Implementation and Evaluation of Improved Secure Index Scheme Using Standard and Counting Bloom Filters

Leyla Tekin, Serap Sahin

Izmir Institute of Technology, Faculty of Engineering, Department of Computer Engineering,
Gulbahce-35430, Urla, Izmir, Turkey. Tel: +90 232 750 7869. e-mail: {leylatekin, serapsahin}@iyte.edu.tr

Abstract—This paper presents an improved Secure Index scheme as a searchable symmetric encryption technique and provides a solution that enables a secure and efficient data storage and retrieval system. Secure Index scheme, conceived by Goh, is based on standard Bloom filters (SBFs). Knowledge of the limitations of SBFs, such as handling insertions but not deletions, helps in understanding the advantages of counting Bloom filters (CBFs). Thus, we have extended this scheme by adding a new algorithm so that CBFs can also be applicable. Unlike the old scheme, our scheme can handle dynamic update of a document by updating the existing index without rebuilding it. Moreover, we give a complementary comparison of both filters in our scheme. Finally, a detailed performance evaluation shows that our scheme exhibits similar performance with regard to the query overhead and the false positive probability and is quite efficient than the old scheme with regard to the update overhead by allocating more space.

Keywords—Searchable symmetric encryption; keyword search; indexes; bloom filters; dynamic update.

1. Introduction

In recent years, vast amounts of data are produced by several sources such as millions of digital sensors, social media applications, smart phones, financial transaction records etc. Thanks to many capabilities offered by cloud computing, data owners and organizations have extensively moved their huge datasets from traditional local data centers to the cloud so that they can utilize the possibilities of greater flexibility and lower cost. However, this requires to be kept their sensitive data on remote untrusted servers and introduces new security and privacy challenges that needs to be handled. Therefore, these data are encrypted before sending to the untrusted servers in order to protect the data confidentiality. Although data encryption ensures data confidentiality, it certainly prevents the server from operating on the data, especially searching over it.

The search functionality enables a data user to receive the related data with a keyword from a remote data server. The proposed solutions to perform a keyword search over the encrypted data are (i) downloading all the stored data to the user side, decrypt it locally and search the keyword over the decrypted data and (ii) allowing the server to decrypt the data and search the keyword, and return the related results to the user. The first approach downloads the entire data when a keyword search is performed, even if a very small part of the data

is related to the search keyword. Hence, it leads to an increase in communication overhead. The second approach allows the server to know the secret key and plaintext.

On the other hand, various searchable encryption schemes have been developed to support searching over encrypted data in a secure and efficient way. Tang [4] presents a systematic study on search in encrypted data. Three application scenarios, which have motivated a number of theoretical searchable encryption schemes, are described in that paper. These application scenarios are: (i) search in outsourced personal database, (ii) email routing service and (iii) matching in internet-based PHR (personal healthcare records) systems.

In the first scenario, there can be a user who may want to access her personal database anytime and anywhere. Thus, the user can outsource her database to a third-party service provider. To provide a privacy-preserving solution, the user can encrypt her database and outsource the ciphertext to the service provider. Then, she can send a search query to the service provider which search the query in the encrypted database and return the encrypted contents related to the search criteria. In the second scenario, there can be an email service provider which offers secure email service to its users. In this situation, a user can have all her mails encrypted using her public key which may be known by every entity. Later on, she can submit a search query to the service provider which search the query in encrypted emails and send back the interesting emails to the user. In the third scenario, an internet-based PHR system can allow users to store, access, edit and also share their PHRs. A PHR data of a user can have a lot of sources. Since PHRs are sensitive information, there should be a secure solution to meet the privacy problem. For this, the user can have her PHR data encrypted under her public key using an encrypted search scheme. Then, the user can authorize third-party servers to match her encrypted data.

In addition to the above application scenarios, two categorizations for search schemes over encrypted data are presented in [4]. The first categorization is based on whether a scheme supports full-domain or index-based search. In full-domain setting, a search will check every data item one by one for some criteria. In index-based search, the search criteria is tested based on index(es) rather than the contents of all data items. Furthermore, in terms of the second categorization, the schemes can be modeled using either symmetric or asymmetric setting. The first symmetric-setting scheme, proposed by Song et al. [5], allows only the client to create the searchable encrypted data and trapdoors. The first asymmetric-setting scheme, introduced by Boneh et al. [6], enables every entity to create the encrypted data, but only the client can generate valid search trapdoors.

The study in this paper matches the first application scenario explained above and focuses on a searchable symmetric encryption scheme which performs index-based search. We have chosen Secure Index scheme, developed in [2], to implement searches on encrypted documents. The scheme is based on (standard) Bloom filters (SBFs) that are fast probabilistic data structures for representation of a set in order to answer membership queries. However, Bloom filters do not support element deletions. Unlike Bloom filters, counting Bloom filters (CBFs) are able to support element additions and deletions dynamically. In this manner, we have proposed an improved scheme of Secure Index that can allow dynamic updates on documents without rebuilding operation.

The rest of the paper is organized as follows. Section 2 gives information about the research background on standard and counting Bloom filters.

Section 3 describes our enhanced Secure Index scheme. Section 4 presents the system algorithms. In Section 5, we point out performance evaluation. Finally, we discuss the related work and conclude the paper in section 6 and 7, respectively.

2. Background

In this section, we will provide a detailed background on standard and counting Bloom filters by describing the properties and operations of the filters, analyzing the mathematical model for false positive probability and deciding trade-offs between performance parameters.

2.1. Bloom Filters

Bloom filters are introduced by Burton Bloom in 1970 [1]. A Bloom filter is a fast probabilistic data structure that allows to test whether an element is a member of a set in a limited memory space. Although it is more space-efficient to represent a set than other data structures like linked lists, arrays, hash tables etc., it does not always produce 100% correct results. It can result in false positives that occur when it suggests that an element is in a set even if the element is not, but the bloom filter does not lead to false negatives. The basic Bloom filter supports two operations that involve adding elements to the set and querying for the membership of elements.

Now, let us look at the detailed description of a Bloom filter. The filter is a bit array of length m to represent a set $S = \{s_1, ..., s_n\}$ of n elements. It uses k distinct independent hash functions $h_1, ..., h_k$, each of them maps some element to the interval $[1, m]$ with a uniform random distribution. All bits are firstly set to 0. Then, to insert each element s_i in the set S, the array bits at positions $h_1(s_i), ..., h_k(s_i)$ are set to 1 on the bit array. Some bits can be set to 1 multiple times by coincidences for different elements. To test whether an element q belongs to the set S, the array bits corresponding to the positions $h_1(q), ..., h_k(q)$ are checked. If at least one bit is set to 0, then q is definitely not a member of S. However, if all the checked bits are set to 1, then q is a member of the set S with a high probability. This means that there is some probability of a false positive.

The false positive probability that occurs in a Bloom filter can be calculated under the assumption that a hash function chooses each array position with equal probability, as specified in [7]. Before quantifying the probability, some notations to be used are examined: m = the number of bits in the Bloom filter, n = the number of elements in the set, k = the number of hash functions, and fp = the false positive probability.

After defining the notations, now we will demonstrate how the false positive probability can be calculated. During the insertion of an element into the filter, the probability that a specific bit is set to 1 by a hash function is $(1/m)$. So, the probability that this specific bit is not set to 1 by a hash function is $(1-1/m)$. Since there are k hash functions, the probability of not setting the bit to 1 after all the hash functions are applied is $(1-1/m)^k$. After inserting all elements of the set into the filter, the probability that the bit is still 0 is $(1-1/m)^{kn}$. Therefore, the probability that the bit is 1 can be found as $1-(1-1/m)^{kn}$. A false positive can occur for an element that is not in the set if each of the k array positions obtained by the hash functions is 1. Hence, the false positive probability fp can be estimated, as in

$$fp = (1-(1-1/m)^{kn})^k \approx (1-e^{-kn/m})^k. \quad (1)$$

For fixed m and n, the value of k that minimizes fp can be computed by setting the derivative of the

equation with respect to k to 0, which gives the optimal value of $k_{opt} = \ln 2 * (m/n)$. So, using the optimal k, the false positive probability is $(1/2)^k \approx 0.6185^{m/n}$. The required array size m for the desired number of elements n and false positive probability fp is given by $m = -(n * \ln(fp))/(\ln 2)^2$. It means that for a fixed false positive probability fp, there is a linear relationship between the array size m and the number of inserted elements n.

As can be seen in the Eq. (1), fp varies according to three parameters: m, n and k. We test the mathematical formula of the false positive probability with different values for these parameters in order to see the behavior of the theoretical mathematical model, and draw the graphs shown in Fig. 1.

As a result, the variation of the fp with respect to the parameters can be illustrated in Fig. 1: (i) if m increases, fp decreases, (ii) if n increases, fp also increases and (iii) if k increases for fixed m and n, fp at first decreases, then reaches a minimum, then increases. Therefore, there exists a trade-off between three performance metrics which are computation time (corresponding to k), storage cost (corresponding to m) and probability of error (corresponding to fp).

2.2. *Counting Bloom Filters*

Standard Bloom filters do not allow element deletions by resetting ones back to zeros because there can be coincidences and a bit can be set by multiple elements. To address such a problem, Fan et al. [3] proposed counting Bloom filters in which an array of counters are used instead of bits. Initially, all counters are set to 0. When an element is inserted, the relevant counters are incremented and when an element is deleted, the relevant counters are decremented. To answer whether an element is contained in a set, check if all the counters corresponding to the hash functions are nonzero. In this context, a counter keeps track of the number of elements currently hashed to that location. The selection of counter size is also important to avoid counter overflow. According to the work in [7], four bits are enough for most applications.

The structure of a counting Bloom filter is similar to that of a standard Bloom filter. Hence, it can represent a set of n elements with m counters using k independent hash functions. Also, it can yield a false positive probability, which does not depend on the counter size, as in Eq. (1).

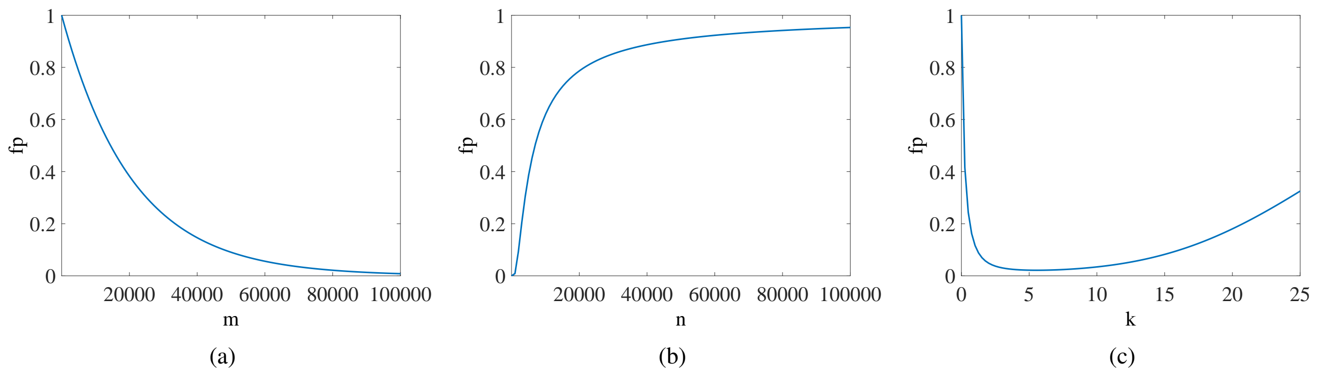

Fig. 1: The changes of fp with respect to m, n and k. (a) fp as a function of m. (b) fp as a function of n. (c) fp as a function of k using fixed m and n values. In (a) and (b), an optimal number of hash functions has been assumed.

3. Proposed Solution: Improved Secure Index Scheme

Secure Index scheme introduced in [2] consists of four algorithms such as Keygen, Trapdoor, BuildIndex and SearchIndex. The scheme uses (standard) Bloom filters to track words in documents. A Bloom filter represents a static set. Therefore, the scheme updates a document by regenerating the Bloom filter index of the document. On the other hand, a counting Bloom filter can represent a dynamic set. In this study, we add a new algorithm which is UpdateIndex to the scheme. This new algorithm uses counting Bloom filters, and so supports dynamic updates on documents more efficiently by just updating the existing counting Bloom filter index of the document. So, in the below algorithms other than UpdateIndex, both standard and counting Bloom filters can be used.

Keygen(s): The key generation algorithm takes a security parameter s and generates a pseudorandom function f and a master private key $K_{priv} = (k_1, \ldots, k_r)$.

Trapdoor(K_{priv}, w): This algorithm takes the master key K_{priv} and word w as input, and outputs the trapdoor for word w by calculating the r pseudorandom functions which are computed efficiently from the word and one part of the master key. So the trapdoor can be shown as:
$T_w = (f(w, k_1), \ldots, f(w, k_r))$.

BuildIndex(D, K_{priv}): The algorithm focuses on index generation. Given a document D including a unique identifier D_{id} and a list of words, and the master key, it generates an index for the document D_{id}.
The steps to create the index for the given document and master key are given below:
First, for each unique keyword w_i in the document:

1. The trapdoor is calculated using the Trapdoor(K_{priv}, w_i) algorithm, so the trapdoor is:
$T_{w_i} = (x_1 = f(w_i, k_1), \ldots, x_r = f(w_i, k_r))$.
2. Trapdoors are not directly inserted to the Bloom filter against correlation attacks. Therefore, the codeword C_{w_i} is calculated using the generated trapdoor and the identifier of the document, which ensures the creation of different codewords representing the same word for different documents. The codeword for w_i in document D_{id} is:
$C_{w_i} = (y_1 = f(D_{id}, x_1), \ldots, y_r = f(D_{id}, x_r))$.
3. The codeword $\{y_1, \ldots, y_r\}$ can be inserted into the Bloom filter of the document D_{id} by setting 1s to the bit positions corresponding to $\{y_1, \ldots, y_r\}$.

Next, the algorithm continues with the blinding the Bloom filter that starts by computing an upper bound u on the number of tokens in the document. The Goh's paper [2] suggests one token for every byte in the document after it has been encrypted.
Then, v is determined as the number of unique words in the document, and now the bloom filter is blinded by inserting $(u–v) * r$ random 1's. It equals to adding (u–v) random words into the filter, except for computing any pseudorandom function.
Finally, the index $I_D = (D_{id}, BF)$ is returned as the index for the document D_{id}.

SearchIndex(T_w, I_D): It takes the trapdoor $T_w = (x_1, \ldots, x_r)$ for word w and the index $I_D = (D_{id}, BF)$ for document D_{id}.
To test whether the document contains the keyword or not, the following steps are performed:

1. The codeword for word w is calculated using the given trapdoor and D_{id} in a similar manner as described above:
$C_{w_i} = (y_1 = f(D_{id}, x_1), \ldots, y_r = f(D_{id}, x_r))$.
2. Test if all bits at positions $\{y_1, \ldots, y_r\}$ in the

Bloom filter are set to 1.
3 If the Bloom filter's reply is positive, then output 1. Otherwise, output 0.

UpdateIndex(D, D', K_{priv}, I_D): The algorithm takes two versions of a document which are the previous version D and the updated version D', the master key and the index of the document.
This algorithm is valid when counting Bloom filters are used in the scheme. The steps to update the taken index are explained below:
Firstly, the counting Bloom filter CBF is obtained from the index.
Then, for each unique keyword w_i that is included in the previous version D, but not included in the updated version D' of the document:

1 The trapdoor is calculated with the Trapdoor(K_{priv}, w_i) algorithm.
2 The codeword is computed using the trapdoor and the document id.
3 The codeword is deleted from the filter.

Next, for each unique keyword w_i that is not included in the previous version D, but included in the updated version D' of the document, the trapdoor and codeword are calculated, and inserted into the filter.
As the last step, the index $I_{D'} = (D_{id}', CBF)$ is returned as the index for the document D_{id}'.

4. Improved Secure Index Applied To Encrypted Search

Until now, theoretical background on standard and counting Bloom filters are investigated, and our improved Secure Index scheme is given. Now, in this section, we will mention how the search system can be created. Actually, Goh [2] explained how Secure Index scheme can be applied to search on encrypted documents and described the system algorithms using the setup, search and update algorithms. But, our system has some differences, specifically in update algorithm. Then, our system consists of five algorithms: setup, search, add a document, delete a document and update a document.

Algorithm 1 – Setup
This algorithm is run on the user side to set up the system, in which either standard or counting Bloom filters can be used. The user has n documents which will be outsourced to the server. The algorithm consists of the following steps:

1 Firstly optimal Bloom filter parameters should be selected. Then, the user runs the Keygen(s) algorithm with the chosen parameters to get the pseudo-random function f and the master private key K_{priv}.
2 An integer $i \in [1, n]$ is associated with each document as its unique identifier.
3 An index is built for each document D_{id} by invoking the BuildIndex(D_{id}, K_{priv}) algorithm.
4 Each document is compressed and encrypted using standard encryption algorithms. Finally, the encrypted documents along with their indexes are uploaded to the server.

Algorithm 2 – Search
When the user wants to search the document collection stored on the server for the word y, the two steps are required as follow:

1 The user generates the trapdoor T_y using the Trapdoor(K_{priv}, y) algorithm and sends T_y to the server.
2 The server checks every index I_{D_i} by calling SearchIndex(T_y, I_{D_i}) algorithm to find all documents that contain the word y. Then, all matching documents are returned to the user.

Algorithm 3 – Add a document
If the user wants to add a new document to the document collection:

1 A unique identifier is assigned to this document.
2 Then, an index is built for this document by using the BuildIndex algorithm.

Algorithm 4 – Delete a document
The deletion algorithm includes:

1 Deleting the document and its index from the server.

Algorithm 5 – Update a document
When the user wants to update a document, the user takes the encrypted version of the related document from the server. Whether downloading the index for the document or not, depends on using standard or counting Bloom filters.
If standard Bloom filters are used, the steps to update the document are explained in detail below:

1 The encrypted document is decrypted and the document is updated.
2 A new index is created for this document with a new document identifier.
3 The document is encrypted again.
4 The new index and encrypted document are sent to the server.

If counting Bloom filters are used, the user also retrieves the counting bloom filter index of the related document from the server. The steps to update the document can be listed as follows:

1 The first step is similar to that of the standard Bloom filter.
2 The user has the previous and updated version of the document, thereby the UpdateIndex algorithm can be called to update the counting Bloom filter index.
3 The document is re-encrypted.
4 The updated index and the encrypted document are transmitted to the server.

5. Performance Analysis

We implement the system based on our enhanced scheme using java language on an Intel Core i5-2410M 2.30 GHz CPU with 4GB RAM running Windows 10 operating system. Both user and server operations are performed on the single machine so we do not consider latency that may occur in practice. We use HMAC-SHA1 for the keyed hash function, which has been suggested in the original scheme and AES-128 with CBC and PKCS5 padding for encryption.

To evaluate the proposed scheme, we mainly compare standard Bloom filters with counting Bloom filters in our scheme in terms of four performance metrics which are: (i) the false positive probability, (ii) the query overhead, (iii) the storage overhead and (iv) the update overhead. For this, all operations are performed in memory. Furthermore, operations such as encryption, which are the same for both the filters, are not taken into account in comparisons.

We conduct some experiments on a real data set of 500 RFC (Request for comments) [8] files that are numbered from 2001 to 2500 with a total size about 26 MB and some experiments on our own data set which are created from RFC files. The RFC file set includes a large number of technical and organizational keywords about the Internet and many of these keywords are unique to the file in which they are used.

We use Apache Lucene [9] to extract keywords from each RFC file by tokenizing the text, converting the characters to lowercase, removing stop-words and reducing words to a root form, namely stemming. Therefore, when a keyword is searched, initially all these operations are performed on this keyword, and then the corresponding trapdoor and codeword are computed.

In order to measure the performance of the certain

pieces of our code correctly, we utilize from Java Microbenchmark Harness (JMH) [10] which is a powerful tool to build, run and analyze micro-benchmarks. We write benchmark codes for our scenarios and execute them by specifying some parameters, such as the number of warmup and measurement iterations, the benchmark mode, and so on. Now, we will explain five cases detailedly below.

Case 1: Average query time (μs) against file size (or number of words in a document)

In this experiment, 3 files of different lengths, such as 100, 1000 and 10000 keywords, are derived from the keywords in the RFC files. The keyword "algorithm" is selected to search on the encrypted files. In both the Bloom filters, number of hash functions r is kept at 5. This search procedure is repeated 100 times and the average results are calculated.

Figure 2 gives information about how much average query time of two filter types in μs is spent for different file sizes. From the figure we demonstrate that SBF has almost the same average query time as CBF for all file sizes. This reason is that the number of hash functions in both of the filters is kept at the same value which is 5. We also see that file size does not affect the query time.

Case 2: Total query time (ms) against number of documents in the document set

For this experiment, we use 5 subsets of 20%, 40%, 60%, 80%, and 100% of the RFC files to show impact of the number of documents in the document set on the total query time. We choose the keyword "communicate" to search on the encrypted subsets of files. In the both Bloom filters, number of hash functions r is kept at 5 as in case 1. The search procedure is repeated 25 times and then total query times are computed.

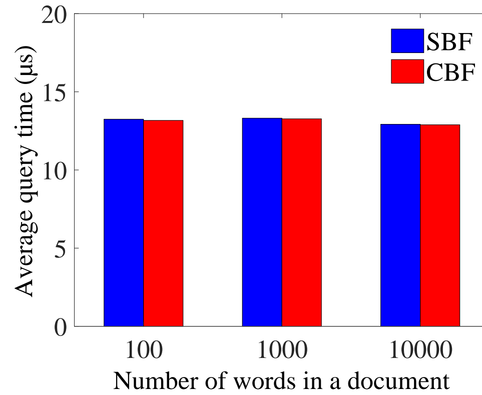

Fig. 2: Average query time (μs) vs. number of words in a document

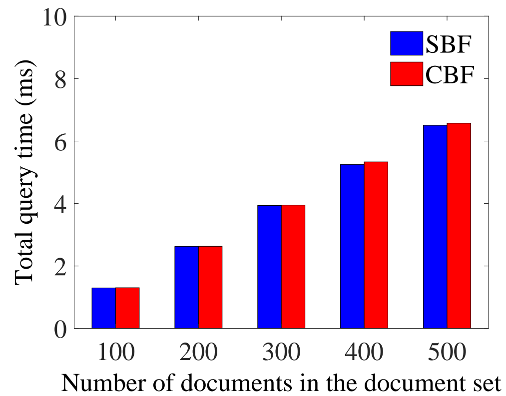

Fig. 3: Total query time (ms) vs. number of documents in the document set

Figure 3 illustrates total query time of the filters for different number of documents in the document set. According to the figure, SBF has almost the same total query time as CBF for different number of documents due to keeping the number of hash functions in both of the filters at a certain level. Moreover, as shown in the figure, the query time of the filters increases linearly with the number of documents.

Case 3: Update time (ms) against number of added words

In this experiment, 3 different-length files are derived from the keywords in the RFC files as in case

1. We add 1, 10, 20 and 50 different words to these files in order to measure the update overhead of the two filter types. This update procedure is repeated 100 times.

Figure 4 shows the update overhead of the filters when different number of words are added to the files in three cases of 100,1000 and 10000-word files. It can be seen that CBF outperforms SBF for all cases. We also demonstrate that the update overhead of SBF dramatically increases as file size increases.

Case 4: Update time (ms) against number of deleted words

In contrast to case 3, now we delete 1, 10, 20 and 50 different words from the created files in order to measure the update overhead of the two filter types. This procedure is also repeated 100 times.

Figure 5 depicts the update overhead of the filters when different number of words are deleted from the files in three cases of 100,1000 and 10000-word files. As in the case 3, it can be viewed from the Fig.5 that CBF outperforms SBF for all cases. Also, the update overhead of SBF grew more quickly as file size increases.

Case 5: Expected, currentAdded and currentDeleted false positive probability of CBF against number of added/deleted words

Figure 6 illustrates expected, currentAdded and currentDeleted false positive probability of CBF against number of added/deleted words to/from varying-

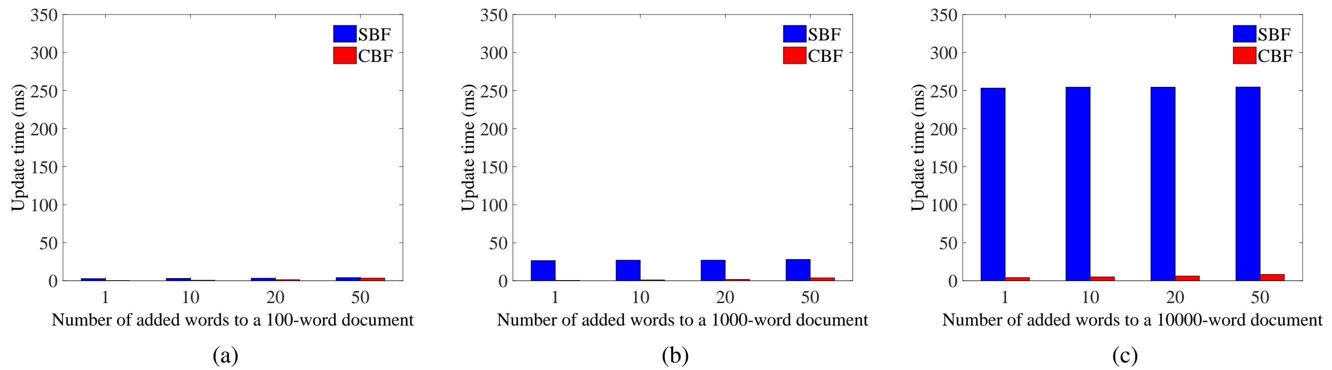

Fig. 4: Update time (ms) vs. number of added words for varying-length documents

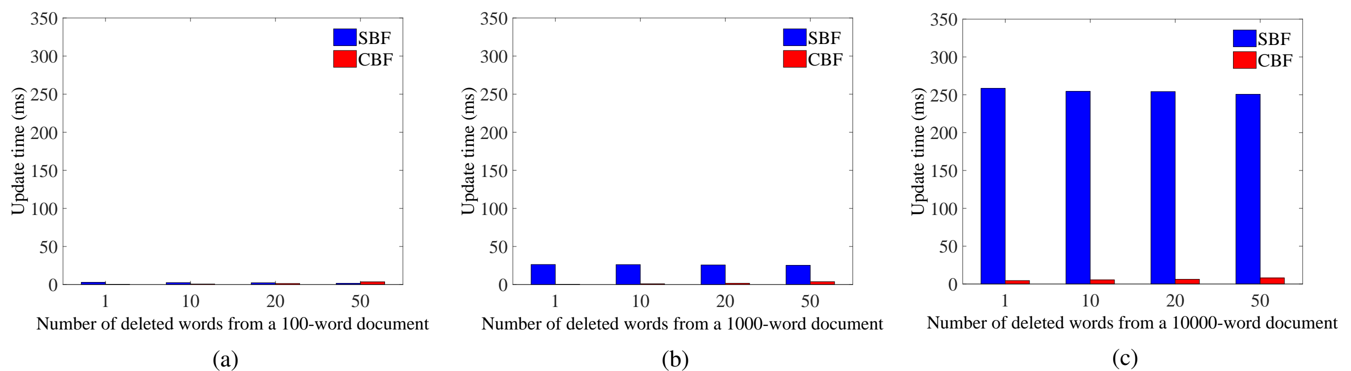

Fig. 5: Update time (ms) vs. number of deleted words for varying-length documents

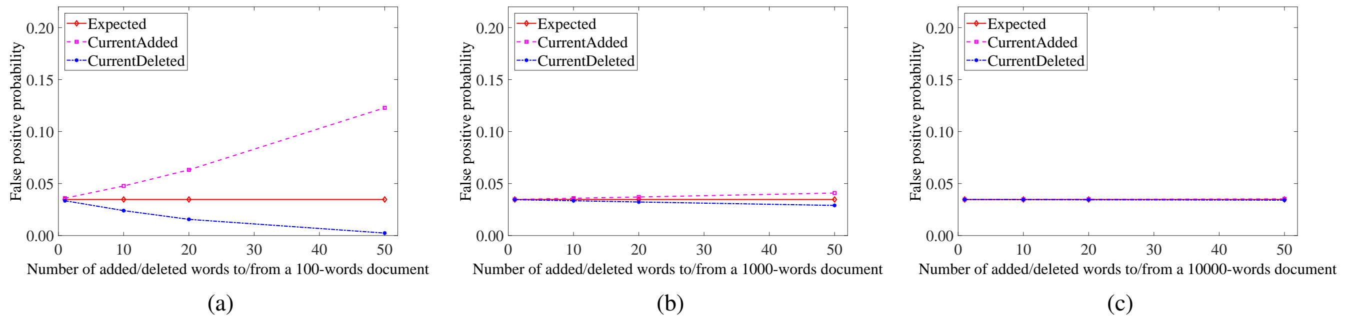

Fig. 6: False positive probability of CBF vs. number of added/deleted words to/from varying-length documents

length documents, such as 100, 1000 and 1000 words. According to the figure, false positive probability of CBF increases as words are added to the filter, and decreases as words are deleted from the filter. Additionally, the probability changes much more rapidly in the small documents.

6. Discussion

Dynamic Searchable Symmetric Encryption schemes allow the document collection to be modified after setup phase. Chang and Mitzenmacher [11] proposed two schemes in which a keyword dictionary can be stored or not stored on the mobile device of the user. A masked index string is created for each document in their approach, and so the search time is linear in the number of documents as in this paper. Also, they studied secure updating of the documents. In their approach, deletion of a document along with its encrypted index is simple, but updating a document is required to delete this document with its encrypted index, and then building a new encrypted index for a new document. Whereas, in our approach updating a document can be performed by only updating the corresponding existing index of it.

The dynamic SSE schemes [12, 13, 14] are based on inverted-index so for each unique keyword a searchable index is generated. [13] and [14] support the ability to add and delete documents efficiently, however they do not take into account updating the contents of documents. If a document is added, the user will generate the add token for this document by producing values for each unique keyword in it, and then send the add token to the server which will update the encrypted index. If a document is deleted, this time the user will create a delete token and send it to the server which will update the encrypted index. [12] enables only adding new documents to the document collection as updates. In this scheme, search is logarithmic in the number of keywords, but the size of the encrypted index is large and the number of updates supported is limited.

As a result, although our improved scheme has linear search time and IND2-CKA security model which provides security if search queries are independent of the previous queries, it is efficient in terms of update time.

7. Conclusion

This paper suggests an improved Secure Index scheme to perform searches and updates on encrypted documents. The old scheme supports updating a document but it requires to rebuild the

standard Bloom filter index of the document. On the other hand, our scheme can handle updating a document by only updating the counting Bloom filter index. Then, we implement our scheme with standard and counting Bloom filters, and compare the performance of the filters regarding different metrics. Comprehensive experiments demonstrate that the proposed scheme performs better in terms of the update overhead by achieving the same accuracy and almost the same query overhead, and using slightly larger space. Especially if large documents are used, update operation takes much less time.

Acknowledgments

This work was presented at the ISCTurkey 2017 Conference.

References

[1] 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.

[2] E.-J. Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216. http://eprint.iacr.org/2003/216, 2003.

[3] 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 (TON)*, Vol.8, No.3, pp.281-293, 2000.

[4] Q. Tang. Search in Encrypted Data: Theoretical Models and Practical Applications. Cryptology ePrint Archive, Report 2012/648. http://eprint.iacr.org/2012/648, 2012.

[5] D.X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data", *IEEE Symposium on Security and Privacy*, pp.44-55, 2000.

[6] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. *Public key encryption with keyword search*. Eurocrypt, Vol.3027, pp.506-522, 2004.

[7] S. Tarkoma, C.E. Rothenberg, E. Lagerspetz, "Theory and practice of bloom filters for distributed systems", *IEEE Communications Surveys and Tutorials*, Vol.14, No.1, pp.131-155, 2012.

[8] http://www.ietf.org/rfc.html, RFC, Request For Comments Database, Latest Access Time is 14 July 2017.

[9] https://lucene.apache.org, ApacheLucene, Latest Access Time is 30 June 2017.

[10] http://openjdk.java.net/projects/code-tools/jmh/, JMH, Java Microbenchmark Harness, Latest Access Time is 27 August 2017.

[11] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data", *ACNS*, Vol.5, pp.442-455, 2005.

[12] P. van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker, "Computationally Efficient Searchable Symmetric Encryption", *Secure data management*, Vol.6358, pp.87-100, 2010.

[13] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption", *ACM Conference on Computer and Communications Security*, pp.965-976, 2012.

[14] R. Ramasamy, S.S. Vivek, P. George, and B.S.R. Kshatriya. Dynamic Verifiable Encrypted Keyword Search Using Bitmap Index and Homomorphic MAC. Cryptology ePrint Archive, Report 2017/676. http://eprint.iacr.org/2017/676, 2017.