# Similarity-Based Encrypted Data Search in the Cloud

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

***Abstract*—Consumer data, such as documents, photos and videos, are increasingly being stored in the cloud. Much of this data is confidential and hence needs to be encrypted to protect it from intruders in case the cloud service provider gets compromised. 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, these techniques limit search capabilities either to a small set of pre-defined keywords or only allow exact keyword matching.**

**This paper addresses the problem of similarity-based search over outsourced encrypted data while ensuring end-to-end privacy. This enables users to search using keywords that only partially match the originally stored words. We implement partial matching using sliding window bloom filters and secure search over arbitrary keywords using homomorphic encryption. Crucially, by avoiding the need to send back the entire encrypted bloom filter index to the client, we reduce the cost of communicating results to the client by over 95%. Our results demonstrate that search queries with 2 to 10 keywords only cost $0.00001 to $0.00004 per 1000 similar requests.**

## I. Introduction

Cloud computing enables its subcscribers to use computing services over the Internet and provides tailored computing resources on-demand. Subscribers of cloud storage can outsource their data on public cloud servers without worrying about the availability and reliability of the data. It is then the cloud service provider's (CSP) 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 have included users' photos and videos getting stolen from compromised cloud servers resulting in severe breaches of privacy. Credit card numbers, dates of birth and medical records are other examples of user data whose privacy needs to be ensured.

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

Existing approaches for searching ovedddr encrypted data often rely on so-called "trapdoors". 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 [1] as a solution to this problem. However, homomorphic encryption used naively only allows searching for exact keyword matches over the encrypted data.

In this paper, we present a novel system for performing searches on encrypted data in the cloud. In addition to exact keyword matching, our system also supports similarity-based searching: finding documents with partially matching keywords rather than only those with exact keywords. Privacy and search capabilities are enabled using homomorphic encryption, while sliding window bloom filters allow us to search for partially matching keywords. Finally, we use a novel compression technique which allows us to avoid sending back 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.

The next section overviews related work in more detail. Section III presents a detailed description of our system and the algorithms it uses. Section IV demonstrates the benefits of using our approaches on a dataset of encrypted textual data. Section VI then concludes with a treatment of some implications and trade-offs of using our system.

## 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. [2] 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 [3] 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. [4] 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. [6] 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. [7] proposed an encrypted search scheme using bloom filters. They also used soundex coding [8] 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. [9] 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.** Zeeshan et al. [10] proposed an inverted index based encrypted data search scheme. They used homomorphic encryption [1] to provide end-to-end privacy. Only authorized users can execute queries using their personal proxy re-encryption keys in order to manipulate the index. They used a trusted third party for the ranking of searched results and query modelling. While this approach does not rely on trapdoors, it still supports only exact keyword matching.

Finally, CryptDB [11] and MONOMI [12] improve the performance of searching over encrypted data using a split client/server querying paradigm. 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 to various extents.

## III. System Design and Implementation

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

### A. System Model

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

### B. Threat Model

The data owner and end user are considered as trusted entities while CSP is considered as a non-trusted entity because it is maintained by an arbitrary third party. Data communication between end user and CSP should be considered as non-trusted as all requests are routed via the open Internet. The CSP hosts the encrypted data file F and the encrypted index I. As they are encrypted, the CSP cannot learn any information regarding 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.

| 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 |
| $\mathcal{I}$ | 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 |
| $\sigma_{pk}$ , $\sigma_{sk}$ | Homomorphic encryption public and private keys |
| $\mathcal{K}_S$ | Symmetric encryption and decryption key. It is used to encrypted index file name |
| $\mathcal{S}_q$ | Similarity score of a query |
| $\mathcal{R}_q$ | Compressed resultant term |
| $\mathcal{M}_q$ | Compressed resultant term |
| $\mathcal{T}_b$ , $\mathcal{O}_b$ | Number of 2s and 1ns in a resultant term |
| $\mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_n$ | Variables of a polynomial $\mathcal{P}$ where $n$ is the length of the bloom filter |
| $3^1, 3^2, \ldots, 3^{n-1}$ | Coefficients of a polynomial $\mathcal{P}$ where $n$ is the length of the bloom filter |

TABLE I
NOTATIONS USED IN MATHEMATICAL AND DESCRIPTIVE DETAILS

### C. Assumptions and Notations

We ignore key exchange mechanism between the data owner, CSP and the end user, 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.

### 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 mapped to the bloom filter. For example, suppose we have a word *cloud* and *window size = 2*. Now *cloud* will 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 high matching score 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 $\mathcal{I}_i$ in an index file $\mathcal{I}$ has the structure:

$$\mathcal{I}_i = \mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_n, f\omega, \mathcal{O}_b \tag{1}$$

where $\mathcal{O}_b$ is the number of 1's in the bloom filter.

### 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 $\sigma_{pk}$ of the Pascal Paillier algorithm. Note that this while 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 will match 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

**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 $I = \langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n \rangle$ 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**
  $\mathcal{K}\omega \leftarrow extractAllKeywords(\mathcal{F}_i)$;
  $\forall k\omega_i \in \mathcal{K}\omega$;
  **while** $k\omega_i \in \mathcal{K}\omega$ **do**
    $SBF \leftarrow createSWBF(k\omega_i)$;
    $f_i \leftarrow getKeywordFrequency(k\omega_i)$;
    $\mathcal{O}_b \leftarrow getOnes(\mathcal{BF}_i)$;
    $\forall bf_i \in SBF$;
    **while** $bf_i \in SBF$ **do**
      $\mathcal{BF}_i \leftarrow \mathcal{E}_h(\sigma_{pk}, bf_i)$;
      $IndexEntry \leftarrow IndexEntry, \mathcal{BF}_i$;
      $bf_i \leftarrow getNextBit(SBF)$;
    $IndexEntry \leftarrow IndexEntry, f_i, \mathcal{O}_b$;
    $\mathcal{I}_i \leftarrow writeToIndexFile(IndexEntry)$;
    $k\omega_i \leftarrow getNextKeyword(\mathcal{K})$;
  $\mathcal{F}_i \leftarrow getNextFile(C)$;
**return** $\mathcal{I}_i$;

difficult.

Fortunately, homomorphic encryption provides a way to perform mathematical operations over ciphertext. Since our bloom filter bits are encrypted using Pascal Paillier homomorphic encryption, we can just multiply their ciphertexts, which when decrypted, would yield 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 measure of the similarity 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 query have been added together, we can simply return their sums back to the client. The client can then decrypt the sums and compute similarity scores. While this straightforward approach works and is used by many existing systems [10], 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. In quantitative terms, the summed index entry for *all* keywords 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 Equation 2.

$$\mathcal{P} = 3^0 \cdot \mathcal{BF}_1 + 3^1 \cdot \mathcal{BF}_2 + 3^2 \cdot \mathcal{BF}_3 + \ldots + 3^{n-1} \cdot \mathcal{BF}_n \quad (2)$$

where $\mathcal{BF}_i$ represent variables for the polynomial $\mathcal{P}$ and $\mathcal{BF}_i \in \langle 0, 1, 2 \rangle$.

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

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

In this equation, $\mathcal{BF}_i$ are the Paillier sums of the corresponding bits of an index entry and the query and $3^{i-1}$ are constants. From Eq.3 we get a single compressed term $\mathcal{R}_q$ and its size will be the size of Pascal Paillier key size. This multiplication by constants and addition is again possible because our bloom filter bits 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. We omit the proof of correctness of the algorithm in the interest of space.

**Algorithm 2:** Index entry decompression

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

```
b ← 3 ;
i ← 0;
while R_q > 0 do
    M_q[i] ← log_b R_q ;
    R_q ← R_q − b^i ;
    i ← 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. Results

We now 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 [14] 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. The server was a Google App Engine B4 class instance with a 2.4 GHz CPU and 512 MB 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 both 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 size of each index entry will increase, which will result in a larger overall index file size. The increase in indexing time with increasing bloom filter size is shown in Figure 2

A larger Paillier key size can help strengthen system security. However, with increasing Paillier key size, index file size also increases proportionally, as shown in Figure 3.

### *B. Search Performance*

Search performance is evaluated in terms of data returned by a query, response time, CPU cycles used to process a search

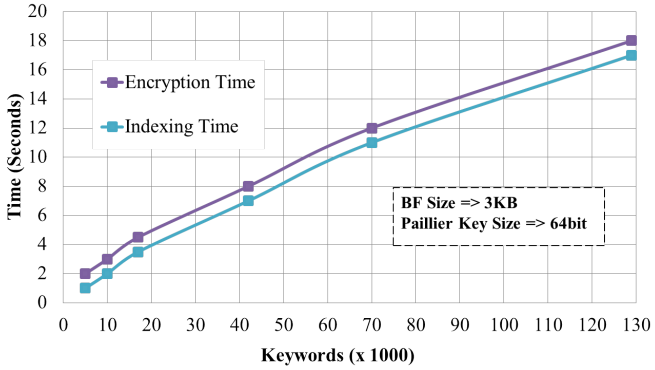


Fig. 1. Index Generation Time

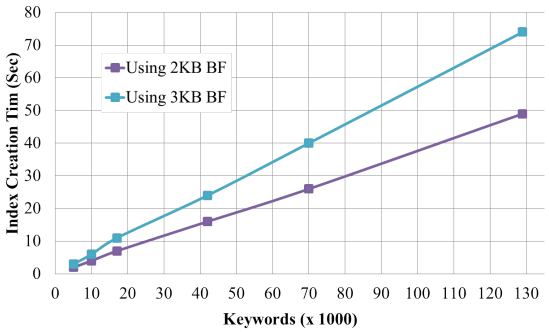


Fig. 2. Impact of Bloom Filter size on index generation time

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 96% less than traditional approaches and remains constant even when the size of the bloom filter increases. This is shown in Figure 4.

We also measure search performance in terms of response time to a search query. Table II show the results. As keyword count on the server increases, the response time increases only linearly showing that our approach scales reasonably well.

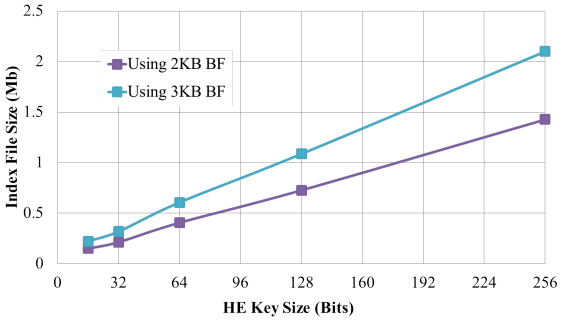


Fig. 3. Paillier Key Size Impact on Index File Size

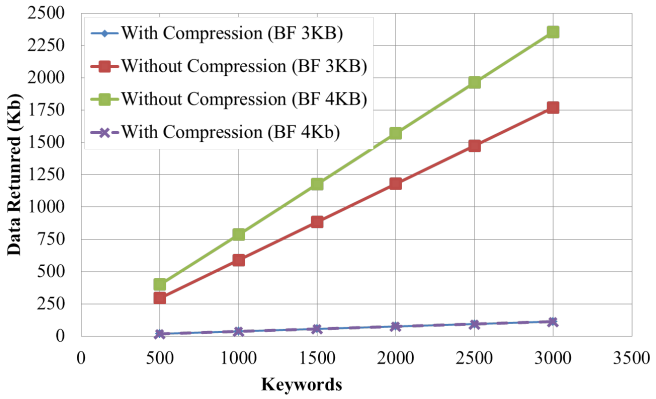


Fig. 4. Data returned by a search query: compressed vs uncompressed results

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

TABLE II
RESPONSE TIME INCREASES ALMOST LINEARLY WITH KEYWORD COUNT.

### C. Cost Estimation

Finally, we evaluate the cost($) that CSP will charge to data owners for search queries. Our results, shown in Figure 5, demonstrate that a query searching for a single keyword in a dataset having 500-3500 index entries will cost only $0.000002 to $0.00002 per 1000 similar queries. Furthermore, all of these metrics increase linearly with number of index entries. We obtained this cost from Google App Engine logs for each data point. In each log entry *ms*, *cpu_ms* and *cpm_usd* depicts response time, number of clock cycles used by CPU to process the request, and cost($) incurred for 1000 similar requests.

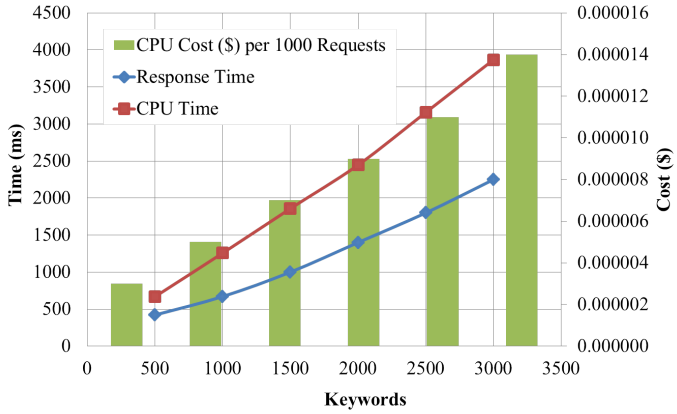


Fig. 5. Searching Cost($) with Single Keyword Query

## V. Future Work

While homomorphic encryption allows us to implement this system effectively, it is still much slower than symmetric and public-key encryption. In this paper, we have demonstrated and tested our system with a comparatively small dataset. With larger amounts of data, the indexer will likely take much more time to generate the encrypted index. One approach to handle this could be to use partition the dataset and use scalable systems like Hadoop [15] for indexing purposes. In addition, it would be useful to explore CryptDB and MONOMI as ways to improve the performance of both the indexer and the searcher, while sacrificing privacy to a minimal extent.

## VI. Conclusion

In this paper, we presented a privacy-aware similarity-based searching scheme over encrypted data residing in an untrusted cloud domain. 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. Since our system does not rely on trapdoors, it allows end users to formulate arbitrary queries and perform search. Finally, 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%.

## References


[1] Craig Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," https://www.cs.cmu.edu/ odonnell/hits09/gentry-homomorphic-encryption.pdf, 2009.

[2] Dawn Xiaodong Song, David Wagner, and Adrian Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000, pp. 44–55.

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

[4] Brent R Waters, Dirk Balfanz, Glenn Durfee, and Diana K Smetters, "Building an encrypted and searchable audit log.," in *NDSS*, 2004, vol. 4, pp. 5–6.

[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] 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, pp. 1–9.

[7] 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.

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

[9] 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, pp. 1156–1167.

[10] 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.

[11] 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, pp. 85–100.

[12] 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, pp. 289–300.

[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, p. 223238.

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

[15] Tom White, *Hadoop: The definitive guide*, " O'Reilly Media, Inc.", 2012.