Coeus: A System for Oblivious Document Ranking and Retrieval

Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta

University of California, Santa Barbara

Abstract

Given a private string q and a remote server that holds a set of public documents $\mathcal{D}$, how can one of the K most relevant documents to q in $\mathcal{D}$ be selected and viewed without anyone (not even the server) learning anything about q or the document? This is the *oblivious document ranking and retrieval problem*. In this paper, we describe Coeus, a system that solves this problem. At a high level, Coeus composes two cryptographic primitives: secure matrix-vector product for scoring document relevance using the widely-used term frequency-inverse document frequency (tf-idf) method, and private information retrieval (PIR) for obliviously retrieving documents. However, Coeus reduces the time to run these protocols, thereby improving the user-perceived latency, which is a key performance metric. Coeus first reduces the PIR overhead by separating out private metadata retrieval from document retrieval, and it then scales secure matrix-vector product to tf-idf matrices with several hundred billion elements through a series of novel cryptographic refinements. For a corpus of English Wikipedia containing 5 million documents, a keyword dictionary with 64K keywords, and on a cluster of 143 machines on AWS, Coeus enables a user to obliviously rank and retrieve a document in 3.9 seconds—a 24× improvement over a baseline system.

1 Introduction

As a motivating example, consider Ziv, who identifies with a non-binary gender, chooses to keep this preference secret from a conservative family, and considers Wikipedia a reliable source of information. Ziv wants to attend a gender-specific event and wishes to read about the event's history before attending it. As usual, Ziv opens Wikipedia, enters a search query (e.g., "History of ____ event in San Francisco"), and selects one of the links to get the desired information. However, this time Ziv feels concerned about privacy due to recent, high-profile data breaches, via insider attacks [39], external hacks [17, 52, 63], mass surveillance by an ISP [14], and even financial pressure [71]. *Can we enable Ziv to search for and retrieve documents from Wikipedia, or more generally any public document repository, privately? Furthermore, can Ziv get peace-of-mind with provable privacy guarantees?*

Ziv's situation is one example of a fundamental problem this paper addresses: *the oblivious document ranking and retrieval problem*. An abstract formulation of the problem is as follows. A user holds a search query q containing multiple keywords, while a server holds a set of public documents $\mathcal{D}$. The user enters q in a web browser (or app), which interacts with the server to enable the user to select and view one of the K documents that have highest relevance to q. The privacy requirement is that nobody besides the user (neither the server nor a network eavesdropper) must learn any information about q or the document viewed by the user.

We emphasize that this problem is quite different from the problem of searching and ranking on encrypted data that has received much attention in the literature (e.g., [8, 20, 24, 29–31, 48, 49, 53, 57, 67, 67, 79, 80, 82, 83, 88, 90, 91, 94, 99]; §7). In searching on encrypted data, the data is private (owned by the user), while in our problem the documents are public and known to the server (for example, the Wikipedia server owns the documents). This difference in setting enables fundamentally different techniques; for example, if the documents are private, then the owner may encrypt and arrange them in a tree data structure, as in oblivious RAMs [42, 81]. Such encryption is not possible if the documents are public.

Instead, oblivious document ranking and retrieval is more closely related to the problem of private information retrieval (PIR) [26, 59], although with significant differences. PIR, in its most basic form, allows a user to privately retrieve a document by specifying an index in a list (e.g., retrieve the 34-th document from the list of 1,000 documents). In contrast, in our problem a user specifies a multi-keyword search query and not an index. Two extensions to PIR, namely PIR-by-keywords [25] and private stream searching [19, 68], allow the user to retrieve documents that match keywords—but without consideration of ranking. As an example, suppose the user's string is "Cristiano Ronaldo". Then, with PIR-by-keywords, the user will get *one* of the many articles that contain the name of the famous soccer player. On the other hand, with private stream searching, the user will get *all* documents that mention Ronaldo, without any ranking or ordering, possibly overwhelming the user.

This paper describes Coeus, the first system for oblivious document ranking and retrieval over public documents under a strong threat model that does not make assumptions about the server. Coeus ranks a document's relevance given a user query using the term frequency-inverse document frequency (tf-idf) statistical method [72, 74] (§3.1), which

This is an unmodified version of the paper originally published in *the 28th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '21)*, October 26–29, 2021, Virtual Event, Germany.
SOSP '21, October 26–29, 2021, Virtual Event, Germany

https://doi.org/10.1145/3477132.3483586

is used commonly in text-based recommender systems in digital libraries. Coeus imposes a latency of a few seconds on a user while providing provable guarantees.

At a high-level, Coeus composes the secure matrix-vector product primitive [41, 46, 47, 58] with PIR. One natural way to do their composition is a two-round protocol, where in the first round the user securely multiplies the query q with the tf-idf matrix to obtain scores for all the documents, and then in the second round retrieves the top-K documents obliviously using PIR. The challenge though is the high server-side overhead, imposed by both secure matrix-vector product and PIR. Fundamentally, if the server must learn no information about the user query or the retrieved document, then it must process its *entire* state comprising the tf-idf matrix and the document library; else, the server will learn information about keywords that are *not* in the query, or the documents that are *not* retrieved by the user (§2.3).

Coeus responds to this challenge in two ways. First, at the protocol-design level, instead of using the natural two-round protocol, Coeus uses a new three-round protocol that separates out metadata retrieval from document retrieval. In the first round, as in the two-round protocol, a user converts the query q into an encrypted vector, sends it to the server, and obtains encrypted relevance scores for the documents by securely multiplying the vector with the tf-idf matrix. Then, in the second round, the user retrieves short descriptions and title (metadata) for top-K scoring documents from a metadata library using multi-retrieval PIR that can concurrently retrieve multiple objects [12, 50]. Finally, in the third round, the user retrieves a single document that the user wants to view in detail using a single-retrieval PIR [7, 12].

Coeus's three round protocol reduces PIR overhead relative to the two-round protocol. Not only does a user retrieve K smaller metadata instead of K documents, but the separation of metadata from document retrieval enables the server to pack variable-sized documents and compress the document library, thereby reducing PIR compute time.

Coeus's second idea further improves the overhead of the first round through a new secure matrix-vector product primitive that fundamentally reduces server-side work (§4.2, §4.3), and distributes this work efficiently across a cluster of machines (§4.4). Coeus starts from the state-of-the-art construction of Halevi and Shoup that works for a square matrix block with few thousand rows and the same number of columns [46, 47] (§3.2). First, Coeus observes and eliminates redundancy in the calls to an underlying homomorphic rotation operation; this optimization reduces overhead by a constant factor of approximately four (§4.2). Second, Coeus amortizes the overhead of homomorphic rotations across multiple blocks of the tf-idf matrix (§4.3). Third, Coeus efficiently distributes the computation for thousands of matrix blocks (the tf-idf matrix is large consisting of several hundred billion elements) onto a cluster of machines arranged in a master-worker-aggregator architecture. During workload distribution, Coeus preserves the benefits of amortization while keeping in check the network transfer overhead, by including an optimizer that finds the optimal shape of the submatrices at the worker nodes (§4.4). Although Coeus's secure matrix-vector product scheme is designed keeping Coeus's scale in mind, it may find uses in other applications especially where matrices are large.

We have implemented (§5) and evaluated (§6) a prototype of Coeus. On an Amazon EC2 cluster (97 machines for document relevance scoring, 7 for metadata retrieval, and 39 for document retrieval), and for a document library consisting of a corpus of English Wikipedia with 5M documents, Coeus's latency is 3.9s for oblivious document ranking and retrieval. In contrast, without Coeus's two techniques, its latency over the same cluster would be 93.9s—thus an improvement of 24×. If Coeus's resource overheads are converted to dollars, then it costs 6.5 cents per request, in contrast to 1.62 dollars for the baseline.

Coeus's absolute overheads are substantial: each request keeps a cluster of machines busy for up to a few seconds. Thus, it may not be used for every request. However, Coeus scales horizontally, as one can replicate its setup, for example, at various CDNs. But more importantly, Coeus shows that Ziv could *choose* to get strong privacy guarantees while retrieving documents from Wikipedia, without waiting for tens of seconds for the webpage to load, and without draining wallet balance (e.g., hundred private requests per month would cost Ziv 6.5 dollars rather than 162 dollars).

2 Architecture and overview

Coeus is designed for private retrieval of public documents. Abstractly, a user holds a multi-keyword query q and a server holds a library of n documents and their metadata (information such as the document title and a short text description). Similar to how search engines work, Coeus takes as input the query q and enables the user to select and view one of the $K \geq 1$ documents that rank highest for q. In the process, an adversary who may compromise the server hosting the library or the network learns no information about q.

2.1 Approach and architecture

An approach to realize the picture described above is to incorporate fully homomorphic encryption (FHE) [38]: the user encrypts q using FHE and sends it to the server, who homomorphically ranks and sends the top-K documents back to the user. On the plus side, the user retrieves the documents in a single round of communication, but on the negative side, the server's computational work is prohibitively high due to the large expense of the homomorphic comparison operation [54, 61].

An alternative to the single-round approach is to split document ranking and retrieval into separate protocol rounds. In the first round, the user retrieves scores for each of the

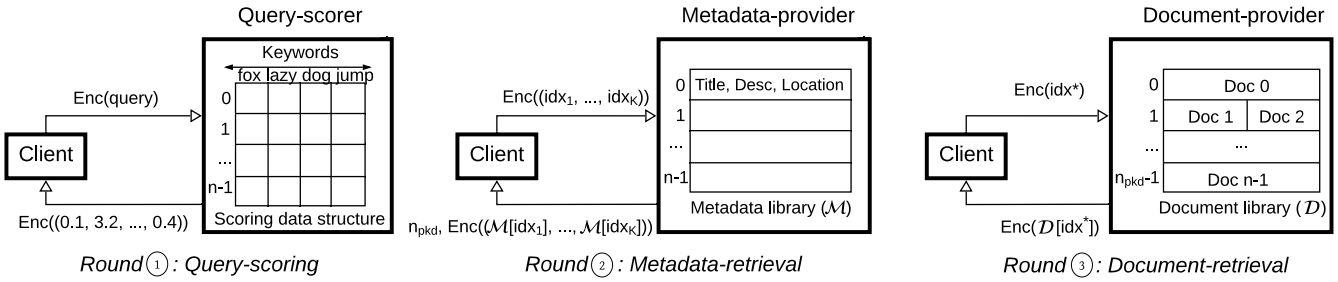

Figure 1. An overview of Coeus's three-round protocol: query-scoring, metadata-retrieval, and document-retrieval.

n documents, and *locally* compares them to learn indices for the top-K documents. Then, in the second round, the user obliviously retrieves the K documents from the server's library. A downside of this two-round protocol is that the user's device downloads K documents rather than the one document the user eventually views in detail.

Coeus instead follows an approach consisting of three rounds of *query-scoring*, *metadata-retrieval*, and *document-retrieval* that run in succession. These rounds are depicted in the three sub-figures of Figure 1, that also shows Coeus's client-server architecture, and the server's three components: a *query-scorer*, a *metadata-provider*, and a *document-provider*.

In the query-scoring round, Coeus's client, running on a user's device, encodes the user query q into a suitable format (for example, a Boolean vector), encrypts it, and sends it to the query-scorer component of the server. The query-scorer maintains a data structure to score documents against user queries and returns an encrypted vector whose i-th component contains the query's score for the i-th document in the server's library. The client then locally processes the score vector to obtain the K indices $\{idx_1, idx_2, \ldots, idx_K\}$ for the K vector entries that have the highest values.

Next, in the metadata-retrieval round, the client takes the K indices, encodes and encrypts them in a specific way that enables oblivious retrieval of the metadata, and sends them to the metadata-provider (the middle diagram in Figure 1). The metadata-provider sends back the entries in the metadata library $\mathcal{M}$ corresponding to the K indices. The client presents the metadata of the top-K documents to the user and asks the user to select one of the documents.

Finally, in the document-retrieval round, the client uses the metadata from the previous round to get a document from the document-provider component of the server. Since document sizes vary and Coeus must not reveal the length of the retrieved document, the document-provider packs the n documents in the document library $\mathcal{D}$ into $n_{pkd} \leq n$ equal-sized *objects*. Such packing is possible as Coeus can add a document's location (e.g., the index of the object into which a document is packed) to the metadata of the document that is retrieved before the document. The user's device downloads an entire object and locally selects the required document.

2.2 Assumptions and guarantees

Threat model. Coeus assumes a strong adversary who may arbitrarily compromise the server or the network. For instance, it may log and process network packets, or the requests received and the responses sent by the server.

We assume that the adversary cannot break standard cryptographic assumptions, such as the semantic security of encryption. We also assume that the adversary does not compromise the user device.

Although we consider server-side side channels (disk access patterns, memory access patterns, etc.), we do not consider side channels that exist due to a client's participation in the system. In particular, we let the adversary learn the number of queries a user makes, the wall-clock times at which the user makes these queries, and the time the user spends in selecting one of the K documents to view in detail. A user who wishes to hide this information can send queries at a fixed schedule, and send dummy queries (e.g., "Cristiano Ronaldo") if needed, as in communication metadata hiding systems [9, 13, 60, 86].

Privacy guarantee. Coeus guarantees *query privacy* (Appendix A). Informally, an adversary learns no information about the user query q (which also means it learns no information about the metadata or the document returned by the server). This notion of privacy is formalized via a security game between a challenger and an adversary, in which the adversary supplies two queries, the challenger simulates Coeus's protocol for one of them, and the adversary guesses which query the challenger picked. In Coeus, the adversary cannot identify the query choice with probability significantly better than that of random guessing ($\frac{1}{2}$).

Non-guarantees. Coeus does not guarantee content integrity that undermines correctness but not privacy. Indeed, the server may compute scores incorrectly, or return documents that do not match the requested indices. Coeus could be extended to add protection against these attacks through additional techniques such as verifiable computation [23, 69].

2.3 Challenges

Coeus's three round protocol already improves over alternatives such the one-round or the two-round protocol (§2.1). But still, Coeus must manage the high server-side compute

overhead. This challenge is fundamental and best illustrated by an example. Suppose that a client makes a query through Coeus. Then, the three server components, namely, the query-scorer, the metadata-provider, and the document-provider must process their *entire* state (the data structure for scoring, and the libraries $\mathcal{M}$ and $\mathcal{D}$) to service the user query. Indeed, if the server were given an information that would allow it to process a subset of the scoring data structure or the libraries (say leaving out a particular document of $\mathcal{D}$), then the server would learn information about the query keywords or the document that the user is *not* interested in. Although, we cannot break this fundamental lower bound [18], our goal is to improve the concrete efficiency and provide low-latency, affordable ranking and document retrieval.

3 Background and protocol

This section describes the scoring method Coeus uses to determine a document's relevance given a user's query (§3.1), cryptographic primitives Coeus builds on (§3.2), and Coeus's protocol (§3.3) that composes these primitives to provide query privacy (§2.2). This protocol is an intermediate design point for Coeus, as one of the protocol components requires further optimizations (§4).

3.1 Term frequency-inverse document frequency

Coeus uses the term frequency-inverse document frequency (tf-idf) measure [72, 74, 101] to determine document relevance given a user query. This method is used popularly in the information retrieval community. It also expresses the scoring function as a matrix-vector product, which is a linear computation that can be performed somewhat efficiently over encrypted data (§3.2). Given that tf-idf is well-studied, we do not go into its lower-level details, but instead focus on the matrix-vector computation structure.

The main idea behind tf-idf is to assign a weight to each (term, document) pair, where a term is, a keyword or a phrase, and the weight reflects how important or relevant the term is to a document in a collection of documents. Thus, a corpus of documents is represented by a tf-idf matrix, where the matrix rows correspond to the documents in the corpus, and the columns correspond to terms in the corpus.

With this arrangement, a common way to score a document d for a query q is to add the tf-idf weights for all terms in the query. This computation can be expressed as a matrix-vector product. The query is converted to a binary vector, whose j-th component is 1 if the j-th term in the corpus is present in the query. Then, the score of a document is the dot product of the query vector with the row vector for the document in the tf-idf matrix. More generally, the scores for all documents are computed by taking the matrix-vector product of the tf-idf matrix and the query vector.

3.2 Cryptographic building blocks

Coeus obliviously performs the scoring computation and retrieves the best matching documents and their metadata (§2.1), using two cryptographic primitives: secure matrix-vector product [41] and private information retrieval (PIR) [26, 59]. The state-of-the-art constructions of these primitives [12, 46, 47] in turn rely on an underlying homomorphic encryption (HE) scheme based on lattices. The literature offers many lattice-based HE schemes [21, 22, 35, 38, 62]; we use and describe the BFV scheme [21, 35] due to its maturity [75] and involvement as a leading candidate in homomorphic encryption standardization efforts [10].

BFV homomorphic encryption scheme. In the more efficient vectorized version of BFV, a plaintext is a vector with N components and a ciphertext is a vector with $2N$ components, where N is of the form 2^x in $\{2^{11}, \ldots, 2^{15}\}$ [10]. For the plaintext, each component is an element of $\mathbb{Z}_p$, which is the set of integers modulo p. Meanwhile, each component of the ciphertext vector is an element of $\mathbb{Z}_{p\prime}$. The parameters N, p, p' can be tuned for a desired security level [10](§5).

The BFV encryption algorithm Enc adds "noise" when encrypting a plaintext into a ciphertext. [1] This noise grows as homomorphic operations are performed on the ciphertext. To ensure that the noise does not grow to a point where the ciphertext cannot be decrypted, $p\prime \gg p$ must be ensured.

The BFV scheme supports three homomorphic operations, Add, ScalarMult, and Rotate, that are used in the higher-level secure matrix-vector product and PIR primitives.

- **Add** takes as input two ciphertexts c_1 and c_2 and produces a ciphertext c_{out} that decrypts to the component-wise sum of the plaintext vectors in c_1 and c_2.
- **ScalarMult** takes as input a plaintext vector s (of same domain as a BFV plaintext) and a ciphertext vector c and produces a ciphertext c_{out} that decrypts to the component-wise product of s with the plaintext vector in c.
- **Rotate** takes as input a ciphertext c, an integer $1 \leq i \leq N-1$, and a set of *rotation keys RK*, and produces a ciphertext c_{out} that decrypts to the plaintext in c rotated left cyclically by i positions. For instance, if c encrypts the plaintext (a, b, c, d), then a rotation by $i = 3$ produces a ciphertext that decrypts to (d, a, b, c).

The set of rotation keys RK is configurable and the rotation is performed as a combination of the rotation keys, each of which indicates the number of positions to rotate. On the one extreme, $RK = \{rk_1\}$ contains a single rotation key, where rk_1 performs rotations by one position. In this configuration, each call to Rotate resolves into i single position rotations. Although the size of RK is small, this configuration is impractical due to significant noise growth. On the other extreme, the set $RK = \{rk_1, \ldots, rk_{N-1}\}$ contains $N-1$ keys,

[1] For the readers familiar with differential privacy [34], we remark that the noise in the context of homomorphic encryption is semantically much different from the noise added for differential privacy.

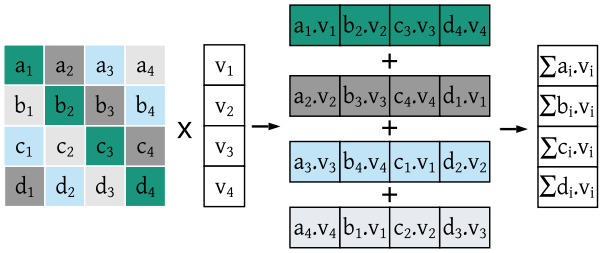

Figure 2. An illustration of secure matrix-vector product construction of Halevi and Shoup [46, 47] for a 4×4 matrix.

and Rotate calls a single i position rotation with the key rk_i. This configuration keeps noise growth in check (and reduces cpu time of Rotate), but drastically increases the size of RK needed for Rotate (with our parameters, all $N-1$ keys in RK would be ≈1.5 GiB). So we assume the default configuration implemented in the state-of-the-art library for BFV [75], where $RK = \{rk_{2^0}, rk_{2^1}, \ldots, rk_{2^{\log(N)-1}}\}$ contains $\log(N)$ keys for all powers of two between 1 and $N-1$, and rotations by i are performed using the rotation keys corresponding to positions of 1s in i's binary representation. Thus, rotation by i uses as many keys as the number of 1's in i's binary representation (i.e., i's Hamming weight); we call such internal calls to a primitive rotation operation that rotates by a power-of-two amount as *PRot*. Further, since the set of rotation keys is fixed, we will assume the set of rotation keys is implicit when specifying the rotation operation.

Secure matrix-vector product. A protocol for secure matrix-vector product runs between a client and a server, where the client has a vector, the server has a matrix, and at the end of the protocol the client learns the result of the matrix-vector product. In the process, the server learns no information about the values in the client vector.

The literature on cryptography offers many constructions for secure matrix-vector product (e.g., [11, 33, 41, 46, 47, 56, 58]). The state-of-the-art construction is that of Halevi and Shoup [46, 47]. It operates over square matrices of dimension $N \times N$ (where N is the number of components in plaintext vectors of a lattice-based HE scheme).

The main idea of the Halevi-Shoup construction is illustrated in Figure 2. The client starts by encrypting its vector of dimension $N \times 1$ using a lattice-based HE scheme and calling its Enc function. The server then performs the product of the vector with its plaintext matrix using the ScalarMult homomorphic operation. The key point here is that the server multiplies the *diagonals* of the matrix with the *rotations* of the client vector. For instance, say $N = 4$ and the matrix is as shown in Figure 2. Then, the server first scalar-multiplies the client vector that encrypts (v_1, v_2, v_3, v_4) with the matrix's main diagonal (a_1, b_2, c_3, d_4) to get a ciphertext that encrypts $(a_1 \cdot v_1, b_2 \cdot v_2, c_3 \cdot v_3, d_4 \cdot v_4)$. Then, the server rotates the client vector by one position using Rotate and multiplies the rotated vector with the matrix diagonal adjacent to the main diagonal to get encryption of $(a_2 \cdot v_2, b_3 \cdot v_3, c_4 \cdot v_4, d_1 \cdot v_1)$. And so on. Finally, the server adds (using Add) all the intermediate ciphertexts to get one ciphertext containing the result of the matrix-vector product.

We emphasize that the Halevi-Shoup method is much more efficient than naive matrix-vector multiplication that multiplies the input vector with the *rows* of the matrix (rather than its diagonals). In the naive scheme, the server would have to perform $\log(N)$ rotations for each row to add all components of the dot product and allocate the result correctly in the output vector. The Halevi-Shoup construction reduces these $\log(N)$ rotations per-row down to 1 by performing multiplications in diagonal order. In total, the construction makes N calls each to ScalarMult, Add, and Rotate.

One can trivially support matrices larger than $N \times N$, say of dimension $(m \cdot N) \times (\ell \cdot N)$, by partitioning it into square blocks of size $N \times N$. (In case the original matrix dimensions are not multiples of N, then the matrix can be padded.) In this case, the aforementioned costs get multiplied by the number of blocks $m \cdot \ell$ in the larger matrix.

Private information retrieval (PIR). A PIR protocol [26, 59] runs between a client and a server, where a client has an index i between 1 and n, and the server holds a set of n items. The protocol allows the client to retrieve the i-th item while hiding the value of i from the server.

PIR exists in two flavors: computational PIR (CPIR) [59] and information-theoretic PIR (ITPIR) [26]. CPIR protocols are computationally more expensive but make no assumptions about the server (except standard cryptographic assumptions). On the other hand, ITPIR protocols are more efficient, but require non-colluding servers. For Coeus, we use a CPIR protocol due the alignment of CPIR assumptions with Coeus's threat model (§2.2).

Although a PIR protocol allows a client to retrieve one document, it can be extended to retrieving $K > 1$ documents without naively running K parallel instances of a single-retrieval PIR protocol. These more efficient schemes for multiple retrievals are called multi-retrieval PIR [12, 50].

3.3 Coeus's protocol

Coeus composes secure matrix-vector product with PIR. Specifically, the query-scorer in Coeus's server (§2.1) maintains a tf-idf matrix, and during the query-scoring round of Coeus's protocol, uses secure matrix-vector product to score documents against a user query. This is possible as the scoring computation with tf-idf is a matrix-vector product (§3.1).

In rounds two and three, a Coeus client and the server use PIR. Specifically, in round two, Coeus runs multi-retrieval PIR between the client who has K indices and the metadata-provider who has the metadata library. For round three, the client and the document-provider use single-retrieval PIR.

A subtle issue for the third round is that of document sizes, which can vary. But PIR expects all objects in the server's library to be of the same size. Coeus addresses this issue by

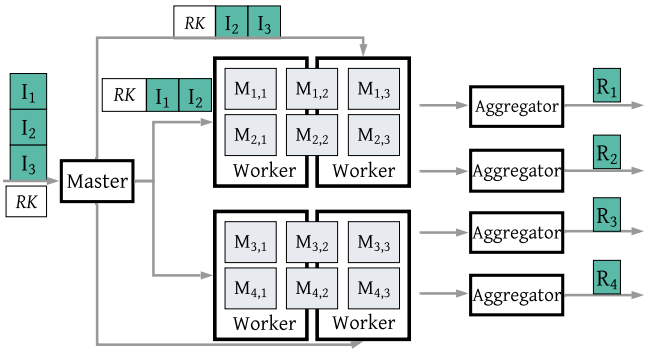

Figure 3. How Coeus partitions secure matrix-vector product onto a single master node, and a set of worker and aggregator nodes. I is the input vector from the client containing ℓ ciphertexts, one for each block along the width of the matrix. M is the matrix with $m \times \ell$ blocks. R is the result vector containing m ciphertexts. RK is the set of cryptographic keys for the Rotate homomorphic operation.

using a mix of concatenation and zero-padding, while taking inspiration from prior work on PIR with variable document sizes [44, 50]. In particular, Coeus uses bin packing to pack multiple documents into the least number of bins such that the "capacity" of each bin is equal to the size of the largest document in the document library. After bin packing, Coeus fills unfilled space in each bin with zeros. A consequence of packing is that a Coeus client needs start and end offsets of a document to extract it from a larger (binned) object. Coeus includes this information in the metadata for each document.

We remark that had Coeus not used a three-round protocol that separates out metadata retrieval from document retrieval, Coeus would have had to forego the packing technique described above. Instead, to make document sizes uniform, the natural option would have been to pad each document to the size of the largest document, thereby increasing the size of the document library and the overhead of PIR.

Security analysis. Appendix A contains a rigorous proof that Coeus's protocol provides query privacy (§2.2). Briefly, during round one, the client sends an encrypted vector after converting a query into a binary vector and encrypting it. Thus, the server learns no information about the query due to the semantic security of encryption. For rounds two and three, the security of PIR ensures that the server learns no information about the indices for which the client is retrieving objects from the metadata or the document library.

4 Large-scale secure matrix-vector product

The server-side scalability of PIR has received significant attention recently [9, 12, 13]. Besides, the metadata and document libraries are not large, at least in relation to the tf-idf matrix. But the tf-idf matrix can have millions of rows and tens of thousands of columns—a total of several hundred billion elements—corresponding to the documents and keywords in the server's document library. Thus, a fundamental question Coeus must answer is: how can its server compute the secure matrix-vector product with the tf-idf matrix while keeping the client-perceived latency small?

One option is to process the tf-idf matrix block-by-block using the Halevi-Shoup construction (§3.2), where each block is of dimension $N \times N$, and N is of the form 2^x for some integer $x \in \{11, \ldots, 15\}$ for the security of the underlying homomorphic encryption scheme [10]. This solution, however, does not meet the small latency requirement. First, the processing time for each block is several seconds even on a machine with tens of CPUs (§6.3). This expense is due to the high cost of the underlying homomorphic operations, particularly Rotate (§3.2). Second, a tf-idf matrix with billions of elements comprises of thousands of blocks. Naturally, we do not want to provision thousands of machines for the computation. Thus, how should Coeus scale the secure matrix-vector product?

Coeus reduces the work the server has to perform (§4.2, §4.3), and distributes this work efficiently over a cluster of machines (§4.4). We begin with an abstract overview of Coeus's scheme that will help set the stage for the optimizations.

4.1 Overview

Computation. Coeus's server multiplies a matrix M of dimension $(m \cdot N) \times (\ell \cdot N)$ consisting of $m \cdot \ell$ blocks each of dimension $N \times N$, with a client input vector I comprising of ℓ ciphertexts (recall each ciphertext itself encrypts a vector of dimension N) to produce a result vector R comprising of m ciphertexts. The i-th ciphertext in R is computed as

$$R_i = \sum_{j=1}^{\ell} \textsc{Block-Mult}(M_{i,j}, I_j, RK),$$

where the sum operation is the homomorphic Add operation, Block-Mult is a block-level secure matrix-vector multiplication algorithm, $M_{i,j}$ is a matrix block, I_j is a ciphertext in the client input vector, and RK is a set of client-supplied keys for the Rotate homomorphic operation.

Architecture. Coeus projects this computation onto a *master* node, and a set of *worker* and *aggregator* nodes (Figure 3). The master receives I and RK from the client. It then copies the keys RK to every worker. It also distributes one or more ciphertexts in I to each worker. The workers together compute Block-Mult$(M_{i,j}, I_j, RK)$ for all $i \in \{1, \ldots, m\}$ and for all $j \in \{1, \ldots, \ell\}$. Each worker, however, performs only part of this computation—corresponding to a *submatrix* of M. An aggregator produces one or more ciphertexts in R by adding outputs from one or more workers.

Division of matrix into submatrices. If Coeus were performing a plain matrix-vector product, it could partition the matrix into submatrices arbitrarily: a submatrix could be a single cell of dimension 1×1, or the entire matrix of dimension $(m \cdot N) \times (\ell \cdot N)$, or any dimension in between. However, the Halevi-Shoup block multiplication algorithm that Coeus builds on imposes certain restrictions due the vectorization of the underlying homomorphic operations: each diagonal

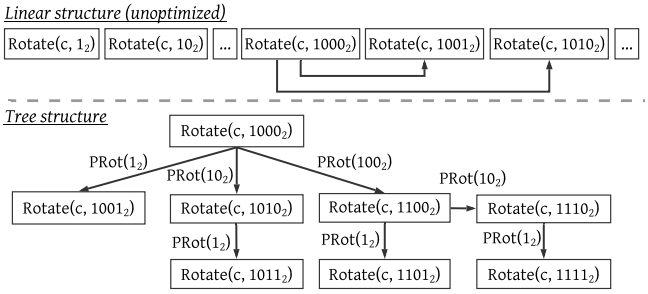

Figure 4. How Coeus conserves calls to the PRot operation.

of a $N \times N$ matrix block is encoded into a single, indivisible unit (§3.2). This means that although the submatrix width w can be any value between 1 and $\ell \cdot N$, the submatrix height h must be a multiple of N. One way to visualize this constraint is to imagine that each matrix block is transformed by taking its diagonals one-by-one and putting them as columns of the block; after this transformation, one can slice the block vertically but not horizontally.

A toy example of the computation. Suppose the matrix M has dimension 4×3 in terms of blocks. Then, the client input I has three ciphertexts, and the result vector R has four ciphertexts. Also, suppose that one of the workers gets assigned the submatrix consisting of block $M_{1,1}$ and half of block $M_{1,2}$ (first $N/2$ diagonals of $M_{1,2}$). Then, this worker receives ciphertexts I_1, I_2 from the master, multiplies I_1 and I_2 with $M_{1,1}$ and the $N/2$ diagonals of $M_{1,2}$, respectively, to obtain two ciphertexts, and sends their sum to an aggregator. This aggregator adds this ciphertext to a similar ciphertext from another worker who is responsible for the remaining half of $M_{1,2}$ and the whole of $M_{1,3}$. This final sum is R_1.

4.2 Reducing expense of homomorphic rotations

We first drill into the computation performed by a single worker, and further into the computation for a single block of the worker's submatrix. For now, assume that the width w and height h of the submatrix are both multiples of N so that the submatrix is an exact multiple of some number of blocks; we will relax this simplifying assumption shortly.

Consider the Halevi-Shoup computation for a block. It comprises of N steps, where each step rotates the plaintext in an input ciphertext c by one position (§3.2, Figure 2). As an example, if $N = 4$, and the input ciphertext c encrypts the plaintext (v_1, v_2, v_3, v_4), then the algorithm calls $\textsc{Rotate}(c, 1)$, $\textsc{Rotate}(c, 2)$, and $\textsc{Rotate}(c, 3)$ in succession. These $N - 1$ rotations consume the bulk ($\approx 90\%$) of the CPU time.

As mentioned earlier (§3.2), each call to Rotate resolves into a set of calls to a primitive rotation operation PRot that performs rotations with power-of-two amounts. For instance, $\textsc{Rotate}(c, 3)$ resolves into a call to $c' \leftarrow \textsc{PRot}(c, 2)$ followed by a call to $\textsc{PRot}(c', 1)$. In total, all $N-1$ calls to Rotate in the Halevi-Shoup algorithm make $\sum_{i=1}^{N-1} \textsc{HammingWt}(i) = (N-2) \cdot \log(N)/2$ calls to PRot, where HammingWt() returns the number of 1's in the binary representation of its input.

But observe there is significant redundancy across multiple calls to Rotate. For instance, $\textsc{Rotate}(c, 1100_2)$ calls PRot for rotation amounts eight and four, while $\textsc{Rotate}(c, 1111_2)$ calls PRot over the same ciphertext for rotation amounts eight, four, two, and one. Coeus eliminates these redundant calls to PRot and resolves the $N - 1$ calls to Rotate in the Halevi-Shoup algorithm into $N - 1$ calls to PRot.

Details. Define $\textit{Parent}(i)$ as the logical AND of the binary representation i_2 and the negation of the smallest non-zero suffix of i_2. For example, if i is 1100_2 in binary, then its smallest non-zero suffix is 100_2, and its parent is 1100_2 & $\sim 100_2 = 1000_2$. It is easy to see that the hamming distance between i_2 and $\textsc{Parent}(i)$ is one. Thus, we can obtain $c' \leftarrow \textsc{Rotate}(c, i)$ by performing one PRot over the ciphertext $\textsc{Rotate}(c, \textsc{Parent}(i))$, where the primitive rotation is for an amount equal to the smallest non-zero suffix of i_2.

A first-cut solution to leveraging this parent-child relationship is to generate all rotations of a ciphertext c, that is, $\textsc{Rotate}(c, i)$ for all $i \in \{1, \ldots, N-1\}$, sequentially, as depicted by a toy example for $N = 16$ in the top part of Figure 4. In particular, we can generate $\textsc{Rotate}(c, i+1)$ from its parent, which is one of the ciphertexts from $\textsc{Rotate}(c, 1)$ to $\textsc{Rotate}(c, i)$. This solution does eliminate redundant calls to PRot, but it has a downside that it increases memory pressure as it requires storing up to N ciphertexts in memory.

However, observe in the toy example that once the ciphertext $\textsc{Rotate}(c, 1000_2)$ is generated, the ciphertexts from $\textsc{Rotate}(c, 1_2)$ to $\textsc{Rotate}(c, 0111_2)$ can be discarded as they cannot be parents for any ciphertext after $\textsc{Rotate}(c, 1000_2)$. Similarly, once $\textsc{Rotate}(c, 1100_2)$ is generated, all ciphertexts prior to (and including) $\textsc{Rotate}(c, 1011_2)$ can be discarded, and same for $\textsc{Rotate}(c, 1110_2)$ as the parent for the next value of $i = 1111_2$ is 1110_2.

Leveraging this intuition, Coeus collapses the linear structure into an efficient tree structure that eliminates redundant PRot without increasing memory pressure, as depicted in the bottom part of Figure 4. Coeus performs a depth-first traversal through the tree and at each step in the traversal, generates a child ciphertext from its parent using one call to PRot. Coeus's algorithm garbage collects any ciphertext in a branch of the tree that has been completely traversed. Hence at any given point, the maximum number of intermediate ciphertexts stored is $\log(N)$ as the height of the tree is $\log(N)$, the number of bits in N. However, further observe that once the algorithm traverses all siblings of a given ciphertext, it can also garbage collect the parent. Hence the number of stored intermediate ciphertexts further reduces to $\lceil \log(N)/2 \rceil$.

This optimization to conserve calls to PRot applies even to fractional blocks (recall the simplifying assumption at the beginning of this subsection) that contain $d < N$ adjacent

diagonals and require performing up to d consecutive rotations. The computation for d diagonals maps to generating a subtree of the overall tree.

Cost savings. The original Halevi-Shoup algorithm applied to a $(m \cdot N) \times (\ell \cdot N)$ dimension matrix makes $m \cdot \ell \cdot N$ calls to the ScalarMult and Add homomorphic operations (§3.2), and $m \cdot \ell \cdot \sum_{i=1}^{N-1} \text{HammingWt}(i) = m \cdot \ell \cdot (N-2) \cdot \log(N)/2$ calls to PRot. Coeus's optimization reduces the calls to the expensive PRot to $m \cdot \ell \cdot (N-1)$—an improvement by a factor of $\approx \log(N)/2$.

4.3 Amortizing rotations across blocks

This subsection zooms out of block-level savings, and considers the entire submatrix at a worker (§4.1). Having potentially many matrix blocks to process raises a natural question: can we amortize the overhead *across* blocks? It turns out that the cost of rotations can be amortized.

As with the last subsection, we begin by making a simplifying assumption that the width w and height h of the submatrix are multiples of N; we will relax this assumption towards the end of this subsection.

Consider the computation imposed by the Halevi-Shoup algorithm on a set of matrix blocks that are *vertically aligned* in the submatrix: that is, the blocks $\{M_{i,j}\}$ for a fixed j and different values of i (up to h/N values of i, which is the number of vertically-stacked blocks in a submatrix of height h). First, these blocks are multiplied by the same input ciphertext: the j-th ciphertext I_j in the client input vector I. Second, when these blocks are multiplied by I_j, the Halevi-Shoup algorithm produces the same sequence of rotations for each block: $\text{Rotate}(I_j, 0), \text{Rotate}(I_j, 1), \ldots, \text{Rotate}(I_j, N-1)$.

Coeus eliminates this redundancy in rotations by reordering homomorphic operations. If Coeus were to process each of the vertically-aligned blocks independently, then it would perform a computation structured as: for each of the h/N blocks, perform a sequence of N Rotate, N ScalarMult, and N Add. Instead, Coeus restructures this computation along the diagonals of the blocks: for each of the N diagonals, perform one Rotate followed by h/N ScalarMult's and h/N Add's for the h/N blocks.

This optimization extends to fractional blocks that are vertically aligned and contain $d < N$ diagonals each. These diagonals are multiplied by consecutive d rotations of the *same* input ciphertext. Thus, the homomorphic operations can be reordered as before to amortize the costs of rotation.

Cost savings. Let h be the height of the submatrix and w be its width. Then, the submatrix has $f = (h/N) \cdot \lfloor w/N \rfloor$ full blocks and $t = (h/N) \cdot (w - N \cdot \lfloor w/N \rfloor)$ diagonals in the fractional blocks. Without the optimization presented in this subsection, Coeus would make $f \cdot N + t$ calls to each of ScalarMult, Add, and PRot. With the optimization, the number of calls to PRot reduces by a factor of h/N.

4.4 Setting submatrix dimensions optimally

So far, we have discussed the matrix-vector product while keeping submatrix dimensions abstract: width w and height h. But, how should these values be set?

A strawman design is to partition the matrix into submatrices by using a strategy that is commonly used for *plaintext* matrix-vector multiplication. In plaintext multiplication, the compute time to process a submatrix is proportional to the area of the submatrix—and does not depend on the *shape* of the submatrix. This performance characteristic leads to a common strategy of breaking up the matrix into square submatrices [73, 93].

However, for Coeus, this strategy is sub-optimal as the compute time to process a submatrix *depends* on the shape of the submatrix: taller (but less wide) submatrices have lower compute overhead due to the amortization of rotations (§4.3). A downside of making submatrices less wide, however, is the increase in aggregator overhead to combine results from each worker. Thus, one needs to find a submatrix shape that minimizes the total time to compute the matrix-vector product considering both per-worker and across-worker work.

We first present an analytical model for the time to compute the matrix-vector product. This model has limitations and makes several simplifying assumptions, and thus cannot be directly used, but serves as a tool to understand the system behavior. We then use this analytical model to present Coeus's empirical method to determine the submatrix shape.

Analytical model. Our goal is to minimize the total time for computing the matrix-vector product. This time is the sum of three components, $t_{distribute}$, $t_{compute}$, and $t_{aggregate}$, which correspond to the times for the three stages of computation: distributing inputs from the master to the workers, processing each submatrix parallelly at the workers, and aggregating worker outputs (§4.1, Figure 3).

The first component $t_{distribute}$ is the sum of the time for copying rotation keys RK from the master to each worker, and copying parts of the input vector I as needed to the workers. If the total number of workers is $n_{workers}$, and the time to transfer one copy of RK out of the master is $t_{key\_transfer}$, then the total time for the copying of keys is $n_{workers} \cdot t_{key\_transfer}$. For the remaining cost of copying parts of the input vector I, observe that for a submatrix of width w, a worker needs $\lceil w/N \rceil$ ciphertexts. Thus, if $t_{ct\_transfer}$ is the time to transfer one ciphertext, the total time for input distribution phase is

$$t_{distribute} = n_{workers} \cdot (t_{key\_transfer} + \cdot \lceil w/N \rceil \cdot t_{ct\_transfer}). \quad (1)$$

The second component of the total time, $t_{compute}$, is the time taken by a worker to process its submatrix. This time follows from the number of per-worker homomorphic operations executed. This number was analyzed in the previous subsection (§4.3). If t_{add}, t_{mult}, and t_{rot} are the times to perform one homomorphic ScalarMult, Add, and PRot, then

$$t_{compute} = (h \cdot w)/N \cdot (t_{mult} + t_{add}) + w \cdot t_{rot}. \quad (2)$$

Finally, the aggregation time $t_{aggregate}$ equals the sum of the times to transfer intermediate ciphertexts from workers to the aggregators, and the time each aggregator takes to add the ciphertexts. The former equals $m \cdot \lceil (\ell \cdot N)/w \rceil \cdot t_{ct\_transfer}$, and the latter equals $m \cdot \lceil (\ell \cdot N)/w \rceil \cdot t_{add}/n_{agg}$, where m is the number of blocks across the height of the original matrix M, and n_{agg} is the number of aggregators. The rationale is that the matrix has $\lceil (\ell \cdot N)/w \rceil$ vertical partitions (recall that matrix dimensions are $(m \cdot N) \times (\ell \cdot N)$), and each generates m ciphertexts. Thus,

$$t_{aggregate} = m \cdot \lceil (\ell \cdot N)/w) \rceil \cdot (t_{ct\_transfer} + t_{add}/n_{agg}). \quad (3)$$

Observe that $t_{distribute}$ and $t_{compute}$ depend linearly on the value w ($h \cdot w$ in Equation 2 is the area of each submatrix and is fixed depending on the total area of M and $n_{workers}$). Thus, wider submatrices increase input distribution and computation time. In contrast, $t_{aggregate}$ depends inversely on w, and reduces with the width of the submatrix. Due to these opposing forces, the total time is a convex function of w.

Ideally, we would like to derive an optimal value for w (the lowest point of the convex function) that would minimize the total time. However, there are two issues. First, the model uses uniform values for network transfer times for both keys and ciphertexts that do not account for load, network conditions, and the topology in which workers and aggregators are connected. Second, the total time function is not continuous and differentiable. Hence, in Coeus we develop an empirical method to determine the submatrix width value.

Coeus's empirical method. One tempting option is to configure and deploy a prototype of Coeus for all possible values of w and measure the total time to compute the matrix-vector product. But observe that the total time is a convex function. Thus, we can perform a more efficient directional search inspired by gradient descent in machine learning [43]. Coeus starts by measuring the time for any value of w, say w_{start}; then takes a step in an increasing or decreasing direction of w and measuring the time for a new value of w; then, if the time decreases, it continues in the same direction; otherwise, it goes back to w_{start} and takes a step in the opposite direction. Coeus repeats this process until steps in both directions increase time. Besides following this search approach, Coeus explores only select values of w such that either N is divisible by w, or $\ell \cdot N$ is divisible by w (when $w > N$). These constraints allow Coeus to more easily deal with the boundary conditions due to the ceil function.

5 Implementation details

Query-scorer. Coeus's query-scorer (§2) is written in ≈2200 lines of C++. Its main piece is a distributed implementation of Coeus's secure matrix-vector product (§4) that uses the state-of-the-art Microsoft SEAL library [75] for BFV homomorphic encryption. Recall that the BFV scheme has three parameters: the bound p on each component of the plaintext vector, the dimension N of this vector, and the bound $p\prime$ on each component of the ciphertext vector (§3.2). We set p as a 46-bit prime (`0x3FFFFF84001`), $p\prime$ as a product of three 60-bit primes {`0xFFFFFFFFFD8001`, `0xFFFFFFFFFE8001`, and `0xFFFFFFFFFFC001`}, and N as 2^{13}. These values provide 128-bit security [10]. Furthermore, they satisfy the constraint $p\prime \gg p$ such that the query-scorer can perform the required number of homomorphic operations for the large tf-idf matrix while staying within the noise budget (§3.2).

tf-idf matrix preparation and encoding. The query-scorer converts a document library into a tf-idf matrix (§3.1) using the Gensim Python library for natural language processing [1, 70]. The query-scorer must also encode the tf-idf matrix into plaintext vectors in the BFV scheme (§3.3). One way to perform this encoding is to map each matrix element individually into a single component (of size $\log(p)$) of the plaintext vector. However, this method is wasteful as p is a 46-bit prime and tf-idf values are within a small range. Instead, Coeus uses the standard ideas of quantization [40] and input packing [6, 45] to map multiple (three in our prototype) matrix elements into a single component of the plaintext. For example, if a_1, b_1, and c_1 are the beginning elements of the first three rows of the tf-idf matrix, then Coeus first quantizes each one to one of 2^{10} levels, and then packs them into the value $a_1 \cdot d^2 + b_1 \cdot d + c_1$ made of three "digits" of size $\log d = 15$ bits each. As long as the number of keywords in user queries is less than 2^5, this arrangement ensures that additions of packed values happen digit-wise without overflow.

Metadata and document providers. Coeus's metadata and document provider are written in ≈1200 and ≈1000 lines of C++, respectively. Underneath, the metadata-provider contains our implementation of the multi-query PIR protocol of Angel et al. [12], which in turn builds on the state-of-the-art SealPIR PIR library [2]. Meanwhile, the document-provider directly uses the SealPIR library (which, by default, provides single-retrieval capability). Both the metadata and document provider use a master-worker architecture for the PIR server, where the master receives client request and distributes work to the workers. Similarly, both providers configure SealPIR to provide 128-bit security. Finally, the document-provider implements the first-fit-decreasing bin packing algorithm to pack the set of variable-sized documents into a PIR library with equal-sized objects (§3.3).

6 Evaluation

Our evaluation focuses on highlighting Coeus's latency for a user request, Coeus's resource overheads (CPU, network, and dollars) for both its server and clients, and the benefits of Coeus's techniques in reducing these overheads. A summary of our main results is as follows:

- For a corpus of 5M documents from English Wikipedia and a dictionary with 65,536 keywords, Coeus's latency is 2.81 s, 0.55 s, and 0.54 s for its three protocol rounds of

query-scoring, metadata-retrieval, and document-retrieval (§2.1). For the same configuration, a baseline system with two rounds incurs a total latency of 93.9 seconds.

- For 5M documents and 65,536 keywords, Coeus's resource consumption (CPU and network) is substantial. However, when converted to a dollar amount, this cost is 6.5 cents per request. In contrast, the baseline costs 1.62 dollars.
- Both system-level design techniques (§2.1, §3.3) and optimizations to secure matrix-vector product (§4.2–§4.4) significantly improve Coeus's performance.

Baselines. We compare Coeus to two baseline systems. *B1* composes the secure matrix-vector product construction of Halevi and Shoup for query-scoring with PIR (specifically, SealPIR [2]) for document retrieval to form a two-round protocol (§2.1, §3.3). *B2* improves on B1 by incorporating Coeus's technique of splitting the document retrieval round into separate rounds for metadata and document retrieval (§3.3). Notably, both B1 and B2 apply the Halevi-Shoup algorithm for query-scoring to the tf-idf matrix block-by-block, and distribute this computation onto a cluster of machines by assigning square, equal-sized submatrices to each worker machine. The difference between B2 and Coeus is the improvements to secure-matrix vector product (§4.2–§4.4).

Dataset. Our seed corpus is an English Wikipedia articles dump from Feb 1, 2021 [3]. It contains ≈6M articles. However, Coeus's topic modeling library Gensim [1, 70] (§5) removes small re-directional articles, which leaves 4,965,789 articles. We form a keyword dictionary from these articles by picking keywords that have the highest idf (specificity) (§3.1).

Experiment configurations. We vary the number of documents (n) in the server's document library, the number of keywords in the dictionary, and the number of machines assigned to the server. To vary n, we sample documents from the seed corpus uniformly at random. This sampling dictates the size of the document library. For the baseline B1, we pad each sampled document to the size of largest document in the set of sampled documents. In contrast, for B2 and Coeus, we pack (concatenate) smaller documents before padding (§3.3). Each document's metadata is 320 bytes, which includes 255 bytes of title [5], and 40 bytes of a short description [4], among other information such as the document's location in the (packed) document library in the case of B2 and Coeus (§3.3). For the baseline B1, we set $K = 16$ as the number of documents the client retrieves in the second protocol round, while for B2 and Coeus, K equals the number of documents for which the client receives metadata in the second protocol round. Finally, we set the number of tf-idf matrix columns equal to the number of keywords, and the number of rows of the matrix equal to $\lceil n/3 \rceil$ after taking into account tf-idf matrix preparation from the document data (§5).

Testbed. We run Coeus's server and a client over a set of machines in the US East (Ohio) AWS EC2 data center. Each component of the server (query-scorer, metadata-provider, and document-provider) uses one machine of type `c5.24xlarge` (96 vCPU, 192 GiB RAM, and 25 Gbps network bandwidth) to host its master, and a variable number of machines of type `c5.12xlarge` (48 vCPU, 96 GiB RAM, 12 Gbps network bandwidth) to run its workers. For the query-scorer, we also run an aggregator on each of the worker machines. The client uses a single vCPU of a machine of type `c5.12xlarge`.

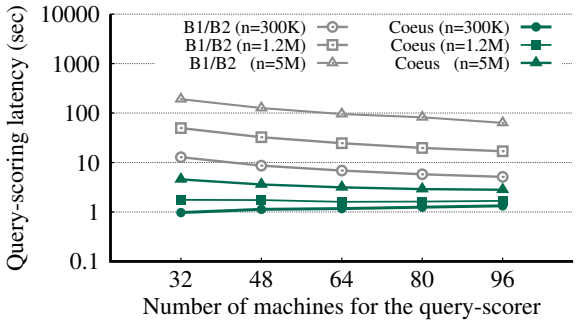

Figure 5. User-perceived latency for Coeus's query-scoring round. n is the number of documents in the document library. The number of keywords is set to 65,536.

6.1 Latency performance of Coeus

We first focus on the query-scoring round of Coeus's protocol as it is different for Coeus and both the baselines, and then on the other two rounds (metadata and document retrieval), which are different for Coeus and only the B1 baseline.

Coeus versus the baselines for query scoring. Figure 5 shows the user-perceived latency of Coeus and the baselines for their query-scoring round, while keeping the number of keywords fixed to 65,536 but varying both the number of documents n in the document library and the number of worker machines for the query-scorer. Coeus's latency is, in general, much lower than the baseline latency. For example, for 5M documents and 96 machines, Coeus's latency is 2.8s, while the baseline's latency is 63.4s, which is 22.6× higher. These improvements are due to Coeus's optimizations to secure matrix-vector product that fundamentally reduce, and efficiently distribute, the server's work (§4.2–§4.4). We will evaluate these optimizations individually in §6.3.

Variation with the number of machines. The latency of query-scoring initially decreases with the number of machines for the query-scorer, then reaches an inflection point, and then increases with more machines. This trend is most clear to see for Coeus when n = 1.2M: the latency is 1.75s for 32 machines, decreases to 1.60s for 64 machines, and then increases to 1.68s for 96 machines. The reason is that although the per-machine compute time decreases with an increase in the number of machines due to a reduction in the size of the submatrix assigned to a machine, the overhead of aggregating intermediate outputs increases (§4.4). Thus, adding more machines does not necessarily improve latency. (For n = 300K and n = 5M, the curves for Coeus are to the right and left of the inflection point, respectively.)

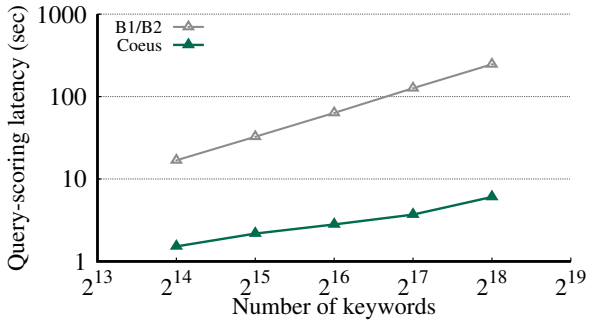

Figure 6. User-perceived latency for Coeus's query-scoring round with the number of keywords. The number of documents is set to 5M, and the number of machines for the query-scorer is 96.

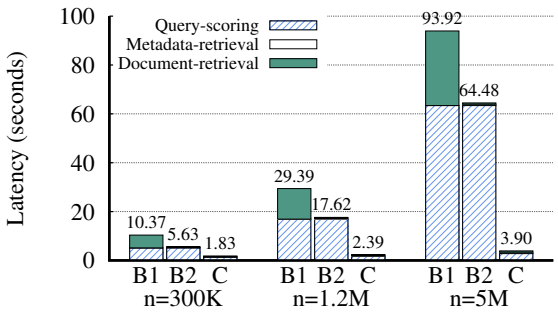

Figure 7. User-perceived latency for Coeus (C) and the baseline systems (B1 and B2) with a varying number of documents (n) in the document library. The number of keywords is 65,536. The text provides details of the machines for these experiments.

Variation with the number of documents. Coeus's latency for query-scoring increases with the number of documents, but not linearly. This is due to the amortization of the cost of Rotate operations across matrix blocks (§4.3). For instance, for 32 server machines, latency for Coeus grows from 0.97s for 300K documents to 1.75s for 1.2M documents—an increase of 1.8×. In contrast, the corresponding latency for the baselines increases from 12.8s to 49.7s (an increase of 3.88×). This linear growth for the baselines is expected as they perform the secure matrix-vector product block-by-block, without any amortization of costs across blocks.

Variation with the number of keywords. Figure 6 shows how Coeus's query-scoring latency changes with the number of keywords when n = 5M and the query-scorer runs over 96 worker machines. Coeus's latency increases linearly with the number of keywords with a slope smaller than one. For instance, it increases by 4.1× from 1.5s to 6.1s when the number of keywords increase by 16× from 2^{14} to 2^{18}. The reason the latency does not increase sixteen times (even though the matrix increases by that factor) is that Coeus readjusts submatrix dimensions to make submatrices taller, which reduces server's work by further amortizing the cost of Rotate operations (§4.4, §4.3). In contrast, the baseline latency increases with a slope of ≈ 1 as the baseline secure matrix-vector product computation time increases linearly with the width of the matrix.

Latency for metadata and document retrieval. Figure 7 shows user-perceived latency for Coeus and the baselines for the rounds of metadata-retrieval (if applicable) and document-retrieval. (For completeness, the figure also shows query-scoring latency from Figure 5.)

The baseline B1 does not have an explicit metadata-retrieval round. It uses 48 worker machines to retrieve metadata and data together for K = 16 documents. This choice of 48 machines is based on parameters for SealPIR and the size of the document library. For instance, SealPIR's multi-retrieval scheme requires partitioning the document library into a number of buckets that is a multiple of K. We choose 48 buckets and assign each bucket to a distinct worker machine.

In contrast to B1, the baseline B2 and Coeus have an explicit metadata-retrieval round. We configure these systems to use 6 worker machines for the metadata-provider and 38 machines for the document-provider. Again, these choices are based on SealPIR parameters and the size of the metadata and document libraries. For instance, the largest object after document packing is 142.5 KiB, which encrypts into 38 BFV ciphertexts in SealPIR, where each is processed in parallel.

Coeus's (and B2's) separation of metadata retrieval from document retrieval significantly improves latency over B1. For example, for n = 5M, B1 takes 30.5s, while Coeus takes 0.55s for metadata retrieval and 0.54s for document retrieval. This gain is for two reasons. First, B1 retrieves K = 16 documents each of size 140.7 KiB privately from a document library via multi-retrieval PIR, whereas Coeus retrieves a single document and 320 byte metadata for each of the K documents. Second, B1's document library is much larger than Coeus's: 670.8 GiB versus 13.1 GiB. This is because B1 pads each document in its library to the size of the largest document (140.7 KiB), whereas Coeus packs multiple smaller documents into 96,151 objects each of size 142.5 KiB (§3.3).

Summary. Coeus's latency is dominated by that of query-scoring. Further, Coeus's techniques are effective: the decoupling of metadata from document retrieval reduces latency from 93.9s to 63.5s for 5M documents and 65,536 keywords, and the optimizations to secure matrix-vector product further reduce this latency to 3.9s (an improvement of 24×).

6.2 Resource overheads of Coeus

This section explores the overhead Coeus imposes on clients and estimates the combined overhead in terms of dollars.

Client-side overhead. Figure 8 shows client-side CPU time, network upload, and network download for Coeus and the baselines with a varying number of documents (n) in the server's document library. Coeus's network overhead is substantial and thus Coeus requires significant download bandwidth at the client. This is because the query-scoring response contains a score for each document, and thus grows with the number of documents (§2.1, §3.3). Meanwhile, the

	n=300K	n=1.2M	n=5M
Client CPU (sec)			
B1	4.04	4.43	5.54
B2/Coeus	0.34	0.61	1.64
Upload (MiB)			
B1	12.29	12.29	17.89
B2/Coeus	14.31	14.31	14.31
Download (MiB)			
B1	460.27	470.02	508.02
B2/Coeus	18.78	28.53	66.53

Figure 8. Client-side costs per request for Coeus and the baseline systems (B1 and B2) for a keyword dictionary with 65,536 keywords and a varying number of documents (n).

upload bandwidth does not change with n, as (a) the length of the input vector to query-scoring depends on the number of keywords (and not on the number of documents), and b) the size of the inputs to PIR (specifically, SealPIR) follows a step function and changes only for a value $n > 16$M.

Coeus's overheads, particularly the network downloads, are significantly lower than that of the baseline B1. The reason is that B1 privately downloads $K = 16$ documents while Coeus retrieves a single object and K smaller metadata.

Dollar cost. We convert both the network and the server-side resource overhead to a dollar amount. For the former, we use a network pricing model of \$0.05 per GiB, which is Amazon's price for bulk network downloads (Amazon does not charge for uploads) [77]. For the server's cost, we multiply the machine rent for Amazon EC2 (`c5.12xlarge` and `c5.24xlarge` machines cost \$0.744 and \$1.488 per hour, respectively [76]) with the number and type of machines we use and the time for which we use them to service a request.

For Coeus, the per-request dollar cost for the configuration of 5M documents and 65,536 keywords is 6.5 cents, of which 5.9 cents is due to query-scoring. The baseline B2 increases this cost to 1.29 dollars, of which 1.28 dollars is due to query scoring. Further, B1 increases this cost to 1.62 dollars, where the additional 34 cents is due to the more expensive document retrieval. Thus, Coeus's improvements take oblivious document ranking and retrieval a level up in affordability.

6.3 Performance of secure matrix-vector product

A major part of Coeus's gain over the baselines is courtesy of the improvements to secure matrix-vector product (§4). This section zooms into the performance of this primitive in isolation. We first focus on a matrix that fits into a single machine, and then on Coeus's distributed implementation over a cluster of machines. The results also shed light on when Coeus's construction could be beneficial to other applications.

Single machine performance. We run the server component of the secure matrix-vector product on a single CPU of an AWS machine of type `c5.12xlarge`. We compare (a) the baseline Halevi-Shoup construction extended to process multiple blocks block-by-block, (b) this baseline plus Coeus's first optimization (§4.2) to reduce the overhead of rotations (Coeus-opt1), and c) this previous variant extended with Coeus's technique (§4.3) to amortize rotation time across blocks (Coeus-opt1-opt2).

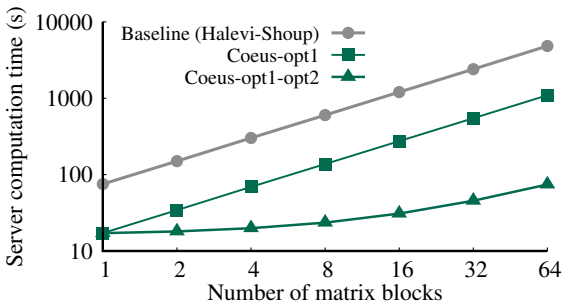

Figure 9. Server CPU time to perform secure matrix-vector product.

Figure 9 shows the CPU time to compute the product. Each block is of dimension $N \times N$, where $N = 2^{13}$, and new blocks are added vertically on top of existing blocks.

Coeus-opt1 reduces computation time by a constant factor of $\approx 4.4\times$ relative to the baseline. This reduction in time is due to the constant $\log(N)/2 = 6.5$ factor savings in the time for the Rotate operations (§4.2). Coeus-opt1-opt2 further reduces overhead by amortizing the cost of rotations across blocks (§4.3). For instance, for the baseline Halevi-Shoup construction, increasing the number of blocks from one to sixty-four increases time linearly from 75s to 4,834s (an increase of 64.4×), but for Coeus-opt1-opt2, the time increases from 17.1s to 74.2s (a factor of 4.34). Overall, for the data point with 64 blocks, the baseline Halevi-Shoup construction takes 4,834s, Coeus's first variant (Coeus-opt1) reduces that time to 1,094s, and Coeus's version with both optimizations (Coeus-opt1-opt2) reduces time to 74.2s.

Multiple machine performance. Coeus distributes secure matrix-vector product computation efficiently onto a cluster of machines, by optimally shaping the submatrices for the worker nodes (§4.4). We compare Coeus's performance with and without this optimization. We run the server component of the secure matrix-vector product over a cluster of 64 machines of type `c5.12xlarge` while utilizing all CPUs on each machine. We measure the wall-clock time for the computation while varying the submatrix width.

Figure 10 shows the wall-clock time for various phases of secure-matrix vector computation (input distribution from the master to the workers, processing of submatrices at all worker nodes, and the aggregation of intermediate outputs generated by the workers) for an example matrix with 2^{20} rows and 2^{16} columns. The figure also shows the end-to-end (total) time measured by the client.

Overall, the total time curve is convex: the time is higher than its lowest value when submatrices are either too thin (left side of the x-axis) or too wide (right side of the x-axis). This convex shape is due to two competing forces. On the one hand, the time to process the submatrices increases with

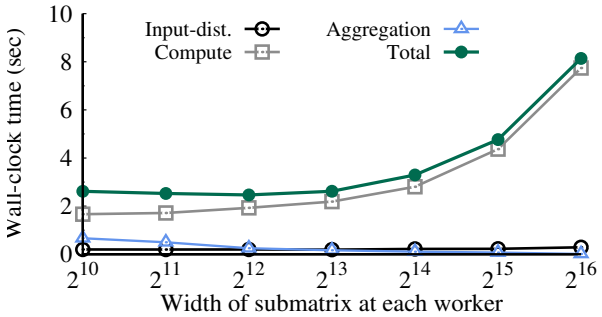

Figure 10. Wall-clock time for various phases of computation of Coeus's secure matrix-vector product: input distribution, computation at the workers, and aggregation of intermediate results. The curve labeled "total" is the end-to-end time measured at a client.

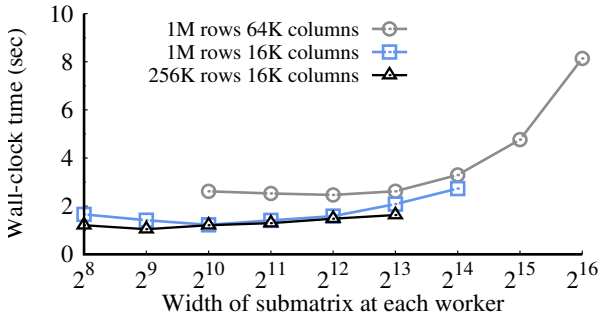

Figure 11. Wall-clock time for Coeus's secure matrix vector product protocol over 64 machines for different matrix dimensions and varying submatrix widths.

width due to a reduction in the amortization of ROTATE cost. (The time for input distribution also increases with width but slowly.) On the other hand, the cost of aggregation decreases with width. Coeus balances these two forces by finding and setting the optimal width for the submatrices (§4.4). Indeed, if Coeus had used the solution of square submatrices, then its time would be 4.76s (the point with width of 2^{15}) rather than 2.46s (width of 2^{12})—an improvement of 1.93×.

The above experiment clarifies that statically setting square submatrices is suboptimal. But could we statically set submatrices to a rectangular shape and get most of the benefit provided by Coeus's scheme? This question is especially compelling as the total time curve (Figure 10) changes slowly around the optimal point of 2^{12} width. Thus, as a concrete example, could one always set submatrix width to 2^{12}?

To answer this question, we rerun the experiment above for three different matrix dimensions: 1M rows and 64K columns, 1M rows and 16K columns, and 256K rows and 16K columns. Figure 11 shows the results. The inflection (optimal) point differs significantly—4096, 1024, and 512, respectively—for the three dimensions. Further, statically picking either of these widths is detrimental for the other configurations. For instance, if we pick 4096 as the submatrix width, then Coeus would incur 41% more latency (1.47s instead of 1.04s) relative to the optimal point for the matrix with 256K rows and 16K columns. On the other hand, picking a submatrix width of 512 will be optimal for this matrix with 256K rows and 16K columns but increase latency by 16% for the matrix with dimensions 1M rows and 16K columns. In general, the optimal point depends on various factors such as matrix dimensions, machine performance characteristics, and network connectivity between machines. Besides, these factors change over time due to updates to the document library and upgrades to the infrastructure.

6.4 Comparison of Coeus to a non-private baseline

We implemented a tf-idf based system that does not hide user query and the matched documents. This baseline implements a two-round protocol. In the first round, a client sends a query to the server in plaintext. The server computes the tf-idf scores for each document and returns metadata for the top $K = 16$ documents. In the second round, the client selects one document from the top-K and retrieves it from the server. With 5M documents and 65,536 keywords in the tf-idf matrix, and after distributing the server's workload over 48 machines of type `c5.12xlarge`, the end-to-end latency experienced by a client is ≈90ms, which is 44× lower than Coeus. The dollar cost for a single query is 0.09 cents, 72× cheaper than Coeus.

Coeus is different to a non-private baseline also in terms of expressiveness of queries. It supports tf-idf-based ranking over a multi-keyword query, but not other forms of queries such as Boolean queries with AND, OR, and NOT operators, fuzzy queries that auto-correct words that are spelled incorrectly, and wildcard and regular expression queries that enable search for patterns. Supporting these queries in Coeus requires future research, though we note that limited query processing, e.g., checking for typographical errors for fuzzy queries, could be done at the client-side.

7 Related work

Searching over encrypted private data. Starting with the seminal work of Song et al. [80], a large body of literature has focused on searching on encrypted *private* data held at a remote server (we refer the reader to surveys and recent papers on this problem [20, 24, 29, 30, 49, 91]).

Two characteristics differentiate this problem from the problem Coeus addresses. First, this problem considers a scenario where the documents are owned by one or more users, while their storage is outsourced. Thus, the data owner can encrypt its documents using a *symmetric* encryption scheme (e.g., [27]) or include an encrypted index that later helps with search. As noted earlier (§1), such encryption is not possible when data is public. Second, schemes in this category focus on searching rather than ranking. For example, a recent system DORY [29] supports retrieval of documents exactly matching a keyword.

Ranking over encrypted private data. A body of literature extends the capability of searching on private encrypted data with the capability to rank the search results [8, 31, 48, 53, 57,

67, 67, 79, 82, 83, 88, 90, 94, 99]. Among these, the schemes of Yu et al. (two-round searchable encryption or TRSE) [99] and Strizhov and Ray [82] are related to Coeus.

Both these schemes support ranking using tf-idf over a two-round protocol that is similar to the two-round baseline B1 discussed and evaluated in this paper (§2.1, §6). In the first round, a user sends a homomorphically encrypted query to a remote server and learns relevance scores for each document. Then, in the second round, the user retrieves the top-K documents. Despite the similarities to B1, we compare Coeus to B1 rather than these existing schemes, for two reasons.

First, in these existing schemes, the tf-idf matrix is encrypted as the data is private and owned by a data owner. Thus, the remote server multiplies an encrypted matrix with an encrypted vector. In contrast, in the baseline B1 (and in Coeus), the matrix is in plaintext and only the vector is encrypted which results in cheaper server-side operations. Second, these existing schemes inefficiently compute the matrix-vector product. TRSE uses the homomorphic encryption scheme of van Dijk et al. [87] which has large parameters and lacks support for vectorized operations. Meanwhile, Strizhov and Ray's scheme uses the more efficient BGV scheme [22] but computes the product naively by multiplying the vector with each matrix row. In contrast, B1 uses the state-of-the-art construction of Halevi and Shoup [46, 58] (§3.2).

Searching over public data. PIR [26, 59] and its extensions are designed for *public* data. Indeed, PIR in its basic form allows retrieval by index from a public library. With PIR-by-keywords [25, 37], a user specifies a keyword and retrieves *one* of the documents that contains the keyword. Therefore, PIR-by-keywords is most applicable to a setting where keywords are unique, for example, key-value stores [13]. SQL-PIR [66] and Splinter [89] extend the PIR-by-keywords interface to support data retrieval using a subset of SQL. However, they do not support selective, oblivious aggregation across columns as in tf-idf scoring computations. Moreover, they assume non-colluding servers unlike Coeus which does not make such assumptions about the server (§2.2). Finally, private stream searching [19, 28, 68] extends search to a stream of public documents such as Google News alerts [36, 65, 95–98, 100]. But, as mentioned earlier (§1), these works do not consider ranking. In contrast to all these works, one can view Coeus as an extension to PIR that prefixes a ranking stage to the private document retrieval stage.

Other related work. Other approaches to searching or ranking privately include trusted execution environments (TEEs) such as Intel SGX [51, 55, 64, 78, 84], anonymous communication systems such as Tor [85], and obfuscation-based techniques that send dummy queries besides real queries [15, 32, 92]. These approaches are either orthogonal or do not provide strong guarantees: TEE-based solutions require trusting the manufacturer of the TEE, anonymous communication systems hide identity but reveal the personally identifiable information (PII) in the query that can in turn reveal a user's identity [16], and obfuscation-based techniques are heuristic in nature and thus susceptible to attacks that separate out real queries from dummy queries [15, 92]. In contrast to these solutions, Coeus hides the content of user queries (and not user identity), and does so provably by incorporating and refining advanced primitives from cryptography.

8 Summary and future work

Coeus, to our knowledge, is the first end-to-end system that supports oblivious ranked retrieval over large scale public data and an untrusted infrastructure. Prior approaches either did not support ranking or only managed private data. One can view Coeus as an extension to the PIR domain that efficiently supports ranking by exploiting standard tf-idf statistical methods. At Coeus's core is a new three round protocol that separates metadata retrieval from document retrieval (§2.1, §3.3), and a novel secure and efficient matrix-vector product protocol (§4) based on the Halevi and Shoup method. This latter scheme, although designed primarily for oblivious document retrieval may be useful in other application contexts. Coeus demonstrates that oblivious ranked document retrieval, which up to now was practically impossible due its high overhead costs, has come to the realm of the possible. Our hypothetical, privacy conscious Ziv can now use Coeus to obliviously retrieve from Wikipedia, with its corpus of about 5 million documents, the history of any event of interest in under 4 seconds (rather than minutes) and at a cost of single digit cents (rather than dollars). Needless to say, this is not a panacea, but a significant improvement that paves the way for a practical future where privacy is within the reach of the masses.

In terms of further improvements, one avenue is to reduce the server-side compute overhead, which is still the main bottleneck. Here, accelerators such as GPUs may drive down costs for both secure matrix-vector product and PIR. The sparsity of the tf-idf matrix too presents an opportunity as it contains many zero entries. One can also consider concurrent queries and batch processing opportunities that are not applicable with a single query. Finally, besides performance, one can improve expressiveness by adding more types of queries such as fuzzy queries, as discussed earlier (§6.4).

Coeus's source code is available at
https://github.com/ishtiyaque/Coeus_artifact.

Acknowledgments

We thank Dheeraj Baby, John Gilbert, our shepherd Jon Crowcroft, and the anonymous reviewers of SOSP 2021 for their feedback and insightful comments that helped improve this paper. This work is funded in part by DARPA under agreement number HR001118C0060 and NSF grants CNS-1703560 and CNS-1815733.

A Security proof (not peer-reviewed)

This appendix is included to show that Coeus's protocol (§3.3, §4) for oblivious document ranking and retrieval satisfies the notion of query privacy (§2.2), however readers should note that this section has not been peer-reviewed.

We first define an abstract protocol for oblivious document ranking and retrieval, then describe a cryptographic security game that captures the notion of query privacy, and finally show why an adversary cannot win this game with non-negligible probability when the abstract protocol is instantiated with Coeus's protocol (§3.3).

A.1 An abstract description of protocol

A protocol for oblivious document ranking and retrieval runs between a server and a client. The server begins with a data structure for scoring document relevance given a client query, a metadata library containing n metadata objects for n documents, and a document library that contains the n documents packed into n_{pkd} equal-sized objects, where a document does not span more than one object. Meanwhile, the client begins with a multi-keyword search query q. At the end of the protocol, the client receives one object in the document library.

This protocol consists of three algorithms: SQuery, MQuery, and DQuery.

SQuery$(1^\lambda, \text{Dict}, q)$ takes as inputs a security parameter 1^λ, a dictionary of keywords Dict, and a multi-keyword search query q, and outputs a query q_s for scoring relevance of q against all n documents in the server's document library.

MQuery$(1^\lambda, n, \{idx_1, \ldots, idx_K\})$ takes as input a security parameter 1^λ, an integer n that represents the number of objects in the server's metadata library, and a set of K indices whose values are between (and inclusive of) 1 and n, and outputs a query q_m that is suitable for retrieving objects in the metadata library whose indices are those in the set $\{idx_1, \ldots, idx_K\}$.

DQuery$(1^\lambda, n_{pkd}, idx)$ takes as input a security parameter 1^λ, an integer n_{pkd} representing the number of objects in a suitable encoding of the document library, and an index whose value is between 1 and n_{pkd}, and outputs a query q_d for retrieving one of the objects from the document library.

The protocol proceeds in three rounds. In the first round, the client runs the SQuery algorithm and sends q_s to the server. The server responds with an answer which consists of relevance scores for n documents in the server's document library. The client processes these scores to extract the value n and a set of K indices corresponding to the highest scoring documents. In the second round, the client feeds the outputs from the first round as inputs to MQuery, and sends the output q_m to the server. The server processes this query and returns metadata for K documents. The server also returns the number of objects n_{pkd} in the encoded document library. The client postprocesses the metadata returned by the server to obtain an integer idx whose value is between 1 and n_{pkd}. Finally, in the third round, the client runs DQuery by feeding the outputs of the second step as inputs to DQuery. The client sends the output of DQuery to the server, who processes it against the document library to return one object from this library.

Simulate$(\mathcal{A}, \pi, \text{Dict}, K, q_b)$

1: // Run query-scoring round of protocol
2: $q_s \leftarrow \pi$.SQuery$(1^\lambda, \text{Dict}, q_b)$
3: $scores \leftarrow \mathcal{A}$.GetScores$(q_s)$
4: $n \leftarrow |scores|$ // n is the number of documents
5: // Obtain indices for the highest scoring documents
6: // if $K > n$, Top-K fills missing values randomly
7: $\{idx_1, \ldots, idx_K\} \leftarrow$ Top-K$(scores)$

8: // Run the metadata-retrieval round of the protocol
9: $q_m \leftarrow \pi$.MQuery$(1^\lambda, n, \{idx_1, \ldots, idx_K\})$
10: $(n_{pkd}, \{\mathcal{M}_{idx_1}, \ldots, \mathcal{M}_{idx_K}\}) \leftarrow \mathcal{A}$.GetMetadata$(q_m)$
11: // Process metadata to get an integer between 1 and n_{pkd}
12: $idx \leftarrow$ SelectDocument$(n_{pkd}, \{\mathcal{M}_{idx_1}, \ldots, \mathcal{M}_{idx_K}\})$

13: // Run the document-retrieval round of the protocol
14: $q_d \leftarrow \pi$.DQuery$(1^\lambda, n_{pkd}, idx)$
15: $obj \leftarrow \mathcal{A}$.GetDocument$(q_d)$

16: return all messages sent to or received from $\mathcal{A}$

Figure 12. Pseudocode for the challenger to simulate the protocol π for one of the queries supplied by the adversary.

A.2 The security game for query privacy

We define a security game that a challenger and an adversary play that captures the notion of query privacy. We denote this game as $\mathcal{G}_{\mathcal{A},\pi,\text{Dict},K}(1^\lambda)$, where $\mathcal{A}$ is an probabilistic polynomial time adversary, π is a protocol for oblivious document ranking and retrieval consisting of the three algorithms of SQuery, MQuery, and DQuery, Dict is a set of keywords, K is an integer greater than or equal to 1 that represents the number of metadata objects a client wants to get, and λ is a security parameter. The game has three phases: setup, simulation, and guess.

During setup, the adversary chooses two multi-keyword queries q_0 and q_1 and sends them to the challenger. These queries may or may not contain keywords in the Dict.

During simulation, the challenger simulates the protocol π for one of the queries. The challenger first flips a coin and picks either q_0 or q_1 depending on the outcome of coin flip ($b \in \{0, 1\}$). It then simulates π for q_b using the Simulate function described in Figure 12. Finally, the challenger shares the output of simulate with the adversary.

During the final guess phase, the adversary takes the output of Simulate corresponding to the scenario which the challenger simulated and outputs b', which is the adversary's guess for whether the challenger simulated the protocol for q_0 or q_1. The adversary wins the game if $b' = b$.

We note that the SIMULATE algorithm calls several functions exposed by the adversary. For instance, it calls the GETSCORES function to learn the relevance scores for q_s. Similarly, it calls the GETMETADATA and GETDOCUMENT functions to retrieve K metadata objects and one object from the document library. The adversary may arbitrarily misbehave when responding to these calls. For instance, when replying to GETSCORES, it may or may not send the actual scores for the scoring query. It may even send back less or more number of scores than n, which is the number of documents in the server library. Similarly, GETMETADATA may return an incorrect value of n_{pkd}, and incorrect number or incorrect content of metadata objects.

A.3 Proof of query privacy

We want to show that the adversary's advantage in winning the game is negligible when π is instantiated with Coeus's protocol. We use a series of hybrid games to calculate the adversary's advantage.

Game 0: This game is the original game as described above with π instantiated with Coeus's protocol. The SQUERY converts q to a binary vector using the dictionary of keywords DICT and then encrypts this binary vector using a secure-matrix vector product primitive (§3.2, §4). The MQUERY algorithm calls the query generation function of a multi-retrieval PIR query (e.g., [12]) with n as the total number of objects in the library and the K indices as the positions of the metadata objects client wants to retrieve. The DQUERY algorithm calls the query generation function of a single-retrieval PIR with n_{pkd} as the number of objects in the library and idx as the position of the object that is retrieved.

Game 1: This game is the same as game 0 except that the DQUERY algorithm calls the query generation function of a single-retrieval PIR with n_{pkd} and an index sampled uniformly at random from the range 1 to n_{pkd}.

Game 2: This game is the same as game 1 except that the MQUERY algorithm calls the query generation function of a multi-retrieval PIR with K indices sampled uniformly at random from the set $\{1, \ldots, n\}$.

Game 3: This game is the same as game 2 except that the SQUERY algorithm calls the request generation algorithm of secure matrix-vector product with a binary vector of length |DICT| whose each element is sampled uniformly at random from $\{0, 1\}$.

Let S_0 be the event that $b = b'$ in Game 0, where q_b is the query chosen by the challenger, and b' is the adversary's guess. Similarly, let S_1 be the event $b = b'$ in Game 1, S_2 be the event $b = b'$ in Game 2, and S_3 be the event that $b = b'$ in Game 3.

Lemma A.1. $Pr[S_3] = 1/2$.

Observe that in game 3 none of the requests to the adversary depend on the query picked by the challenger. Specifically, DQUERY and MQUERY generate PIR queries for uniformly sampled indices, and SQUERY generates a query for a uniformly sampled binary vector. Therefore, an adversary participating in game 3 cannot distinguish between the two scenarios.

Lemma A.2. $|Pr[S_2] - Pr[S_3]| \leq \epsilon_{Secure-matrix-vec}$

The difference between game 3 and game 2 is the input to secure matrix-vector product. Specifically, in game 3, the input is dependent on the actual query selected by the challenger, while in game 2, it is a uniformly sampled Boolean vector. But given the security of secure matrix-vector product (which in turn depends on the semantic security of an underlying encryption scheme), the adversary cannot differentiate the two cases with non-negligible probability.

Lemma A.3. $|Pr[S_1] - Pr[S_2]| \leq \epsilon_{Multi-retrieval-PIR}$

The difference between game 2 and game 1 are the indices input to multi-retrieval CPIR: in game 2, the indices are sampled uniformly at random while in game 1 they are dependent on the scores returned by the query-scoring round. However, given the security of multi-retrieval CPIR that hides the value of these indices, the adversary cannot distinguish between the two games with non-negligible probability.

Lemma A.4. $|Pr[S_0] - Pr[S_1]| \leq \epsilon_{Single-retrieval-PIR}$

The difference between game 1 and game 0 is the index input to single-retrieval PIR: in game 1, the index is sampled uniformly at random while in game 0 it is dependent on the metadata returned by metadata-retrieval round. Again, given the security of single-retrieval CPIR that hides the value of the index, the adversary cannot distinguish between the two games with non-negligible probability.

Combining the four lemmas, we get the proof that $|Pr[S0] - 1/2| <= \epsilon_{Secure-matrix-vec} + \epsilon_{Multi-retrieval-PIR} + \epsilon_{single-retrieval-PIR}$. Therefore, an adversary cannot win the security game with non-negligible probability.

References

[1] gensim – topic modelling in python. https://github.com/RaRe-Technologies/gensim/.

[2] SealPIR: A computational PIR library that achieves low communication costs and high performance. https://github.com/microsoft/SealPIR. Microsoft Research, Redmond, WA.

[3] Wikimedia downloads: enwiki dump progress on 20210201. https://dumps.wikimedia.org/enwiki/20210201/enwiki-20210201-pages-articles-multistream.xml.bz2/.

[4] Wikipedia:short description. https://en.wikipedia.org/wiki/Wikipedia:Short_description#Formatting/.

[5] Wikipedia:wikipedia_records. https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_records#Title_length/.

[6] S. Agrawal, S. Badrinarayanan, P. Mukherjee, and P. Rindal. Game-Set-MATCH: Using mobile devices for seamless external-facing biometric matching. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1351–1370, 2020.

[7] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private Information Retrieval for Everyone. In *Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[8] D. Agun, J. Shao, S. Ji, S. Tessaro, and T. Yang. Privacy and efficiency tradeoffs for multiword top k search with linear additive rank scoring. In *International World Wide Web Conference (WWW)*, pages 1725–1734, 2018.

[9] I. Ahmad, Y. Yang, D. Agrawal, A. El Abbadi, and T. Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 313–329, 2021.

[10] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, November 2018.

[11] A. Amirbekyan and V. Estivill-Castro. A new efficient privacy-preserving scalar product protocol. In *The Australasian Data Mining Conference (AusDM)*, 2007.

[12] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 962–979, 2018.

[13] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–569, 2016.

[14] J. Angwin, C. Savage, J. Larson, H. Moltke, L. Poitras, and J. Risen. AT&T helped US spy on Internet on a vast scale. *The New York Times*, 2015.

[15] E. Balsa, C. Troncoso, and C. Diaz. OB-PWS: Obfuscation-based private web search. In *IEEE Symposium on Security and Privacy (S&P)*, pages 491–505, 2012.

[16] M. Barbaro and T. Zeller Jr. A Face Is Exposed for AOL Searcher No. 4417749. *The New York Times*, Aug. 2006.

[17] B. Barrett. Security news this week: Russia's SolarWinds hack is a historic mess. *Wired*, Dec. 2020. https://www.wired.com/story/russia-solarwinds-hack-roundup/.

[18] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Advances in Cryptology—CRYPTO*, pages 55–73, 2000.

[19] J. Bethencourt, D. Song, and B. Waters. New techniques for private stream searching. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–32, 2009.

[20] C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):1–51, 2014.

[21] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology—CRYPTO*, pages 868–886, 2012.

[22] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[23] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[24] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[25] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998. https://eprint.iacr.org/1998/003.

[26] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, 1995.

[27] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[28] G. Danezis and C. Diaz. Space-efficient private search with applications to rateless codes. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 148–162. Springer, 2007.

[29] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica. DORY: An encrypted search system with distributed trust. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1101–1119, 2020.

[30] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security Symposium (SEC)*, pages 2433–2450, 2020.

[31] X. Ding, H. Pang, and J. Lai. Verifiable and private top-k monitoring. In *ACM ASIA Conference on Computer and Communications Security (CCS)*, pages 553–558, 2013.

[32] J. Domingo-Ferrer, A. Solanas, and J. Castellà-Roca. h(k)-private information retrieval from privacy-uncooperative queryable databases. *Online Information Review*, 2009.

[33] C. Dong and L. Chen. A fast secure dot product protocol with application to privacy preserving association rule mining. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2014.

[34] C. Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*, pages 1–12, 2006.

[35] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[36] R. A. Fink, D. R. Zaret, R. B. Stonehirsch, R. M. Seng, and S. M. Tyson. Streaming, plaintext private information retrieval using regular expressions on arbitrary length search strings. In *IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 107–118. IEEE, 2017.

[37] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference*, pages 303–324. Springer, 2005.

[38] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[39] T. George. Takeaways from the Shopify hack. *SecurityWeek*, Sept. 2020. https://www.securityweek.com/takeaways-shopify-hack.

[40] A. Gersho. Principles of quantization. *IEEE Transactions on circuits and systems*, 25(7):427–436, 1978.

[41] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikäinen. On private scalar product computation for privacy-preserving data mining. In *International Conference on Information Security and Cryptology (ICISC)*. 2004.

[42] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[43] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[44] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 91–107, 2016.

[45] T. Gupta, H. Fingler, L. Alvisi, and M. Walfish. Pretzel: Email encryption and provider-supplied functions are compatible. In *ACM SIGCOMM Conference*, 2017.

[46] S. Halevi and V. Shoup. Algorithms in HElib. In *Advances in Cryptology—CRYPTO*, pages 554–571. Springer, 2014.

[47] S. Halevi and V. Shoup. Faster homomorphic linear transformations in HElib. In *Advances in Cryptology—CRYPTO*, pages 93–120. Springer, 2018.

[48] R. Handa, C. R. Krishna, and N. Aggarwal. Document clustering for efficient and secure information retrieval from cloud. *Concurrency and Computation: Practice and Experience*, 31(15):e5127, 2019.

[49] R. Handa, C. R. Krishna, and N. Aggarwal. Searchable encryption: A survey on privacy-preserving search schemes on encrypted outsourced data. *Concurrency and Computation: Practice and Experience*, 31(17):e5201, 2019.

[50] R. Henry, Y. Huang, and I. Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[51] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Privacy Enhancing Technologies Symposium (PETS)*, 2019(1), 2019.

[52] A. Holmes. 533 million facebook users' phone numbers and personal data have been leaked online. *Business Insider*, Apr. 2021. https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4.

[53] A. Ibrahim, H. Jin, A. A. Yassin, and D. Zou. Secure rank-ordered search of multi-keyword trapdoor over encrypted cloud data. In *2012 IEEE Asia-Pacific Services Computing Conference*, pages 263–270. IEEE, 2012.

[54] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. *Privacy Enhancing Technologies Symposium (PETS)*, 3:246–264, 2021.

[55] Intel. Intel Software Guard Extensions. https://software.intel.com/en-us/sgx.

[56] I. Ioannidis, A. Grama, and M. Atallah. A secure protocol for computing dot-products in clustered and distributed environments. In *International Conference on Parallel Processing (ICPP)*, 2002.

[57] S. Ji, J. Shao, D. Agun, and T. Yang. Privacy-aware ranking with tree ensembles on the cloud. In *International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 315–324, 2018.

[58] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security Symposium (SEC)*, pages 1651–1669, 2018.

[59] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, 1997.

[60] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 711–725, 2018.

[61] Q. Lou, B. Feng, G. C. Fox, and L. Jiang. Glyph: Fast and accurately training deep neural networks on encrypted data. *arXiv preprint arXiv:1911.07101*, 2019.

[62] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–23, 2010.

[63] L. McKenzie. Secure file sharing compromises university security. *Inside Higher Ed*, Apr. 2021. https://www.insidehighered.com/news/2021/04/07/accellion-data-security-breach-latest-hit-universities.

[64] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *IEEE Symposium on Security and Privacy (S&P)*, pages 279–296. IEEE, 2018.

[65] M. Oehler and D. S. Phatak. A conjunction for private stream searching. In *International Conference on Social Computing*, pages 441–447. IEEE, 2013.

[66] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *Privacy Enhancing Technologies Symposium (PETS)*, pages 75–92. Springer, 2010.

[67] C. Örencik and E. Savaş. An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking. *Distributed and Parallel Databases*, 32(1):119–160, 2014.

[68] R. Ostrovsky and W. E. Skeith. Private searching on streaming data. *Journal of Cryptology*, 20(4):397–430, 2007.

[69] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[70] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *LREC Workshop on New Challenges for NLP Frameworks*, pages 45–50, May 2010.

[71] D. Rushe. Yahoo $250,000 daily fine over NSA data refusal was set to double "every week". *The Guardian*, 2014.

[72] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[73] M. D. Schatz, R. A. Van de Geijn, and J. Poulson. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 38(6):C748–C781, 2016.

[74] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[75] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Apr. 2020. Microsoft Research, Redmond, WA.

[76] A. W. Services. Amazon EC2 Instance Savings Plans. https://aws.amazon.com/savingsplans/compute-pricing/.

[77] A. W. Services. Amazon EC2 On-Demand Pricing (Data transfer). https://aws.amazon.com/ec2/pricing/on-demand/.

[78] J. Shao, S. Ji, A. O. Glova, Y. Qiao, T. Yang, and T. Sherwood. Index obfuscation for oblivious document retrieval in a trusted execution environment. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 1345–1354, 2020.

[79] J. Shao, S. Ji, and T. Yang. Privacy-aware document ranking with neural signals. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 305–314, 2019.

[80] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy (S&P)*, pages 44–55, 2000.

[81] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[82] M. Strizhov and I. Ray. Multi-keyword similarity search over encrypted cloud data. In *IFIP international information security conference*, pages 52–65. Springer, 2014.

[83] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 71–82, 2013.

[84] W. Sun, R. Zhang, W. Lou, and Y. T. Hou. REARGUARD: Secure keyword search using trusted hardware. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 801–809. IEEE, 2018.

[85] P. Syverson, R. Dingledine, and N. Mathewson. Tor: The second-generation onion router. In *USENIX Security Symposium (SEC)*, pages 303–320, 2004.

[86] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–440, 2017.

[87] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 24–43. Springer, 2010.

[88] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 253–262. IEEE, 2010.

[89] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 299–313, 2017.

[90] Y. Wang, H. Pang, Y. Yang, and X. Ding. Secure server-aided top-k monitoring. *Information Sciences*, 420:345–363, 2017.

[91] Y. Wang, J. Wang, and X. Chen. Secure searchable encryption: a survey. *Journal of communications and information networks*, 1(4):52–65, 2016.

[92] C. Wei, Q. Gu, S. Ji, W. Chen, Z. Wang, and R. Beyah. OB-WSPES: A uniform evaluation system for obfuscation-based web search privacy. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[93] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel computing*, 27(1-2):3–35, 2001.

[94] Z. Xia, X. Wang, X. Sun, and Q. Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE transactions on parallel and distributed systems*, 27(2):340–352, 2015.

[95] X. Yi and E. Bertino. Private searching for single and conjunctive keywords on streaming data. In *Workshop on Privacy in the Electronic Society (WPES)*, pages 153–158, 2011.

[96] X. Yi, E. Bertino, J. Vaidya, and C. Xing. Private searching on streaming data based on keyword frequency. *IEEE Transactions on Dependable and Secure Computing*, 11(2):155–167, 2013.

[97] X. Yi, R. Paulet, and E. Bertino. Private searching on streaming data. In *Homomorphic Encryption and Applications*, pages 101–126. Springer, 2014.

[98] X. Yi and C. Xing. Private (t, n) threshold searching on streaming data. In *International Conference on Privacy, Security, Risk and Trust and International Conference on Social Computing*, pages 676–683. IEEE, 2012.

[99] J. Yu, P. Lu, Y. Zhu, G. Xue, and M. Li. Toward secure multikeyword top-k retrieval over encrypted cloud data. *IEEE transactions on dependable and secure computing*, 10(4):239–250, 2013.

[100] P. Zhang, Y. Li, Q. Liu, and H. Lin. A scalable distributed private stream search system. In *International Conference on Distributed Computing Systems Workshops*, pages 128–135. IEEE, 2015.

[101] J. Zobel and A. Moffat. Exploring the similarity space. In *ACM SIGIR Forum*, volume 32, pages 18–34, 1998.