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, 57]. 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 [37], the query itself can contain personally identifying information, and similarities across queries can link requests and deanonymize the user [102, 103].

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, 52, 137]. 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 [29, 70] generally require the servers to scan the entire data set in response to each query [5, 9, 136]—otherwise, the servers would learn which parts of the data set were not relevant to the query [17]. 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 [111], 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 [97, 98] and private information retrieval [59], 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 [41, 94, 141], 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 [71, 84, 117]. 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 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 [59, 115] 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 [111]. 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 [122].

To evaluate the quality of Tiptoe’s search results, we use the MS MARCO search-quality benchmark for end-to-end document retrieval [95]. 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 [51]:

Definition 2.1 (Query privacy). For a query string \( q \in \{0, 1\}^* \) and an adversarial search service \( A \), let \( M_{A,q} \) 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 \( A \), the corresponding probability distributions of \( M_{A,q_0} \) and \( M_{A,q_1} \) 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 [37], mix-nets [23]) 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 [79]. 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.

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
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.

2. Embeddings allow Tiptoe to natively support expressive queries—without any special machinery for keyword intersection or term-frequency analysis.

Diagram:

![Diagram of Tiptoe's semantic search with embeddings.]

- Document embedding
- Cluster of documents
- Cluster centroid
- Embedding of search query
- Cluster to search over
- Nearest neighbor in cluster

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

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 [88], including transformer models [117].

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% of 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 [46].

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 [59].)

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 [29] 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.

Tiptoe implements this ranking step using a new private nearest-neighbor search protocol. On a corpus of \( N \) documents with \( d \)-dimensional 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.

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 \( q \) 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 \( q \). For now, we think of the embedding as consisting of \( d \) integers; we discuss how to handle real-valued vectors in §4.3.

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 \( q \).
which is zero everywhere except that it contains the client’s vector \( \mathbf{v} \) chosen cluster \( i^* \). Let \( M \) be the augmented query vector of \( dC \) integers, which is zero everywhere except that it contains the client’s query vector \( \mathbf{v} \) 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 \( \mathbf{c}_t = \text{Enc}(\mathbf{v}) \). The client sends this ciphertext \( \mathbf{c}_t \) 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 \( \mathbf{v} \), 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 \( M \). If there are \( C \) total document embeddings, grouped into \( C \) clusters, the ranking service arranges these vectors into a matrix \( M \) 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 \( M \) 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 \( \mathbf{c}_t \) from the client, the server computes the matrix-vector product \( \mathbf{M} \cdot \mathbf{c}_t \). Since \( \text{Enc} \) is a linearly homomorphic encryption scheme, it holds that \( \mathbf{c}_t' = \mathbf{M} \cdot \text{Enc}(\mathbf{v}) = \text{Enc}(\mathbf{M} \cdot \mathbf{v}) \)—meaning that the ranking service has just computed the product \( \mathbf{M} \cdot \mathbf{v} \) under encryption. The server returns the resulting ciphertext \( \mathbf{c}_t' \) to the client.

**Step 3: Client decryption.** The client decrypts \( \mathbf{c}_t' \) to recover \( \mathbf{M} \cdot \mathbf{v} \). Based on how we construct the ranking service’s matrix \( \mathbf{M} \) and the client’s vector \( \mathbf{v} \), the product \( \mathbf{M} \cdot \mathbf{v} \) 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 \( \mathbf{q} \), under a secret key known only to the client. Provided that the underlying encryption scheme is semantically secure [51], 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 (\( C \) 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 \( \text{Enc}(\mathbf{q}) \), 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 [71, 117], 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 $M$ by the client’s encrypted query vector $\text{Enc}(\mathbf{q})$. For a corpus with hundreds of millions of documents, the matrix $M$ can be 50 GiB or more in size. Sharding $M$ 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 $M = (M_1 \parallel \cdots \parallel M_W)$, and we store matrix $M_i$ on server $i$.

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

Our implementation ships each ciphertext chunk $\mathbf{ct}_i$ 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 [29, 70] 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 [59], 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 §§ 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 [59], at the cost of increasing the per-query communication by roughly 4x.

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 [59], which is in turn based on Regen’s lattice-based encryption scheme [115]. This encryption scheme allows the server to preprocess a matrix $M$ ahead of time. Thereafter, given a ciphertext $\text{Enc}(\vec{q})$ encrypting a vector $\vec{q}$ (i.e., a query vector), the server can compute the matrix-vector product $M \cdot \text{Enc}(\vec{q}) = \text{Enc}(M \cdot \vec{q})$ almost as quickly as computing the same product on plaintext values. The server can compute many subsequent matrix-vector products (with the same $M$), 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 efficiency of the SimplePIR-style encryption scheme in Tiptoe, we sketch the concrete costs. Throughout this section, we take the embedding dimension $d = 1$ to simplify the expressions. (To obtain the costs for embedding dimension $d > 1$, replace all instances of $N$ in this section with $dN$.) If the matrix $M$ is a $\sqrt{N}$-by-$\sqrt{N}$ matrix, the query vector $\vec{q}$ consists of $\sqrt{N}$ 16-bit integers. Furthermore, if 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 $M$, the cost of computing $M \cdot \text{Enc}(\vec{q})$ is small: only $2N$ 64-bit operations. For comparison, computing $M \cdot \vec{q}$ on plaintext (unencrypted) values requires $2N$ 16-bit operations.

**Large communication.** After computing on the ciphertexts, they become large: the ciphertext encrypting the product $\text{Enc}(M \cdot \vec{q})$ is a factor of $\left(\frac{\lambda}{128}\right)$ larger than the plaintext vector $M \cdot \vec{q}$. 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 $M$.

- **Homomorphic matrix-vector product.** The server computes $M \cdot \text{Enc}(\vec{q})$ with $2N$ 64-bit additions and multiplications.

- **Ciphertext size before homomorphic operation.** The ciphertext $\text{Enc}(\vec{q})$ consists of $\sqrt{N}$ 64-bit integers.

- **Ciphertext size after homomorphic operation.** The resulting ciphertext $\text{Enc}(M \cdot \vec{q})$ 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 $4.4 \approx 4096$ 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 [46].

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

Our new technique, inspired by theoretical work on fully homomorphic encryption [20, 47, 78, 81], is to “outsource” the work of decrypting the large ciphertext $C$ to the server:

1. The client encrypts its secret key $s$ using a second linearly homomorphic encryption scheme $\text{Enc}_2$, which allows encrypting vectors of 64-bit integers. The client sends $\text{Enc}_2(s)$ to the server.

2. The server, holding ciphertext $C$, computes the matrix-vector product $C \cdot \text{Enc}_2(s) = \text{Enc}_2(C \cdot s) = \text{Enc}_2(y)$ under encryption. That is, the server “decrypts” $C$ under encryption.

3. The server returns $\text{Enc}_2(y)$ to the client, who decrypts it.

The crucial observation here is that the encryption scheme $\text{Enc}_2$ can be slow as long as it has compact ciphertexts after homomorphic evaluation. More specifically, the computation $C \cdot \text{Enc}_2(s)$ involves a matrix $C$ of size $\lambda \sqrt{N}$—much smaller than the original matrix $M$, which has size $N$. Thus, the homomorphic operations using $\text{Enc}_2$ will not be a computational bottleneck, even if they are slow.

We instantiate $\text{Enc}_2$ with an encryption scheme based on the ring learning-with-errors assumption [77]. Along the way, we introduce new techniques to shrink the communication and storage in SimplePIR (Appendix A). We detail all cryptographic parameters used 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 $\text{Enc}_2(s)$ to the server and downloads the product $C \cdot \text{Enc}_2(s)$. 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
C is also fixed—it just depends on the document corpus. (This portion of the matrix C corresponds to the hint in the SimplePIR encryption scheme [59].) Therefore, the ranking service can compute and return most of the bits of the product \( C \cdot \text{Enc}_2(s) \) 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 \( s \) 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 [59] and on Microsoft SEAL [124]. 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 [60, 61]. 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 [110], 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 [87], 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 [65, 66] 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 [26, 64]. Following prior work [26], 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× 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× larger corpus and uses a 2× 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 r5.8xlarge 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.
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.

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 [111,112]. The Common Crawl 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 [122]. 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 [95]. 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 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) [69], 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 [83]; for BM25, we report the Anserini BM25 baseline document ranking with the default parameters \((k_1 = 0.9, b = 0.4)\) [138]; for tf-idf, we use the Gensim library for stemming and building the tf-idf matrix [116]; 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 \(\sqrt{100} \times \) (Per §4.2, Tiptoe sets the cluster size proportionally to the square-root of the corpus size.) On average, the top search result appears 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
Alternatively, the client could directly download a search index. (We estimate the size of the ColBERT and BM25 indices by 

\[ \text{index size} = N \times \log(\text{dictionary size}) \]

\(N\) is the number of documents, \(\log(\text{dictionary size})\) is the logarithm of 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 [59] 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):

<table>
<thead>
<tr>
<th>Cost metrics:</th>
<th>Client storage (GiB)</th>
<th>Compute cost (MiB/query)</th>
<th>Server comp. cost (cores/query)</th>
<th>Index compression (MiB/query)</th>
<th>AWS cost ($/query)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wikipedia search over 5 million docs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coeus query-scoring [5]</td>
<td>0</td>
<td>50</td>
<td>12,900</td>
<td>2.8</td>
<td>0.059</td>
</tr>
<tr>
<td><strong>Text search over 360 million docs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client-side Tiptoe index</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tiptoe</td>
<td>0.3</td>
<td>42 + 15</td>
<td>145</td>
<td>2.7</td>
<td>0.003</td>
</tr>
<tr>
<td><strong>Image search over 400 million docs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client-side Tiptoe index</td>
<td>98</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tiptoe</td>
<td>0.7</td>
<td>50 + 21</td>
<td>339</td>
<td>3.5</td>
<td>0.008</td>
</tr>
</tbody>
</table>

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**. This traffic occurs before the client enters its search query (§6.3).

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 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 [59] used by Tiptoe, which are roughly an order of magnitude faster than prior single-server private-information-retrieval schemes.
Table 7: Breakdown of Tiptoe’s 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. Estimates are per image embedding. Client downloads are in terms of the number of documents downloaded. We estimated that the LAION-400M data set contains roughly 400 million documents. $5$ Estimated GPU-hours on a V100 GPU. $6$ This step occurs before the client reads the search query ($§6.3$).

### 8.8 Scaling to tens of billions of documents

Popular search engines index tens of billions of documents or more [53]. 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 1900 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 pre-processing).

### 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 $\circ$, the client must retrieve an inner-product score for every document, resulting in communication similar to that of Coeus’s query scoring. With clustering $\circ$, computation is constant, but communication shrinks by $20\times$ 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 $\circ$ reduces communication and computation by $4\times$ at the cost of a drop of 0.04 in MRR@100. Clustering URLs into semantically similar batches $\circ$ does not affect communication and computation, but improves MRR@100 by 0.04.

Assigning documents at cluster boundaries to two clusters $\odot$ increases index size by $1.2\times$ but improves MRR@100 by 0.015. Finally, $\odot$ reducing the embedding dimension by $3\times$ with PCA improves the bandwidth and the computation by roughly $2\times$, 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.

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 [54]. 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 [55, 130].

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. Within a cluster, the client would share an encoding of 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 [55, 130].

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 [18].

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 \( \mathbf{q} \) in Figure 10) using a distributed point function [19]. 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, 28] 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 strengthen privacy for web search. The first is to send search queries to a conventional search engine via an anonymizing proxy, such as Tor [37] or a mix-net [23]. The second approach uses dummy queries or obfuscates queries [11, 15, 38, 92, 96, 104, 114, 139]. Both techniques still reveal some version of the client’s query to the search engine, allowing it to link queries and deanonymize users [48, 102, 103]. The third approach is to use trusted hardware [73, 86, 105], which provides security guarantees that are only as strong as the underlying hardware [24, 27, 56, 63, 91, 101, 113, 129, 132, 133, 134, 135]. Because trusted execution environments leak memory-access patterns, a trusted-hardware-based approach would need to use oblivious RAM [50] to hide memory-access patterns, incurring additional overheads [33, 85, 121].

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 [29, 70] 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, 59, 80, 82, 89], taking advantage of fast lattice-based encryption techniques [115]. 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 [30, 76] or require streaming the entire database to the client [90, 140], 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).

Splitter [136] 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. Splitter 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 [21, 22, 32, 36, 45, 49, 67, 68, 93, 119, 127, 128], their realization in encrypted databases [44, 99, 100, 106, 107, 131], and related systems [34, 35]. 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 [25, 62, 108, 125], rely on computationally heavy cryptographic tools, which practically limits the corpus to a few thousand entries [126, 142], or leak some information about the client’s query to the server [12, 14, 118]. In contrast, Tiptoe requires only a single infrastructure provider, searches over hundreds of millions of documents, and leaks no information about the client’s query.

Some prior work also uses embeddings with homomorphic encryption for text search [12] or image matching [40], 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 [40, 108, 125, 126] 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 disk accesses: both cases aim to minimize the number of reads to the index [26, 64].

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 [61, 74, 109, 117]. An alternative approach uses a sparse document representation, which can capture informa-
tion at the token level, such as the classic BM25 algorithm or the newer SPLADE model [43, 72].

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 Mazieres, 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. 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


Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common


A Details on linearly homomorphic encryption with preprocessing

As described in §6, Tiptoe implements both the ranking service (§4) and the URL service (§5) using a linearly homomorphic encryption scheme that simultaneously has high throughput and compact ciphertexts. At a high level, this scheme consists of two “layers” of encryption:

1. First, the client encrypts its message using a high-throughput linearly homomorphic encryption scheme that supports preprocessing, under a fresh secret key $s$.

   We use the encryption scheme from SimplePIR [59], which is based on the learning-with-errors assumption [115]. This encryption scheme achieves high throughput by letting the server preprocess the linear function that it applies to the ciphertext; however, it has relatively large ciphertexts after homomorphic evaluation.

2. Then, the client encrypts the secret key $s$ with a second linearly homomorphic encryption scheme that has compact ciphertexts, again under a fresh secret key.

   We use standard linearly homomorphic encryption from ring learning-with-errors [77].

A.1 Syntax

Formally, a linearly homomorphic encryption scheme with preprocessing is parameterized by dimensions $\ell, m \in \mathbb{N}$, plaintext modulus $p \in \mathbb{N}$, and key space $K$. The message space is $\mathbb{Z}_p^m$. The scheme then consists of the following algorithms, which all take some public parameters as an implicit input:

- $\text{Enc}(sk, v) \rightarrow c_v$. Given a secret key $k \in K$ and a message vector $v \in \mathbb{Z}_p^m$, output a ciphertext $c_v$ that encrypts the message.
- $\text{Preproc}(M) \rightarrow \text{hint}_M$. Preprocess a linear function, represented as a matrix $M \in \mathbb{Z}_p^{\ell \times m}$, and output a data structure used for applying $M$ to encrypted vectors.
- $\text{Apply}(M, \text{hint}_M, c_v) \rightarrow c_{Mv}$. Given a linear function represented as a matrix $M \in \mathbb{Z}_p^{\ell \times m}$, a preprocessed hint $\text{hint}_M$ for $M$, and a ciphertext $c_v$ encrypting a message vector $v \in \mathbb{Z}_p^m$, output a ciphertext $c_{Mv}$ encrypting the result of the matrix-vector product $M \cdot v \in \mathbb{Z}_p^\ell$.
- $\text{Dec}(sk, c_{Mv}) \rightarrow v$. Given a secret key $sk \in K$ and a ciphertext $c_{Mv}$ encrypting the matrix-vector product $M \cdot v \in \mathbb{Z}_p^\ell$, output the decrypted product $M \cdot v$.

The correctness requirement is standard: encrypting a message vector, applying a matrix to it under encryption, and decrypting the resulting ciphertext yields the original message with the same sequence of operations applied, except with some small failure probability. Formally, we say that the scheme has correctness error $\delta$ if, for all message vectors $v \in \mathbb{Z}_p^m$ and all linear functions $M \in \mathbb{Z}_p^{\ell \times m}$, the following probability is at least $1 - \delta$:

\[
\Pr \left[ (M \cdot v) = \text{Dec}(sk, c_{Mv}) : c_v \leftarrow \text{Enc}(sk, v), \text{hint}_M \leftarrow \text{Preproc}(M), c_{Mv} \leftarrow \text{Apply}(M, \text{hint}_M, v) \right] \approx 0.5\cdot\delta.
\]

We also demand that this encryption scheme satisfy the standard notion of semantic security [51]. That is, for all $v_0, v_1 \in \mathbb{Z}_p^m$, and for $sk \in K$, the following probability distributions are computationally indistinguishable: 

\[
\{\text{Enc}(sk, v_0)\} \approx_c \{\text{Enc}(sk, v_1)\}.
\]

A.2 Construction

Our construction is a modification of the SimplePIR linearly homomorphic encryption scheme with preprocessing [59], which is based on Regev encryption [115].

In that scheme, if $M \in \mathbb{Z}_q^{\ell \times m}$ is the linear function that we wish to apply to encrypted vectors:

- The preprocessing algorithm outputs a “hint” matrix $H \in \mathbb{Z}_q^{\ell \times n}$, on security parameter $n \approx 2048$ and ciphertext modulus $q$. The hint matrix depends on the linear function $M$.
- The “apply” algorithm outputs a vector $c \in \mathbb{Z}_q^n$.
- The decryption algorithm, on hint matrix $H \in \mathbb{Z}_q^{\ell \times n}$, secret key $s \in \mathbb{Z}_q^n$, and ciphertext $c \in \mathbb{Z}_q^n$, computes $f(H \cdot s - c)$, where $f$ is a non-linear function.

A limitation of the SimplePIR encryption scheme is that the decryptor must somehow obtain the large hint matrix $H$. In Tiptoe, the hint matrix $H$ depends on the search engine index which means that: (1) $H$ changes every time the search index changes and (2) $H$ is very large—a gigabyte or more.

To avoid the decryptor having to obtain the hint matrix $H$, we tweak the SimplePIR encryption scheme to allow the decryptor to “outsource” much of the work of decryption to the evaluator. To do so, we make use of a second linearly homomorphic encryption scheme ($\text{Enc}_2, \text{Apply}_2, \text{Dec}_2$) whose plaintext space is $\mathbb{Z}_q^n$. We call this second scheme the “outer” encryption scheme. (Here $n$ is the SimplePIR security parameter, i.e., the dimension of the SimplePIR decryption key.) That is, the encryption scheme $\text{Enc}_2$ encrypts vectors in $\mathbb{Z}_q^n$ and allows adding vectors under encryption, and also scaling an encrypted vector component-wise by a vector of constants.

We augment the SimplePIR encryption-scheme routines ($\text{Enc}, \text{Preproc}, \text{Apply}, \text{Dec}$) as follows:

- **Key generation.** The augmented encryption scheme chooses a fresh random secret key $sk_2$ for the outer encryption scheme $\text{Enc}_2$, in addition to the SimplePIR secret key $sk$.
- **$\text{Enc}(sk, m)$:** The augmented encryption scheme takes as input a secret key $sk = (sk, sk_2)$, consisting of the SimplePIR secret key $sk$ and a key $sk_2$ for the outer encryption scheme $\text{Enc}_2$. 


Let \((s_1, \ldots, s_n) = s \in \mathbb{Z}_q^n\) be the components of the SimplePIR secret key. In addition to outputting the SimplePIR ciphertext \(c\), the augmented encryption routine outputs \(n\) ciphertexts \((z_1, \ldots, z_n)\), where for all \(i \in [n]\), \(z_i\) is an encryption of \(n\) copies of the \(i\)th component of the SimplePIR secret key \(s\):

\[
z_i \leftarrow \text{Enc}_2(sk, [s_i, s_i, \ldots, s_i, s_i]).
\]

The \(z_i\) values become part of the augmented ciphertext.

- **Apply**\((M, \text{hint}_M, c_v)\): The Apply algorithm runs the SimplePIR “apply” algorithm on the matrix \(M\) and the SimplePIR component of the ciphertext \(c_v\). The result is a vector \(c \in \mathbb{Z}_q^\ell\).

The augmented Apply algorithm then runs the linear part of the SimplePIR decryption operation under encryption. In particular, the hint matrix \(\text{hint}_M\) has dimension \(\ell \times n\). The apply algorithm considers each chunk of \(n\) rows \(H \in \mathbb{Z}_q^{\ell \times n}\) at a time.

For each chunk, using the vectors \((z_1, \ldots, z_n)\)—which encrypt the SimplePIR secret key \(s\)—the augmented “apply” algorithm computes \(H \cdot s - c \in \mathbb{Z}_q^{\ell}\) under the outer encryption scheme \(\text{Enc}_2\). This step uses the linear homomorphism of the underlying encryption scheme \(\text{Enc}_2\).

The augmented apply algorithm then returns the resulting \(\text{Enc}_2\) ciphertexts as \(c_{M,s}\). There are \(\lceil \ell/n \rceil\) such ciphertexts.

- **Dec**\((sk, c_{M,s})\): The augmented decryption scheme uses the secret key \(sk\) for the outer encryption scheme to decrypt each of the \(\lceil \ell/n \rceil\) outer ciphertexts. The decryption routine then applies the non-linear function \(f\) (from SimplePIR decryption) to each component of the resulting plaintexts.

We now sketch the properties that this augmented encryption scheme provides:

**Correctness.** Follows by construction. The output of the decryption routine is exactly the output of the SimplePIR decryption routine.

**Security.** The augmented ciphertexts consist of two sets of ciphertexts, encrypted under independent keys. Semantic security thus follows directly from the security of the underlying encryption schemes.

**Ciphertext size.** We instantiate the inner encryption scheme with SimplePIR, which encrypts a dimension-\(m\) plaintext vector to a dimension-\(m\) ciphertext vector, using a secret key of dimension \(n\). We instantiate the outer encryption scheme with ring-LWE-based encryption, which encrypts a dimension-\(n\) plaintext vector to a dimension-\(O(n)\) ciphertext vector. Since the augmented ciphertext consists of \(n\) ring-LWE ciphertexts (encrypting each entry of the SimplePIR secret key), plus one SimplePIR ciphertext, the total ciphertext length is roughly \(m + n^2\). In our setting, \(n \ll m\).

After homomorphic evaluation, the ciphertext consists of \(\lceil \ell/n \rceil\) ring-LWE ciphertexts, each of dimension \(n\). So the total evaluated ciphertext size is roughly \(\ell\).

**Computational cost.** The SimplePIR Apply operation requires roughly \(\ell \cdot m\) ring operations, where the matrix \(M\) applied to encrypted vectors has dimension \(\ell \times m\). The augmented scheme requires an additional \(\ell \cdot n\) operations to perform the decryption under encryption. The total computational cost of Apply is then roughly \(\ell \cdot (m + n)\). Since \(m \gg n\) in our application, the computational cost roughly matches that of SimplePIR.

### A.3 Optimizations

Finally, we apply the following optimizations to reduce the computation and communication costs of our encryption scheme:

**Reducing latency with tokens.** We have the client build and send its “outer” ciphertexts \((z_1, \ldots, z_n)\) to the server ahead of time, since they do not depend on the message being encrypted. Then, the server computes \(H \cdot s\) under the outer encryption scheme \(\text{Enc}_2\) ahead of time, and returns it to the user. With this optimization, the client only has to send its original SimplePIR ciphertext, \(c_v\), to the server, and the server only has to compute and respond with the output of the SimplePIR Apply algorithm, \(c\), on the latency-critical path. We call the server’s offline response, which encrypts \(H \cdot s\), a query “token”.

**Using the same secret key for both services.** We additionally observe that we can use the same encryption \((z_1, \ldots, z_n)\) (which is the encryption of a Regev secret vector \(s\) under a second encryption scheme \(\text{Enc}_2\)) for both the ranking service and the URL service, since each service uses independent public parameters and this upload does not depend on the message being encrypted. This optimization saves a factor of \(2\times\) in the per-query upload that occurs ahead of time; concretely, this saves roughly 30 MiB.

**Dropping the lowest-order bits of the hint matrix.** Inspired by prior work on learning-with-errors-based cryptosystems [16, 39], we observe that the non-linear part of the SimplePIR decryption routine, \(f\), performs a rounding step. Because of this rounding, the lowest-order bits of each entry of the hint matrix \(H\) do not actually affect the client’s output—they are always rounded away. So, we have the server drop the lowest-order bits of each entry of \(H\), since the server does not need to compute over them. This optimization saves a factor of roughly \(2\times\) in the per-query, token generation work (on the server) and a factor of \(2\times\) in the server-to-client token download. (This trick effectively shrinks the size of the hint in SimplePIR; it could additionally be used to compress SimplePIR’s online download.)
Figure 10: Tiptoe’s private nearest-neighbor protocol. The protocol is parameterized by: a vector dimension $d \in \mathbb{N}$, a number of vectors $N \in \mathbb{N}$, a number of clusters $C \in \mathbb{N}$, and a linearly-homomorphic encryption scheme $\text{Enc}$ with plaintext modulus $p$ (defined in Appendix A).

Server’s input: A matrix $M$ of $N$ row vectors in $\mathbb{Z}_p^d$, arranged into $C$ columns, one per cluster, and $N/C$ rows. (Assume that $N/C$ is an integer.)

Client’s input: A vector $q \in \mathbb{Z}_p^d$ and an index $i^* \in \{1, \ldots, C\}$. Client’s output: The inner product of its vector $q$ with each of the $N/C$ vectors in column $i^*$ of the server’s matrix $M$.

For simplicity, throughout the rest of our description of the protocol, we say that the matrix $M$ has dimension $N/C$ by $dC$.

Per-query protocol. Run when the client issues a new query.

   - The client builds a vector $\tilde{q} \in \mathbb{Z}_p^{dC}$, that is zero everywhere, except that it contains the client’s query vector $q \in \mathbb{Z}_p^d$ in place of the $i^*$ th size-$d$ block of zeros.
   - The client encrypts $\tilde{q}$ using the linearly homomorphic encryption scheme with a fresh secret key. The client sends the resulting ciphertext $\text{Enc}(\tilde{q})$ to the server.

2. Server answer. The server receives the ciphertext $\text{Enc}(\tilde{q})$ and applies the matrix $M$ to it to obtain $\text{Enc}(M \cdot \tilde{q})$, which the server returns to the client.

3. Client reconstruction. The client receives $\text{Enc}(M \cdot \tilde{q})$ from the server and decrypts it. Finally, the client outputs $M \cdot \tilde{q} \in \mathbb{Z}^{N/C}$.

B Details on private nearest-neighbor protocol

B.1 Fixed-precision embedding representation

To use the above linearly homomorphic encryption scheme for embedding-based search, Tiptoe maps the floating-point embedding values into the message space $\mathbb{Z}_p$.

To do so, Tiptoe first represents each embedding value using $b$ bits of precision with a sign. That is, Tiptoe represents each real number $x \in [-1, 1] \subseteq \mathbb{R}$ as the $\mathbb{Z}_p$ element $[x \cdot 2^b]$. (Our text search embedding function occasionally generates values outside of $[-1, 1]$, and we simply clip these to fall within $[-1, 1]$ with no significant impact to search quality.) Second, we associate the elements of $\mathbb{Z}_p$ with the set $\{-p, \ldots, -1, 0, 1, \ldots, p\}$. We can then add and multiply numbers in their $\mathbb{Z}_p$ representation as long as the maximum possible value never grows larger than $p/2$ or smaller than $-p/2$. To compute inner products of $d$-dimensional vectors under encryption, without “wrapping around” the modulus, we need $p/2 > d \cdot (2^b)^2$.

B.2 Private nearest-neighbor search protocol

We give a formal description of Tiptoe’s private nearest-neighbor search protocol in Figure 10.

C Parameters for cryptographic protocols

We choose parameters to achieve 128-bit security and correctness error roughly $2^{-40}$.

Learning-with-errors parameters. For the portion of our encryption scheme based on LWE, we use the following parameters for Regev-style secret-key encryption [115]:

- We take the ciphertext modulus to be $q = 2^{64}$, so that we can represent ciphertexts as unsigned 64-bit integers.
- We set the secret dimension, $n$, to be 2048. We sample errors from the discrete Gaussian distribution with standard deviation $\sigma = 81,920$, and sample the secret key from the ternary distribution modulo $q$. This parameter choice guarantees 128-bit security for encrypted vectors of dimension $\leq 2^{37}$ [6].
- Finally, we select a plaintext modulus $p$ such that (a) our batch inner-product computation does not “wrap-around” modulo $p$, and (b) our encryption scheme supports sufficiently many homomorphic operations for the given data set size. Concretely:
  - For text search, we use plaintext modulus $p = 2^{17}$, as this avoids overflow in the inner-product computation with embeddings of dimension $d = 192$ consisting of 4-bit signed integers. This parameter setting in turn supports up to $2^{21}$ homomorphic additions with roughly $2^{-40}$ correctness-failure probability. With $d = 192$, this allows Tiptoe to scale to up to $C \approx 10K$ clusters.
  - For image search, we use plaintext modulus $p = 2^{15}$, as this avoids overflow in the inner-product computation with normalized embeddings of dimension $d = 384$ consisting of 4-bit signed integers. To achieve roughly $2^{-40}$ correctness-failure probability, the scheme can now support up to $2^{37}$ homomorphic additions. With $d = 384$, this allows Tiptoe to scale to up to $C \approx 350K$ clusters.

For our private-information-retrieval scheme (§5), we take the ciphertext modulus to be $q = 2^{32}$ (as in the original SimplePIR). We set the secret dimension to be $n = 1,408$. We use discrete Gaussian errors with standard deviation $\sigma = 6.4$ and ternary secrets to achieve 128-bit security for encrypted vectors up to dimension $2^{20}$ [6].

Ring learning-with-errors parameters. For the portion of our encryption scheme based on RLWE, we use BFV encryption [42] and we take:

- the security parameter, $n$, to be 2048.
- the plaintext modulus, $p$, to be 65,537.
- the ciphertext modulus, $q$, to be a 38-bit prime.

We map the inputs (LWE ciphertexts that are 64-bit or 32-bit values) into the plaintext space using standard base-16 decomposition.

Database dimensions. We set the dimensions of the matrix
M in Figures 3 and 10 to ensure that the cost of the first “layer” of homomorphic encryption (which we implement with SimplePIR) dominates the homomorphic evaluation cost. Similarly, in the URL retrieval step, we “unbalance” the dimensions of the database over which we run SimplePIR. This parameter choice effectively sets the height of the matrix M (or the SimplePIR database), which must be roughly \(10\times\) as wide as it is tall. For text search, this implies that Tiptoe has clusters of at most 50K documents and that batches of compressed URLs do not exceed 40 KiB.

**Scaling to larger database sizes.** In §8.5, we estimate Tiptoe’s performance scaling to larger database sizes. To do so, we use the LWE parameters given in Tables 11 and 12. We use the same RLWE parameters as previously, since the LWE lattice dimension always remains at most 2048.

### D Security analysis

**Syntax.** A private-search scheme with query space \(Q \subseteq \{0, 1\}^*\) is a two-party interactive protocol that takes place between a client \(C\) and a server \(S\), where:
- the client \(C\) takes as input a query string \(q \in Q\), and
- the server \(S\) takes as input a dataset \(D \in \{0, 1\}^*\).

For an interactive algorithm \(S^*\) and a query string \(q \in Q\), let \(\text{View}_{S^*}[C(q)]\) denote \(S^*\)’s view of its interaction with \(C\) on query input \(q\).

**Definition D.1** (Query privacy, restated). We say that a private-search scheme \((C, S)\) achieves query privacy if, for all efficient adversaries \(S^*\), and all query strings \(q_0, q_1 \in Q\), the following probability distributions are computationally indistinguishable in the implicit security parameter:

\[
\{\text{View}_{S^*}[C(q_0)]\} \approx \{\text{View}_{S^*}[C(q_1)]\}
\]

**Remark D.2** (Correctness). For typical cryptographic primitives, we define both correctness and security. Since it is not clear how to formally define correctness of an embedding-based search, we rely on our empirical evaluation to show correctness. The security properties of our scheme hold in the precise formal sense of Definition D.1.

We now prove that Tiptoe satisfies query privacy, in the sense of Definition D.1. The Tiptoe client sends exactly two messages during an execution of the search protocol:

1. During its interaction with the ranking service, the client sends one ciphertext, encrypted using the linearly homomorphic encryption scheme of §6.
2. During its interaction with the URL service, the client sends one encrypted query using the SimplePIR private-information-retrieval scheme [59], optimized as in §6.

Both messages have fixed length and the client sends both of these messages irrespective of the output of the server.

To complete the proof, we consider the following hybrid distributions:

- \(H_0\): The server \(S^*\)’s view in the real interaction with \(C(q_0)\). This is \(\text{View}_{S^*}[C(q_0)]\), by definition.
- \(H_1\): The same as \(H_0\) with the client’s first message replaced with the first message that client \(C(q_1)\) would send.
- \(H_2\): The same as \(H_1\) with the client’s second message replaced with the second message that client \(C(q_1)\) would send. This is exactly \(\text{View}_{S^*}[C(q_1)]\), by definition.

Let \(\mathcal{A}\) be any efficient algorithm and let \(q_0, q_1 \in Q\) be any pair of queries. For \(i \in \{0, 1, 2\}\), let \(W_i\) be the event that \(\mathcal{A}\) outputs 1 when fed a sample from distribution \(H_i\), defined with respect to queries \(q_0\) and \(q_1\). Then, the advantage of \(\mathcal{A}\) at distinguishing \(\text{View}_{S^*}[C(q_0)]\) from \(\text{View}_{S^*}[C(q_1)]\) is \(|\Pr[W_0] - \Pr[W_2]|\). We now know that:

- \(|\Pr[W_1] - \Pr[W_2]|\) is negligible in the (implicit) security parameter, by security of the linearly homomorphic encryption scheme under the ring learning-with-errors assumption.

Therefore \(|\Pr[W_0] - \Pr[W_2]|\) is negligible, and we are done.
Table 11: Parameters for ciphertext modulus $q = 2^{32}$ (used for URL retrieval step)

<table>
<thead>
<tr>
<th>Upload dim. $m$:</th>
<th>$2^{13}$</th>
<th>$2^{14}$</th>
<th>$2^{15}$</th>
<th>$2^{16}$</th>
<th>$2^{17}$</th>
<th>$2^{18}$</th>
<th>$2^{19}$</th>
<th>$2^{20}$</th>
<th>$2^{21}$</th>
<th>$2^{22}$</th>
<th>$2^{23}$</th>
<th>$2^{24}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext mod. $p$:</td>
<td>991</td>
<td>833</td>
<td>701</td>
<td>589</td>
<td>495</td>
<td>416</td>
<td>350</td>
<td>294</td>
<td>887</td>
<td>745</td>
<td>627</td>
<td>527</td>
</tr>
<tr>
<td>Lattice dim. $n$:</td>
<td>1408</td>
<td>1408</td>
<td>1408</td>
<td>1408</td>
<td>1408</td>
<td>1408</td>
<td>1408</td>
<td>1608</td>
<td>1608</td>
<td>1608</td>
<td>1608</td>
<td>1608</td>
</tr>
<tr>
<td>Error dev. $\sigma$:</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 12: Parameters for ciphertext modulus $q = 2^{64}$ (used for ranking step)

<table>
<thead>
<tr>
<th>Upload dim. $m$:</th>
<th>$2^{13}$</th>
<th>$2^{14}$</th>
<th>$2^{15}$</th>
<th>$2^{16}$</th>
<th>$2^{17}$</th>
<th>$2^{18}$</th>
<th>$2^{19}$</th>
<th>$2^{20}$</th>
<th>$2^{21}$</th>
<th>$2^{22}$</th>
<th>$2^{23}$</th>
<th>$2^{24}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext mod. $p$:</td>
<td>$2^{19}$</td>
<td>$2^{18}$</td>
<td>$2^{18}$</td>
<td>$2^{18}$</td>
<td>$2^{18}$</td>
<td>$2^{17}$</td>
<td>$2^{17}$</td>
<td>$2^{17}$</td>
<td>$2^{17}$</td>
<td>$2^{18}$</td>
<td>$2^{18}$</td>
<td>$2^{18}$</td>
</tr>
<tr>
<td>Lattice dim. $n$:</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Error dev. $\sigma$:</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>81920</td>
<td>4096</td>
<td>4096</td>
<td>4096</td>
</tr>
</tbody>
</table>

**E Sample queries**

**E.1 Text search**

We run queries from the MS MARCO document ranking dev set over the documents in the text Common Crawl C4 data set from 2019. We include the output of all queries here: https://anonymfile.com/XPyAA/tiptoe-msmarcoqueries.txt. We conservatively redact URLs that might contain offensive language. Below we show the output of 20 randomly sampled queries.

Q: **What test are relevant for heart screenings**
2. [http://cardiocppa.com/faq/?s=](http://cardiocppa.com/faq/?s=)
3. [https://bookinghawk.com/events/heartcare-clinic/](https://bookinghawk.com/events/heartcare-clinic/)
10. [https://www.faythclinic.com/24hours-holter-study/](https://www.faythclinic.com/24hours-holter-study/)

Q: **What is the IGE antibody**
6. [https://spectrumhealth.testcatalog.org/show/LAB1230099-1](https://spectrumhealth.testcatalog.org/show/LAB1230099-1)
8. [https://www.hyqvia.com/about/how-hyqvia-works](https://www.hyqvia.com/about/how-hyqvia-works)

Q: **Foodborne trematodiases symptoms**
1. [https://bowenmedicallibrary.wordpress.com/2017/04/04/foodborne-trematodiases/](https://bowenmedicallibrary.wordpress.com/2017/04/04/foodborne-trematodiases/)
4. [https://westsidedog nanny.com/category/illness/](https://westsidedog nanny.com/category/illness/)
5. [https://m.petmd.com/reptile/conditions/skin/c rp_skin_shell_infections](https://m.petmd.com/reptile/conditions/skin/c rp_skin_shell_infections)
7. [https://totalrisksa.co.za/a-list-of-diseases-you-should-be-aware-of-excursion](https://totalrisksa.co.za/a-list-of-diseases-you-should-be-aware-of-excursion)
8. [https://activities.timeout.com/en-us/tel-
\protect\penalty0z@aviv-l487/jerusalem-business-class-tour-full-day-from-tel-aviv-t20352/?referrer_view_id=49ed3cf5b860e66f003a4cb4b9590004&referrer_view_position=10](https://activities.timeout.com/en-us/tel-
\protect\penalty0z@aviv-l487/jerusalem-business-class-tour-full-day-from-tel-aviv-t20352/?referrer_view_id=49ed3cf5b860e66f003a4cb4b9590004&referrer_view_position=10)
9. [https://www.petra-trip.com/product/jewish-jerusalem-3-days](https://www.petra-trip.com/product/jewish-jerusalem-3-days)
Q: How long has Earth been habitable

6. https://sciencesprings.wordpress.com/tag/are-we-alone/

Q: Is the Judicial branch of the UN.

8. http://eprints.nottingham.ac.uk/13282/

Q: What is the most important solid dissolved in ocean water?

4. https://allinonehighschool.com/ocean-water/
7. https://www.cencoos.org/data/parameters/oxygen
8. https://ultimate-animals.com/what-do-we-really-know-about-the-ocean/
10. http://marinebio.net/marinescience/02ocean/swcomposition.htm

Q: What is the LCD math


Q: What's the Establishment clause

1. https://scholarship.law.missouri.edu/facpubs/162/
Q: what is coffee flour
10. https://www.kitanda.co.com/coffee

Q: largest snail in the world
7. https://www.lighthhearth.com/blog/gastropods

Q: disease caused by blood worms

Q: the meaning of stinchcombe
1. https://www.webster-dictionary.org/definition/scab
4. https://www.wordunscrambler.net/word-meaning/worm
5. https://1828.mshaffer.com/d/word/neck
6. https://webster-dictionary.org/definition/neck
Q: is home improvement interest deductible
7. https://umasoudana.com/2019/04/03/

Q: is burning sage evil
1. http://familiaterrority.us/2019/03/28/protect-your-home-or-workspace/

Q: what does otg in a usb stands for

Q: the meaning of haploid cell
3. https://guillaumeantzenberger.com/chapter-6-terms
4. https://www.palaeontologyonline.com/glossary/m/meiosis/
5. http://webster-dictionary.net/definition/chromatin
7. http://academic.pgcc.edu/~kroberts/Lecture/Chapter%207/euchromosome.html
10. https://rwguild.com/collections/sculptures-
Q: how long before eagles get feathers

8. https://eaa35.org/?q=content/young-eagles