Privacy and Efficiency Tradeoffs for Multiword Top K Search with Linear Additive Rank Scoring

Daniel Agun, Jinjin Shao, Shiyu Ji, Stefano Tessaro, Tao Yang
Department of Computer Science
University of California at Santa Barbara

ABSTRACT

This paper proposes a private ranking scheme with linear additive scoring for efficient top K keyword search on modest-sized cloud datasets. This scheme strikes for tradeoffs between privacy and efficiency by proposing single-round client-server collaboration with server-side partial ranking based on blinded feature weights with random masks. Client-side preprocessing includes query decomposition with chunked postings to facilitate earlier range intersection and fast access of server-side key-value stores. Server-side query processing deals with feature vector sparsity through optional feature matching and enables result filtering with query-dependent chunk-wide random masks for queries that yield too many matched documents. This paper provides details on indexing and run-time conjunctive query processing and presents an evaluation that assesses the accuracy, efficiency, and privacy tradeoffs of this scheme through five datasets with various sizes.

ACM Reference Format:
Daniel Agun, Jinjin Shao, Shiyu Ji, Stefano Tessaro, Tao Yang Department of Computer Science University of California at Santa Barbara. 2018. Privacy and Efficiency Tradeoffs for Multiword Top K Search with Linear Additive Rank Scoring. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/https://doi.org/10.1145/3178876.3186084

1 INTRODUCTION AND RELATED WORK

As sensitive information is increasingly stored on the cloud, preserving privacy is a critical factor for users to adopt cloud-based information services including keyword search. A cloud server is often considered as *honest-but-curious*: namely such a server honestly executes protocol specification and hosted programs, but it may observe and infer the private information of a client during execution or by inspecting hosted data. To deal with such a server, searchable encryption [20, 38] can be used to conduct privacy-preserving server-side query matching with single keyword [29, 30], conjunctive multiwords using the OXT protocol [16–18], or disjunctive multiwords [28]. The studies including OXT do not support ranking, and efficient and secure top K ranking is an open research problem because of the challenge in achieving both.

The main challenge to perform server-side privacy-preserving top K ranking is that advanced ranking involves arithmetic computation based on raw features (defined in Section 2) and hiding feature information through encryption prevents the server from performing effective scoring and result comparison. On the other hand, unencrypted feature values can lend themselves to privacy attacks [15, 26]. Homomorphic encryption [22, 33] is one idea offered to secure data while letting the server perform arithmetic calculations without decrypting the underlying data. For example, given feature values f_1 and f_2, the server that uses a homomorphic encryption $E()$ can compute $E(f_1+f_2)$ using $E(f_1)$ and $E(f_2)$ without knowing f_1 and f_2. But such a scheme is still not computationally feasible when many numbers are involved, because each addition or multiplication is extremely slow, not mentioning the ability of comparing two results using scores computed with homomorphic encryption. For example, homomorphic encryption does not allow the efficient comparison of $f_1 + f_2$ and $f_1' + f_2'$ at the server even it can securely compute $E(f_1 + f_2)$ and $E(f_1' + f_2')$. Order-preserving encryption (e.g. [2, 9, 34, 35]) allows a sever to compare two encrypted numbers without knowing the actual numbers but it does not support additions or multiplication of encrypted numbers.

Another challenge is that advanced ranking considers a variety of features (e.g. [1, 8, 12, 41, 42, 49]) and feature vectors are often sparse with many zero values. Space usage can grow explosively if zeros are explicitly stored. Using a compact data structure representation without a proper defense can leak statistic information about index, which may lead to leakage-abuse attacks [15, 26, 36].

The previous work on similarity-based secure ranking [13, 32, 40] converts each TFIDF-based feature vector into two random vectors. Index construction builds forward index after multiplying feature vectors with secret matrices. Online query processing transforms the given query vector with matrix multiplication also, and derives dot similarity of a transformed query vector with all encrypted document vectors, which results in time complexity as $O(T^2 + DT)$ where D is the number of documents and T is the dictionary size. To extend this model to consider proximity features such as word pairs, parameter T becomes very large. Inverted indexing is not feasible because offline and online matrix transformation with randomization yields dense feature and query vectors, and the space cost of the index becomes $O(DT)$. Recent work in [45, 48] follows the above matrix transformation while considering multi-user data ownership and dynamic document update. The datasets tested in [13, 32, 40] are small with only thousands of documents or terms and high search cost with no inverted index support prohibits such an approach from handling a slightly larger dataset.

It appears implausible to develop a completely secure ranking scheme without resorting to heavyweight cryptography such as functional encryption [22, 23, 39]. With a practical restriction towards fast response time, a server-hosted algorithm must deploy a compromised lightweight scheme to compute ranking scores and select results. Our strategy to address this private ranking open

ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/https://doi.org/10.1145/3178876.3186084

problem is to leverage the previous searchable encryption research and strike various tradeoffs in developing an efficient scheme with limited information leakage to a server. We adopt a client-server collaborative approach in which the server conducts efficient matching, additive scoring, and partial ranking while the client does query preprocessing and the final top result selection. Our design only uses one round of client-server communication since multi-round active communication between the server and client (e.g. [25, 31]) incurs a much higher communication cost and response latency.

We assume that a client owns a dataset and places the corresponding index on the cloud to be searchable by the client. This paper does not consider the multi-ownership of documents [48] and it assumes the index on the cloud can be periodically refreshed, but without considering dynamic index update [30, 45]. The work on query scrambling in [4, 5] protects user privacy in issuing queries for private web search while assuming that the server owns searchable data and thus hosts unencrypted index. Our work is complementary with a different data ownership assumption. Our presentation assumes that a query contains at least two words because it is relatively easy to handle single-word queries by storing encrypted pre-ranked document IDs for each word on the server.

The **contribution** of this paper is an indexing and online search scheme with linear additive scoring for this open private top K search problem. It seeks a tradeoff with partial server-side result filtering in handling a modest-sized dataset. It is several orders of magnitude faster than the previous baseline solution [13] while accommodating four categories of previously-developed ranking signals. Due to the page limit, this paper lists formal properties without presenting a detailed proof.

2 PROBLEM DEFINITION AND DESIGN CONSIDERATIONS

Problem Definition: Given D document feature vectors with T features that a client owns, each document d has many feature values denoted as f_i^d and the client builds a searchable index and places it to the server. We focus on developing an indexing and top K search scheme so that the server can access encrypted matched document features for a query and compute their rank without knowing the underlying feature values within a reasonable response time for a modest-sized dataset. The server also should not learn the meaningful information when features are not involved in search.

We assume each search query contains a conjunction of keywords and adopt the posting intersection of query words based on a searchable encryption algorithm called OXT [16, 17]. Faster traditional intersection algorithms that traverse two or more postings [19] simultaneously are not adopted because such a traversal leaks more information to the server. During the search process, a standard technique to ensure privacy is to use a deterministic pseudo random function (called PRF) to hide information including term IDs and document IDs.

Ranking formula. We opt for a simple but popular rank score computing scheme which is a linear combination of document features. While such scoring [47] delivers decent relevancy performance, multiple additive trees (e.g. [44]) can achieve better relevance. However, making such a nonlinear ranking method private involves not only score addition and but also value comparison. There is no known encryption method available that can solve both issues within a single framework. A linear combination formula computes ranking score in a form of $\sum \alpha_i f_i^d$ where α_i is a coefficient of feature value f_i^d for document d. We assume all coefficients are static and can be embedded into feature values during index setup. Thus for the rest of the paper we will ignore these coefficients, and the linear additive rank formula is simplified as $\sum f_i^d$.

Raw ranking features. We call a rank feature as *raw* if it is explicitly stored in the index and a rank feature as *composite* if it is computed based on other raw features. Our design is to make each basic feature in the above additive formula as a raw feature and the server retrieves and simply adds them without knowing the role of these values. This minimizes the chance that the server understands their semantic contributions to ranking signals.

We use the above additive formula to support four categories of ranking features used in the information retrieval literature: 1) Term-frequency based composite features such TFIDF and BM25 [27]. They can be represented as the summation of weighted raw word frequency and thus the above additive scheme supports such features. 2) Proximity composite features based on the sum of raw proximity term features. Such a proximity term can be a n-gram within a certain distance [8] or a word pair [21, 49]. A traditional inverted index with positional information stores word positions explicitly [7] and online ranking computes composite proximity features based on word positions in each document. While this scheme is space efficient, computing composite proximity features requires both order comparison and arithmetic calculation from word positions. As discussed in Section 1, there is a lack of encryption techniques to support both secure calculation and order comparison. Leaking relative word positions of each document may enable statistical attacks which reveal document content structure. Thus we opt to use a proximity formula expressed as the summation of raw features that directly model proximity terms. As a result, some of the previous proximity formulas are not be supported, for example, minimum aggregation [42] and span coverage [41]. 3) Document query specific features such as document-query click through rate. 4) Document specific features such as freshness and document quality. Both Category 3) and 4) are raw features and can fit naturally in the additive rank formula.

Handling sparsity of raw ranking features. Raw features in Category 2), 3) and 4) often have many zero values. If all encrypted zero values are stored explicitly, it would greatly simplify the privacy preserving design, but the space cost would be explosively high and such a scheme would become impractical. One option of handling feature sparsity is to compactly store nonzero feature values for each document as a forward index and once documents that match a query are identified during query processing, encrypted document features can be retrieved using a hashed document ID. Another option is to embed feature values in the posting entries of the traditional inverted index [7]. These options without a proper defense have a privacy risk that the server can inspect document feature vectors or postings directly and gather document statistical information such as word frequency distribution without client authorization and launch leakage-abuse attacks [15, 36].

Our design for optional features which are often sparse is to use an online key-value store. Namely each matched document d

involves the following two types of weights. 1) Required individual feature weight f_i^d for document d. When a feature is required, every matched document has a feature value stored in the index. 2) Optional feature weights O_t^d where $1 \le t \le m$. When a feature is optional, the default value is zero when this feature value of a document is not available from the index.

Feature encryption with mask blinding. To preserve privacy of feature values, we blind each feature value using a random mask with modular addition to hide this value from the server. This mask is generated in a deterministic way using a PRF, and is known to the client only. Formally, we store each feature value f in the index as $[f + R]$ which is defined as $f + R \bmod N$ where R is the feature mask computed as a PRF of the term ID and the document ID in the range from 0 to $N - 1$. N is set to 2^{32} for our implementation.

The square bracket notation in $[f + R]$ also emphasizes that the server sees the computed value of expression $f + R \bmod N$ but the server is not able to derive individual f or R value. As we explain below, the above scheme will allow the server to compute the rank score by adding masked feature values and to conduct partial comparison if choosing masks judiciously with a tradeoff. We did not adopt a homomorphic encryption scheme (e.g. [33]) for feature blinding because it does not bring visible advantage while being less efficient and cannot support partial server-side ranking.

More specifically, for document d, weight of feature f_i^d in index is an integer and is stored as $[f_i^d + R_i^d]$ with a random noise mask R_i^d and optional weight O_t^d is stored as $[O_t^d + RO_t^d]$ with a random noise mask RO_t^d. Given a query with q required features and m optional weights, the total rank score F plus the total score mask for document d including these masks under modulo N is:

$$[F + M] = [\sum_{i=1}^{q}[f_i^d + R_i^d] + \sum_{1 \le t \le m, O_t \in X} [O_t^d + RO_t^d]]$$

where $M = \sum_{1 \le t \le m, O_t \in X} O_t^d + \sum_{1 \le t \le m, O_t \in X} RO_t^d$ and $O_t \in X$ means that the corresponding optional feature can be found in the key-value store of index.

While the server can compute the above encrypted sum, as the feature masks are random and independent from one document to another, the server is not able to compare the relative rank order among matched documents. When the number of matched documents is modest, the server can send these results to the client along with a bitmap of optional features used for each document, which assists the client to remove the sum of masks in each rank score. When there is a large number of matched documents in the server or client-server bandwidth is low, we want the server to conduct partial ranking so that results with low scores can be filtered out first before sending back to the client. That essentially becomes a two-stage ranking as a type of cascade ranking [43]. In next section, we explore this possibility with several tradeoffs necessary to balance privacy and query response time efficiency.

3 SERVER-SIDE PARTIAL RANKING

Server-side partial ranking with uniform random masks. To allow the server to compare rank scores of matched documents, one option is to choose the same random masks for all documents of the same feature. With this relaxation, we change mask symbol R_i^d as R_i, and RO_i^d as RO_i in rank score formula of Section 2 so that the total mask sum is the same for all matched documents under the same set of matching optional terms and then the server can order these documents.

To support the above strategy, we need to address two issues. 1) As each random feature mask is selected uniformly from 0 to $N - 1$, the masked feature weights and their summation may wrap around under modulo N, which can mislead server-side rank ordering. To let the server handle this wraparound without knowing actual rank scores or additional client-server interaction, we impose a constraint that the final rank score without feature masks is to be less than $\frac{N}{2}$. Given $q+m$ required and optional features, each feature weight w should satisfy $\max_{q,m}(q + m)w < \frac{N}{2}$, namely its upper bound is $\frac{N}{\max_{q,m}(2q+2m)}$. For example, when $q+m \le 32$ and $N = 2^{32}$, the upper bound of each feature weight is 2^{26}. When $q + m \le 1024$, the upper bound is 2^{21}. Our evaluation shows that ranking with 21-bit integer features is sufficient for the tested datasets. We will discuss the possible value of $q + m$ at the end of this section.

THEOREM 3.1. *Given $q + m$ required and optional features, each feature weight w satisfies* $\max_{q,m}(q + m)w < \frac{N}{2}$. *Let F_1 and F_2 be the rank scores of two documents under the same sum mask M where $0 \le M < N$.*

- *If $|[F_1 + M] - [F_2 + M]| < \frac{N}{2}$, then $[F_1 + M] \le [F_2 + M]$ if and only if $F_1 \le F_2$.*
- *If $|[F_1 + M] - [F_2 + M]| > \frac{N}{2}$, then $[F_1 + M] \le [F_2 + M]$ if and only if $F_1 \ge F_2$.*

The above theorem shows that when the server obtains two masked rank scores $[F_1 + M]$ and $[F_2 + M]$ with a possibility of wraparound, it can determine which score F_1 or F_2 is bigger without knowing actual values of F_1 and F_2. That is done by checking if the difference of $[F_1 + M]$ and $[F_2 + M]$ is within $\frac{N}{2}$ or not. Notice the difference cannot be equal to $\frac{N}{2}$.

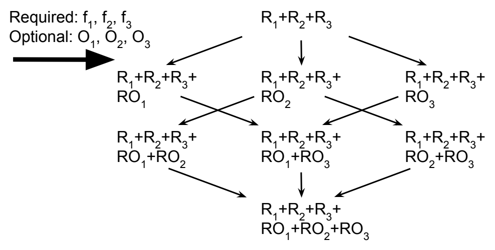


Figure 1: The lattice relationship for optional feature matching cases when $q = 3$ and $m = 3$.

2) Under the same query, different documents may be matched with a different set of optional features. There are 2^m optional feature matching cases representing different combinations of matching these m optional terms. Note that the score masks are different across cases, so the server cannot directly compare between optional cases. Figure 1 illustrates the lattice relationship of score masks among optional feature matching cases for a query with 3 required terms and 3 optional terms. Each edge represents the subsuming relationship and namely a parent case subsumes terms used in the child case. All documents that correspond to the same

optional feature matching case are comparable because their score masks have the same value. Thus the server can sort and select the top K results matching the same case. We use the following property of the lattice to remove unnecessary results across cases.

Theorem 3.2. *Let U and V be two vertices in the lattice of optional feature matching cases and V is a parent node of U (V points to U in Figure 1). Let $topK(U)$ be the top K documents matched under Case U. For any document d matched in U, and $d \notin topK(U)$, any document in V ranked below d or equally under Case V can be removed safely from V.*

After the above lattice-based suppression, the server sends the top results from different matching cases to the client where further score deblinding and final top K result selection will take place.

Query-dependent deblinding using chunk-wide runtime random masks. The above uniform masking strategy allows server-side partial ranking, but leads to the following leakage of information to the server.

- Given two documents d and d' from the same feature mask R, the server can learn their relative weight difference $[f^d+R]-[f^{d'}+R]$ to get $f^d - f^{d'}$.
- The server may also use $\frac{[f^d+R]-[f^s+R]}{[f^{d'}+R]-[f^s+R]}$ to approximate $\frac{f^d}{f^{d'}}$ where f^s is the smallest feature value. Such an approximation can be reasonable if f^s happens to be small (e.g. 0).

To restrict the above leakage to a smaller scope, we propose a dynamic chunk-wide random masking strategy based on runtime query choices. Each feature has a *posting* list of document IDs containing nonzero feature values. Index construction divides this list into chunks and the random mask for each feature is the sum of a document specific mask and a chunk-specific mask. For certain queries when server-side partial ranking is triggered, runtime query processing removes document-specific masks in all involved features. In this case, matched documents within the same chunk share the same mask for the same feature and then the server is able to order documents matched under the same chunk.

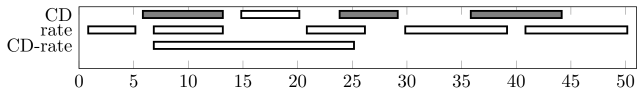
A client triggers partial ranking on the server side only when the matched results can potentially exceed a threshold. To detect the above condition, we call a feature *popular* when the total number of documents for this feature with nonzero values is greater than a popularity threshold. Otherwise call this feature *unpopular*. When one of required features in a query is unpopular, the number of matched results cannot exceed the above threshold.

Query decomposition. Following the above design and given a sorted document list of a feature, we decompose this list as a set of chunks with non-overlapping document ID ranges. Conceptually we view each feature ID as a term and each chunk as a subterm of the original term. Figure 2(a) gives an example of feature chunking and query decomposition with query "CD rate". The optional word-pair feature added is "CD-rate". The posting of "CD" is decomposed into 4 chunks corresponding to 4 subterms a_1, a_2, a_3, and a_4 and the posting of "rate" is decomposed into 5 chunks corresponding to b_1, b_2, b_3, b_4, and b_5. Similarly, the document list of optional feature "CD rate" is decomposed with only one chunk c_1 based on document ID range partitioning.

Knowing the ID ranges of posting chunks, a client can perform an earlier range intersection and decompose the original query as

CD	6 7 8 10 13	15 16 18 19 20	24 25 27 28 29	36 40 41 44	
rate	1 2 3 4 5	7 9 10 12 13	21 22 23 25 26	30 31 33 36 39	41 46 48 49 50
CD-rate	7 25				

(a)Chunking examples



(b) Client-side range intersection for query decomposition

Figure 2: Chunking and query decomposition

a set of **subqueries**. Any document that matches any subquery satisfies the original query. For Figure 2(b), the client can learn that only documents in ID ranges (6,13), (24,26), and (36,44) are possible to contain all required keywords, and equivalently, original query "CD rate" is converted into a disjunction of the 4 subqueries: a_1b_2 with optional c_1, a_3b_3 with optional c_1, a_4b_4, and a_4b_5.

Note that the proposed client-side query decomposition brings the following two extra benefits in addition to server-side partial ranking. 1) As we discuss in the next section, feature access is implemented as a hash table lookup. Query decomposition helps exploit locality-aware data partitioning, and thus reduces the memory footprint size during intersection and feature lookup when loading the relevant part of hash tables. We also have considered to avoid over exploitation of locality in data partitioning as there is a privacy tradeoff for blocked data access operations [18]. 2) An intersection algorithm selects one required feature to start enumerating possible document candidates and typically it is the one with the smallest posting length [19]. This is called s-term in OXT [17] in which the server does learn the length of the s-term posting for each query during intersection. As the server accumulates such information after processing many queries, that may open a door to leakage-abuse attacks, and one counter measure is to pad the posting by adding bogus IDs, which disguises true counts [15]. In our setting, as a query is decomposed into a sequence of subqueries, a client can select a start feature (essentially s-term) differently from one subquery to another, which greatly reduces the chance of leaking the posting length of single words.

Tradeoff by limiting the number of optional features. As shown in next section, the server uses a bitmap record to memorize which optional features are matched for each document. Having a large number of optional features causes two disadvantages: 1) space overhead to maintain a long bitmap. 2) There are less matched results comparable at the server side based on the above lattice discussion. Both factors lead to more server-client communication overhead in sending back partially-ranked results.

When word pairs are used as raw text proximity features, to reduce the total number of word pairs, we combine word pair weights appearing in different sections of a document. Following the previous work [8], indexing can impose a constraint that a word pair is included in the index only when the word distance of such a pair is within a limit. Our evaluation has used limit 9 in processing three test datasets. At runtime, given q words in a query, there are $\binom{q}{2}$ word pairs, and if we let $m = \binom{q}{2}$, there are $2^{\binom{q}{2}}$ lattice cases of using these word-pair features. For example, when $q = 5$, there are already 2^{10} cases. When $q = 7$, the lattice size explodes to 2^{21}. We impose a constraint that a client only demands optional word pairs that are within distance limit L in the query. For example with a 5-word query $w_1, \cdots, w_5$, and $L = 2$, only distance pairs (w_1, w_2), (w_1, w_3), (w_2, w_3), (w_2, w_4), (w_3, w_4), (w_3, w_5), and (w_4, w_5) are taken as optional features. In general, the number of optional features m to consider based on word pairs is reduced from $\binom{q}{2}$ to $\binom{L}{2} + L(q-L)$ if $q \geq L$, otherwise $\binom{q}{2}$. When we choose $L = 2$, then $m = 2q - 3$. As shown in Section 5 the relevance with $L = 2$ is acceptable in the tested datasets. In practice, we can assume $q < 10$ because typically trimming a query longer than such a length does not affect the relevance. Thus $m \leq 15$ when optional features are only based on word pairs. In some applications, additional optional terms may be needed to improve ranking relevance such as freshness or authoritativeness of a document, thus m is typically under 24 and we use 3 bytes for the optional bit map in our evaluation.

4 INDEXING AND QUERY PROCESSING

In this section, we adopt the OXT searchable encryption [16, 17] for document matching and extend it for ranking. We present an indexing and query processing scheme to support feature blinding with dynamic chunk-wide random masking, and enable a client to safely trigger server-side partial ranking for selected queries involving popular features. This design prevents the server from learning anything formation about the features and posting of an encrypted word if such a word has never been searched.

Data structure and high level search flow. Our search flow involves 3 key-value stores: 1) R-store saves meta information in feature posting chunks such as document ID range of chunks. That facilitates query decomposition at the client side. 2) S-store contains required feature values and is used by the search algorithm to identify the candidate documents. 3) X-store contains feature values accessible using a pair of document ID and feature ID.

Given a query, a client converts it into a sequence of subqueries using R-store. For each subquery, the client generates n required and optional feature IDs as $w_1, w_2, \cdots, w_n$ for the server to match documents and fetch their features. Let d be a candidate document ID that appears in the posting list of start feature w_1. To fetch another feature w_i of d where $2 \leq i \leq n$, the online algorithm pairs two IDs w_i and d together as the key to access X-store. To preserve privacy, all IDs and intermediate values are hashed or encrypted using operators summarized in Table 1 so that for any feature name and document ID, the server cannot access the posting list or feature values from the hosted X-store and R-store without authorization from the client. The server should not learn the identity of the query terms used during online search.

Encrypted inverted index setup. The input data set with feature vectors is converted into an inverted index format. The indexer controlled by a client builds the encrypted index and lets the server host the S-store and X-store. For each feature w of document d with value f, this indexer constructs a key-value pair for the X-store as follows. Define p as the position count of d in the posting and c is the chunk ID where $c = \lfloor p/csize \rfloor$ and $csize$ is the chunk size.

- The key is $xtag = g^{PRF(k_5,w)PRF(k_2,d)}$ where k_5 and k_2 are secret hashing keys. Base g is a Diffie-Hellman constant for pseudo-random mapping [10].
- The value accessible through above key is: $S(xtag) = f + R_c + R_x \mod N$ where $R_x = PRF(k_u, g^{PRF(k_1,w)PRF(k_2,d)})$ and $R_c = PRF(k_0, w||c)$ represent the chunk specific and document specific masks, respectively.

$\|\|$	Bitwise concatenation of two values.
$PRF(k, a)$	A deterministic pseudo-random function of "a" using key k (e.g. SHA256 of $a\|\|k$). Among PRF keys used, $k_0, \ldots, k_9$ are secret known by the client only. k_u is public to generate feature weight mask R_x.
$E(k, a)$	Non-deterministic symmetric encryption of plaintext a using key k (e.g. AES using random initialization vector)
$D(k, b)$	Symmetric decryption of b using key k s.t. $D(k, E(k, a)) = a$ for any plaintext a and key k
$S(stag)$	Look up S-store with key $stag$, and return a chunk of encrypted feature posting.
$X(xtag)$	Look up X-store with key $xtag$, and return the encrypted feature value if exists.
$(x_i)_{i=1}^n$	A list of elements x_i from $i = 1$ to n.

Table 1: Function and operator symbols

When the above feature is required, the d and f values are also saved in the posting chunk c of document w in the S-store. The corresponding S-store key is $stag = PRF(k_7, w||c)$. The corresponding value is a chunk list of posting entries and each posting entry is an encrypted tuple (e, y, f_s) defined as follows.

(1) $e = E(k_e, d)$ which represents the encrypted document ID d. The encryption for e is semantically secure so that the server cannot learn anything except the length of the original document ID.
(2) $y = PRF(k_4, w||p)^{-1}PRF(k_2, d)$ is a blinded bridging number used when w is selected as a start feature for deriving the key to access the X-store during intersection proposed in [17]. We extend its usage for feature fetching and will elaborate it later in the example below.
(3) Blinded feature $f_s = u_d + f + R_c + R_s \mod N$ where u_d is the sum of document-specific Category 4 feature weights discussed in Section 2. R_s is the document-specific mask computed as $PRF(k_3, w||p)$.

Online search procedure. Figure 3 shows the three phases of query processing. **Phase 1**, which is conducted at client side, forms required and optional features for a given query and performs an earlier range intersection to derive subqueries after R-store lookup. Assume each subquery has n features with n corresponding chunk IDs: $(w_i, c_i)_{i=1}^n$. All required features are placed before optional features in this feature sequence and w_1 is selected as the start required feature. Then the key to access S-store is $stag = PRF(k_7, w_1||c_1)$ and the key used in the server to decrypt an entry of posting $S(stag)$ is $skey = PRF(k_6, w_1||c_1)$. In addition, the client builds a special 2-dimensional token array for the server and each array element $tokens[i][j]$ is defined as: $xtoken_i = g^{PRF(k_5,w_i)PRF(k_4,w_1||p)}$, and $mtoken_i = g^{PRF(k_1,w_i)PRF(k_4,w_1||p)}$ where $2 \leq i \leq n$, $1 \leq j \leq csize$, $csize$ is the chunk size, and posting position $p = c_1 * csize + j$. The corresponding document specific mask for the start feature is $R_s = PRF(k_3, w_1||p)$. If the predicted result length does not exceed a threshold, the client will not turn on server-side result filtering and in that case, the second entry $mtoken_i$ of each token array element and R_s information are voided, Finally the client sends $stag$ and $skey$ along with the token array and related mask R_s if needed to the server for the above subquery.

At **Phase 2**, after receiving the control information for each subquery, the server fetches the posting of w_1 from S-store based on function call $S(stag)$. For every posting entry (assume j-th entry) for w_1 and i-th other feature, it uses the received token array $tokens[i][j]$ to compute $xtag$ to access X-store and also the document specific mask R_x when needed. The server accumulates the rank score with mask subtraction when needed (e.g. Line 21 of Figure 3). When partial ranking is turned on, document scores under each optional feature lattice case are compared and filtered, and at most top K results are returned for each case. For each matched document sent back from the server to the client, the result tuple is in a format of (e, F, O, j), representing an encrypted document ID, an encrypted score, a bitmap on the optional features used, and the position of this document appeared in the posting chunk of w_1.

Phase 3 client post-processing removes score masks and compares all received documents to select the final top K results.

Example. Figure 4 gives a runtime processing example for query "CD rate" with optional term "CD-rate". After query decomposition and subqueries are generated, assume "CD" is the start feature of this subquery. The key to access S-store is $stag = PRF(k, \text{"}CD\text{"}\|c)$ for chunk c of this feature's posting. After accessing S-store, assume d is a candidate document found from the posting of "CD" with corresponding tuple (e, y, f_s). To further access the X-store for features "rate", the server computes the access key as $xtag = xtoken_2^y$ and $R_x = PRF(k_u, mtoken_2{}^y)$. Since $y = PRF(k_4, \text{"}CD\text{"}\|p)^{-1}PRF(k_2, d)$ based on the index setup algorithm, we can verify that the server actually obtains the same values $xtag = g^{PRF(k_5,\text{"}rate\text{"})PRF(k_2,d)}$ and $R_x = PRF(k_u, g^{PRF(k_1,\text{"}rate\text{"})PRF(k_2,d)})$, which match the index setup algorithm in building X-store. It means that without knowing values k_1, k_2, and k_5, the server can fetch and add the feature weight of word "rate" successfully after the query-specific authorization information such as the token array is issued by the client.

Search complexity. The index space cost after encryption is proportional to the summation of all nonzero optional feature values plus the number of required feature IDs multiplied by the number of documents. The encrypted numbers cannot be compressed well because of randomization. For a query with n features and let $\sum(Posting(w_1))$ denote the sum of the posting length of the start feature w_1 for each subquery. The time cost for this query is $O((n-1)\sum|Posting(w_1)|)$. It is slower than that of [19] which does not need to consider privacy-preserving and this represents a tradeoff for private search.

Properties of query processing and leakage discussion. We present a more formal analysis of the worst-case leakage profile in Appendix A. Two privacy properties of our scheme following that are listed below.

Theorem 4.1. *If a feature ID has never been used in any search query, the server cannot learn the corresponding feature weight of any document.*

Theorem 4.2. *The server cannot learn the feature weight of a document for any unpopular word during or after query processing. That is true also for any popular word of a search query which involves at least one unpopular required word.*

The above theorems show that if a feature such as a word that has never appeared in any past search query, the server is not able to inspect its feature value and learn any leakage of information on this feature used in any document. During the query processing,

```
Phase 1: Client-side query preprocessing
1:  Build query terms, decompose query, and enable server-side partial
    ranking if needed.
2:  for each subquery do
3:      Send S-store access key $stag$, decryption key $skey$, $tokens$ array, and
        related start feature mask $R_s$ if needed to the server.
4:  end for
Phase 2: Server-side subquery processing with scoring
5:  Let $stag, skey, R_s, tokens$ be information received for one subquery.
6:  for $j$-th entry $Z$ in posting chunk returned by $S(stag)$ where $1 \le j \le csize$ do
7:      Decrypted tuple $(e, y, f_s) = D(skey, Z)$
8:      Initialize Score $F = f_s$. Initialize a bitmap $O$.
9:      if partial ranking is enabled then
10:         $F \leftarrow f_s - R_s \mod N$
11:     end if
12:     for $i$ = 2, 3, ..., and n, $(xtoken_i, mtoken_i)$ in $tokens[j][i]$ do
13:         $xtag_i \leftarrow xtoken_i{}^y$
14:         if $X(xtag_i)$ does not exist and $w_i$ is required then
15:             Skip this document for conjunctive requirement
16:         end if
17:         if $X(xtag_i)$ exists then
18:             Add $i$ to optional feature bitmap $O$.
19:             if partial ranking is enabled then
20:                 Let feature mask $R_x \leftarrow PRF(k_u, mtoken_i{}^y)$
21:                 $F = F + X(xtag_i) - R_x \mod N$
22:             else
23:                 $F = F + X(xtag_i) \mod N$
24:             end if
25:         end if
26:     end for
27:     Add the tuple $(e, F, O, j)$ to the results list.
28: end for
29: if partial ranking is enabled then
30:     Select top $K$ results for set of optional bitmap $O$ and suppress
        unnecessary results.
31: end if
32: Send matched documents as list of $(e, F, O, j)$ to client.
33: Repeat the above steps for all subqueries.
Phase 3: Client-side post processing
34: for each $(e, F, O, j)$ of the result list from the server do
35:     Client decrypts document IDs and subtracts masks from their rank
        scores.
36: end for
37: Sort the results list and select top-K results.
```

Figure 3: Top K search

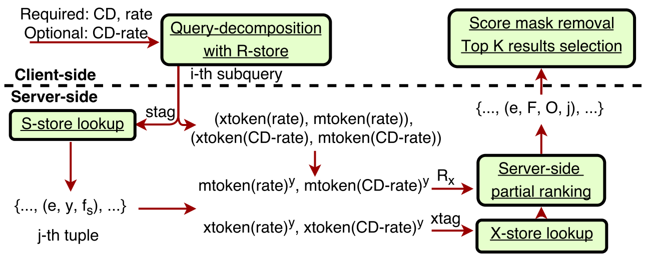


Figure 4: Example of client-server query processing

if a query that does not trigger partial server ranking (e.g. involve at least one unpopular word), the server cannot learn information on features or the final score of each document. In our evaluation with chunk-wide masks is 10,000, the percentage of popular words for CSIRO, TREC45, Aquaint, Clueweb and Enron are respectively 0.07%, 0.31%, 0.47%, 0.07%, and 0.1%. When sever-side result filtering is triggered for queries that involve only these popular words, there is some leakage of rank order and chunk-wide feature difference as discussed in Section 3 which will be further discussed in Appendix A. To restrict such leakage to a smaller scope, one strategy is make the relative ratio of chunk size over the posting length threshold as small as possible. In our tested datasets, this ratio is set to 2.1% with chunk size 210, since this is only for words that appear in over 10,000 documents.

5 EVALUATION

Implementation and datasets. We have implemented a prototype system with C++. Experiments are conducted on Linux Ubuntu 16.04 servers with 8 cores of 2.4 GHz AMD FX8320, 16GB memory. The code is compiled with optimization flag -O3. The following datasets are used: the TREC Disk 4&5 dataset with TREC Robust 2004 topics 301-450 &601-700, the CSIRO dataset with the TREC 2008 Enterprise Track Topics (CE051-CE127) queries, and the Aquaint dataset with TREC Robust 2005 query set of 50 topics. Table 2 summarizes their key characteristics after stemming and index generation. Row 2 is the number of documents. Row 3 is the number of term and document ID pairs. Row 4 is the number of composite term and document ID pairs. An optional term composed of two words appears in the X-store with a document if these two words appear in a document within distance 9. Note the majority of the encrypted index size comes from these optional terms. If lower ranking relevance is acceptable the distance limit could be reduced to significantly reduce the size because X-store size is linearly proportional to the total number of document-feature pairs. Row 5, 6 and 7 are the size of R-store, S-store and X-store in bytes. Row 8 is the total index size in bytes. As the machines we run experiments are shared in a cloud environment, we control the search memory usage to under 2GB per machine. Due to memory constraints, we opt to use 1 server for CSIRO while distributing data to 8 and 5 servers respectively for Aquaint and TREC45. Intersection and score accumulation is conducted in parallel on those servers. At the end of this section, we also present our investigation in using a larger dataset from ClueWeb and the Enron email dataset.

Dataset	CSIRO	TREC45	Aquaint
#Doc	0.37M	0.53M	1.03M
word-doc	22M	109M	216M
wordpair-doc	146M	712M	1,357M
R-Store	0.31GB	1.25GB	1.13GB
S-Store	1.12GB	5.56GB	11.02GB
X-Store	2.42GB	11.82GB	22.53GB
Total Size	3.85GB	18.63GB	34.68GB

Table 2: Size characteristics of datasets

For each document, we select features that perform well for text ranking based on the past work [8, 12, 41, 42, 49]. We generate the feature value of a word that this document contains based on the BM25 coefficient differentiated by where they appear (title and body). We also compute optional features based on word pairs that appear within a limit in a document. The feature value for the word pairs is based on the squared inverse of word distance following the work in [49] and if such a pair appears in the title or body of its document. We use AdaRank [47] to guide feature weighting and relevance evaluation follows 3-fold cross-validation. Notice that with the conjunctive query requirement, our document matching retrieves may miss some labeled relevant results in some queries compared to a disjunctive assumption, which results in up-to 10% relevance difference in some cases.

Each document result returned from the server on average uses about 16 bytes and this includes an 8-byte encrypted document ID, 4-byte blinded score, 3-byte bitmap, and 1-byte position. We select 10,000 as the result size threshold that triggers server-side partial ranking. Notice that the average Internet connection globally has reached an average speed of 7.2Mbps in 2017 [3]. With this average connection speed, it takes 0.18 seconds to transmit 10,000 matched results with 16 bytes per result.

Multiword search time. Table 3 lists the average search response time and its cost breakdown when the number of search words varies. In this setting, X-store has 1024 partitions. A query response time includes the client-side time and the server side time for S-store lookup of candidate documents and X-store lookup for intersection and score accumulation. The client time reported includes R-store intersection and the token generation for the first subquery, and the server starts to search as soon as it receives the first subquery. There is no cost involved in X-store operations for $q = 1$ and its time is listed so we can observe the average time difference between single-word and multiword queries. When q increases from 2 to 3, the X-store time increases as more optional features are involved. But as there are less candidates found from S-store when q is 4 and 5 after earlier range intersection, the X-store time drops in all three query sets. The client-side time is modest, and it increases in all three datasets with more query words because there is more cost for range intersection and x-token generation. Time to compute a x-token is not insignificant because of integer exponentiation involved.

# Query words q		1	2	3	4-5
CSIRO	Client	30.53	59.98	74.89	101.16
	S-store	58.31	121.57	140.40	37.60
	X-store	0	59.21	137.87	64.09
	Total(ms)	89.62	283.86	427.98	248.51
TREC45	Client	18.89	45.18	65.56	107.22
	S-Store	146.79	191.30	222.33	119.67
	X-Store	0	85.56	284.15	260.11
	Total(ms)	166.42	405.64	717.34	693.00
Aquaint	Client	25.87	47.38	54.17	66.35
	S-store	261.81	147.60	146.67	90.60
	X-store	0	89.47	218.52	200.95
	Total(ms)	289.35	337.06	496.20	400.26

Table 3: Query processing cost of three datasets

Table 4 lists the average search response time for 5 query posting length buckets. Each query falls into one column bucket based on the sum of the start feature posting length of its subqueries. This table validates that the query response time is approximately proportional to the posting length sum of the selected start features of all subqueries as discussed in Section 4.

A comparison with the baseline. We have also implemented and evaluated the baseline approach based on matrix transformation in [13, 45] with the same machine resource setting. It takes

# Posting length (thousands)	0 - 2	2 - 4	4 - 6	6 - 8	8 - 10
CSIRO (ms)	186.00	392.34	499.50	801.40	973.65
TREC45 (ms)	217.81	453.23	558.27	838.78	1049.26
Aquaint (ms)	134.47	343.56	479.21	770.15	909.97

Table 4: Impact of posting length on search time

over 48, 26, and 46 hours to process a query on average for CSIRO, TREC45, and Aquaint with 1, 5, and 8 machines, respectively. The forward index space cost is 105, 285, and 815 terabytes respectively as encrypted feature vectors are dense. The high cost is caused by the large number of features (T), which is about 39M, 73M, and 108M respectively in these datasets. When restricting the offline word-pair distance to be within 3 instead of 10, T is reduced approximately by half on average and the above cost is also shortened by half. Still our approach is several orders of magnitude faster. An optimization using tree search to speed up similarity computing as $O(T \log D)$ is discussed in [45], but the worst case is still $O(TD)$. Recall that D is the number of documents.

Impact of query decomposition. Query decomposition with chunked postings enables earlier range intersection and S-store and X-store locality-aware partitioning. For range earlier intersection during query decomposition, we find that the intersection scope is reduced by 6%, 12%, and 21% for CSIRO, TREC45, and Aquaint, respectively. Without partitioning X-store, compared to a 1024-partition, search is about 42.5x, 17x, and 19.1x slower for queries with 2 words, 3 words, and 4-5 words, respectively. Thus the above decomposition is important to sustain low query response times.

#Results returned		TREC Queries	Synthetic
CSIRO	No filter	11,607	33,093
	Chunk 105	2,504	8,884
	Chunk 210	1,288	6,159
TREC45	No filter	20,985	127,538
	Chunk 105	14,151	28,876
	Chunk 210	8,708	20,596
Aquaint	No filter	32,896	185,139
	Chunk 105	25,561	38,333
	Chunk 210	16,112	22,437

Table 5: Return result reduction in top-10 search with threshold 10,000.

Effectiveness of server partial ranking. The average number of results returned from the sever for all 154 TREC queries without top K filtering in CSIRO, TREC45, and Aquaint is 849, 5490, and 5704 respectively. Table 5 shows the average number of results for top K search with $K = 10$. The threshold to trigger partial ranking is 10,000 as discussed above. Column 3 is the average number of results for the 23 TREC queries that trigger partial ranking with posting chunk size 105 and 201 respectively. Rows marked as "No filter" are for the setting that search does not use server-side partial ranking. For Aquaint, there are 13 TREC queries triggered result filtering and 16,112 results are returned on average with chunk size 210 after filtering out 51% of all matched results. For a larger chunk size, there are more comparable matched documents under each subquery and there are more results filtered during partial ranking.

To evaluate the extreme situations, Column 4 is for handling 15 synthetic queries per dataset built based on only popular and stop words with a much larger number of return results. The partial ranking scheme is able to filter out more unnecessary return results also. For Aquaint, the server returns 22,437 results on average for synthetic queries after filtering out 87.9% of all matched results. In this case with a 7.2Mbps average global Internet connection speed [3], result communication takes about 0.39 seconds.

Relevancy impact with linear additive scoring and the number of optional terms. Table 6 illustrates the NDCG@10 score of the linear model and LambdaMART [44] under different query word distance constraints $L = 2, 5, \infty$. The distance restriction does not negatively degrade performance. Compared to LambdaMART, the linear scoring model delivers decent performance.

NDCG@10		L=2	L=5	L=∞
CSIRO	LambdaMART	0.4836	0.4842	0.4789
	Linear	0.4311	0.4046	0.4317
TREC45	LambdaMART	0.3823	0.3898	0.3866
	Linear	0.4121	0.4012	0.3808
Aquaint	LambdaMART	0.3256	0.3246	0.3131
	Linear	0.3118	0.3227	0.317

Table 6: Relevance with different numbers of word-pairs

ClueWeb. We have also investigated how our scheme handles a larger dataset using ClueWeb09 Category B dataset [6] with 50 million web documents. The relevancy raw features include BM25-weighted single word weights and word pairs in the title and body sections, and document PageRank scores. The top team in the 2011 Web TREC has accomplished NDCG@20 of 0.24282 in [11] using the TREC 2011 Queries plus 450 out of the 684 queries from Million Query (MQ) TREC 2009 [14]. LambdaMART using our raw features obtains NDCG@20 of 0.2547, 0.2554, and 0.2601 with query word distance restriction $L = 2$, $L = 5$, and $L = \infty$, respectively. The additive ranking scheme we use obtains 0.2512, 0.2530, and 0.2585 under these L values.

The search time for ClueWeb data is still reasonable as the client-side query processing does not require large memory and the server side can use more machines to distribute S-store and X-store for parallel processing. The client-server communication cost is a key concern and thus we select several different subsets of this ClueWeb dataset to assess the impact of database size change on server-side partial ranking. For all 684 queries from MQ TREC 2009 with judgment labels, Table 7 lists the average return result size from Row 3 to Row 4 with two chunk sizes when the database size chosen varies from 3M to 50M ClubWeb documents. Row 6 and Row 7 show the average return result size for queries for top 10% of the largest result sizes after server-side filtering. When the database size is 5M and chunk size is 210, it would take about 0.46 seconds on average for the server to send 25,394 results back to the client with today's global average Internet speed 7.2 Mbps. The slowest 10% of queries can take 1.93 seconds with 107,195 results. Notice the top country-level average Internet connection is 28.6 Mbps [3] and with such a speed, the slowest 10% would take 0.486 seconds to deliver.

Enron email dataset. We have also studied this corporate email collection [37] with about 0.5M messages from about 150 users. The average posting length of words in this dataset is 65 with 0.1% of them as popular words. Like other datasets we have studied, the posting length follows a long-tail Zipf distribution. We use 100 personal names from the email collection as test queries, and find all queries only contain unpopular words. Namely all of them

# Results returned		3M	5M	10M	30M	50M
All	No Filter	65,449	81,445	113,173	218,027	321,769
	Chunk 105	25,273	31,866	46,442	90,138	134,240
	Chunk 210	19,992	25,394	37,413	73,017	108,892
Top 10%	No Filter	393,650	506,452	752,328	1,619,203	2,465,084
	Chunk 105	97,710	147,156	234,872	508,292	812,515
	Chunk 210	69,056	107,376	173,195	374,309	608,736

Table 7: Return result sizes in top-10 search with threshold 10,000 for ClueWeb subsets varying from 3M to 50M

have results less than 10,000 and the server only computes rank scores without filtering. If we lower the threshold to 1000, then the result filtering ratio after server-side partial ranking is over 50% on average. Because each email message is short, searching this dataset is much faster than TREC45 with a similar size.

6 CONCLUDING REMARKS

The contribution of this work is a novel private top K ranking scheme with various tradeoffs to efficiently search cloud hosted data. A privacy analysis on limited leakage is provided. Our experiment shows that the response time is reasonable, and partial server ranking can effectively reduce the return result size for the tested datasets. The developed techniques address this open problem for a modest sized dataset. For a significantly larger dataset, the current techniques require fast client-server connection since the result size returned from the server even after partial ranking can be significant when all query words are popular. Deployment of faster Internet connection such as mobile 5G networks could extend the applicability of the proposed approach together with additional improvements. Another potential strategy to support a larger dataset is to adopt a trusted proxy with a fast server-proxy communication connection, if available. Then final supplemental ranking could be completed at such a proxy and the ranked results can be sent to the client with a smaller communication cost. Recently there is an advancement in neural network based ranking (e.g. [24, 46]) and our techniques can be applicable for document pre-ranking used in such work or directly integrating neural features.

Acknowledgments. We thank Olaoluwa Osuntokun for initial code development, and the anonymous referees for their thorough comments. This work is supported in part by NSF IIS-1528041. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

A PRIVACY ANALYSIS

The previous work [17, 28] analyzes the correctness of the derived leakage in a searchable encryption system by showing the information that any adversary can acquire during the execution of the real search protocol can be well simulated by an interactive experiment between this adversary and a simulation algorithm which only knows the leakage of the secure search system. This appendix analyzes the privacy of our ranking scheme following such semantic security with leakage.

Notion: Let Π be a searchable symmetric encryption scheme [17, 28], which consists of two algorithms: Algorithm "EDBSetup" distributes the keys to the client and server and encrypts inverted index; Algorithm "Search" executes the search protocol between the client and server. Let $\mathcal{L}(\mathsf{DB}, \mathsf{q})$ be a leakage profile function, which takes the plaintext inverted index DB and a query sequence q as input, and outputs the leakage information of the whole system. For a security parameter λ (roughly related to the number of bits in the key), for two efficient algorithms A (for Adversary) and S (for Simulator), define experiments $\mathbf{Real}^{\Pi}_{A}(\lambda)$ and $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$ as:

- $\mathbf{Real}^{\Pi}_{A}(\lambda)$: $A(1^{\lambda})$ chooses the plaintext inverted index DB and a list of queries q. The experiment runs $(K, \mathsf{EDB}) \leftarrow \mathsf{EDBSetup}(\mathsf{DB})$, where K is the set of secret keys known only to the client, and EDB is the encrypted index. For each query in q, it secretly runs the algorithm Search with client input K and server input EDB and stores the transcript (i.e., all the communications between the client and server). Then the experiment gives EDB and the transcript to A, which returns and outputs a Boolean value in $\{0, 1\}$ answering whether the transcript is produced by the real search protocol, which happens here, or the simulator.
- $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$: $A(1^{\lambda})$ chooses DB and a list of queries q. The experiment secretly runs $S(\mathcal{L}(\mathsf{DB}, \mathsf{q}))$ and gives its output (a synthetic transcript) to A, which returns and outputs a Boolean value in $\{0, 1\}$ answering whether this output is produced by the real search protocol or the simulator S.

Definition: Π is $\mathcal{L}$-semantically-secure against non-adaptive attacks if for any efficient adversary A there exists an algorithm S such that the probability difference $\Pr[\mathbf{Real}^{\Pi}_{A}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{A,S}(\lambda) = 1]$ is negligible, i.e., asymptotically less than λ^{-c} for any integer c.

Following the above definition, our objective is to make the attacker A unable to tell apart the communications of the real secure protocol and the communications between the client and the simulator S. This notion is only for non-adaptive attacks, since in each experiment the attacker can only choose the queries at the beginning, and can never change the chosen queries during the interaction with the server. This assumption fits most scenarios since often a third-party attacker has little control over the queries that users issue. The only thing the attacker can know is the distribution of the queries. With the above definition, we can prove the following sketched theorem.

Theorem (sketched): With standard cryptographic assumptions, the search scheme in Section 4 is $\mathcal{L}$-semantically-secure against non-adaptive attackers with the leakage profile $\mathcal{L}$ (i.e., the information leaked to the attacker) given as follows.

- Size patterns: 1) The chunk size for partitioned posting lists. 2) The number of documents matching each subquery. 3) The length of the posting list for the s-term in each subquery.
- Rank and feature patterns: 1) Rank order of encrypted document IDs for answering a query. 2) The difference between two document weights of the same feature within the same chunk, if the corresponding word has been searched.
- Intersection patterns: 1) The overlapping pattern of s-tags and x-tokens sent during search. For any subquery, given any chunk in the posting of the s-term, the server knows which chunks in the posting of any non-s-term in the subquery have non-empty intersections with this chunk. 2) The identification of any two subqueries sharing the same s-term. 3) The intersection of the posting lists of two s-terms from two subqueries that the server knows they share at least one non-s-term.

REFERENCES

[1] E. Agichtein, E. Brill, S. Dumais, and R. Ragno. Learning user interaction models for predicting web search result preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 3–10, New York, NY, USA, 2006. ACM.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *SIGMOD 2004*, pages 563–574, 2004.

[3] Akamai.com. Akamai's State of the Internet. Q1 2017 Executive Summary, 2017.

[4] A. Arampatzis, G. Drosatos, and P. S. Efraimidis. Versatile query scrambling for private web search. *Information Retrieval Journal*, 18(4):331–358, 2015.

[5] A. Arampatzis, P. Efraimidis, and G. Drosatos. Enhancing deniability against query-logs. In *European Conference on Information Retrieval*, pages 117–128. Springer, 2011.

[6] L. T. I. at Carnegie Mellon University. The clueweb09 dataset, http://boston.lti.cs.cmu.edu/data/clueweb09.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval (2nd Edition)*. Addison Wesley, 2011.

[8] J. Bai, Y. Chang, H. Cui, Z. Zheng, G. Sun, and X. Li. Investigation of partial query proximity in web search. WWW '08, pages 1183–1184, 2008.

[9] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. EUROCRYPT '09, pages 224–241, 2009.

[10] D. Boneh. The decision diffie-hellman problem. *Algorithmic number theory*, pages 48–63, 1998.

[11] L. Boytsov and A. Belova. Evaluating learning-to-rank methods in the web track adhoc task. In *TREC*, 2011.

[12] S. Büttcher, C. L. A. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. SIGIR '06, pages 621–622. ACM.

[13] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):222–233, 2014.

[14] B. Carterette, V. Pavlu, H. Fang, and E. Kanoulas. Million query track 2009 overview. In *TREC*, 2009.

[15] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS'15*, pages 668–679. ACM, 2015.

[16] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

[17] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*, pages 353–373, 2013.

[18] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT 2014*, pages 351–368, 2014.

[19] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1:1–1:25, Dec. 2010.

[20] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. CCS '06, pages 79–88. ACM, 2006.

[21] T. Elsayed, N. Asadi, L. Wang, J. J. Lin, and D. Metzler. UMD and USC/ISI: TREC 2010 web track experiments with ivory. In *Proceedings of The Nineteenth Text REtrieval Conference, TREC 2010, Gaithersburg, Maryland, USA*, 2010.

[22] C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, 2009.

[23] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[24] J. Guo, Y. Fan, Q. Ai, and W. B. Croft. A deep relevance matching model for ad-hoc retrieval. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 55–64, New York, NY, USA, 2016. ACM.

[25] H. Hu, J. Xu, C. Ren, and B. Choi. Processing private queries over untrusted data cloud through privacy homomorphism. In *ICDE*, pages 601–612, 2011.

[26] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

[27] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, Nov. 2000.

[28] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 94–124. Springer, 2017.

[29] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, pages 258–274, 2013.

[30] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS'12*, pages 965–976, 2012.

[31] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. SP '14, pages 639–654. IEEE Computer Society, 2014.

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

[33] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT '99*, pages 223–238, 1999.

[34] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. SP '13, pages 463–477. IEEE Computer Society, 2013.

[35] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. SOSP '11, pages 85–100. ACM, 2011.

[36] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS'16*, pages 1341–1352. ACM, 2016.

[37] C. Project. The enron email dataset, https://www.cs.cmu.edu/ enron/.

[38] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. SP '00. IEEE Computer Society, 2000.

[39] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS 2012*, 2012.

[40] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *IEEE Trans. Parallel Distrib. Syst.*, 25(11):3025–3035, 2014.

[41] K. M. Svore, P. H. Kanani, and N. Khan. How good is a span of terms?: exploiting proximity to improve web retrieval. SIGIR '10, pages 154–161. ACM, 2010.

[42] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. SIGIR '07, pages 295–302. ACM, 2007.

[43] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 105–114, New York, NY, USA, 2011. ACM.

[44] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, June 2010.

[45] Z. Xia, X. Wang, X. Sun, and Q. Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):340–352, 2016.

[46] C. Xiong, Z. Dai, J. Callan, Z. Liu, and R. Power. End-to-End Neural Ad-hoc Ranking with Kernel Pooling. In *Proceedings of the 40th International ACM SIGIR Conference on Research & Development in Information Retrieval.* ACM, 2017.

[47] J. Xu and H. Li. Adarank: a boosting algorithm for information retrieval. SIGIR '07, pages 391–398. ACM, 2007.

[48] W. Zhang, Y. Lin, S. Xiao, J. Wu, and S. Zhou. Privacy preserving ranked multi-keyword search for multiple data owners in cloud computing. *IEEE Trans. Computers*, 65(5):1566–1577, 2016.

[49] J. Zhao and J. X. Huang. An enhanced context-sensitive proximity model for probabilistic information retrieval. In *SIGIR*, pages 1131–1134, 2014.