $k = 100$ URLs for the best-ranked documents in this batch. We show in §8 that retrieving a single batch of URLs (rather than k batches of URLs) does not significantly reduce the search quality.

6 Cryptographic optimizations

The ranking service (§4) accounts for the bulk of the per-query computational cost in Tiptoe. Its main computational overhead comes from the use of linearly homomorphic encryption; for each query, the service must multiply an encrypted vector by a matrix as large as the entire search index. We accelerate this matrix-vector product with the following ideas:

1. We use an off-the-shelf high-throughput linearly homomorphic encryption scheme that has large ciphertexts (§6.1).
2. We compress the ciphertexts using a second layer of homomorphic encryption (§6.2).
3. We perform the bulk of the compression work in a per-query setup phase that runs before the client makes a search query, i.e., off of the latency-critical path (§6.3).

We also apply optimizations 2 and 3 to the private-information-retrieval step used for URL retrieval (§5). These techniques effectively eliminate the client-side "hint" storage required by SimplePIR [56], at the cost of increasing the per-query communication by roughly 4×.

We give a formal description of the resulting linearly homomorphic encryption scheme in Appendix A; we thank Yael

Kalai, Ryan Lehmkuhl, and Vinod Vaikuntanathan for helpful discussions pertaining to this section.

6.1 Preprocessing to reduce per-query computation

Tiptoe uses the high-throughput linearly homomorphic encryption scheme developed in SimplePIR [56], which is in turn based on Regev's lattice-based encryption scheme [112]. This encryption scheme allows the server to *preprocess* a matrix $\mathbf{M}$ ahead of time. Thereafter, given a ciphertext $\mathsf{Enc}(\tilde{\mathbf{q}})$ encrypting a vector $\tilde{\mathbf{q}}$ (i.e., a query vector), the server can compute the matrix-vector product $\mathbf{M} \cdot \mathsf{Enc}(\tilde{\mathbf{q}}) = \mathsf{Enc}(\mathbf{M} \cdot \tilde{\mathbf{q}})$ almost as quickly as computing the same product on plaintext values. The server can compute many subsequent matrix-vector products (with the same $\mathbf{M}$), amortizing away the cost of its one-time preprocessing step.

In Tiptoe, the matrix held by the ranking service does not depend on the client's query—it is a fixed function of the search index. Thus, the Tiptoe servers can preprocess this matrix during the data-loading batch processing phase that occurs whenever the document corpus changes (§3.2). To give a sense of the concrete efficiency of the SimplePIR-style encryption scheme in Tiptoe, if the matrix $\mathbf{M}$ is a $\sqrt{N}$-by-$\sqrt{N}$ matrix, the query vector $\tilde{\mathbf{q}}$ consists of $\sqrt{N}$ 16-bit integers, and the security parameter (i.e., lattice dimension) is $\lambda \approx 1024$, then the linearly homomorphic encryption scheme has the following performance characteristics:

Small computation. After the one-time preprocessing of the matrix $\mathbf{M}$, the cost of computing $\mathbf{M} \cdot \mathsf{Enc}(\tilde{\mathbf{q}})$ is small: only $2N$ 64-bit operations. For comparison, computing $\mathbf{M} \cdot \tilde{\mathbf{q}}$ on plaintext (unencrypted) values requires $2N$ 16-bit operations.

Large communication. After computing on the ciphertexts, they become large: the ciphertext encrypting the product $\mathsf{Enc}(\mathbf{M} \cdot \tilde{\mathbf{q}})$ is a factor of $(\frac{64}{16}) \cdot \lambda \approx 4096$ larger than the plaintext vector $\mathbf{M} \cdot \tilde{\mathbf{q}}$. In the context of Tiptoe's ranking service, this would amount to an impractical 0.75 GiB download to search over 360 million documents. (If the client makes multiple queries to the same corpus, SimplePIR can re-use a large portion—namely, 99.9%—of this download; however, the *total* download would still be at least as large.)

The exact cost expressions are:

- *Matrix preprocessing.* The server executes $\lambda\sqrt{N}$ 64-bit operations for the one-time preprocessing of the matrix $\mathbf{M}$.
- *Homomorphic matrix-vector product.* The server computes $\mathbf{M} \cdot \mathsf{Enc}(\tilde{\mathbf{q}})$ with $2N$ 64-bit additions and multiplications.
- *Ciphertext size before homomorphic operation.* The ciphertext $\mathsf{Enc}(\tilde{\mathbf{q}})$ consists of $\sqrt{N}$ 64-bit integers.
- *Ciphertext size after homomorphic operation.* The resulting ciphertext $\mathsf{Enc}(\mathbf{M} \cdot \tilde{\mathbf{q}})$ consists of $\lambda\sqrt{N}$ 64-bit integers.

6.2 Compressing the download

While the linearly homomorphic encryption scheme that we use makes homomorphic operations cheap, it has large ciphertexts—roughly $4\lambda \approx 4096$ times larger than the corresponding plaintext. To shrink the size of these ciphertexts, we introduce a trick inspired by the "bootstrapping" technique used in fully homomorphic encryption schemes [44].

To be concrete, let $\mathbf{C}$ be a ciphertext encrypting the dimension-$\sqrt{N}$ result of a matrix-vector product $\mathsf{Enc}(\mathbf{M} \cdot \tilde{\mathbf{q}})$. In the Regev scheme, the ciphertext $\mathbf{C}$ is a matrix of 64-bit integers of dimension roughly $\sqrt{N} \times \lambda$. The Regev secret key $\mathbf{s}$ is a vector of λ 64-bit integers. To decrypt the ciphertext $\mathbf{C}$ with secret key $\mathbf{s}$, the client just computes the matrix-vector product $\mathbf{y} = \mathbf{C} \cdot \mathbf{s}$, where all arithmetic is modulo 2^{64}. The decrypted message is in the high-order bits of each entry of the vector $\mathbf{y}$. In sum, decryption essentially requires computing a matrix-vector product modulo 2^{64}.

Our new technique, inspired by theoretical work on fully homomorphic encryption [19, 45, 75, 78], is to "outsource" the work of decrypting the large ciphertext $\mathbf{C}$ to the server:

1. The client encrypts its secret key $\mathbf{s}$ using a second linearly homomorphic encryption scheme Enc_2, which allows encrypting vectors of 64-bit integers. The client sends $\mathsf{Enc}_2(\mathbf{s})$ to the server.
2. The server, holding ciphertext $\mathbf{C}$, computes the matrix-vector product $\mathbf{C} \cdot \mathsf{Enc}_2(\mathbf{s}) = \mathsf{Enc}_2(\mathbf{C} \cdot \mathbf{s}) = \mathsf{Enc}_2(\mathbf{y})$ *under encryption*. That is, the server "decrypts" $\mathbf{C}$ under encryption.
3. The server returns $\mathsf{Enc}_2(\mathbf{y})$ to the client, who decrypts it.

The crucial observation here is that the encryption scheme Enc_2 can be *slow* as long as it has *compact ciphertexts* after homomorphic evaluation. More specifically, the computation $\mathbf{C} \cdot \mathsf{Enc}_2(\mathbf{s})$ involves a matrix $\mathbf{C}$ of size $\lambda\sqrt{N}$—much smaller than the original matrix $\mathbf{M}$, which has size N. Thus, the homomorphic operations using Enc_2 will not be a computational bottleneck, even if they are slow. We instantiate Enc_2 with an encryption scheme based on the ring learning-with-errors assumption [74]. We detail all cryptographic parameters used, as well as additional low-level optimizations, in Appendix C.

6.3 Reducing latency with query tokens

Finally, we push much of the client-to-server communication to a per-query preprocessing step. This optimization reduces the client-perceived latency between the moment that the client submits a search query and receives a response.

Using the optimization of §6.2, the client sends an encrypted secret key $\mathsf{Enc}_2(\mathbf{s})$ to the server and downloads the product $\mathbf{C} \cdot \mathsf{Enc}_2(\mathbf{s})$. Since the encryption of the secret key does not depend on the client's query, the client can send it to the Tiptoe services in advance. In Tiptoe, 99.9% of the ciphertext matrix $\mathbf{C}$ is also fixed—it just depends on the document corpus. (This portion of the matrix $\mathbf{C}$ corresponds to the *hint* in the SimplePIR encryption scheme [56].) Therefore, the ranking service can compute and return most of the bits of the product $\mathbf{C} \cdot \mathsf{Enc}_2(\mathbf{s})$ to the client *before* the client has decided on its query string.

We refer to the chunk of bits that the client downloads ahead of time as a "query token." The client can execute one search query per token; once the client has used a token, it may never use it again. (Otherwise, the security guarantees of the encryption scheme break down: the client would be using the same secret key $\mathbf{s}$ to encrypt multiple query vectors.) The client can fetch as many tokens as it wants in advance; these tokens are usable until the document corpus changes.

7 Implementation

The source for our Tiptoe prototype is available at `github.com/ahenzinger/tiptoe`. Our prototype consists of roughly 5 200 lines of code: 3 700 lines of Go and Python for the Tiptoe client and services, 1 500 lines of Python for the batch jobs, and 1 000 lines of Python for cluster management.

We implemented the core cryptosystem (§6) in a separate library in 1 000 lines of Go and 300 lines of C/C++, building on the SimplePIR codebase [56] and on Microsoft SEAL [121]. It is available at `github.com/ahenzinger/underhood`.

Embedding models. For text search, we use the `msmarco-distilbert-base-tas-b` text model, which outputs embedding vectors of dimension 768 [57, 58]. We compute each document's embedding over its first 512 tokens (the maximum that the model supports). We chose this embedding as it supports fast inference.

For text-to-image search, we use the CLIP embedding function [107], which maps text and images to the same dimension-512 vector-embedding space. Modifying our Tiptoe prototype to support plain image search (i.e., using an image to find similar images) only requires changing a few lines of code at the client.

Dimensionality reduction. Following prior work [84], Tiptoe reduces the dimension of its document embeddings by performing principal component analysis on the embeddings for the entire corpus. The principal-component-analysis algorithm outputs a linear function that projects the original embeddings down to a vector space of smaller dimension. The client downloads this function (0.6 MiB in size) and applies it locally to its query embedding before interacting with the Tiptoe services. We reduce the embedding dimension to 192 (from 768) for text search and to 384 (from 512) for image search; we measure the effect on search quality in §8.

Clustering. Tiptoe uses the Faiss library [62, 63] to group documents into clusters; for both text and image search, these clusters consist of approximately 50 000 documents. Tiptoe computes the clusters using a variant of k-means: we first compute centroids by running k-means over a subset of the data set (roughly 10 million documents), and then assign every document to the cluster with the closest centroid. To obtain roughly balanced clusters, we recursively split large clusters into multiple smaller ones.

A common technique to increase search quality in cluster-based nearest-neighbor-search is to assign a single document to multiple clusters [25, 61]. Following prior work [25], Tiptoe assigns documents to multiple clusters if they are close to cluster boundaries. In particular, Tiptoe assigns 20% of the documents to two clusters and the remaining 80% only to a single cluster, resulting in a roughly $1.2\times$ overhead in server computation and $\sqrt{1.2}\times$ overhead in communication. We show in §8 that this optimization improves search quality.

8 Evaluation

In this section, we answer the following questions:

- How good are Tiptoe's text-search results? (§8.2)
- What is the performance and cost of Tiptoe? (§8.3)
- How do Tiptoe's costs compare to those of other private-search systems? (§8.4)
- How well does Tiptoe scale to larger corpuses? (§8.5)
- To what extent do our optimizations reduce Tiptoe's search costs and affect its search quality? (§8.6)

8.1 Experimental setup

System configuration. We run the Tiptoe services on a cluster of many memory-optimized `r5.xlarge` AWS instances (with 4 vCPUs and 32 GiB of RAM each), since Tiptoe's server workload is bottlenecked by DRAM bandwidth. We allocate enough servers to each service to allow each machine's shard of the search index to fit in RAM, and to keep the client-perceived latency on the order of seconds. For text search, the ranking service runs on 40 servers, and the URL service runs on four servers. For image search, which runs over a $1.2\times$ larger corpus and uses a $2\times$ larger embedding dimension, the ranking service runs on 80 servers and the URL service on 8 servers. Each server holds roughly 8-12 GiB of data.

We additionally run a single front-end coordinator server, shared among both services, on a `r5.8xlarge` AWS instance (32 vCPUs, 256 GiB RAM). The coordinator performs all the server-side work in the query-token-generation step (§6.3), but fans out the client's queries to the corresponding machines in the ranking step and the URL-retrieval step.

Finally, we run a single Tiptoe client on a `r5.xlarge` AWS instance (4 vCPUs, 32 GiB of RAM) for text search, and on a `r5.2xlarge` AWS instance (8 vCPUs, 64 GiB of RAM) for image search. The simulated link between the client and the coordinator has 100 Mbps bandwidth with a 50 ms RTT. To measure query throughput, we simulate running up to 19 clients on one `r5.8xlarge` instance (32 vCPUs, 256 GiB of RAM), which generates enough load to saturate the servers.

To be conservative, when we report compute costs in "core-seconds," we measure the total number of AWS vCPUs-seconds paid for (counting idle cores).

Data sets. For text search, we search over the C4 data set, a cleaned version of the Common Crawl's English web crawl corpus with 364M web pages [108, 109]. The Common Crawl

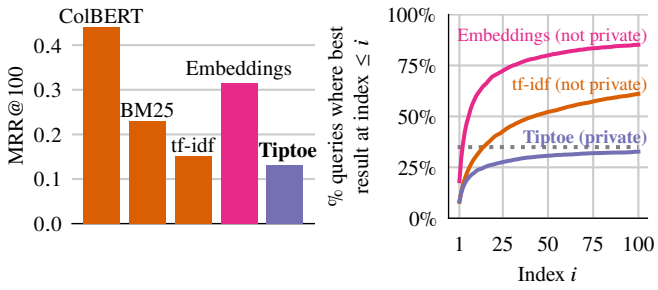

Figure 4: Search quality for the full-retrieval document ranking MS MARCO task. For tf-idf, we report the score with an unrestricted dictionary. On the left, we show MRR@100 scores. On the right, we show the percent of queries where the best (human-chosen) result appears at location $\leq i$. The dotted gray line represents the fraction of queries on which Tiptoe searches in the cluster containing the human-chosen answer, which bounds Tiptoe's search quality.

data set is not as comprehensive as the crawls used by commercial search engines such as Bing and Google. At the same time, this data set spans much of the web and is far larger than those considered by prior work (e.g., Coeus's search over 5M Wikipedia articles [5]). In §8.5, we analytically estimate how Tiptoe would scale to handle more documents.

For image search, we use the LAION-400M data set of 400 million images and English captions [119]. We deduplicate images and discard captions.

Since neither of these data sets contains ground-truth labels for search, we evaluate Tiptoe's search quality on the smaller MS MARCO document-ranking "dev" data set [92]. This data set contains 3.2M documents, along with 520 000 query-document pairs consisting of real Bing queries and human-chosen answers. We use the standard MRR@100 ("mean reciprocal rank at 100") search quality metric, which is the the inverse of the rank at which the true-best result appeared in the top 100 returned results, averaged over the test queries.

8.2 Search quality

In Figure 4, we compare the search quality on the MS MARCO document-ranking task for multiple search algorithms: a deep-learning-based state-of-the-art retrieval system (ColBERT) [66], a standard keyword-based retrieval system (BM25), the classic term frequency-inverse document frequency (tf-idf) algorithm, an exhaustive search over embeddings that ranks the documents by inner-product score (no clustering), and Tiptoe.

On the left, Figure 4 shows each algorithm's MRR@100 score: for ColBERT, we report the score from the MS MARCO leaderboard [80]; for BM25, we report the Anserini BM25 baseline document ranking with the default parameters ($k_1 = 0.9, b = 0.4$) [135]; for tf-idf, we use the Gensim library for stemming and building the tf-idf matrix [113]; for embedding search, we use the same embedding function as Tiptoe (but do not cluster the embeddings). As the MS MARCO data set is roughly 100× smaller than the C4 data set, for Tiptoe, we reduce the size of embedding and URL clusters by $\sqrt{100}\times = 10\times$. (Per §4.2, Tiptoe sets the cluster size proportionally to the square-root of the corpus size.) On average, the top search result apears at rank 7.7 with Tiptoe. This is worse than the search quality achieved with ColBERT, BM25, and exhaustive embedding search, but is comparable to that of tf-idf with an unrestricted dictionary size. In particular, Tiptoe's MRR@100 score is within 0.02 of tf-idf's.

The state-of-the-art system for private search over Wikipedia, Coeus [5], uses tf-idf with a dictionary restricted to only the 65K stemmed words with the highest inverse-document-frequency score (that is, the words that appear in the fewest documents). As the MS MARCO data set contains many document-specific keywords, we find that the MRR@100 score of tf-idf with Coeus's method of restricting the dictionary size is 0. By comparison, Tiptoe's use of embeddings allows it to generalize to large and diverse corpuses and vocabularies more effectively.

On the right, Figure 4 compares Tiptoe's distribution of search results to tf-idf and exhaustive embedding search. Tiptoe correctly identifies and searches within the cluster containing the best (human-chosen) result on roughly 35% of the queries. When it does so, Tiptoe roughly matches the search quality of the exhaustive search and ranks the human-chosen result higher on average than tf-idf. However, when it does not, the human-chosen result does not appear in the 100 search results returned by Tiptoe (because the human-chosen result does not appear in the cluster that the Tiptoe client is searching over). Querying more clusters could improve search quality, but would substantially increase Tiptoe's costs. One avenue for improving Tiptoe's search quality is thus to avoid the need for clustering: clustering allows Tiptoe to operate at web scale by drastically reducing communication costs, but accounts for a large share of the search-quality loss.

In Figure 5, we show Tiptoe's top search results on several randomly sampled queries, for both text and image search. Appendix E lists additional queries and results.

8.3 Tiptoe end-to-end performance

Table 6 shows the end-to-end performance of Tiptoe, as well as several private search baselines for comparison. Tiptoe's text search costs $0.003 per query ($0.008 for image search) and achieves an end-to-end query latency of 2.7 s (3.5 s for image search). Tiptoe's performance compares favorably to client-side baselines and to Coeus, as we now describe.

Baseline: Client-side search index. One approach for private search is to download and store a search index for the entire data set on the client. As shown in Table 6, locally storing Tiptoe's text search index requires at least 48 GiB of client storage. Alternatively, the client could directly download a search index for a state-of-the-art retrieval scheme like ColBERT or a keyword-based retrieval scheme like BM25. However,

```
Q: what test are relvant for heart screenings
A: https://newyorkcardiac.com/best-heart-palpitati
   ons-cardiac-doctor-nyc
Q: what is the ige antibody
A: https://www.empr.com/home/news/new-sc-inj-for-pri
   mary-immunodeficiency-approved/
Q: foodborne trematodiases symptoms
A: https://bowenmedicalibrary.wordpress.com/2017/
   04/04/foodborne-trematodiases/
```

```
Q: A train is next to an enclosed train station
A: https://en.wikipedia.org/wiki/File:Waynejunction
   0810a.JPG
Q: A man and a woman pose next to a small dog which
is wearing a life jacket
A: https://commons.wikimedia.org/wiki/File:Jack_Lon
   don_age_9_-_crop.jpg
Q: A young man wearing a tie and a blue shirt
A: https://commons.wikimedia.org/wiki/File:Emil_Jann
   ings_-_no_watermark.jpg
```

Figure 5: Sample of Tiptoe search results. At top, answers to random text-search queries from the MS MARCO dev document retrieval data set. At bottom, answers to random image-search queries from the MS COCO caption data set [72]; we used rejection sampling to select queries whose top results are public-domain images.

such client-side indexes would be orders of magnitude larger: roughly 4.6 TiB for BM25 or 6.4 TiB for ColBERT, perhaps reduced down to 0.9 TiB using techniques from PLAID [117]. (We estimate the size of the ColBERT and BM25 indices by scaling up the index sizes reported on the much smaller MS MARCO document and passage ranking data sets, with the same configuration we use to report search quality.) Existing tools [30] could compress the client-side index further at the expense of search quality, but at an absolute minimum would require 7.4 GiB of storage just for the compressed URLs.

Baseline: Coeus query-scoring. Coeus is a state-of-the-art text-search system that uses tf-idf (with a limited vocabulary) to privately search over five million Wikipedia articles, running on 96 machines with 48 vCPUs each [5]. Coeus supports private search ("query scoring," in their terminology) and private document retrieval; to compare against Tiptoe, we use Coeus's query-scoring costs only. Like Tiptoe, Coeus's server-side work grows linearly with the number of documents in the corpus. We estimate that, searching over N documents, Coeus's query-scoring requires $10.66 \cdot N$ bytes of communication. (We obtained this formula via private communication with the Coeus authors.) Scaling Coeus's performance to the size of the C4 web crawl, which is roughly 72× larger than Wikipedia, we estimate that each query with Coeus would require more than 3 GiB of download, 900 000 core-seconds of server compute, and $4.00 in AWS cost.

In comparison to Coeus, Tiptoe has more than 1 000× lower

Cost metrics:	Client storage (GiB)	Comm. (MiB/query)	Server comp. (core-s/query)	End-to-end latency (s)	AWS cost ($/query)
Wikipedia search over 5 million docs					
Coeus query-scoring [5]	0	50	12 900	2.8	0.059
Text search over 360 million docs					
Client-side Tiptoe index	48	0	0	0	0
Tiptoe	0.3	42$^\diamond$ + 15	145	2.7	0.003
Image search over 400 million docs					
Client-side Tiptoe index	98	0	0	0	0
Tiptoe	0.7	50$^\diamond$ + 21	339	3.5	0.008

Table 6: Comparison of Tiptoe to private search alternatives: (1) Coeus and (2) downloading the Tiptoe index to the client. We give Coeus's reported costs [5]. We compute Tiptoe's AWS cost using list prices ($0.252/hour for `r5.xlarge`, $2.016/hour for `r5.8xlarge`, and $0.09/GiB of egress bandwidth). AWS costs do not include one-time download costs that may be amortized over any number of queries (i.e., the embedding model and centroid metadata). We highlight Tiptoe's performance in yellow. $^\diamond$ This traffic occurs before the client enters its search query (§6.3).

AWS operating costs. We attribute Tiptoe's performance improvements over Coeus to: (1) semantic embeddings, which are two orders of magnitude smaller than the rows in a tf-idf matrix (whose dimension must scale with the size of the dictionary used); (2) clustering, which allows Tiptoe's communication to scale sublinearly with the number of documents; and (3) the high-throughput cryptographic protocols [56] used by Tiptoe, which are roughly an order of magnitude faster than prior single-server private-information-retrieval schemes.

Image search. A Tiptoe search over the LAION-400M image data set, which is 1.2× larger than the C4 text data set and uses 2× larger embeddings, is roughly twice as expensive as text search in AWS cost: it uses 2.3× more compute (339 core-seconds/query) and 1.2× more communication (71 MiB/query, of which 50 MiB can occur ahead of time).

8.4 Tiptoe cost breakdown

We now detail the operating costs for each of the entities in a Tiptoe deployment (§3.2):

Data-loading and setup costs. In Table 7, we report approximate core-hours for Tiptoe's data-loading batch jobs. To assign documents to clusters, we used a single, large shared machine; the running time varied widely with the number of concurrent active jobs. We balanced the embedding clusters and ran principal component analysis using a set of 100 `r5.2xlarge` instances (32 vCPUs, 128 GiB of RAM). Finally, we constructed the data structures held by the Tiptoe services on one `r5.24xlarge` instance (96 vCPUs, 768 GiB of RAM). In total, Tiptoe's data-processing pipeline requires roughly 0.01-0.02 core-seconds per document.

Setup cost	Text	Image
Corpus size		
Documents	364M	400M
Embedding dim.	192	384
Index preprocessing (core hours)		
Embed (est.)♡	92	583
Build centroids	224	262
Cluster assign.	703	1,231
Balance, PCA	312	290
Crypto.	50	120
Total (core-s/doc)	*0.013*	*0.022*
Client download (GiB)		
Model	0.27	0.59
Centroids	0.02	0.02
System configuration (vCPUs)		
Token♢	32	32
Ranking	160	320
URL	16	32

Query cost	Text	Image
Communication (MiB/query)		
Up, token♢	32.4	32.4
Up, ranking	11.6	16.2
Up, URL	2.4	3.2
Down, token♢	9.8	17.4
Down, ranking	0.5	1.0
Down, URL	0.1	0.2
Preprocessing (s/query)		
Client (§6)	37.7	36.7
Query latency (s)		
Token♢	6.5	8.7
Ranking	1.9	2.5
URL	0.6	0.6
Tot. perceived (s)	2.7	3.5
Throughput (query/s)		
Token♢	0.5	0.2
Ranking	2.9	2.3
URL	5.0	7.0

Table 7: Breakdown of Tiptoe costs for text and image search. We computed text embeddings on a heterogeneous GPU cluster and we sourced image embeddings from the LAION-400M data set. ♡ Estimated GPU-hours on a V100 GPU. ♢ This step occurs before the client enters its search query (§6.3).

Search costs. Table 7 additionally gives Tiptoe's search costs. With Tiptoe's optimizations (§6), many of the cryptographic operations, as well as more than 70% of the client-server communication, happen before the client has decided on its search query. For text search, our cluster of machines can sustain a throughput of 0.5 queries/s for token generation, 2.9 queries/s for ranking, and 5 queries/s for URL retrieval.

8.5 Scaling to tens of billions of documents

Popular search engines index tens of billions of documents or more [51]. In Figure 8, we analytically compute how Tiptoe's search costs would scale to handle data sets of this size. If we increase the corpus size by a factor of T, the server compute increases by roughly a factor of T and the communication increases by roughly a factor of $\sqrt{T}$. For example, on a corpus of 8 billion documents, the number of Google knowledge graph entities (as of March 2023) [3], a Tiptoe search query would require roughly 1 900 core-seconds of compute and 140 MiB of communication.

Moreover, Tiptoe's throughput scales linearly with the numbers of physical machines used: doubling the machines allocated to each service roughly doubles the measured throughput. Tiptoe can support dynamically adding more physical machines at runtime, by either having more machines serve each shard of the database or having each machine serve a smaller shard (without repeating any of the preprocessing).

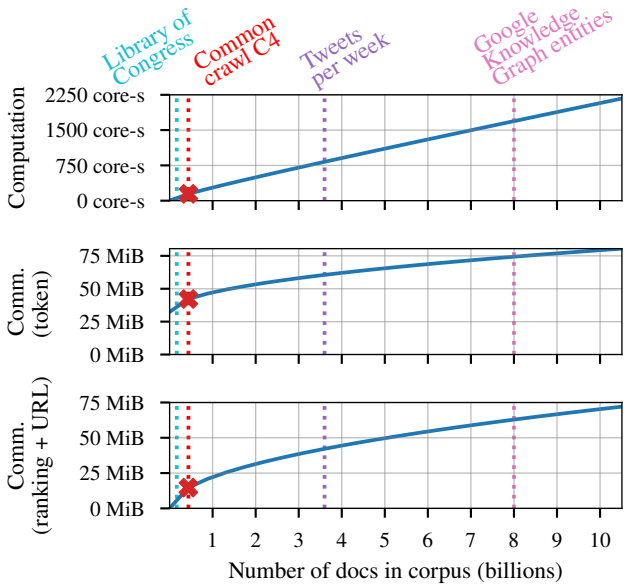

Figure 8: Analytical Tiptoe per-query performance, scaling to large text corpuses [1,2,3,108]. The cross indicates measured performance.

8.6 Impact of optimizations

Figure 9 shows how Tiptoe's optimizations trade off between text-search quality and performance. First (not shown in Figure 9), we reduce the embedding precision from floating point values to signed 4-bit integers, decreasing MRR@100 by 0.005. Without other optimizations ❶, the client must retrieve an inner-product score for every document, resulting in communication similar to that of Coeus's query scoring. With clustering ❷, computation is constant, but communication shrinks by 20× since the client now only downloads inner-product scores for one cluster. MRR@100 also decreases by 0.2, as the best result might not be in the chosen cluster.

Without any URL optimizations, the client must run SimplePIR to individually retrieve each of the 100 URLs displayed to the client. (We use 100 because MRR@100 considers the top 100 results.) Compressing chunks of URLs and retrieving only the chunk with the top result ❸ reduces communication and computation by 4× at the cost of a drop of 0.04 in MRR@100. Clustering URLs into semantically similar batches ❹ does not affect communication and computation, but improves MRR@100 by 0.04.

Assigning documents at cluster boundaries to two clusters ❺ increases index size by 1.2× but improves MRR@100 by 0.015. Finally, ❻ reducing the embedding dimension by 3× with PCA improves the bandwidth and the computation by roughly 2×, but decreases MRR@100 by 0.02. Overall, Tiptoe's optimizations improve communication by two orders of magnitude and computation by one order of magnitude, at the cost of a 0.2 drop in MRR@100—on average, the top result appears at position 7.7 rather than at position 3.

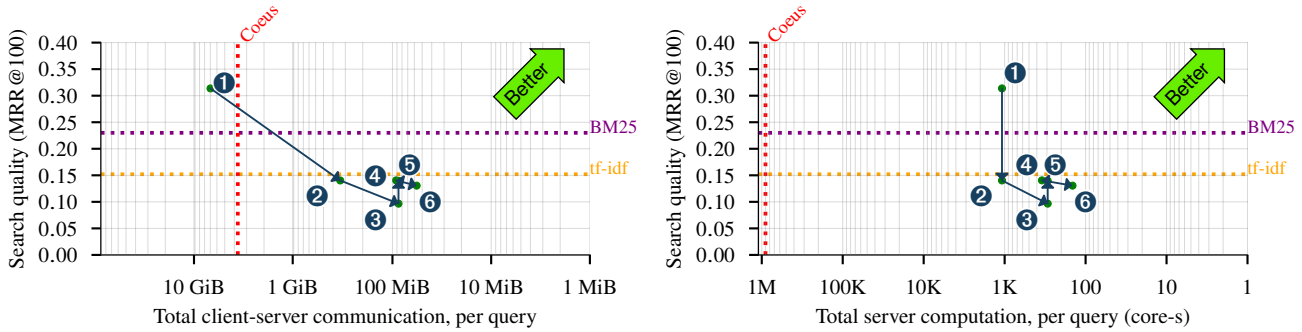

Figure 9: Analytical impact of optimizations on Tiptoe's text search performance. We compute MRR@100 scores (y-axis) on the MS MARCO data set and performance numbers (x-axis) on the C4 data set. We report measured performance for full Tiptoe, but expected performance for (a) Coeus and (b) Tiptoe without some optimizations. ❶ No optimizations (return inner-product score for every document and run a private-information-retrieval scheme to retrieve the top 100 results, using a SEAL PIR-like homomorphic encryption scheme [8]). ❷ Cluster embeddings and only return inner-product scores for a cluster. ❸ Within a cluster, retrieve a random chunk of URLs containing the top result and output the top 100 results from this chunk. ❹ Cluster the URLs to retrieve a batch of related URLs containing the top result and output the top 100 results from this batch. ❺ Assign documents at cluster boundaries to 2 clusters. ❻ Reduce the embedding size by 3× with PCA (full Tiptoe).

9 Discussion

In this section, we discuss extensions and related topics pertaining to Tiptoe.

Private advertising. Search engines make money by displaying relevant ads alongside search results [52]. Tiptoe is compatible with this business model: just as a client uses Tiptoe to fetch relevant webpages, a client could use Tiptoe to fetch relevant textual ads. The search provider could embed each ad using an embedding function. The client would then use Tiptoe to identify the ads most relevant to its query—instead of privately fetching a URL in the last protocol step, the client would privately fetch the text of the ad.

The privacy guarantees here hold only until the client clicks on the ad. This type of private ad retrieval may be compatible with techniques for private impression reporting [53, 127].

Private recommendations. Tiptoe's private nearest-neighbor search protocol may be useful in applications beyond private web search, for example in private recommendation engines. Just like text and images, items can also be represented by embeddings that map semantic proximity to vector-space proximity. In a recommendation system, the client can hold a vector representing its profile or its recently viewed items. Then, with Tiptoe's private nearest-neighbor search protocol, the client can privately retrieve similar items from the recommendation system's servers.

Private search on encrypted data. Tiptoe can be extended to search over *encrypted* documents. To do so, the client processes the corpus (as done, in our prototype, by the Tiptoe batch jobs): the client embeds each document, clusters the embeddings, and stores the centroids locally. Instead of storing the plaintext embeddings and URLs on the Tiptoe servers, the client encrypts the embeddings and URLs and stores the *encrypted* search data structures on the Tiptoe servers.

Later on, the client can search over these documents while revealing no information about its query or the corpus (apart from the total corpus size) to the server. The only difference to Tiptoe is that, in the ranking step, the server must now compute the inner product of the client's encrypted query embedding with each *encrypted* document vector. This is possible using a homomorphic encryption scheme that supports *degree-two computations* on encrypted data [17].

Reducing communication with non-colluding services. In Tiptoe, the client interacts with a single logical server, which may be adversarial. If instead the client can communicate with two search services assumed to be non-colluding, we can forgo the use of encryption to substantially reduce the communication costs. To execute the nearest-neighbor search within a cluster, the client would share an encoding of its query embedding (vector $\tilde{\mathbf{q}}$ in Figure 10) using a distributed point function [18]. The servers could execute the nearest-neighbor search protocol of §4 on a secret-shared query, instead of an encrypted one. No server-to-server communication would be necessary, as the servers only perform linear operations. URL-fetching would work exactly as in Tiptoe, except with two-server private information retrieval. We estimate that the per-query communication on the C4 data set would be roughly 1 MiB (instead of Tiptoe's 56.9 MiB).

Exact keyword search. Tiptoe's embedding-based search algorithm does not perform well on textual queries for rare strings, such as phone numbers, addresses, and uncommon names. One way to extend Tiptoe to handle such queries would be to construct a suite of search backends—one for each common type of exact-string query. For example, there would be a private phone-number search backend, a private address-search backend, etc. Each backend would implement a simple private key-value store mapping each string in the corpus (e.g., each phone number) in some canonical format to

the IDs of documents containing that string. The client could use a standard keyword-based private-information-retrieval scheme [4, 27] to privately query this key-value store.

Upon receiving a query string, the Tiptoe client software would attempt to extract a string of each supported type (phone number, address, etc.) from the query string. The client software would canonicalize the query string and use it to make a key-value lookup to the corresponding backend.

Personalized search. Tiptoe could potentially support personalized search by incorporating a client-side embedding function that takes as input not only the user's query, but also the user's search profile. As an example, the embedding function could take as input the user's location so that a query for "restaurants" would return restaurants that are nearby. The servers could continue using their embedding function that does not take a search profile as input, but that preserves the distance to outputs of the client's embedding function.

10 Related work

Three popular non-cryptographic methods have been used to strenghten privacy for web search. The first is to send search queries to a conventional search engine via an anonymizing proxy, such as Tor [36] or a mix-net [22]. The second approach uses dummy queries or obfuscates queries [11, 15, 37, 89, 93, 101, 111, 136]. Both techniques still reveal some version of the client's query to the search engine, allowing it to link queries and deanonymize users [46, 99, 100]. The third approach is to use trusted hardware [70, 83, 102], which provides security guarantees that are only as strong as the underlying hardware [23, 26, 54, 60, 88, 98, 110, 120, 126, 129, 130, 131, 132]. Because trusted execution environments leak memory-access patterns, a trusted-hardware-based approach would need to use oblivious RAM [48] to hide memory-access patterns, incurring additional overheads [32, 82, 118].

DuckDuckGo and other privacy-preserving search engines do not track users and route queries to search back-ends without identifying information. However, the search back-ends similarly still learn the client's search query in plaintext.

Coeus [5] is a private Wikipedia search system with security properties matching those of Tiptoe, but at orders of magnitude higher costs (Table 6). While Tiptoe uses embedding-based search, which works well for conceptual queries, Coeus uses tf-idf search, which better handles exact-string matching but cannot support non-text search.

Cryptographic private-information-retrieval protocols [28, 67] also perform private lookups, though they only natively support private array lookups or key-value queries, rather than full-text string queries. The performance of private-information-retrieval systems has improved by orders of magnitude in recent years [4, 7, 8, 9, 56, 77, 79, 86], taking advantage of fast lattice-based encryption techniques [112]. A number of recent works have shown how to build single-server sublinear-time private-information-retrieval protocols. However, these schemes still have high per-query overheads in practice [29, 73] or require streaming the entire database to the client [87, 137], which is impractical for a large database that changes regularly. Tiptoe also exploits lattice-based encryption schemes (§4) and uses private information retrieval as a subroutine (§5).

Splinter [133] is a system that extends private information retrieval to support more expressive query types (e.g., range queries with various aggregation functions), provided that the client can communicate with two non-colluding database replicas. Splinter does not support full-text search.

An orthogonal line of work has developed cryptographic protocols and systems for private search on *private* data. These techniques include symmetric searchable encryption [20, 21, 31, 35, 43, 47, 64, 65, 90, 116, 124, 125], their realization in encrypted databases [42, 96, 97, 103, 104, 128], and related systems [33, 34]. In contrast, Tiptoe allows private queries to a *public* document corpus.

Tiptoe uses semantic embeddings to reduce the problem of text (or image) search to that of private nearest-neighbor search. Many prior works have constructed protocols for private nearest-neighbor search, though these systems in general require multiple non-colluding services [24, 59, 105, 122], rely on computationally heavy cryptographic tools, which practically limits the corpus to a few thousand entries [123, 139], or leak some information about the client's query to the server [12, 14, 115]. In contrast, Tiptoe requires only a single infrastructure provider, searches over hundreds of millions of documents, and leaks no infomation about the client's query.

Some prior work also uses embeddings with homomorphic encryption for text search [12] or image matching [38], although these either leak information about the client's query or incur communication costs linear in the number of documents and high computation costs (as much as three hours of computation with ten cores for 100 million records). A distinguishing point is that some of these works [38, 105, 122, 123] provide two-sided privacy: they reveal little about the server's data set to the client, apart from the query result. Tiptoe, on the other hand, assumes that the server's data set is public.

Tiptoe uses clustering to implement nearest-neighbor-search with few client-server round trips. The task of designing a clustering-based index for private search is similar to that of designing a (non-private) nearest-neighbor-search algorithm optimized to minimize for disk accesses: both cases aim to minimize the number of reads to the index [25, 61].

Tiptoe draws inspiration from non-private embedding-based information-retrieval systems. Many such systems rely on dense document representations and leverage approximate nearest neighbors techniques to efficiently find the closest documents [58, 71, 106, 114]. An alternative approach uses a sparse document representation, which can capture information at the token level, such as the classic BM25 algorithm or the newer SPLADE model [41, 69].

11 Conclusion

Tiptoe demonstrates that a client can search over hundreds of millions of server-held documents at modest cost, while hiding its query from the server. The key idea in Tiptoe is to use semantic embeddings to reduce a complex problem (text/image search) to a simple and crypto-friendly one (nearest-neighbor search). We expect this technique may be broadly useful in privacy-protecting systems.

Acknowledgements. We thank the anonymous SOSP reviewers for their thoughtful feedback, and we thank Stefan Savage for serving as our shepherd for this paper. Eric Rescorla encouraged us to work on this problem from the start. We thank Leo de Castro, Ryan Lehmkuhl, Vinod Vaikuntanathan and Yael Kalai for helpful discussions about fully homomorphic encryption, and we thank Jacob Andreas for tips on natural-language embeddings. Anish Athalye, Christian Mouchet, David Mazières, Dima Kogan, Frans Kaashoek, Matei Zaharia, and Raluca Ada Popa gave suggestions that improved the presentation. Dan Boneh, Sam Hopkins, Stefan Groha, and Stefan Heule gave helpful comments on half-baked early versions of this work. Larry Gibbs suggested the system name. This work was funded in part by gifts from Capital One, Facebook, Google, Mozilla, NASDAQ, and MIT's FinTech@CSAIL Initiative. We also received support under NSF Award CNS-2054869 and from the RISE and Sky labs at UC Berkeley. Alexandra Henzinger was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 2141064 and an EECS Great Educators Fellowship. Emma Dauterman was supported by an NSF Graduate Research Fellowship and a Microsoft Ada Lovelace Research Fellowship.

References

[1] New tweets per second record, and how! https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how, accessed 17 April 2023, 2013.

[2] Annual report of the librarian of congress. https://www.loc.gov/static/portals/about/reports-and-budgets/documents/annual-reports/fy2021.pdf, accessed 17 April 2023, 2021.

[3] Google knowledge graph. https://en.wikipedia.org/wiki/Google_Knowledge_Graph, accessed 17 April 2023, 2023.

[4] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Pantheon: Private retrieval from public key-value store. *Proceedings of the VLDB Endowment*, 16(4):643–656, 2022.

[5] 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 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 672–690, Virtual conference, October 2021.

[6] Martin Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[7] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation trade-offs in PIR. In *Proceedings of the 30th USENIX Security Symposium*, pages 1811–1828, Vancouver, Canada, August 2021.

[8] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, pages 962–979, San Francisco, CA, May 2018.

[9] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–569, Savannah, GA, November 2016.

[10] Apache Software Foundation. Lucene, 2023. https://lucene.apache.org/.

[11] Avi Arampatzis, George Drosatos, and Pavlos S. Efraimidis. Versatile query scrambling for private web search. *Information Retrieval Journal*, 18:331–358, 2015.

[12] 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.

[13] Michael Arrington. AOL proudly releases massive amounts of private data, August 2006. https://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/.

[14] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Francesco Silvestri. Distance-sensitive hashing. In *Proceedings of the 37th ACM Symposium on Principles of Database Systems (PODS)*, pages 89–104, Houston, TX, June 2018.

[15] Ero Balsa, Carmela Troncoso, and Claudia Diaz. OB-PWS: Obfuscation-based private web search. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 491–505, San Francisco, CA, May 2012.

[16] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, 2004.

[17] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proceedings of the 2nd IACR Theory of Cryptography Conference (TCC)*, pages 325–341, Cambridge, MA, February 2005.

[18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 1292–1303, Vienna, Austria, October 2016.

[19] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *Proceedings of the 17th IACR Theory of Cryptography Conference (TCC)*, Nuremberg, Germany, December 2019.

[20] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, pages 23–26, San Diego, CA, February 2014.

[21] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of the 11th Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 442–455, Chennai, India, December 2005.

[22] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), February 1981.

[23] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE: Stealing intel secrets from SGX enclaves via speculative execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy*, Stockholm, Sweden, June 2019.

[24] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M Sadegh Riazi. SANNS: Scaling up secure approximate k-nearest neighbors search. In *Proceedings of the 29th USENIX Security Symposium*, pages 2111–2128, Virtual conference, August 2020.

[25] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient billion-scale approximate nearest neighbor search, 2021. `https://arxiv.org/abs/2111.08566`.

[26] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *Proceedings of the 30th USENIX Security Symposium*, Vancouver, Canada, August 2021.

[27] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Paper 1998/003, February 1998. `https://eprint.iacr.org/1998/003`.

[28] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[29] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Trondheim, Norway, May 2022.

[30] Criteo. AutoFaiss. `https://github.com/criteo/autofaiss`, accessed 16 April 2023, 2023.

[31] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[32] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 655–671, Virtual conference, October 2021.

[33] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1101–1119, Virtual conference, November 2020.

[34] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2022.

[35] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *Proceedings of the 29th USENIX Security Symposium*, Virtual conference, August 2020.

[36] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, August 2004.

[37] Josep Domingo-Ferrer, Agusti Solanas, and Jordi Castellà-Roca. h(k)-private information retrieval from privacy-uncooperative queryable databases. *Online Information Review*, 33(4):720–744, 2009.

[38] 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.

[39] Hugging Face. Semantic search with FAISS, 2023. `https://huggingface.co/learn/nlp-course/chapter5/6?fw=tf`.

[40] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, March 2012. `https://eprint.iacr.org/2012/144`.

[41] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. SPLADE: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 2288–2292, Virtual conference, July 2021.

[42] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. SoK: Cryptographically protected database search. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, pages 172–191, San Jose, CA, May 2017.

[43] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, pages 563–592, Santa Barbara, CA, August 2016.

[44] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178, Bethesda, MD, May–June 2009.

[45] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In *Proceedings of the 17th IACR Theory of Cryptography Conference (TCC)*, Nuremberg, Germany, December 2019.

[46] Arthur Gervais, Reza Shokri, Adish Singla, Srdjan Capkun, and Vincent Lenders. Quantifying web-search privacy. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 966–977, Scottsdale, AZ, November 2014.

[47] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Paper 2003/216, October 2003. `https://eprint.iacr.org/2003/216`.

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

[49] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[50] Google. In-depth guide to how google search works. `https://developers.google.com/search/docs/fundamentals/how-search-works`.

[51] Google. How Google Search organizes information. `https://www.google.com/search/howsearchworks/how-search-works/organizing-information/`, accessed 17 April 2023, 2023.

[52] Google. How our business works. `https://about.google/how-our-business-works/`, accessed 17 April 2023, 2023.

[53] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[54] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of Rowhammer defenses. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2018.

[55] Katie Hafner. Researchers yearn to use AOL logs, but they hesitate, August 2006. `https://www.nytimes.com/2006/08/23/technology/23search.html`.

[56] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In

Proceedings of the 32nd USENIX Security Symposium, Anaheim, CA, August 2022.

[57] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. msmarco-distilbert-base-tas-b model. https://huggingface.co/sentence-transformers/msmarco-distilbert-base-tas-b.

[58] Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. Efficiently teaching an effective dense retriever with balanced topic aware sampling. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 113–122, Virtual conference, July 2021.

[59] Piotr Indyk and David Woodruff. Polylogarithmic private approximations and efficient matching. In *Proceedings of the 3rd IACR Theory of Cryptography Conference (TCC)*, pages 245–264, New York, NY, March 2006.

[60] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, October 2017.

[61] 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.

[62] Hervé Jegou, Matthijs Douze, and Jeff Johnson. Faiss: A library for efficient similarity search. https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/, March 29, 2017.

[63] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[64] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Proceedings of the 17th International Financial Cryptography and Data Security Conference*, pages 258–274, Okinawa, Japan, April 2013.

[65] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 965–976, Raleigh, NC, October 2012.

[66] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 39–48, Virtual conference, July 2020.

[67] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, Miami Beach, FL, October 1997.

[68] UKP Lab. Sentence transformers: Multilingual sentence, paragraph, and image embeddings using BERT & co. https://github.com/UKPLab/sentence-transformers.

[69] Carlos Lassance and Stéphane Clinchant. An efficiency study for SPLADE models. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 2220–2226, Madrid, Spain, July 2022.

[70] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. Bringing decentralized search to decentralized services. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 331–347, Virtual conference, July 2021.

[71] Sheng-Chieh Lin, Jheng-Hong Yang, and Jimmy Lin. In-batch negatives for knowledge distillation with tightly-coupled teachers for dense retrieval. In *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP)*, pages 163–173, 2021.

[72] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Proceedings of the 13th European Conference on Computer Vision*, pages 740–755, Zurich, Switzerland, September 2014.

[73] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 595–608, Orlando, FL, June 2023.

[74] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Proceedings of the 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Monaco and Nice, France, May–June 2010.

[75] Rasoul Akhavan Mahdavi, Abdulrahman Diaa, and Florian Kerschbaum. HE is all you need: Compressing FHE ciphertexts using additive HE, 2023.

[76] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM computing surveys (CSUR)*, 54(6):1–35, 2021.

[77] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the 16th Privacy Enhancing Technologies Symposium*, Darmstadt, Germany, July 2016.

[78] Carlos Aguilar Melchor, Philippe Gaborit, and Javier Herranz. Additively homomorphic encryption with d-operand multiplications. In *Proceedings of the 30th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, August 2010.

[79] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2022.

[80] Microsoft. MS MARCO document ranking leaderboard. `https://microsoft.github.io/MSMARCO-Document-Ranking-Submissions/leaderboard/`, accessed 15 April 2023.

[81] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR)*, Scottsdale, AZ, May 2013.

[82] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2018.

[83] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. X-search: revisiting private web search using Intel SGX. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 198–208, 2017.

[84] Jiaqi Mu and Pramod Viswanath. All-but-the-top: Simple and effective post-processing for word representations. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, Vancouver, Canada, April–May 2018.

[85] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. MTEB: Massive text embedding benchmark, 2023. `https://arxiv.org/abs/2210.07316`.

[86] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, page 2292–2306, Virtual conference, November 2021.

[87] Muhammad Haris Mughees and Ling Ren. Simple and practical single-server sublinear private information retrieval. *Cryptology ePrint Archive*, 2023.

[88] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2020.

[89] Mummoorthy Murugesan and Chris Clifton. Providing privacy through plausibly deniable search. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 768–779. SIAM, 2009.

[90] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 639–654, San Jose, CA, May 2014.

[91] Pandu Nayak. Understanding searches better than ever before, 2019. `https://blog.google/products/search/search-language-understanding-bert/`.

[92] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. MS MARCO: A human generated machine reading comprehension dataset. In *Proceedings of the Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches (CoCo@NIPS)*, Barcelona, Spain, December 2016.

[93] Helen Nissenbaum and Howe Daniel. TrackMeNot: Resisting surveillance in web search. In *Lessons from the Identity Trail: Anonymity, Privacy, and Identity in a Networked Society*. Oxford University Press, 2009.

[94] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *Proceedings of the 17th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Helsinki, Finland, May–June 1998.

[95] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, Prague, Czech Republic, May 1999.

[96] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 587–602, Savannah, GA, November 2016.

[97] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 359–374, San Jose, CA, May 2014.

[98] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, Oakland, CA, May 2011.

[99] Sai Teja Peddinti and Nitesh Saxena. Web search query privacy: Evaluating query obfuscation and anonymizing networks. *Journal of Computer Security*, 22(1):155–199, 2014.

[100] Albin Petit, Thomas Cerqueus, Antoine Boutet, Sonia Ben Mokhtar, David Coquil, Lionel Brunie, and Harald Kosch. SimAttack: private web search under fire. *Journal of Internet Services and Applications*, 7(1):1–17, 2016.

[101] Albin Petit, Thomas Cerqueus, Sonia Ben Mokhtar, Lionel Brunie, and Harald Kosch. PEAS: Private, efficient and accurate web search. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, pages 571–580, 2015.

[102] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. CYCLOSA: Decentralizing private web search through SGX-based browser extensions. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 467–477, Vienna, Austria, July 2018.

[103] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, pages 1664–1678, Los Angeles, CA, August 2019.

[104] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, October 2011.

[105] Yinian Qi and Mikhail J. Atallah. Efficient privacy-preserving k-nearest neighbor search. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 311–319, Beijing, China, June 2008.

[106] Yingqi Qu, Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Wayne Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. RocketQA: An optimized training approach to dense passage retrieval for open-domain question answering. 2020. `https://arxiv.org/abs/2010.08191`.

[107] 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. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 8748–8763, Virtual conference, July 2021.

[108] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. C4 model. `https://huggingface.co/datasets/c4`.

[109] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019. `https://arxiv.org/abs/1910.10683`.

[110] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, Virtual conference, May 2021.

[111] David Rebollo-Monedero and Jordi Forné. Optimized query forgery for private information retrieval. *IEEE Transactions on Information Theory*, 56(9):4631–4642, 2010.

[112] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, 2009.

[113] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. `http://is.muni.cz/publication/884893/en`.

[114] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks, 2019. https://arxiv.org/abs/1908.10084.

[115] M Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan Wallach, and Farinaz Koushanfar. Sub-linear privacy-preserving near-neighbor search, 2016. https://arxiv.org/abs/1612.01835.

[116] Panagiotis Rizomiliotis and Stefanos Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Cloud Computing Security Workshop (CCSW)*, pages 65–76, Denver, CO, October 2015.

[117] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. Plaid: an efficient engine for late interaction retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 1747–1756, 2022.

[118] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[119] 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. In *Proceedings of the 2021 NeurIPS Data-Centric AI Workshop*, December 2021.

[120] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, United Kingdom, November 2019.

[121] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[122] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. Private approximate nearest neighbor search with sublinear communication. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, pages 911–929, San Francisco, CA, May 2022.

[123] Hayim Shaul, Dan Feldman, and Daniela Rus. Secure k-ish nearest neighbors classifier, 2018. https://arxiv.org/abs/1801.07301.

[124] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, May 2000.

[125] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, pages 72–75, San Diego, CA, February 2014.

[126] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *Proceedings of the 26th USENIX Security Symposium*, Vancouver, Canada, August 2017.

[127] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2010.

[128] Stephen Tu, M. Frans Kaashoek, Samuel R. Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[129] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, August 2018.

[130] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2020.

[131] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[132] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking

data on Intel CPUs via cache evictions. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, Virtual conference, May 2021.

[133] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 299–313, Boston, MA, March 2017.

[134] Xapian. Xapian. `https://xapian.org/`.

[135] Peilin Yang, Hui Fang, and Jimmy Lin. Anserini: Enabling the use of Lucene for information retrieval research. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 1253–1256, Tokyo, Japan, August 2017.

[136] Shaozhi Ye, Felix Wu, Raju Pandey, and Hao Chen. Noise injection for search privacy protection. In *Proceedings of the 2009 International Conference on Computational Science and Engineering*, August 2009.

[137] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. *Cryptology ePrint Archive*, 2023.

[138] Jeffrey Zhu. Bing delivers its largest improvement in search experience using Azure GPUs, 2019. `https://azure.microsoft.com/en-us/blog/bing-delivers-its-largest-improvement-in-search-experience-using-azure-gpus/`.

[139] Martin Zuber and Renaud Sirdey. Efficient homomorphic evaluation of k-NN classifiers. In *Proceedings of the 21st Privacy Enhancing Technologies Symposium*, pages 111–129, Virtual conference, July 2021.