Privacy-Preserving Ranked Search on Public-Key Encrypted Data

Sahin Buyrukbilen
The Graduate Center
City University of New York
Email: sbuyrukbilen@gc.cuny.edu

Spiridon Bakiras
John Jay College
City University of New York
Email: sbakiras@jjay.cuny.edu

***Abstract*—The massive production of digital data and the complexity of the underlying data management, motivate individuals and enterprises to outsource their computational needs to the cloud. While popular cloud computing platforms provide flexible and inexpensive solutions, they do so with minimal support for data security and privacy. As a result, owners of sensitive information may be skeptical in purchasing such services, given the risks associated with the unauthorized access to their data. To this end, *searchable encryption* is a family of cryptographic protocols that facilitate private keyword searches directly on encrypted data. These protocols allow users to upload encrypted versions of their documents to the cloud, while retaining the ability to query the database with traditional plaintext keyword queries. In this paper, we focus on public-key encrypted data and introduce the *first* method that supports ranked results from multi-keyword searches. Our solution employs a simple indexing structure, and leverages homomorphic encryption and private information retrieval (PIR) protocols to process queries in a privacy-preserving manner. Using measurements from Amazon's Elastic Compute Cloud, we show that our method provides reasonable response times with low communication cost.**

I. Introduction

Cloud computing is the new trend in IT that offers users great flexibility in purchasing off-site, third-party resources (ranging from software to infrastructure) at competitive prices. Popular cloud computing services, such as Amazon's Elastic Compute Cloud (EC2)[1], provide on-demand computing, network, and storage resources in an attractive "pay-as-you-go" pricing model [1]. The flexibility of provisioning services on-demand and the virtually endless amount of resources, enable entrepreneurs to deploy their business instantly, without purchasing expensive hardware/software or hiring technically skilled system administrators [2]. Similarly, cloud storage has become ubiquitous and numerous providers, such as Google Drive, SkyDrive, and iCloud, offer multi-GB storage space at no cost. Consequently, users are motivated to move their personal data to the cloud, which gives them the ability to access them anytime from anywhere.

Despite the aforementioned advantages, most cloud computing platforms do not provide adequate security and privacy for the outsourced data. As a result, owners of sensitive information such as emails, personal health records, financial transactions, etc., may be skeptical in purchasing such services, given the risks associated with the unauthorized access to their data. To mitigate these security risks, sensitive information should be encrypted prior to being transferred to the cloud provider. Nevertheless, outsourcing encrypted data would not be practical if the service provider is unable to process queries, as this would necessitate the transfer and decryption of all records at the client site. Therefore, designing privacy-preserving query mechanisms for encrypted data is of paramount importance.

To this end, *searchable encryption* is a family of cryptographic protocols that facilitate private keyword searches directly on encrypted data. These protocols allow users to upload encrypted versions of their documents to the cloud, while retaining the ability to query the database with traditional plaintext keyword queries. Fully functional searchable encryption can be achieved with oblivious RAMs [3], since they can simulate any data structure in a private manner. Even though oblivious RAMs can hide everything from the server (including the access pattern), they incur a very high computational cost. Therefore, the majority of the research work on searchable encryption has focused on efficiency improvements, by weakening the underlying security definitions. In particular, most searchable encryption schemes aim at hiding all but the *access* and *search* patterns. Access pattern is defined as the documents retrieved during a search, and search pattern refers to the possibility of inferring whether two queries were performed for the same keyword.

In the symmetric key setting, Searchable Symmetric Encryption (SSE) [4], [5] is a well-studied problem and there exist numerous techniques that support different types of keyword searches. The recent work by Cao et al. [6] offers the most practical SSE scheme to date, as it implements *ranked* keyword searches. Specifically, the client can issue an arbitrary multi-keyword query (similar to web search engines) and the sever will return the top-k most relevant documents in the database. The main advantage of symmetric encryption is its computational efficiency that allows the server to perform linear searches on the encrypted documents at low cost. Nevertheless, symmetric encryption has an inherent key management problem, so all SSE methods assume that the data owner is the only entity that may upload encrypted data/indexes to the server.

On the other hand, public-key cryptography allows any user to update an encrypted database (i.e., add a new item) without knowledge of the private key. A real-world scenario that leverages this functionality is given by Boneh et al. in [7]. Alice uses an email gateway for her communications and, as she considers her emails sensitive, she asks her friends to

[1]`http://aws.amazon.com/ec2/`

send their emails encrypted with her public key. A searchable encryption scheme would then enable Alice to retrieve all emails containing a specific keyword (e.g., "from:Bob"). Public-key Encryption with Keyword Search (PEKS) [7] is the most representative solution in this field that works by scanning all keywords from every document in the database. However, PEKS and all of its variants [8], [9] are very restrictive in the types of queries that they allow and, most importantly, none of them implements ranked keyword search.

In this paper, we introduce the *first* method that provides ranked results from multi-keyword searches on public-key encrypted data. Since public-key cryptography is computationally expensive, we incorporate the following two design principles in our algorithms: (i) avoid a linear scan of the documents and (ii) parallelize the computations as much as possible. The first principle clearly necessitates the use of an indexing structure. To this end, we encrypt the keyword information for each document in a Bloom filter [10], and hierarchically aggregate (using homomorphic encryption) the individual indexes into a tree structure. Query processing is performed at the client side, and entails the traversing of the tree in a *best-first* manner. To hide the content of the query from the server, the client utilizes an efficient private information retrieval (PIR) protocol [11] to extract the necessary Bloom filter entries from the tree nodes.

To speed-up the search process, we leverage the abundance of computational resources that are available in most cloud computing platforms. In particular, we split the indexing structure into multiple chunks, and utilize several CPUs in parallel in order to execute the PIR queries efficiently. Using measurements from Amazon's Elastic Compute Cloud, we show that our method provides reasonable response times with low communication cost.

The remainder of the paper is organized as follows. Section II discusses previous work on searchable encryption, and Section III introduces the cryptographic primitives utilized in our method. Section IV presents the details of our privacy-preserving search mechanism, while Section V describes a few optimizations to improve the query processing times. Section VI illustrates the results of our experimental evaluation, and Section VII concludes our work.

II. Related Work

The work of Song et al. [5] is the first Searchable Symmetric Encryption (SSE) scheme proposed in the literature. Their method does not employ indexes, but instead encrypts each document in a way that allows keyword search. Searching takes linear time in the combined size of all documents, as the server has to test the trapdoor against every encrypted block in a document. Although this model is proven to be a secure encryption scheme, it is not a secure searchable scheme, because of a vulnerability against statistical attacks. In particular, every keyword search reveals the exact position(s) in the document where there is a positive match.

Goh [12] introduces secure indexes that are based on Bloom filters and pseudo-random functions as hash functions. Specifically, each document has its own index that is constructed as follows. Every word in the document is first hashed with the master key and the output is hashed again with the document *id*, in order to differentiate a word's hash values in different files. The produced output is called a *codeword* and is inserted into the Bloom filter. Finally, the Bloom filter is blinded by inserting a random uniform distribution of 1's. To determine whether a certain keyword exists in a file, the querier computes and sends the corresponding codeword to the server which, in turn, checks whether the bits associated with the codeword are all set (with a probability of false positives that is inherent in Bloom filters).

Chang and Mitzenmacher [13], on the other hand, use a predefined dictionary of all possible words in the database and, for each document, they build a binary array (the index) that identifies the keywords appearing in the document. Next, the user masks the bits in the index with the output of a pseudo-random function. To search for a keyword, the user sends short seeds for the pseudo-random functions to the server, so as to help recover the necessary parts of the index.

Curtmola et al. [4] maintain, for each keyword, an inverted index (stored as a linked list) comprising of document identifiers. Every node in the list stores information about the position and the decryption key of the next node. Then, the nodes from all inverted indexes are encrypted with random keys and are randomly inserted into an array. With this construction, given the position and decryption key of the first node of an inverted index, it is possible to find all documents which include the corresponding keyword.

To support ranked keyword searches, Wang et al. [14] build an inverted index for every keyword in the dataset. For security, the actual scores are encrypted with a modified Order-Preserving Symmetric Encryption (OPSE) scheme [15], where the numeric ordering of the plaintexts is preserved in the ciphertexts. During query processing, the user sends a trapdoor that allows the server to decrypt the corresponding entry in the inverted index. Then, the server compares the encrypted scores and returns to the client the top-k results. Cao et al. [6] address multi-keyword ranked queries by leveraging inner product similarity as the scoring function. Similar to the work of Chang and Mitzenmacher [13], they use a predefined dictionary of all possible words in the database, and construct an encrypted binary array for each document. To compute the similarity scores from the encrypted indexes, they employ the secure kNN computation method of [16].

In the public-key setting, Boneh et al. [7] first introduced a solution called Public Key Encryption with Keyword Search (PEKS). In their approach, the sender selects a set of keywords related to the message and, for each keyword, he produces its corresponding PEKS encryption. The encrypted message and keywords are subsequently sent to the server for storage. When the owner wants to search for messages containing a specific keyword, he creates a trapdoor for that keyword and sends it to the server. The server tests the trapdoor with each PEKS encrypted keyword and, if the test returns true, the message is returned to the owner.

Baek et al. [8] address several limitations of the PEKS framework by (i) preventing the server from reusing the trapdoors, (ii) eliminating the need for a secure authenticated channel between the owner and the server, and (iii) adding multi-keyword search capabilities. Although this work allows queries with multiple keywords, it does not provide ranked results. Similarly, further studies to improve PEKS [17], [18],

[19] do not include multi-keyword search or ranked result features. Golle et al. [9] address conjunctive keyword searches on public-key encrypted data. However, as noted by the authors, their solution reveals the keyword fields that are searched by the client. Moreover, it does not support ranked results.

Finally, Boneh et al. [20] propose a searchable encryption scheme that provides perfect privacy. They use encrypted Bloom filters to store keyword membership information, and leverage the homomorphic encryption scheme of Boneh, Goh, and Nissim [21] to allow the senders to modify the index in a secure manner. Nevertheless, their solution is computationally expensive, as it hides everything from the server (including the access pattern).

III. Preliminaries

In this section we give a brief description of the cryptographic primitives incorporated in our methods. Section III-A discusses homomorphic encryption and Section III-B introduces private information retrieval. Section III-C describes the underlying threat model and security of our protocol.

A. Homomorphic Encryption

Homomorphic encryption allows certain algebraic operations on two plaintexts to be carried out on their corresponding ciphertexts, without any intermediate decryptions. In particular, *fully* homomorphic encryption [22] enables both addition and multiplication operations on the ciphertext space and, therefore, such cryptosystems could be used to build searchable encryption schemes with perfect privacy. However, research on fully homomorphic encryption is still in its infancy and all existing methods are extremely expensive.

In this work, we utilize Paillier's cryptosystem [23], which is an efficient *additively* homomorphic encryption scheme. Specifically, given the Paillier encryptions $E(m_1)$ and $E(m_2)$ of two plaintext messages m_1 and m_2, we can compute the encryption of $E(m_1+m_2)$ by multiplying the two ciphertexts:

$$E(m_1 + m_2) = E(m_1)E(m_2)$$

Furthermore, any message m can be multiplied with a plaintext constant c as follows:

$$E(cm) = E(m)^c$$

The Paillier cryptosystem is semantically secure, i.e., it is infeasible to derive any information about a plaintext, given its ciphertext and the public key that was used to encrypt it. Its security is based on the decisional composite residuosity assumption. The cryptosystem works as follows.

Key generation. Choose two large primes p and q of equal length, and compute the RSA modulus $n = pq$. For security, each prime should be at least 512 bits in length. The public key is n and the private key is $\varphi(n) = (p-1)(q-1)$.

Encryption. To encrypt a message $m \in \mathbb{Z}_n$, choose a uniformly random integer $r \in \mathbb{Z}_n^*$, and compute the ciphertext $c \in \mathbb{Z}_{n^2}^*$ as $c = (n+1)^m r^n \bmod n^2 = (mn+1)r^n \bmod n^2$.

Decryption. Given a ciphertext c, compute the plaintext

$$m = \frac{(c^{\varphi(n)} \bmod n^2) - 1}{n} \cdot \varphi(n)^{-1} \bmod n$$

where $\varphi(n)^{-1}$ is the multiplicative inverse of $\varphi(n) \bmod n$.

B. Private Information Retrieval

Private information retrieval (PIR) was first introduced by Chor et al. [24], and is formally defined as follows. The server holds a database with N records and the client wants to retrieve the i-th record, without the server knowing the value of index i. *Information theoretic* PIR [24], [25], [26] is secure against computationally unbounded adversaries, but it is not practical as it requires that the database be replicated into multiple *non-colluding* servers. On the other hand, *computational* PIR protocols [11], [27], [28] work with a single server, and employ well known cryptographic primitives that are secure against computationally bounded adversaries.

In our methods, we leverage the computational PIR protocol of Gentry and Ramzan [11], because (i) it has very low communication cost, and (ii) it allows the retrieval of multiple records with a single query. The security of the protocol is based on the φ-hiding assumption, and its operation can be summarized as follows.

Setup. During a setup phase, the server associates each record j with a prime power $\pi_j = p_j^{c_j}$, where p_j is a small prime. Assuming that each record is ℓ bits in size, c_j is the smallest integer such that $\log \pi_j > \ell$. All the above values are public knowledge. Before participating in query processing, the server computes a value β, which is the unique solution to the congruences $\beta \equiv D_j \pmod{\pi_j}$, for all $j \in \{1, 2, \ldots, N\}$, where D_j is the binary representation of record j. This is a straightforward application of the Chinese Remainder Theorem (CRT). Note that all client queries are processed on the transformed database β.

Query generation. As noted by Groth et al. [29], Gentry and Ramzan's scheme can be used to retrieve multiple records with a single query. Let $i_1, i_2, \ldots, i_k$ be the indexes of the records that the client wants to retrieve. Initially, the client computes $\pi = \prod_{j=1}^{k} \pi_{i_j}$. He then chooses two large primes p and q, such that $p = 2\pi r + 1$ and $q = 2st + 1$, where r, s, and t are large random integers. After setting $m = pq$, the client selects a random element $g \in \mathbb{Z}_m^*$ with order πv, where $\gcd(\pi, v) = 1$. Finally, he sends (g, m) to the server. For security, we want m to be at least 1024 bits in size, and $\log m > 4 \log \pi$.

Query processing. The server simply computes $c = g^\beta \bmod m$ and returns the result to the client. Note that β is at least equal to the size of the original database, so the computational complexity at the server is linear in N.

Result extraction. To reconstruct the k records, the client computes, for each i_j, $c_{i_j} = c^{\pi v / \pi_{i_j}} \bmod m$. This value should be equal to $g_{i_j} = (g^{\pi v / \pi_{i_j}})^{D_{i_j}} \bmod m$, and thus, the client can retrieve record D_{i_j} using the Pohlig-Hellman algorithm for discrete logarithms [30].

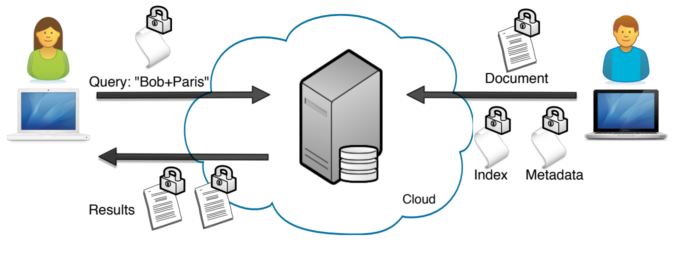


Fig. 1. System architecture

C. Threat Model and Security

We assume that the adversary is the cloud provider (i.e., the server[2]) and its goal is to derive any non-trivial information regarding (i) the plaintext keywords included in an encrypted document and (ii) the plaintext keywords from a client query. We also assume that the adversary runs in polynomial time and is "curious but not malicious," i.e., it will follow the protocol correctly, but will try to gain any advantage by analyzing the information exchanged during the protocol execution.

Similar to most searchable encryption schemes in the literature, we want to hide everything but the access and search pattern. In other words, the server will see the documents that are retrieved during a search, and also the index nodes that were accessed as part of that search. However, the following information will not be leaked:

- The keywords associated with an encrypted document (including their number).
- The keywords contained in a query (including their number).
- The ranking scores of the result set (including their actual order).

IV. Ranked Keyword Search

In this section we present in detail our privacy-preserving search mechanism. Section IV-A describes the system architecture, while Section IV-B presents our indexing scheme and discusses the handling of data updates. Section IV-C introduces our top-k query processing algorithm and Section IV-D discusses the security of our approach.

A. System Architecture

Alice is subscribed to a cloud computing service that allows her friends to send her documents, such as emails, in encrypted form, in order to enforce data confidentiality. When Bob wants to send Alice a new document (Figure 1), he first creates a set of keywords that are stored in a metadata file. For example, if the document is an email, the keywords could be the sender's name, the keywords appearing in the subject line, the most frequent keywords from the email's body, etc. Every keyword is associated with an integer value (score) that indicates the importance of that keyword in the document. The metadata file may also include some additional information about the document, such as a file name, format, size, etc. Bob also creates an appropriate index structure that encodes membership and score information for all the keywords. Finally, Bob uses Alice's public key to encrypt each file separately (document, index, metadata) and transmits all the encrypted files to the cloud provider[3]. If Alice wants to view the emails from Bob's trip to Paris, she can create a query such as "Bob Paris" in order to retrieve the top-k most relevant documents from the database. In particular, the query will trigger a two-party protocol between Alice and the cloud provider, which will allow Alice to download the corresponding encrypted documents.

TABLE I. Summary of symbols

Symbol	Description
N	Database size (number of documents)
M	Size of Bloom filter vector
b	Block size of Bloom filter vector
d	Bit size of Bloom filter counter
f	Node fan-out
e_i	Node i
D_i	Document i
W_i	Keyword set for document i
$\|W\|$	Max number of keywords per document
$\|K\|$	Number of keywords in database
n	RSA modulus for Paillier encryption
m	RSA modulus for PIR scheme

The document index consists of a *counting* Bloom filter [31] with a single hash function H (which is public knowledge). The counting Bloom filter is essentially a vector of counters (where each counter is initialized to zero) that is used to probabilistically encode set membership information. As shown in Figure 2, for every keyword $w \in W_i$ the sender adds w's score to the counter at position $H(w)$. (Table I summarizes the most frequently used symbols in the paper.) Note that, in this work, we assume that every document D_i is allowed to define at most $|W|$ number of keywords. To reduce the number of ciphertexts required to store the Bloom filter, the sender creates groups of b counters that are encrypted together as a single binary value (using the additively homomorphic Paillier cryptosystem). In the example of Figure 2, $M = 9$ and $b = 3$, so the index is stored in $M/b = 3$ Paillier ciphertexts.

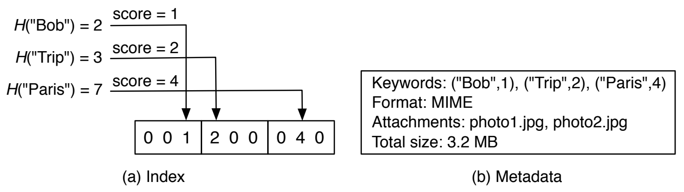


Fig. 2. Document index and metadata

The reason for using both an index and a metadata file is twofold. First, documents can be quite large (e.g., emails with multimedia attachments) and the client may want to see a brief description of the document before downloading it locally. Second, Bloom filters are probabilistic structures and, thus, introduce several false positives in the result set. For instance, any document with a keyword that hashes to position 7 (Figure 2) is a potential match for query "Paris." On the other hand, if we have the client download the (compact) metadata files first, we can guarantee the correctness of the top-k result set.

[2]Henceforth, we use terms *server* and *cloud provider* interchangeably.

[3]For better performance, the document and metadata files can be encrypted with a hybrid cipher, as in the PGP protocol.

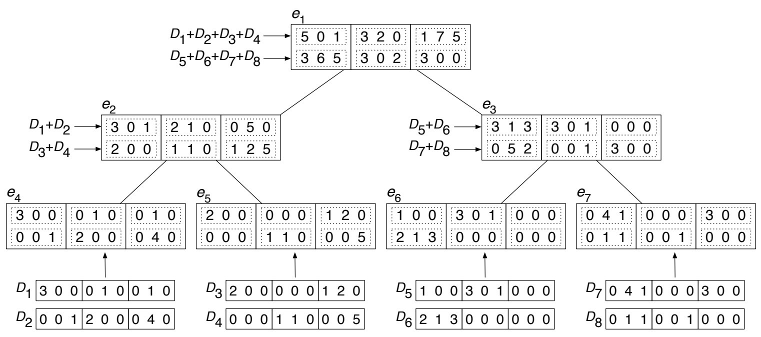


Fig. 3. Database index

B. *Index Construction and Update*

The cloud provider maintains a single index for all the client's documents. It is constructed by hierarchically aggregating the encrypted Bloom filters into a tree structure, as illustrated in Figure 3. Specifically, every *internal* node in the tree stores f Bloom filters (where f is the node fan-out), each being the aggregation of all document indexes in the subtree of the corresponding child. On the other hand, *leaf* nodes store the individual indexes from f distinct documents.

To understand why each node can store multiple Bloom filters, observe that the nodes in Figure 3 are identical to document indexes, i.e., they consist of exactly M/b Paillier encryptions. However, the Paillier cryptosystem allows for the encryption of very large integers that are similar in size to the RSA modulus n (typically a 1024-bit number). Consequently, unlike individual document indexes, each Paillier encryption in the tree structure will utilize most of the available storage space, in order to store f groups of counters. To compute an appropriate value for f, given the group size b, we need to derive a lower bound for the counter size (in bits), such that the overflow probability is negligible. Assuming a database with $|K|$ distinct keywords, the probability that at least x different keywords hash in the same counter is [31]:

$$\Pr(count \geq x) \leq M \left(\frac{e|K|}{xM}\right)^x$$

As an example, if $|K| = 20000$ and $M = 5000$, the probability that more than 30 keywords hash in the same location is less than 3×10^{-10}. If we can also estimate the distribution of the keywords in the document collection, we can determine a suitable bit size d for the individual counters.

Nevertheless, even if we end up underestimating the value of d, there are several ways for the client to detect the resulting overflows. For instance, since the counters are aggregated in a bottom-up fashion, the client can recognize such overflows during query processing (Section IV-C). Alternatively, the client can periodically download the root node locally, and identify counters that experience large drops in their values. If such overflows are detected, the client can reconstruct the index tree from scratch, using the existing document metadata. Note that, in our experiments, we used $d = 15$ bits per counter and never experienced any overflows at the root node. Based on this value, and for a 1024-bit modulus n and groups of size $b = 4$, a single Paillier encryption can store $f = 17$ groups of counters.

Prior to indexing any documents, the cloud provider creates an empty version of the tree structure, i.e., every node on the tree will comprise of Paillier encryptions of 0. Specifically, we assume there is a limit (set by the cloud provider) on the number of encrypted documents N that the client can store, which allows the server to initialize all nodes according to the predetermined fan-out f. However, since all nodes are initially identical, the server can simply construct one version and then create multiple copies, as needed. This is not a security concern, as the encryptions are re-randomized during every aggregation operation.

When the cloud provider receives a new document index from the sender (i.e., Bob), it selects an empty location at the leaf level of the database index where it can be stored (this is trivially done with a bitmap of size N). Next, it determines the nodes at all levels of the tree that need to incorporate the new document information. In the example of Figure 3, the insertion of document D_6 will affect nodes e_6, e_3, and e_1. Since each Paillier ciphertext in a node consists of f groups of counters, the document index counters must be shifted accordingly so that they are added to the correct position at the different nodes. Specifically, each of the M/b plaintext values of the document index must be shifted (to the left) $j \cdot d \cdot b$ times, where $0 \leq j \leq f-1$. Assuming counters of size $d = 4$ bits (Figure 3), the index values of D_6 must be shifted 12 times for nodes e_6 and e_1, and 0 times for node e_3. In the ciphertext space, the shifting operation of the underlying plaintext values is performed with a single modular exponentiation, where the exponent is the value $2^{j \cdot d \cdot b}$.

To summarize, given a ciphertext c_1 from the document index and the corresponding ciphertext c_2 from an arbitrary tree node, the updated ciphertext c_2, following the insertion of the new document, is computed as:

$$c_2 = c_1^{2^{j \cdot d \cdot b}} \cdot c_2$$

where j is calculated by the cloud provider, based on the underlying tree structure. In addition, in order to save storage space at the server site, document indexes can be destroyed after the update operation.

Finally, note that document deletions also necessitate index updates, as the individual keyword scores have to be subtracted from the corresponding tree nodes. In particular, when deleting a stored document, the client first re-constructs the encrypted document index from the metadata file and sends it to the cloud provider. The cloud provider then updates the index in a

Top_$k(Q, k)$

```
1:  $nodeHeap \leftarrow \emptyset$; $resultHeap \leftarrow \emptyset$;
2:  $\theta \leftarrow 0$;
3:  Compute the Bloom filter indexes $\{h_i\}$ associated
    with all keywords in $Q$;
4:  Using PIR, retrieve from the root node the Paillier
    ciphertexts corresponding to all $h_i$'s;
5:  Compute the aggregate score for every child of the
    root node, and insert that node into $nodeHeap$;
6:  while ($nodeHeap$ not empty) do
7:    Remove the top entry from $nodeHeap$ and
      store it into $e$;
8:    if ($e.score \leq \theta$) then
9:      break;
10:   end if
11:   Using PIR, retrieve from node $e$ the Paillier
      ciphertexts corresponding to all $h_i$'s;
12:   Compute the aggregate score for every child of $e$;
13:   if ($e$ is a leaf node) then
14:     Retrieve the metadata files for all documents
        with score $> \theta$;
15:     Compute the actual scores of these documents
        and insert them into $resultHeap$;
16:     $\theta \leftarrow$ score of $k$-th document in $resultHeap$;
17:   else /* $e$ is an internal node */
18:     Insert every child of $e$ into $nodeHeap$;
19:   end if
20: end while
21: return $resultHeap$;
```

Fig. 4. Top-k query processing algorithm

manner similar to the one explained above. The only difference is that the document index values have to be negated, an operation that is trivially performed (in the ciphertext space) by computing the multiplicative inverse modulo n^2 of the Paillier ciphertext.

C. Top-k Query Processing

Query processing is performed at the client side, by traversing the database index with a *best-first* search algorithm, as illustrated in Figure 4. Specifically, the client initializes and maintains two max-heaps: $nodeHeap$ and $resultHeap$ (line 1). The first one is used to visit nodes in decreasing score value, while the later one (which can be of size k) stores the result set that is eventually returned to the client. In addition, the client maintains a threshold value θ (line 2) that keeps track of the score value of the k-th document in $resultHeap$. This threshold is used to terminate the algorithm early, i.e., when there exists no other document that can alter the current top-k result set.

Initially, the client computes the Bloom filter indexes associated with all keywords comprising query Q (line 3). As an example, consider a top-2 query with two keywords, mapping into positions $h_1 = 0$ and $h_2 = 7$ (from left to right) in the Bloom filter vectors shown in Figure 3. For simplicity, assume that there are no false positives, i.e., every keyword hashes into a unique location. The algorithm starts from the root node (e_1), and utilizes the PIR protocol of Section III-B to retrieve the corresponding Bloom filter entries. In our example, the client will retrieve the first and third ciphertexts that contain the required values.

Next, the client computes the aggregate scores of e_1's children (e_2 and e_3) by adding the corresponding values at positions h_1 and h_2. Since e_1 is an internal node, the client simply inserts $\langle e_2, 12\rangle$ and $\langle e_3, 3\rangle$ into $nodeHeap$ (lines 17-18). Node e_2 is visited next, as it has the highest score in the heap (line 7). The same process is repeated, i.e., the client retrieves privately the first and third ciphertexts of e_2 and computes the aggregate scores of e_4 and e_5 (lines 11-12). Subsequently, the client inserts $\langle e_4, 8\rangle$ and $\langle e_5, 4\rangle$ into $nodeHeap$.

The next node removed from $nodeHeap$ is e_4, and the client computes (privately) the $estimated$ scores of D_1 and D_2. Node e_4 is now a leaf node, so the client retrieves the encrypted metadata files of *both* D_1 and D_2 (line 14), since both documents can potentially be part of the result set (i.e., their actual scores could be larger than θ). From the metadata files, the client calculates the actual scores (line 15) and, as in our example we assume the absence of false positives, $resultHeap$ is updated to $\{\langle D_1, 4\rangle, \langle D_2, 4\rangle\}$. Additionally, since there are exactly $k = 2$ documents in $resultHeap$, the threshold value θ is updated to 4 (line 16).

Finally, node e_5 is visited next, which has an aggregate score of 4. Given that document indexes are aggregated in a bottom-up fashion, none of the other documents in the database can have a score larger than 4. Consequently, the two existing documents in $resultHeap$ are at least as "good" as any of the remaining ones, so the algorithm can terminate safely (lines 8-9). At this point, the result set stored in $resultHeap$ is returned to the client, along with the corresponding document metadata. The client may then look into the metadata files and decide whether she wants to retrieve the encrypted full documents from the database server.

D. Security

First, all documents and their respective indexes are encrypted with Alice's public key. Therefore, the server is unable to decrypt them and derive the corresponding keywords. Second, due to the PIR queries, the server cannot determine the Bloom filter indexes that Alice is interested in. Consequently, he cannot derive any information regarding Alice's query. Finally, the top-k query processing algorithm is executed at Alice's local machine, so the server is oblivious to the ranking scores and order of the result set.

V. OPTIMIZATIONS

Public key operations are computationally expensive and, thus, the query processing algorithm described above would be inapplicable to large document collections. Therefore, in this section, we present a number of optimizations, targeting the cryptographic operations of our protocol, which may lead to reasonable query response times, even for large datasets. Section V-A presents several optimizations at the database server, while Section V-B introduces a few optimizations at the client side.

A. Server optimizations

Multiple CPUs. The PIR protocol of Section III-B is the major performance bottleneck in our top-k retrieval algorithm. As shown in a recent study [32], Gentry and Ramzan's scheme (which is arguably one of the more efficient computational PIR protocols) requires many seconds of computing time, even for databases of size less than 100 KB. In our system, a typical index node consists of several thousands of Paillier ciphertexts, each of size 256 bytes. Consequently, executing the PIR protocol in a single CPU is not a practical implementation.

In our work, we adopt the striping technique of [32] that sacrifices some communication cost in order to achieve significantly faster PIR query response times. The idea is to partition each node into t blocks (Figure 5), so that the required Paillier ciphertexts are retrieved by querying each of the t blocks in parallel. This is an ideal scenario for cloud computing platforms (such as Amazon's EC2) that offer large CPU clusters at low cost. Recall that the PIR protocol of Gentry and Ramzan associates a prime power π_j with each database record (in our case, Paillier ciphertext). Therefore, the client can simply construct a single PIR query that is applied to all t blocks comprising the node.

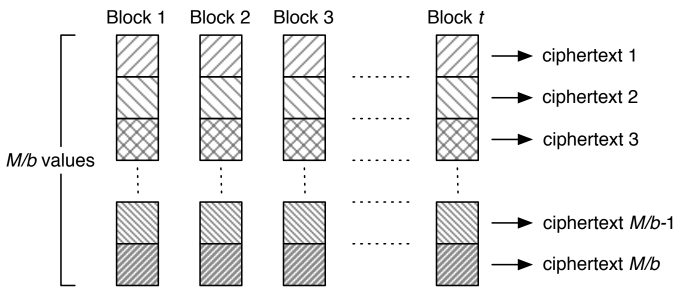


Fig. 5. Node partitioning

In our implementation, nodes are partitioned into $t = 128$ blocks, i.e., every block holds 2 bytes from each of the M/b ciphertexts. (Using the notation of Section III-B, $\ell = 16$.) The reason behind this choice is the security constraint of the query generation algorithm that sets a limit on the bit size of the product of all the prime powers π_j corresponding to the requested records. Setting $\ell = 16$ allows us to securely retrieve up to 10 different ciphertexts with a single PIR query, i.e., our system supports keyword search queries with up to $|Q| = 10$ terms. Finally, note that the multiple CPUs can be leveraged in the index update process as well. Since every Paillier ciphertext is updated independently, we can always process multiple ciphertexts in parallel.

Chinese Remainder Theorem. During the setup phase of the PIR protocol, the server computes the transformed database β by applying the Chinese Remainder Theorem (CRT) on the original database records. In our setting, document insertions and deletions alter the entire node content, so the CRT has to be computed from scratch after each update operation. Consequently, an efficient implementation of the CRT is of paramount importance. In our work, we chose Garner's algorithm [33], which includes an expensive preprocessing step, but is very efficient during data updates. Specifically, as long as the number of records remains constant (which is true for the index nodes) the preprocessing step has to be executed only once and is independent of the database records (it only depends on the prime powers π_j of the PIR protocol). As shown in our experimental results, calculating the CRT using the pre-computed values is very inexpensive.

B. Client optimizations

Discrete logarithm. The most expensive operation at the client side is solving the discrete logarithm problem when extracting the result from the server's reply. This cost is amplified in our implementation, due to the aforementioned node partitioning method. In particular, our approach requires $t|Q|$ discrete logarithm computations per visited node, in order for the client to retrieve $|Q|$ Paillier ciphertexts. Therefore, instead of relying on the Pohlig-Hellman algorithm, we chose to pre-compute all possible results during the query generation algorithm. As it is evident in the protocol description (Section III-B), when the client generates the PIR query she can compute the g_{i_j} values corresponding to all possible (in our case, 2^{16}) database records D_{i_j}. Furthermore, these pre-computations are very efficient if we apply successive modular multiplications.

Multiple CPUs. Similar to the server case, clients can also benefit from the availability of multi-core CPUs and/or high-performance GPU chips. Most operations, including discrete logarithm computations, are fully parallelizable, since they can be performed independently. Nevertheless, we did not explore this possibility in our experiments, i.e., we assumed that the client utilizes a single CPU.

VI. Experimental Results

In this section, we evaluate experimentally the performance of our methods, in terms of storage, communication, and CPU cost at the various entities of our architecture. Section VI-A describes the setup of the experiments, while Section VI-B presents the results.

A. Setup

We developed our programs in C++, utilizing the GMP[4] arithmetic library to handle large integers. We run the client and sender (Alice and Bob) programs on an Intel Core i7, 2.8 GHz CPU, and the server program on an Amazon EC2 instance with 1 Compute Unit. In other words, we did not utilize multiple CPUs at the cloud provider, but rather measured the CPU times required to execute the cryptographic protocols of our methods on Amazon's infrastructure. Table II summarizes the costs of the basic cryptographic primitives at the three parties involved in our system architecture.

For the document collection, we downloaded the Enron email dataset[5], which is an excellent real life example of the scenario described in Section I. We created several versions of the document collection, corresponding to different values of N and $|W|$. When selecting the keywords for a specific document, we used the terms from the "From:" and "Subject:" fields (excluding certain stop words) and, if there was more room, we chose the most frequent terms from the email's body. For each experiment, we generated 1000 random queries and run the top-k query processing algorithm of Figure 4. We measured the average number of index nodes that were accessed per query, and then used Table II to derive the corresponding costs. When creating the queries, we tried to

[4]http://gmplib.org/
[5]http://www.cs.cmu.edu/∼enron/

TABLE II. COST OF CRYPTOGRAPHIC PRIMITIVES AT DIFFERENT ENTITIES

Server (1 Amazon EC2 Compute Unit)	
Paillier Cryptosystem	
Modular exponentiation	2.1 ms
Modular multiplication	0.005 ms
Encryption	6.5 ms
PIR (250 elements, 2 bytes each)	
Preprocessing (for CRT)	103 ms
CRT (per block)	0.4 ms
Query (per block)	7.8 ms
PIR (1250 elements, 2 bytes each)	
Preprocessing (for CRT)	1.6 sec
CRT (per block)	5 ms
Query (per block)	53 ms
PIR (2500 elements, 2 bytes each)	
Preprocessing (for CRT)	5.9 sec
CRT (per block)	19 ms
Query (per block)	143 ms
PIR (5000 elements, 2 bytes each)	
Preprocessing (for CRT)	22 sec
CRT (per block)	81 ms
Query (per block)	377 ms
Sender (Intel Core i7, 2.8 GHz)	
Paillier Cryptosystem	
Encryption	4.6 ms
Client (Intel Core i7, 2.8 GHz)	
Paillier Cryptosystem	
Decryption	4.5 ms
PIR (2 keywords)	
Query generation	441 ms
Result retrieval	192 ms
PIR (3 keywords)	
Query generation	661 ms
Result retrieval	204 ms
PIR (4 keywords)	
Query generation	667 ms
Result retrieval	230 ms
PIR (5 keywords)	
Query generation	822 ms
Result retrieval	262 ms

mimic the typical user behavior by searching based on the sender and subject fields. If there were not enough keywords in these fields to fill the query, we used random keywords from the email's body.

Table III summarizes our system parameters, with their default values appearing in bold face. When testing the effect of a single parameter, we fixed the remaining ones to their default values. In the following plots we illustrate the effectiveness of our approach, based on these performance metrics:

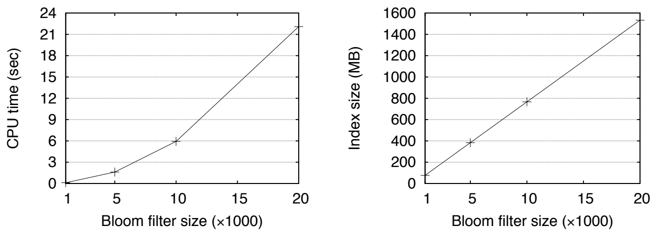
- The offline processing and storage cost at the server to store the database index.
- The CPU time at the server to update the database index.
- The CPU time and communication cost at the sender to send a new document to the server.
- The query response time at the client, i.e., the time that elapses from the instance the query is posed, until the actual answer in obtained.
- The communication cost between the client and the server during query processing. This cost includes the transferred metadata files, whose size is fixed to 512 bytes.

TABLE III. SYSTEM PARAMETERS

Parameter	**Range**
N	1000, 2000, **5000**, 10000
M	1000, **5000**, 10000, 20000
k	1, 2, **5**, 10, 15
$\|Q\|$	2, **3**, 4, 5
f	**17**
b	**4**
$\|W\|$	**10**, **20**
$\log n, \log m$	**1024**
Number of CPUs at server	**128**

B. Results

Figure 6(a) shows the CPU time at the server for constructing the complete index tree (for $N = 10000$ documents), as a function of the Bloom filter size M. As explained in Section IV-B, the index is initially empty, i.e., every node consists of Paillier encryptions of zero. The cost is dominated by the preprocessing step of Garner's CRT algorithm, because the time to construct the Paillier ciphertexts is negligible (the server simply creates a single Paillier ciphertext and populates all index nodes with the same value). This figure also illustrates the computational complexity of the CRT problem that justifies the expensive preprocessing step at the cloud provider. Figure 6(b) depicts the storage cost at the server to hold the database index. As expected, it grows linearly in M, and reaches approximately 1.5 GB for $M = 20000$. Note that, the actual storage cost of the index tree is only half of what is shown in this figure. The reason is that, due to the underlying PIR protocol, the server needs to maintain two versions of the index: the "plaintext" version consisting of the Paillier ciphertexts, and the transformed version β required by Gentry and Ramzan's scheme.



(a) Index construction time (b) Index size

Fig. 6. Index construction cost

Figure 7 plots the processing time at the cloud provider for updating the index, as a function of M. Recall that, for a single node, the update process requires (i) one modular exponentiation and one modular multiplication per Paillier ciphertext, and (ii) one CRT computation per block. Using multiple CPUs in parallel (Section V-A) reduces considerably the processing time at the server, which remains below 350 ms in all cases.

Figure 8(a) illustrates the CPU time at the sender, as a function of M. When sending a new document to the server, the sender has to encode the keyword information on the Bloom filter, and then perform numerous Paillier encryptions to construct the document index. Clearly, the cost of these encryptions can be significant, requiring over 20 sec of processing time for $M = 20000$. One optimization

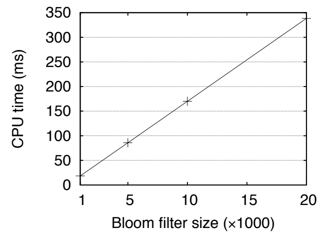
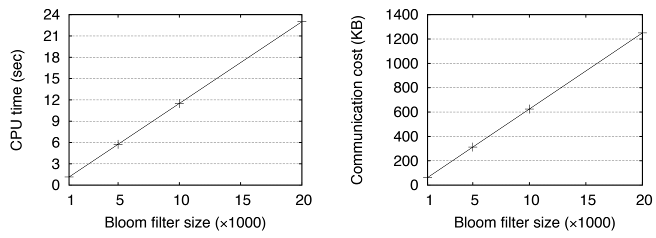


Fig. 7. Index update cost

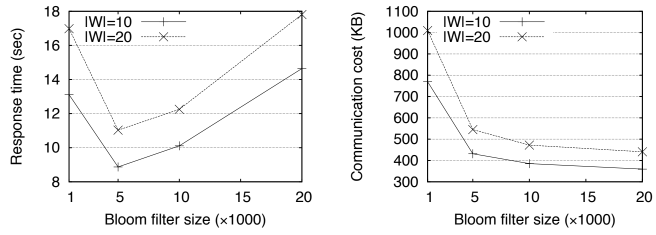
that can be applied to reduce this cost, is for the sender to pre-compute (offline) Paillier encryptions of zero. This may alleviate significantly the online cost, because the vast majority of the Bloom filter counters will always remain zero. Figure 8(b) shows the communication cost for the same experiment. It grows linearly in M, and ranges from 62 KB to 1.2 MB.



(a) CPU time (b) Communication cost

Fig. 8. Cost at the sender

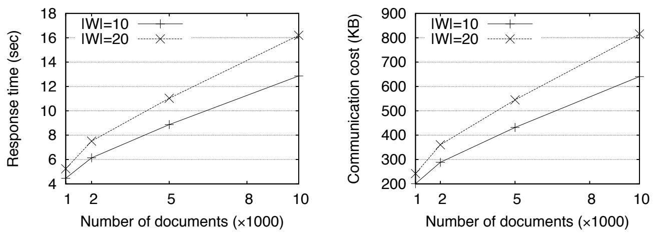
Figure 9(a) shows the response time of our top-k query processing algorithm, with respect to the Bloom filter size M. When M is small, the number of false positives at the individual Bloom filters is high, thus affecting the accuracy of the computed aggregate scores. As a result, the best-first search algorithm has to visit a lot more index nodes than necessary, a fact that increases the overall query response times. On the other hand, larger Bloom filter vectors are more accurate, and the top-k algorithm visits significantly less number of nodes that result in better performance. Nevertheless, as M increases even further, the cost of the PIR retrievals becomes a dominant factor that eliminates the advantage of the reduced false positive rates. Similarly, Figure 9(b) depicts the communication cost at the client for the same experiment. As M increases, the Bloom filters become more accurate, leading to fewer node accesses and, thus, lower communication cost. As evident in Figure 9, the value $M = 5000$ achieves a good trade-off between query response time and communication cost.



(a) Response time (b) Communication cost

Fig. 9. Query processing cost vs. M

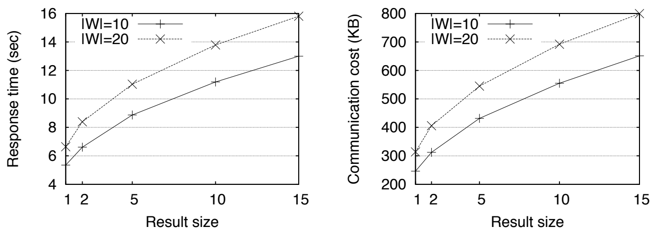
Figure 10 illustrates the response time and communication cost of the query processing algorithm, as a function of the number of documents N. Recall that, in this experiment, M is fixed to 5000, so increasing the number of documents produces more false positives and, therefore, the performance of our algorithm deteriorates. However, even for $N = 10000$ documents, the algorithm terminates in less than 17 sec and incurs less than 900 KB of communication cost.



(a) Response time (b) Communication cost

Fig. 10. Query processing cost vs. N

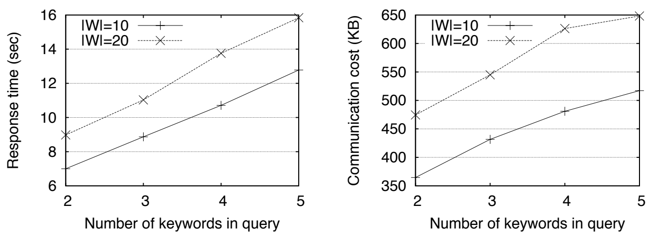
Figure 11 depicts the cost of the query processing algorithm, with respect to the number of requested documents k. When the client wants to retrieve more relevant documents, the threshold value θ that terminates the algorithm (Figure 4) becomes smaller, thus allowing the algorithm to run further and access more nodes. As a result, both the query response time and the communication cost increase with k.



(a) Response time (b) Communication cost

Fig. 11. Query processing cost vs. k

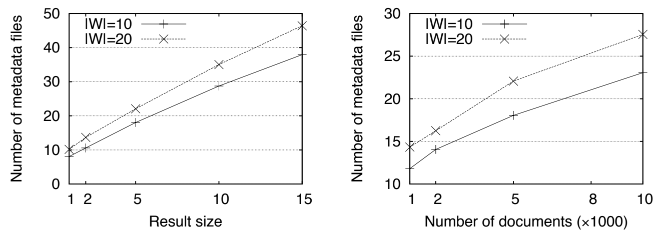
Figure 12 shows the response time and communication cost of the query processing algorithm, as a function of the number of keywords $|Q|$ in the client's query. As $|Q|$ increases, the effect of the false positives is amplified, since more Bloom filter indexes are involved in the score computations. Consequently, the number of visited nodes increases slightly with $|Q|$. Note that, the query response time increases faster than the communication cost, because the PIR-related computations at the client side are more expensive for larger values of $|Q|$ (Table II).



(a) Response time (b) Communication cost

Fig. 12. Query processing cost vs. $|Q|$

Finally, Figure 13 illustrates the number of metadata files that are transferred to the client during query processing, with respect to k and N. As explained previously, increasing either k or N leads to more node accesses and, therefore, more metadata files are sent to the client. Nevertheless, under all settings, less than 1% of the total number of document metadata are transferred to the client. Consequently, the resulting overhead is negligible, thus justifying their use to offset the effect of false positives in the Bloom filters.



(a) Number of metadata files vs. k (b) Number of metadata files vs. N

Fig. 13. Number of metadata files accessed

VII. Conclusions

Searchable encryption is an important cryptographic primitive that facilitates private keyword searches directly on encrypted data. While this problem is studied extensively in the symmetric key setting, existing public-key algorithms are very restrictive in the types of keyword queries that they allow. To this end, our work introduces the first method for privacy-preserving ranked keyword search on public-key encrypted data. Our solution employs a simple indexing structure, and leverages homomorphic encryption and private information retrieval protocols to process queries in a privacy-preserving manner. Furthermore, we introduce several optimizations for the cryptographic primitives of our approach that reduce the query response times by several orders of magnitude. Using measurements from Amazon's EC2 infrastructure, we show that our method can process ranked keyword searches in less than 17 sec, while incurring less than 900 KB of communication cost.

Acknowledgments

This research has been funded by the NSF CAREER Award IIS-0845262.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] L. M. V. Gonzalez, L. Rodero-Merino, J. Caceres, and M. A. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009.

[3] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM CCS*, 2006, pp. 79–88.

[5] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE S&P*, 2000, pp. 44–55.

[6] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *IEEE INFOCOM*, 2011, pp. 829–837.

[7] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *EUROCRYPT*, 2004, pp. 506–522.

[8] J. Baek, R. Safavi-Naini, and W. Susilo, "Public key encryption with keyword search revisited," in *ICCSA*, 2008, pp. 1249–1259.

[9] P. Golle, J. Staddon, and B. R. Waters, "Secure conjunctive keyword search over encrypted data," in *ACNS*, 2004, pp. 31–45.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] C. Gentry and Z. Ramzan, "Single-database private information retrieval with constant communication rate," in *ICALP*, 2005, pp. 803–815.

[12] E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, Report 2003/216, 2003.

[13] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS*, 2005, pp. 442–455.

[14] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *IEEE ICDCS*, 2010, pp. 253–262.

[15] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *EUROCRYPT*, 2009, pp. 224–241.

[16] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis, "Secure kNN computation on encrypted databases," in *ACM SIGMOD*, 2009, pp. 139–152.

[17] G. D. Crescenzo and V. Saraswat, "Public key encryption with searchable keywords based on jacobi symbols," in *INDOCRYPT*, 2007, pp. 282–296.

[18] L. Fang, W. Susilo, C. Ge, and J. Wang, "A secure channel free public key encryption with keyword search scheme without random oracle," in *CANS*, 2009, pp. 248–258.

[19] C. Gu, Y. Zhu, and H. Pan, "Efficient public key encryption with keyword search schemes from pairings," in *Inscrypt*, 2007, pp. 372–383.

[20] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith, "Public key encryption that allows PIR queries," in *CRYPTO*, 2007, pp. 50–67.

[21] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *TCC*, 2005, pp. 325–341.

[22] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC*, 2009, pp. 169–178.

[23] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, 1999, pp. 223–238.

[24] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *IEEE FOCS*, 1995, pp. 41–50.

[25] I. Goldberg, "Improving the robustness of private information retrieval," in *IEEE S&P*, 2007.

[26] D. P. Woodruff and S. Yekhanin, "A geometric approach to information-theoretic private information retrieval," in *IEEE CCC*, 2005.

[27] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *IEEE FOCS*, 1997, pp. 364–373.

[28] H. Lipmaa, "An oblivious transfer protocol with log-squared communication," in *ISC*, 2005, pp. 314–328.

[29] J. Groth, A. Kiayias, and H. Lipmaa, "Multi-query computationally-private information retrieval with constant communication rate," in *PKC*, 2010, pp. 107–123.

[30] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, no. 1, pp. 106–110, 1978.

[31] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[32] S. Papadopoulos, S. Bakiras, and D. Papadias, "pCloud: A distributed system for practical PIR," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 9, no. 1, pp. 115–127, 2012.

[33] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press, 1997.