Reducing Communication Cost of Encrypted Data Search with Compressed Bloom Filters

Muhammad Umer*, Tahir Azim*† and Zeeshan Pervez‡
* National University of Sciences and Technology (NUST),Islamabad, Pakistan
† École Polytechnique Fédérale de Lausanne, Switzerland
‡ University of the West of Scotland, Paisley, Scotland PA1 2BE
Email: 13msitmumer@seecs.edu.pk, tahir.azim@epfl.ch, zeeshan.pervez@uws.ac.uk

***Abstract*—Consumer data, such as documents and photos, are increasingly being stored in the cloud. To ensure confidentiality, the data is encrypted before being outsourced. However, this makes it difficult to search the encrypted data directly. Recent approaches to enable searching of this encrypted data have relied on trapdoors, locally stored indexes, and homomorphic encryption. However, data communication cost for these techniques is very high and they limit search capabilities either to a small set of pre-defined trapdoors or only allow exact keyword matching.**

This paper addresses the problem of high communication cost of bloom-filter based privacy-preserving search for measuring similarity between the search query and the encrypted data. We propose a novel compression algorithm which avoids the need to send the entire encrypted bloom filter index back to the client. This reduces the cost of communicating results to the client by over 95%. Using sliding window bloom filters and homomorphic encryption also enables our system to search over encrypted data using keywords that only partially match the originally stored words. We demonstrate the viability of our system by implementing it on Google Cloud, and our results show that the cost of partially matching search queries on encrypted data scales linearly with the total number of keywords stored on the server.

I. Introduction

Cloud computing enables its subscribers to use computing services over the Internet and provides tailored computing resources on-demand. Subscribers of cloud storage often outsource their data on public cloud servers to ensure that it is highly available and reliable [1]. It is then the cloud service provider's responsibility to coordinate data backup, synchronization and sharing with relevant stakeholders.

As the amount of data stored on the cloud increases dramatically, the security of this data has become an important concern. Recent cases [2] have included users' photos and videos getting stolen from compromised cloud servers resulting in severe breaches of privacy. Credit card numbers and medical records are other examples of user data whose privacy needs to be ensured.

The straightforward way to achieve data protection and confidentiality is to place data on the cloud after encrypting it using public key encryption. This solves the confidentiality and privacy problem, but encryption hides information within the data and searching becomes difficult. The user has to download all of the data from cloud storage and search it after decrypting it locally. This is obviously an expensive proposition. Therefore we need a mechanism that allows a user to search over encrypted data without revealing private information and without downloading excessive data.

Existing approaches for searching over encrypted data often rely on "trapdoors" [3]–[5]. Trapdoors enable a user to search the encrypted data for a small set of pre-defined keywords. These approaches restrict the searching capability of a user to a limited number of trapdoors defined during data encryption. More recent work has focused on homomorphic encryption [6], [7] as a solution to this problem. However, to ensure privacy from the cloud service provider (CSP), all the evaluated results are returned from the cloud server in encrypted form without any compression. This causes high data communication cost but naively allows searching for exact keyword matches over the encrypted data.

In this paper, we present a system for performing searches on encrypted data in the cloud. We propose a novel compression technique which allows us to avoid sending the entire encrypted index back to the client. This reduces the cost of communicating search results back to the client by over 95% resulting in faster response times and less budgeting cost to the owner. Besides exact keyword matching, our system also supports similarity-based searching: finding documents with partially matching keywords. Homomorphic encryption enables private search capability, while sliding window bloom filters enable searching for partially matching keywords.

II. Related Work

Traditionally, confidential information is protected using access control mechanisms. This mechanism only works if confidential information is present on a fully secure, trusted server. But this assumption fails when confidential information is outsourced to remote servers in the cloud.

Although cloud service providers (CSP) allow data to be encrypted in order to secure it, encryption leads to a couple of problems. First, searching for keywords within data encrypted using popular encryption algorithms is not possible. Second, if you provide the CSP the decryption keys in order to make search possible, you cannot prevent the CSP from accessing the confidential data.

Recent work has focused on solving these problems using a variety of techniques. These techniques can be broadly classified into two categories.

Using Trapdoor Encryption. Song et al. [3] proposed a symmetric key based searchable scheme for encrypted data. Each keyword in the document was encrypted independently using trapdoors. Their approach proposed ways to search encrypted data using both an encrypted index as well as searching the original data itself. Goh [4] proposed encrypted search using bloom filters. They generate trapdoors against all the keywords in a file and add them to a bloom filter. In this way, for each file, a single bloom filter is created using trapdoors and stored in the cloud. To search, a user computes the trapdoor for a keyword and sends it to the cloud server. The cloud server checks the trapdoor in the bloom filter, and if it exists, returns the corresponding file identifier. Waters et al. [8] extended the work of Song et al. and proposed a similar technique to secure audit logs. Audit logs contain sensitive information about a series of events, actions and actors who are responsible for triggering particular event or performing an action. Therefore encryption is required for its confidentiality. When it needs to be searched, a trusted third party issues a trapdoor for a specific keyword search. All of the above schemes support only exact keyword matches and rely on trapdoors.

Boneh et al [5] presented the first public key based searchable scheme which enables authorized users having the private key to search in the index. This approach still used trapdoors based indexing and supported only exact keyword matching.

Yu et al. [9] proposed a scheme on encrypted Personal Health Records (PHR) by using Hierarchical Predicate Encryption (HPE). They also used a trusted third party for the distribution of trapdoors. Authorized users obtained trapdoors from the trusted third party and then submitted them to the CSP for evaluation. This scheme is limited because only predefined trapdoors can be used and users cannot model their own queries.

Pal et al. [10] proposed an encrypted search scheme using bloom filters. They also used soundex coding [11] for each word to search words which are pronounced similarly. This work is complementary to our work. However, it does not make an effort to reduce the communication cost of returning results to the client. Kuzu et al. [12] introduced a similarity-based encrypted search scheme, which used locality sensitive hashing for the similarity score calculation. This scheme also used trapdoors, hence leaking private matching information upon which statistical attacks are possible.

Using Homomorphic Encryption. Homomorphic encryption is an encryption technique that allows computations to be carried out on ciphertext, thus generating an encrypted result, which when decrypted, matches the result of operations performed on the plaintext. An example of homomorphic encryption is the Pascal Paillier cryptosystem [13], which provides two useful properties: (i) the product of two ciphertexts will decrypt to the sum of their corresponding plaintexts and (ii) a ciphertext raised to a constant k will decrypt to the product of the corresponding plaintext and the constant.

Pervez et al. [7] proposed an encrypted data search scheme using an inverted index. They used homomorphic encryption [6] to provide end-to-end privacy. Only authorized users could execute queries using their personal proxy re-encryption keys in order to manipulate the index, so a lot of computation is done due to index transformations. They used a trusted third party to rank search results and model queries. While this approach avoids using trapdoors, it still supports only exact keyword matching and has high communication cost.

Finally, CryptDB [14] and MONOMI [15] improve the performance of searching over encrypted data using a split client/server querying paradigm. By employing several forms of encryption including symmetric, public-key and homomorphic encryption, they improve querying performance by orders of magnitude. However, in doing so, they are vulnerable to several kinds of information leakage but with low probability.

III. Proposed System

Before diving into a description of how the system is designed and implemented, we discuss some of our assumptions about the system and the threat model. Then we will describe the main steps involved in the initial index generation and then the actual search process over encrypted data.

A. System Model

Entities involved in the proposed system are the data owner, CSP and end user. A data owner is an entity who wants its confidential data to be stored in cloud storage with the capability of privacy-aware searching. A CSP hosts public cloud storage services for its subscribers on a pay-as-you-use model. The end user is an entity that will perform searches on the encrypted data stored in the cloud. The end user can submit search queries to the CSP, which evaluates the queries and returns results back to the user. However, during query evaluation, the CSP should not be able to learn anything about the query, the stored data or the matching results.

B. Threat Model

The data owner and end user are considered trusted entities while CSP is considered as an untrusted entity because it is maintained by an arbitrary third party. Data communication between the end user and CSP should be considered as untrusted as all requests are routed via the open Internet. The CSP hosts the encrypted data file ($\mathcal{F}$) and the encrypted index (I). As they are encrypted, the CSP cannot learn any information regarding $\mathcal{F}$ and I. Query evaluation is done by CSP homomorphically: that is, the CSP performs evaluation over encrypted text of I and encrypted user query, so it is neither able to learn anything about the matched results, nor can it relate any subsequent queries from other users.

C. Assumptions and Notations

We ignore the key exchange mechanism between the data owner and the end users, assuming that it happens using secure out-of-band channels. Table I illustrates the notations that we use in order to explain the details of our proposed approach.

TABLE I
NOTATIONS USED IN MATHEMATICAL AND DESCRIPTIVE DETAILS

Notation	Description
$\mathcal{F}$	Confidential file that needs to be outsourced
$\mathcal{K}\omega$	List of keywords in a data file $\mathcal{F}$
$f\omega$	Frequency of a keyword $k\omega$
$\mathcal{L}$	Allowed keyword length for extraction of list of keywords from data
I	Index file having encrypted bloom filter for each keyword, frequency and number of 1ns as index entries
$\mathcal{E}_h$, $\mathcal{D}_h$	Homomorphic encryption and decryption functions
$\mathcal{E}_S$, $\mathcal{D}_S$	Symmetric encryption and decryption functions
σ_{pk} , σ_{sk}	Homomorphic encryption public and private keys
$\mathcal{K}_S$	Symmetric encryption and decryption keys. It is used to encrypt the data file.
$\mathcal{S}_q$	Similarity score of a query
$\mathcal{R}_q$	Compressed resultant term after bloom filter matching
$\mathcal{M}_q$	Uncompressed resultant term after bloom filter matching
$\mathcal{BF}_i$	i'th encrypted bit of bloom filter $\mathcal{BF}$
O_b	Number of 1 bits in a bloom filter

D. Index Generation

The data owner generates an encrypted index on the client side to facilitate subsequent searches. At a high level, the index is generated by creating a *sliding window bloom filter* which is then encrypted using the Pascal Paillier homomorphic encryption algorithm [13].

The sliding window bloom filter (SWBF) is a special type of bloom filter for which a window size is defined, and based on that window size, a keyword is sliced and mapped to the bloom filter. For example, suppose we have a word "cloud" and a window size of 2. The word will then be sliced into "cl", "lo", "ou", and "ud" and each of these slices will be independently mapped to the bloom filter. This filter enables us to achieve a partial matching even if the requested keyword does not match completely.

Algorithm 1 describes the indexing procedure. For each file to be uploaded to the server, a separate index file is generated. The index file contains one index entry per keyword. To generate the index file, a sliding window bloom filter is updated for each keyword and the number of 1 bits in the bloom filter are noted. Each bit of the bloom filter is then encrypted using the Pascal Paillier algorithm, and written to the index entry along with frequency and number of 1 bits. At the end of this process, each index entry I_i in an index file I has the structure:

$$I_i = \mathcal{BF}_i, f\omega, O_b \tag{1}$$

where O_b is the number of 1's in the bloom filter $\mathcal{BF}_i$ and $\mathcal{BF}_i$ is consisting of n bits $\mathcal{BF}_i = \mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_n$.

Once the encrypted index file has been generated, it is uploaded to the cloud along with the data files. The data files are encrypted using a symmetric encryption algorithm, as they will not be used during the search process.

E. Search Process

To start the search process, the user first generates a query through a similar process as index creation. Each of the user's specified keywords are used to generate sliding window bloom filters, whose bits are then encrypted using the public key σ_{pk} of the Pascal Paillier algorithm. Note that this process takes place on the end user's machine. After encryption, the query (comprising the encrypted bloom filters) is sent to the cloud for terms matching.

On receiving a query, the CSP matches the incoming query with the index entries of the files it is hosting. For a perfect match, all of the corresponding bits of the index and query bloom filters need to match. However, since both the query and the index bloom filters are encrypted, simple matching is not possible. Furthermore, encrypting the same plaintext repeatedly results in completely different ciphertexts, which improves privacy and security but makes matching difficult.

Algorithm 1: Index Creation

Input: A collection of text files $C = \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n \rangle$
Output: Index files $I = \langle I_1, I_2, \ldots, I_n \rangle$ for each text file in C

$\forall \mathcal{F}_i \in \langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n \rangle$;
while $\mathcal{F}_i \in C$ **do**
  $\mathcal{K}\omega \leftarrow extractAllKeywords(\mathcal{F}_i)$;
  $\forall k\omega_j \in \mathcal{K}\omega$;
  while $k\omega_j \in \mathcal{K}\omega$ **do**
    $SWBF \leftarrow createSWBF(k\omega_j)$;
    $f_j \leftarrow getKeywordFrequency(k\omega_j)$;
    $O_b \leftarrow getOnes(\mathcal{BF}_j)$;
    $\forall bf_k \in SWBF$;
    while $bf_k \in SWBF$ **do**
      $\mathcal{BF}_k \leftarrow \mathcal{E}_h(\sigma_{pk}, bf_k)$;
      $IndexEntry \leftarrow IndexEntry, \mathcal{BF}_k$;
      $bf_k \leftarrow getNextBit(SWBF)$;
    $IndexEntry \leftarrow IndexEntry, f_j, O_b$;
    $I_j \leftarrow writeToIndexFile(IndexEntry)$;
    $I \leftarrow I, I_j$;
    $k\omega_j \leftarrow getNextKeyword(\mathcal{K})$;
  $\mathcal{F}_i \leftarrow getNextFile(C)$;
return I;

Fortunately, homomorphic encryption provides a way to perform mathematical operations over ciphertext. Since our bloom filter entries are encrypted using Pascal Paillier homomorphic encryption, we can just multiply their ciphertexts, which when decrypted, yields the sum of the plaintexts [13]. The sum of the bits will be either 0,1 or 2 after decryption. The twos in the decrypted output then represent the matched bits and the ones represent the unmatched bits. Then, a similarity score can be estimated as the ratio of twos to ones in the decrypted result.

Note that decryption of this result will require users to have the private Pascal Paillier key. Thus, only authorized users who have received the private key from the data owner can actually decrypt the results.

F. Compressing Search Results

Once the encrypted index entries and the query have been multiplied together, we can simply return these products back to the client. The client can then decrypt the products to get the sums and compute similarity scores. While this straightforward approach works and is used by many existing systems (for example, in [7]), it is extremely inefficient. In particular, if there are many documents and keywords in the cloud dataset, this will result in a huge amount of data to be communicated back to the user. Specifically, the product of the query and the encrypted index entry for *every* keyword in the dataset will be returned to the client. This will not only increase the response time over the network, but also the cost of cloud computing for the data owner, since network usage is billed by most cloud service providers.

In order to reduce this communication cost overhead, we devised a method by which we are able to reduce this cost by over 95%. Based on the insight that each returned index entry consists only of 0s, 1s or 2s, we define a polynomial $\mathcal{P}$ given by (2).

$$\mathcal{P} = a^0 \cdot \mathcal{BF}_1 + a^1 \cdot \mathcal{BF}_2 + a^2 \cdot \mathcal{BF}_3 + \ldots + a^{n-1} \cdot \mathcal{BF}_n \quad (2)$$

where $a \in \langle 3,5,7 \cdots \rangle$, $\mathcal{BF}_i$ are the Paillier sums of the corresponding bits of an index entry and represent variables for the polynomial $\mathcal{P}$ and $\mathcal{BF}_i \in \langle 0,1,2 \rangle$. We used $a = 3$ as it yields smaller sums and less computaion.

The sum of this polynomial $\mathcal{P}$ is the resultant compressed term which we return back to client application:

$$\mathcal{R}_q = \sum_{i=0}^{n-1} a^i \cdot \mathcal{BF}_i \quad (3)$$

From (3) we get a single compressed term $\mathcal{R}_q$ whose size is the size of the Pascal Paillier key. This multiplication by constants and addition is again possible because our bloom filter entries are encrypted using Pascal Paillier homomorphic encryption. Therefore, if after decryption, we want the product of the plaintext with a constant, we just need to exponentiate the ciphertext by that constant [13].

This means that irrespective of the size of the bloom filter, we will always return a single number back to the client. The client can then decompress it to find out the original Paillier sums using the routine specified in Algorithm 2. In short, this algorithm works correctly because every $\mathcal{BF}_i$ is at most 2 and $a^n > 2\sum_{j=0}^{n-1} a^j$. This means that taking $log_a \mathcal{R}_q$ repeatedly will yield the position of the next bloom filter entry that needs to be incremented.

Algorithm 2: Index entry decompression

Input: The compressed index sum $\mathcal{R}_q$
Output: Matched result $\mathcal{M}_q$ comprising a sequence of 0s, 1s and 2s

```
a ← 3 ;
i ← 0;
while R_q > 0 do
    i ← log_a R_q ;
    R_q ← R_q − a^i ;
    M_q[i] ← M_q[i] + 1 ;
return M_q;
```

The savings achieved using this algorithm can be evaluated using a simple theoretical formulation. Suppose we use a 32-bit bloom filter and a 64-bit Pascal Paillier key. Then for 1000 keywords, the data owner will create an index file with 1000 entries. Then the size of the index file will be 32*64*1000/8 = 256KB. On the other hand, after compression, the number returned per keyword will just have a size equal to the size of the Paillier key, namely 64 bits. So the total data returned will be 64*1000/8 = 8KB, a saving of over 95%.

IV. Evaluation

We demonstrate the viability of our proposed scheme using a real cloud environment, Google Cloud. We implement an indexing service, a search service and a client application as standard Java services. The search service was deployed on Google App Engine configured as an F4 class instance with a 2.4 GHz CPU and 512 MB of RAM. We utilized Google Blobstore for hosting index files because it allows us to store and retrieve the complete index file efficiently. This is in contrast to Google Datastore whose entities can hold only a few bytes of data. As a result, when index entries increase, read/write operations on index entries in the Datastore become more costly.

We evaluate the results of our approach on a dataset of 150 documents. These documents range in size from 5 MB to 100 MB, containing from 5000 to 129,000 keywords. The keywords were chosen to be at least 8 characters in length and Porter stemming [16] was applied to those keywords. The client was a Lenovo Thinkpad 430 with a 2.6 GHz Intel Core(TM) i5-332M CPU and 8 GB of RAM.

A. Indexing Performance

Figure 1 shows execution time for the index creation and encryption processes over different input dataset sizes using a 64-bit Pascal Paillier key and a 3 KB bloom filter. The graph shows that, despite using sliding window bloom filters, both

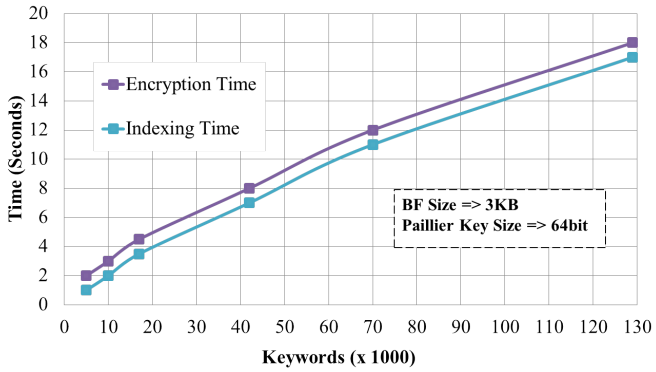

Fig. 1. Index Generation Time

encryption and index creation sizes increase linearly with input dataset size.

Paillier key size and bloom filter size both have a substantial impact on index file size. With the increase of bloom filter size, the false positive rate decreases but the size of each index entry increases. The indexing time also increases, mainly because it takes longer to upload a larger bloom filter to the cloud. A larger Paillier key size can help strengthen system security. However, with increasing Paillier key size, index file size also increases proportionally and our experiments show that index size increases linearly with Paillier key size.

B. Search Performance

Search performance is evaluated in terms of data returned by a query, response time, CPU cycles used to process a search request and the cost($) CSP will charge for search queries. For private term matching and data returned, our implementation demonstrates that by using our compression algorithm, data returned to the client is 95% less than traditional approaches and remains constant even when the size of the bloom filter increases. This is shown in Figure 2.

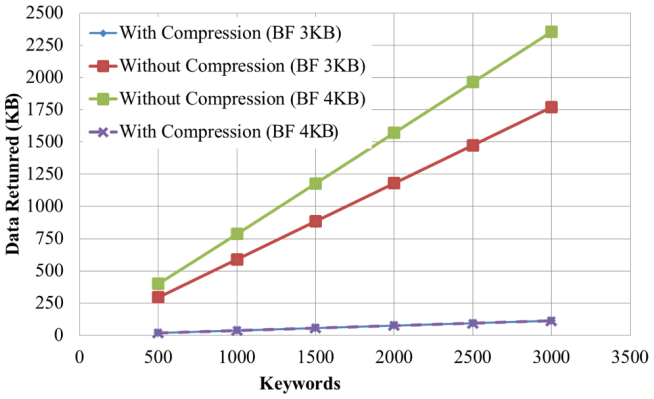

Fig. 2. Data returned by a search query: compressed vs uncompressed results

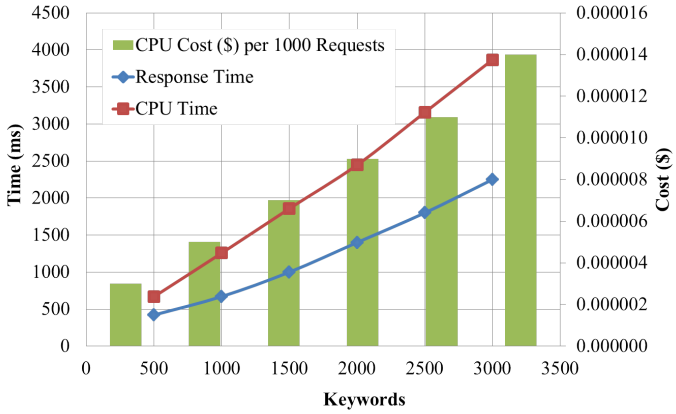

Fig. 3. Searching Cost($) with Single Keyword Query

We also measure the time it takes to extract the matching bloom filter values from the compressed responses. Table II show the results. As expected, the extraction time increases linearly with the keyword count on the server. The reason is that we add more keywords keywords by adding more documents to the dataset and each document has its own bloom filter index associated with it.

TABLE II
RESPONSE TIME INCREASES LINEARLY WITH KEYWORD COUNT.

Keyword Count	Response Extraction Time (ms)
500	60
1000	125
1500	185
2000	250
2500	310
3000	370

C. Cost Estimation and Response Times

Finally, we evaluate the dollar cost that CSP will charge to data owners for search queries with partial matching. Our results, shown in Figure 3, demonstrate that a query searching for a single keyword in a dataset having 500-3500 index entries will cost only $0.000002 to $0.00002 per 1000 similar queries. Due to the sliding window bloom filter approach, all of the metrics increase only linearly ($O(n)$) with the number of keywords on the server. This is substantially better than using a naive algorithm for partial matching, which would result in $O(nm)$ time complexity for n keywords that are on average m characters long. Crucially, privacy is preserved throughout the search process because data is never decrypted at the cloud or by any other untrusted system.

V. Conclusions and Future Directions

In this paper, we presented a privacy-aware similarity-based searching scheme with reduced communication cost over encrypted data residing in an untrusted cloud domain. Using a novel compression algorithm, our system avoids the need to send the bloom filter based index back to the user, reducing communication costs by over 95%. By using homomorphic

encryption for index files, a cloud service provider can not learn the contents of the data files, the index files or the search queries. Furthermore, it cannot discern patterns from incoming queries. Moreover, unlike most existing techniques, our proposed system supports similarity-based searching in addition to exact matching. Finally, by not relying on trapdoors, it allows end users to formulate arbitrary queries to search encrypted data.

REFERENCES

[1] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou, "Toward secure and dependable storage services in cloud computing," *Services Computing, IEEE Transactions on*, vol. 5, no. 2, pp. 220–232, 2012.

[2] Cloud Security Alliance, "The Treacherous Twelve: Cloud Computing Top Threats in 2016," https://cloudsecurityalliance.org/download/the-treacherous-twelve-cloud-computing-top-threats-in-2016/, 2012.

[3] Dawn Xiaodong Song, David Wagner, and Adrian Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of the IEEE Symposium on Security and Privacy, 2000 (S&P 2000)*.

[4] Eu-Jin Goh et al., "Secure indexes.," *IACR Cryptology ePrint Archive*, vol. 2003, pp. 216, 2003.

[5] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano, "Public key encryption with keyword search," in *Advances in Cryptology-Eurocrypt 2004*. Springer, 2004, pp. 506–522.

[6] Craig Gentry et al., "Fully homomorphic encryption using ideal lattices.," in *STOC*, 2009, vol. 9, pp. 169–178.

[7] Zeeshan Pervez, Ammar Ahmad Awan, Asad Masood Khattak, Sungyoung Lee, and Eui-Nam Huh, "Privacy-aware searching with oblivious term matching for cloud storage," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 538–560, 2013.

[8] Brent R Waters, Dirk Balfanz, Glenn Durfee, and Diana K Smetters, "Building an encrypted and searchable audit log.," in *NDSS*, 2004.

[9] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Infocom, 2010 proceedings IEEE*. Ieee, 2010.

[10] Sankar K Pal, Puneet Sardana, and Ankita Sardana, "Efficient search on encrypted data using bloom filter," in *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*. IEEE, 2014, pp. 412–416.

[11] M Odell and RC Russell, "The soundex coding system," *US Patents*, vol. 1261167, 1918.

[12] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu, "Efficient similarity search over encrypted data," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 2012.

[13] Pascal Paillier, "Public key cryptosystems based on composite degree residuosity classes," in *17th international conference on theory and application of cryptographic techniques*. Springer, 1999.

[14] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[15] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the VLDB Endowment*. VLDB Endowment, 2013, vol. 6.

[16] Martin Porter, "The Porter Stemming Algorithm," http://tartarus.org/martin/PorterStemmer/.