Q-GRAM BASED ENCRYPTED CODEWORD DICTIONARY FOR FAST SEARCHES OVER A LARGE COLLECTION OF ENCRYPTED UNSTRUCTURED DOCUMENTS

Mohammed Afzal Khan[1]and Shoab Ahmad Khan[2]

[1, 2] Electrical & Computer Engineering Department, Center of Advance Studies in Engineering, Islamabad, Pakistan

Email: [1]afzalmkhan@yahoo.com, [2]Shoab@carepvtltd.com

ABSTRACT

With the advent of cloud computing, many businesses prefer to store their unstructured documents over the cloud. The preference is to store the encrypted unstructured document over the cloud for security. In most of these instances, one of the main criteria is to support fast searches without requiring any form of decryption. It is thus important to develop methods and architectures that can perform fast searches without compromising security and return the rank results for a client query. Our technique uses the enhanced version of the symmetric encryption algorithm for unstructured documents and develops a novel secure searchable hierarchical in-memory indexing scheme for each encrypted document using multiple Bloom filters and construct a dictionary over a large collection of encrypted unstructured documents. The paper also proposes a dynamic index construction method based on hierarchical in-memory index to perform fast and parallel rank searches over a large collection of encrypted unstructured documents. To the best of our knowledge, this is a novel contribution that propose methodology of constructing a dictionary using hierarchical in-memory index for performing fast and parallel rank searches over a large collection of encrypted unstructured documents. We introduce the concept of Q-gram for building the encrypted searchable index, and provide multiple Bloom filters for a given encrypted unstructured document or a chunk to build encrypted searchable indexes using separate Bloom filter for a set of bytes. Our proposed construction enables fast rank searches over encrypted unstructured documents. A detailed study of 44 billion code-words is worked out using off the shelf serves to demonstrate the effectiveness of Layer Indexing method.

Keywords: Cloud computing, cloud security, searchable encryption, ranked search, encrypted Bloom filters

1.0 INTRODUCTION

The explosive growth of cloud computing provides many organizations a way of storing their information in the clouds and utilizing the cheap computing resources effectively. Many businesses want to ensurethat information stored in clouds remains confidential and not easily accessible to non-privileged users. The preference is to store the encrypted unstructured documents over the cloud. The businesses also want to run searches on encrypted unstructured document databases in a fast and effective manner, is an active area of research among the researchers.

In the literature, searchable symmetric encryption [14] is a helpful technique that allows the user to perform queries over the encrypted documents. However, a faster search over a large collection of encrypted unstructured documents is not possible. There are two high-level approaches to design reasonably efficient and secure schemes using the searchable symmetric encryption; dynamic and static. In Dynamic Searchable Symmetric Encryption (DSSE) [14, 15, 18, 19, 20, 24], encrypted keyword searches should be supported even after the documents are randomly inserted or deleted from the collections. In this approach, one of the first notable papers by Goh presented an algorithm for a document index construction based on the secure word [1, 2, 15]. Goh also presented a technique to perform searches on these secure word indexes without having to decrypt the complete document. This scheme provides linear time search over a document collection.

The second approach is Static Searchable Symmetric Encryption (SSSE) [16, 17, 22, 23], work on static data i.e., there is a setup phase that produces an encrypted index for a set of collection of documents and after that phase no insertion or deletion of documents is supported. In static approach, first notable paper by Curtmola et al. [17]

presented a Searchable Symmetric Encryption (SSE) scheme [17]. The SSE scheme executes in sub-linear time. The formation of the dictionary is through a combination of the lookup table and array. Curtmola et al. [17] first gathers all unique words from a set of documents and then creates the link list using the list of document identifiers for each unique word. The link list for the distinct words is constructed, and then the list is encrypted, flattened and scrambled into an array. Each element is packaged with the key that is used to encrypt the next element in the list. This allows the server to find all the document identifiers from a given word and a key.

The Dynamic schemes based on the index are used in most recent research publications due to its efficiency [14, 18, 19, 20, 24]. In DSSE schemes, there are four areas that require further attention:

1. How to improve search time to make it practical over a large collection of document?
2. What is the computational complexity of searches? Is it practical in multi-user environment?
3. How well your index scheme supports fast rank searches over a collection of encrypted unstructured documents?
4. Lastly, how well your index scheme supports security to minimize *leakage*?

Before we dwell into other discussions lets briefly summarized ideal security environment. Ideally, the DSSE considers secure where there is no partial leakage of information and they are characterized against three parameters:

Table 1: Comparison of Several DSSE Schemes

Scheme	Dynamic	Security -leakage	Search time	Indexing method
Goh [15]	yes	N/A	$O(n/p)$	N/A
Liesdonk et al. [20]	yes	similar	$O(n)$	-
Kamara et al. [19]	yes	similar	$O(r)$	hashing plus link list
Kamara and Papamanthou [18]	yes	similar	$O((r/p)logn)$	hashing plus tree based
Emil et al. [24]	yes	better	$O(min\{\alpha + \log n, rlog^3 n\}))$	hashing using multi-levels
Proposed method	yes	similar	$O(1/plog(\mathcal{W}/\mathfrak{Y}))$	layers indexing (in-memory) plus tier's for ranking

Table 1 compares our construction with the previous DSSE schemes. In Table 1, n is the size of the document collection, r is the number of documents containing keyword $\mathfrak{w}$, p is the number of processors, $\alpha$is the number of times historically the document is added, $\mathcal{W} = \sum\mathfrak{w}$number of words in n collection of documents and $\mathfrak{Y}$ is the number of nodes in a page.

i. The hash of keywords that we used for searches are referred to as search pattern in the literature [21, 24].
ii. The matching document identifiers of a keyword, addition, deletion operations referred to as access pattern in the literature [21, 24].
iii. The number of document in the collection is referred to as size pattern [24].

The above DSSE security parameters are further described and analyzed in the next sections.

DSSE [14, 15, 18, 19, 20, 24] and SSSE [16, 17, 22, 23] schemes allow the client to perform the secure searches over the encrypted data using the search patterns constructed from keywords. All these schemes do not provide any relevance of the documents in their search results. Although [6, 25] provides a methodology to perform relevant searches over encrypted data, both of them fall in the category of static (SSSE) scheme.

The relevance of the document in the search result is important when applied to cloud environment due to two main reasons. Firstly, without relevance of the documents, client needs to go through the entire encrypted documents set against a keyword to find documents of interest after decrypting and analyzing these documents that involve a large amount of post processing overhead. Secondly, invariably sending back all the documents

creates large unnecessary traffic that may not be desirable in the cloud environment due to pay-as-you-use cloud paradigm [6].

Technical Highlights: In this paper, we focused on the problem of constructing DSSE scheme for the purpose of designing fast searchable cloud storage systems to perform searches on encrypted unstructured documents. Some of the technical highlights are:

1. We enhanced the Goh [15] algorithm for building index over a large collection of encrypted unstructured documents using separate Bloom filter (Bloom filter is defined in Section 2) for set of bytes. We improve the search time by describing new method to locate Bloom filter for set of bytes. We further minimize the probability of Bloom filter saturation.
2. Our main contribution is a new dynamic hierarchical in-memory index construction method to perform fast searches over a large collection of encrypted unstructured documents. Searches for keywords runs in $O(log(\mathcal{W}/\mathfrak{Y})$ sub-linear time, where $\mathcal{W} = \sum\mathfrak{w}$ is number of words in n collection of documents, $\mathfrak{Y}$ is the number of nodes in a page.
3. In multi-user environment, searches can be performed in parallel time using multiple processors and the search time is $O(1/plog(\mathcal{W}/\mathfrak{Y})$, where p is the number of processors.
4. We provide relevant results in ranked order for searches. No other dynamic algorithms in our knowledge [14, 15, 18, 19, 20, 24]returns ranked result on client queries.

Related work: Song et al. [14] were the first to propose the Dynamic Searchable Symmetric Encryption and their index searches are linear to the size of data collection. Their constructions provide the addition or deletion of documents in a simple way. Goh [15] address the Dynamic Searchable Symmetric Encryption using the Bloom filter, but Goh construction does not use the dictionary. Goh scheme has the linear search time and results in false positive. Kamara et al. [19] constructed a DSSE scheme that has sub-linear search time. Their construction uses the hash dictionary with complex and difficult list structures. Their construction leaks information for keywords during update operations. The work of Kamara and Papamanthou [18] improves the dictionary construction using the hash dictionary with the KRB tree structure. Their construction requires sub-linear search time and also utilizes the parallel searching, increasing the space requirement for their data structures. Liesdonk et al. [20] were the first to explicitly present the Dynamic Searchable Symmetric Encryption and in worst case has linear search time. Their construction supports a limited number of update operations. Emil et al. [24] proposed DSSE scheme based on constructing dictionary from hash using multiple levels instead of a flat hash table. Their construction achieve better security (called forward privacy) using fresh key for encrypting the entries for new level and thus utilizing new tokens for every new level.

Swaminathan et al. [25] propose confidentially preserving rank order search. Their paper looks at practically building a ranking system into a secure index based search scheme. Their scheme defines a method for ranking documents based on relevance, which is similar to SSSE. Wang et al. [6] define a method for adding new scores for newly created documents or modification of old scores of existing documents in the collection. Their scheme incorporated random size non-overlapping buckets where within each bucket all the document with the same rank are arbitrarily placed.

2.0 NOTATION AND PRELIMINARIES

We use $x \leftarrow S$ to denote random variable x that is drawn uniformly at random from the set S. We denote $x \leftarrow [1, N]$ to denote a random variable $x$is chosen uniformly from the set of integers in $[1, N]$. We denote $x \leftarrow X$ is the output of the algorithm.

We are using different types of data structures, including link lists, arrays and hierarchical index. We give name of our hierarchical index as *layers indexing*. Assume that$l$ is the list andl_k denotes total number of layers that contains the list. The array is presented through A and total number of cells in the array is presented as$\sum A$. A layers *indexing* consists of multiple layers $(l_1, \ldots, l_k)$ and leaf tiers $(t_1, \ldots, t_j)$. Layers $(l_1, \ldots, l_k)$ are in the external memory and leaf tiers $(t_1, \ldots, t_j)$are in the disk. Each layer of $(l_1, \ldots, l_k)$ is made of multiple pages, where each page size is equal to the array size $\sum A$. We further define a *layer node* in a page that store structure at location i $\in [\sum A]$as$A[i]$ and $A[i] := (s_1, \ldots, s_m)$ represent the operation that stores s at location $i$in A. Similarly, a *leaf node* in a page that store structures at location i $\in [\sum A]$as$A[i]$ and $A[i] := (d_1, \ldots, d_m)$ represent operation that stores d at location $i$in A. Each *leaf node* represents a leaf tier t_j and multiple tiers $(t_1, \ldots, t_j)$ are connected together through a link list.

Let W denotes the universe of words. If $d = (w_1, \ldots, w_m) \in W^n$ is a document, then $\sum d$ denotes its total number of words and its bit length is $|d|$.We also assume that d is the document that has the distinct list of words. We use {} to denote string concatenation. The data can be seen as a sequence of $n$documents $\boldsymbol{d} = (d_1, \ldots, d_n)$, where document d_i is a sequence of words $(w_1, \ldots, w_m)$ from a universe W. We assume that each document has a unique identifier$I = (d_i)$.

Let $c = (c_1, \ldots, c_n)$ is a set of encryptions of the documents in $\boldsymbol{d}$. If $d = (w_1, \ldots, w_m) \in W^n$ then these words are represented in equivalent secure form as *codewords*$d_{cw} = (cw_1, \ldots, cw_m) \in W^n$. The sequence of n codeword documents is $\boldsymbol{d_{cw}} = (d_{\mathrm{cw}\,1}, \ldots, d_{cwn})$. We define *binary codeword* as the converted form of *codeword* in binary. We can delete or add or update document at any time as we are working with dynamic data using *codeword*. Given a keyword w, its equivalent *codeword* is cw. To add a document d, the client generates an add *codeword*cw_a and given cw_a and encrypted index γ, the provider can update the encrypted index γ. Similarly, for the delete *codeword*cw_d, document can be deleted from the database.

Definition 1 *(Layers Indexing Scheme):Our construction consists of the following eleven algorithms:*

$K \leftarrow KeyGen(s)$*: Given the security parameter s, outputs the secret key* K.

$(d_{cw}, c) \leftarrow DocEnc(K, d)$*: is an algorithm that takes as input secret key* K*, keywords in a document* d*. It outputs the list of codewords in document* d_{cw}*, and a sequence of ciphertexts* c.

$\Upsilon \leftarrow BuildIndex(d_{cw})$*: is an algorithm that takes as input the list of codewords in document* d_{cw}*. It outputs the layers index* Υ *that contains the list of codewords.*

$cw_s \leftarrow SrchCodeword(K, w)$*: is an algorithm that takes as input a secret key* K *and a keyword* w*. It outputs a search codeword* cw_s.

$nodeAdd_{cw} \leftarrow AddCodeword(d_{cw}, codewordIndex)$*: is an algorithm that takes as input document* d_{cw} *that contains list of codewords and the index number for a particular codeword. It outputs a node* $nodeAdd_{cw}$ *that contains the (codeword, doc id, doc size) for addition.*

$nodeDel_{cw} \leftarrow DelCodeword(d_{cw}, codewordIndex)$*: is an algorithm that takes as input document* d_{cw} *that contains list of codewords and the index number for the codeword. It outputs a node* $nodeDel_{cw}$ *that contains the (codeword, doc id, doc size) for deletion.*

$LeafNode \leftarrow Search(\Upsilon, cw_s, tierEnable, tierAdress)$*: is a deterministic algorithm that takes as input an encrypted layers index* Υ*, a search codeword* cw_s*, a leaf tier enable bit* $tierEnable$*, and the next tier adddress*$tierAdress$ *. It outputs a leaf tier node* $LeafNode$.

$R_{cw} \leftarrow Rank(n, df_w, wf_{w,d})$*: is a deterministic algorithm that takes as input total number of documents in the collection* n,df_w *number of documents in the collection that contains the word* w, $wf_{w,d}$ *is the word frequency. It outputs the ranking number of codeword*R_{cw}.

$\Upsilon' \leftarrow Add(\Upsilon, R_{cw}, nodeAdd_{cw})$*: is a deterministic algorithm that takes as input an encrypted layers index* Υ*,ranking number of codeword* R_{cw} *and a node* $nodeAdd_{cw}$ *that contains the (codeword, doc id, doc size) for addition.It outputs new encrypted layers index* Υ' *that contains the new list of codewords.*

$node_{Del} \leftarrow Del(\Upsilon, nodeDel_{cw})$*: is a deterministic algorithm that takes as input an encrypted layers index* Υ *and a node* $nodeDel_{cw}$ *that contains the (codeword, doc id, doc size) for deletion. It outputs a node* $node_{Del}$ *that has the delete structure.*

$d \leftarrow Dec(K, c)$*: is an algorithm that takes as input secret key* K*and a ciphertext c and outputs a* $document$ d.

A dynamic SSE scheme is correct for all keys generated by $KeyGen(s)$ for all d for all (d_{cw}, c) output by $DocEnc(K, d)$ and for all sequences of add, delete, update or search operations on Υ, search always return the correct set of ranked documents.

Q-gram: A q-gram is a contiguous sequence of 'q' bytes from a given sequence of text. A q-gram of two means that a consecutive sequence of two bytes, similarly, q-gram of sixteen means sixteen consecutive sequences of sixteen bytes. Queries are solved by intersecting/joining the lists of q-gram occurrences depending on whether the query pattern is longer/shorter than q. Q-gram also used for approximate searches.

Binary Codeword Collision: We transform the words into the equivalent binary codewords, and during this transformation, there is a probability that the different words can transform into the same binary codeword. This condition of transformation from multiple words into a single binary codewordis referred as collision.

Bloom filter: Bloom filter is a data structure that is used for fast set membership test that is stored as an array of bits. All the bits of the array are initialized to zeros. Hashes are performed when words are added to the set. The input to each hash function is the word that needed to be added and output from each hash function is the bits to be added to the array that is different for each hash. After hashes are calculated, each bit in the filter at the indexes specified by the hash outputs set to one. To test a word for membership, it is hashed using the same algorithms and the resulting bits are checked against array bits to see if it's set to one. One problem is the possibility of saturation of Bloom filter [1, 2]. In Bloom filter false positive can occur, when the bits specified by the hash outputs are already set to one, even though the word was not added originally.

Ranking: The ranking functions are used in information retrieval to calculate the scores of documents against a given client query. Most commonly used rule for relevance scoring is$wf * idf$, where the wf (word frequency) is the number of times words appear in a document and idf (inverse document frequency) is specified through the total number of documents divided by the number of documents that contains the given word. Among several hundred variations of $wf * idf$ no single combination can out performs the others [6]. So, we will use the formula originally specified by Luhn [26, 27] and Spark Jones [28] that is specified for single word client query searches in (see chapter 6 in [3]):

$$wf - idf_{w,d} = wf_{w,d} * log \frac{n}{df_w} \qquad (1)$$

Where n is the total number of documents, df_wnumber of documents in the collection that contains the word w, $wf_{w,d}$is the word frequency. The above equation assigns to word w a weight in document d.

Security: In Dynamic Symmetric Search Engine (DSSE), we are concerned with, firstly, given an encrypted index γ and ciphertext c, no partial information is leaked to the adversary (server) for document d. Secondly, given a set of adaptive sequence of keywords $w_1, \ldots, w_m$ and corresponding codewords$cw_1, \ldots, cw_m$, no partial information is leaked to the adversary (server) for either document d or keywords w [17, 18]. We achieve adaptive security without using the random oracle model because the server is not doing any decryption [31]. The two conditions mentioned provide ideal conditions for the DSSE construction, but realistically DSSE scheme [14, 18, 19, 20, 24] leaks some limited information about messages and query to the adversary. Therefore, we weaken our security definition and define a *leakage function*, we define *leakage* for DSSE constructions using the definitions given in [17, 18, 14, 24] to determine what is being leaked for the ciphertext and codeword:

Definition 2$(L_{sz}, n, sz) \leftarrow L_1(\Upsilon, d_{cw}, \boldsymbol{c})$: *Given an index Υ and set of ciphertext documents c along with the list of codewords in document d_{cw}, this function outputs the size of the layers indexing L_{sz}, total number of ciphertext documents n and size of each document sz.*

Definition 3 $(cw_s, ap) \leftarrow L_2(\Upsilon, d, w, t)$: *During search phase we will reveal search pattern for a query keyword w. We will define leakage as given an index Υ and set of ciphertexts c for set of documents, codeword cw for a keyword w and time t, this function outputs the search pattern cw_s and access pattern ap. Where search pattern is the binary codeword at time t, for a query keyword w. Search pattern reveals that searches being performed in past with same search pattern. Where access pattern reveals how we are accessing set of documents for a keyword w, at time t.*

Definition 4 $(\Upsilon') \leftarrow L_3(nodeAdd_{cw}, nodeDel_{cw})$*: During add or delete phase of the documents we will reveal the codewordscw that are being added, updated or deleted in Layers Indexing and the corresponding ciphertext for these documents. We will define leakage as given a node $nodeAdd_{cw}$ for addition or a node $nodeDel_{cw}$for deletion in an index Υ at a time t, this function outputs the new index Υ', Where node ($nodeAdd_{cw}$, $nodeDel_{cw}$) contains the (codeword, doc id, doc size).*

We define security using the standard security definition that is described in the [24, 29]. We present the scheme is secure in the semi-honest model where the adversary faithfully follows the given protocol:

Ideal world execution (F, S): *We will devise an experiment in the simulation environment. We will have a client that sends an ideal functionality F a "setup" message to an adversary (also referred to as simulator) S with a Layers Indexing and set of ciphtertext for documents.*

In each time-step, the client sends the search, addition or deletion operations to the adversary S. For search operation client sends the search pattern cw_s for a corresponding keyword and the adversary S sends back the corresponding leaf tier node $LeafNode$. For addition or deletion operations, client sends the $nodeAdd_{cw}$ or $nodeDel_{cw}$ for a corresponding keyword w at a time t and the adversary S adds or deletes the corresponding codeword in the Layers Indexing. $FnotifiesS$ of leakage L_1, L_2 and L_3. Finally client outputs a bit.

Real world execution($\prod_F A$): *We will devise an experiment in real environment. We will have a client that send a "setup" message to an adversary A with a Layers Indexing and set of ciphertext for documents.*

In each time-step, the client sends the search, addition or deletion operations to the adversary A. For search operation client send the search pattern cw_s for a corresponding keyword w and the adversary A sends back the corresponding leaf tier node $LeafNode$. For addition or deletion operations, client sends the $nodeAdd_{cw}$ or $nodeDel_{cw}$ for a corresponding keyword w at a time t and the adversary A adds or deletes the corresponding codeword in the Layers Indexing. Finally client outputs a bit.

In probabilistic, polynomial-time semi-honest model a real-world adversary A, there exists a simulator S, such that for all non-uniform, polynomial-time environments, there exists a negligible function $negl(\lambda)$ such that

$$[P_r\left[Real_{\prod_F,A}(\lambda) = 1\right] - P_r\left[Ideal_{F,S}(\lambda) = 1\right] \leq negl(\lambda) \qquad (2)$$

The definition covers correctness and privacy. Correctness is covered because the ideal-world client either receives the correct answer of the query, or receives an abort. Privacy is covered because the ideal-world adversary has no knowledge of the client's queries other than the leakages defined in 2, 3 and 4.

3.0 METHODOLOGY AND CONSTRUCTION

Q-gram Size Selection: The size of the q-gram is driven, based on underlying system requirements. The q-gram size will mostly be driven by two factors namely, the size of the desired storage system and the collision avoidance in the binary codewords. The most important factors of these two are the collision avoidance in binary codeword.

Each extracted distinct word w_m from document d is converted to required q-gram word. If the keyword w_m is larger than the selected q-gram size than truncate the bytes that are larger than the required q-gram size. However, if the keyword w_m is smaller than the q-gram size than remaining bytes are zeros. In our construction, the q-gram word is further converted into the codeword and then into the binary codeword.

The bigger size of binary codeword lowers the probability of collision in the binary codeword. If we choose, for example, the q-gram size of thirty two bytes than we will have the binary codeword size of 256-bits that means we have collision probability of one in every 2^{128} binary codeword. Using the q-gram of thirty two bytes also provides several terabytes of storage that is suitable for most storage systems. We are focusing on the codeword searches any codeword that is greater than thirty-two, q-grams will be truncated instead of using the multiple q-grams for codewords.

Goh Algorithm Selection: Goh's algorithm [15] is used to generate only the Layers Indexing and the documents are encrypted using the CPA-secure private-key encryption scheme [Please refer to [30] for definition of CPA-security].

Goh algorithm [15] was chosen because it is based on Hash function and Bloom filter. One of the requirements of the Layer Indexing is to minimize the probability of collision (collision-resistance) to enhance overall in-memory search performance. But the Goh's algorithm has false positive and results in filter saturation for Bloom Filter. Both of these shortcomings are minimized in the enhanced Goh Algorithm described below.

Bloom filters have the following property; the time needed to add a query word or to check whether a query word is in the set is a fixed constant. In our enhanced Goh's algorithm, we first check using the Bloom filter to determine if the given query word is the member of Layers Indexing, if it is a member then we perform searches on Layers Indexing that saves run time on queries that are not part of Layers Indexing.

Another application of Bloom filter is to perform wild card searches on Layers Indexing (paper [32] uses secure keyword searches using Bloom filter with specified character positions). We would like to use the enhanced algorithm for multi-word query searches. Let's assume each codeword in our Layers Indexing contains three query words and client can perform query of these three words in any permutation. To save these three permutations in Layers Indexing, we will require nine codeword entries in our Layers Indexing but due to the membership property of the Bloom filter, client need to store only one codeword in the Layers Indexing. The client can determine which member of the query is present in the Layers Indexing through first checking membership using Bloom filter and then can perform search for the codeword that is found in the Bloom filter. Hence Bloom filters save both space and time for client queries.

Enhanced Goh Algorithm: The Goh algorithm [15] takes a private key K, a plaintext document d, a document ID and returns a Bloom filter representing the document index. In the Goh algorithm, document is first split into the set of plaintext words. Then the trapdoor is constructed for each plaintext word in the document using the trapdoor algorithm. The trapdoor is a secure form of each plaintext word. A codeword is then constructed based on the trapdoor after taking each trapdoor input and hash with the document ID. Finally codeword is added in the Bloom filter that is representing the index. In the algorithm, document ID is used to stop two identical documents to generate same index.

Goh algorithm [15] uses Bloom filter for every document. This will work fine for a small collection of documents, where user can perform searches for a given client query over a set of Bloom filters to find out query word belongs to a given collection. When the collection of document is large then usage of Bloom filter for a document is not a viable solution. In a large collection of documents, the query search become difficult as we have to find Bloom filter against a document and then construct query word based on a particular Bloom filter for index searches.

The Goh algorithm is enhanced to construct index for codewords using layers indexing. We have improved the Goh [15] algorithm to use a Bloom filter for a set of bytes. These filters are stored in the location pointed by these set of bytes. Multiple Bloom filters utilization for the set of bytes also helps to minimize the probability of filter saturation and false positives. We store, retrieve and utilize these filters using the first few bytes of the q-gram word, called q-gram segmentation. To show an example, if we choose q-gram segmentation size of three bytes to generate pointers for store, retrieve or utilize Bloom filters, this means we will have the 2^{24}Bloom filters available to build secure index over a collection. These q-gram segmented pointers also used to generate codeword from the trapdoor instead of using the document ID. In our enhanced version of the Goh algorithm, we use the Bloom filter to find out that query word belongs to a collection. We do not use the Bloom filter to represent the overall document index.

Methodology: Our index construction departs from existing dynamic index-based techniques for DSSE [14, 18, 19, 20, 24]that uses an inverted index data structure. Instead, it stores codewords on layers indexing that is similar to hierarchical memory approach mentioned in ORAM [21].

In our index construction, we first extract distinct words $(w_1, \ldots, w_m) \in W^n$ after removal of stop and stemming words from every document over a collection. These extracted words are in the form of tuples *(word, doc ID, identifying text, word frequency).* The alphabetical word list is transformed using enhanced Goh algorithm into the equivalent codeword list. This transformation procedure is repeated for all the documents in a collection. The codeword tuples list from all documents are then merged. The codewords are alphabetically sorted and same codewords are combined into a single codeword and there parameters are merged. Then these codewords are converted into binary codewords. These binary codewords are the search patterns or addresses of q-gram size, for example, if the selected q-gram size is 256-bit then the binary codeword address range is from 0to 2^{256}.

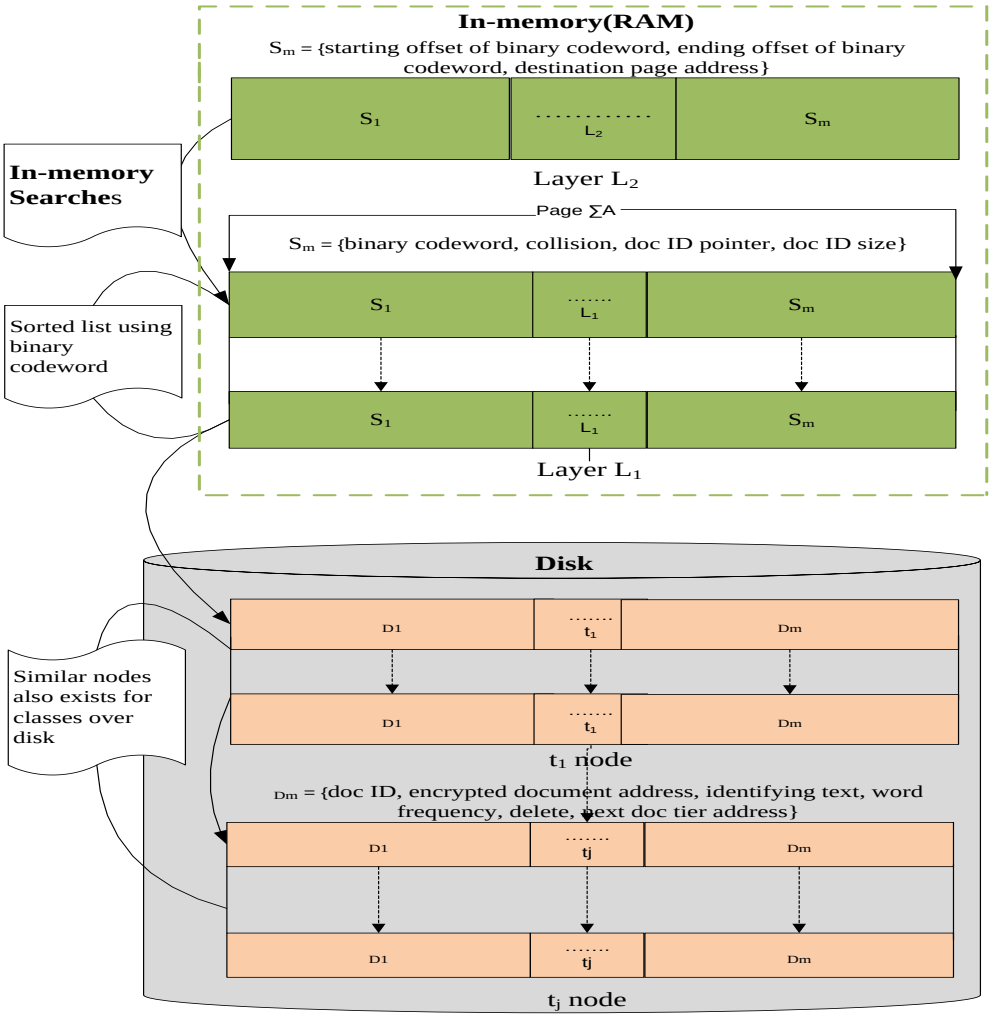

Fig. 1: Layers Indexing Construction

We build the layers indexing using these binary codewords. The layers indexing is shown in Fig. 1. The design goal for our construction is to perform in-memory index searches for layers and I/O leaf operation for rank list of documents. Layers indexing consists of layers $(l_1, \ldots, l_k)$ and leaf tiers $(t_1, \ldots, t_j)$. Layers $(l_1, \ldots, l_k)$ are in the external memory and leaf tiers $(t_1, \ldots, t_j)$are in the disk. Optionally, client provides a key to a server to encrypt the layers $(l_1, \ldots, l_k)$ and the leaf tiers $(t_1, \ldots, t_j)$.

Layers $(l_1, \ldots, l_k)$ have the sorted search pattern (binary codeword) and the destination address for the leaf tier. Layers $(l_1, \ldots, l_k)$ are formed using memory where the top layers nodes have the address ranges based on binary codeword along with the destination pointer for a lower layer page and the bottom layer nodes have the binary codeword and the destination pointer for a leaf tier page. Each layer is further divided into pages and layers are built-in hierarchical fashion, bottom layer l_1 pages are rolled up to layer l_2 and l_2 pages are rolled up to l_3 until we left with one page at the topmost layer.

Leaf tiers $(t_1, \ldots, t_j)$ are formed using the link list. Each *leaf node* has the set of rank structures that contains doc ID, identifying text, word frequency, delete and the destination tier address. Leaf tiers $(t_1, \ldots, t_j)$ have the order list of doc ID structures based on word frequency. All parameters are encrypted except the destination tier address. During search operations, client sees the latency of tier $t_1$fetch from disk, whereas remaining $(t_2, \ldots, t_j)$ tiers latencies are hidden from the client as they are being pre-fetched while client is analyzing tier $t_1$rank outputs.

We can add or delete document at any time as we are working with dynamic data. The addition operation is supported using the percentage of free space (spare area) in each page of layers $(l_1, \ldots, l_k)$ and leaf tiers

$(t_1, \ldots, t_j)$. We provide an extra (encrypted) data structure A_s called the storage array that the client can query to keep track of which locations in A_s are free so that client can add new document and same array is used for new space addition that results after deletion operation. The deletion operation is handled using the *delete* bit in the *leaf node*. To avoid risk of running out of free space and for better memory utilization layers indexing are periodically rebuild.

In another variant of our layers indexing scheme that is not discuss here, documents are categorized into different classes and documents are ranked using class parameter. The documents are classified using the classification algorithm such as Naive Bayes, KNN and others [3]. On a client query side, probabilistic information retrieval algorithm is used to categorize the client query into a particular class. The client query and its classification is used to retrieve the document from a subset of documents.

Construction: The index construction using the multiple Bloom filters for a document is given in Table 2:

Algorithm 1: To construct keys

$\boldsymbol{K \leftarrow KeyGen(s)}$**:** Given a security parameter s, choose a pseudo-random function f such as SHA-512, and using the function generate the secret key $K = (k_1, \ldots, k_r)$. Where $(k_1, \ldots, k_r)$ is the secret key K split into multiple keys, for example, set of 32 keys.

Algorithm 2: To construct codeword and ciphertext

$(\boldsymbol{d_{cw}, c}) \leftarrow \boldsymbol{DocEnc(K, d)}$**:**

1. Input document: Keywords are extracted from the document. All distinct key words are kept.
2. Sorting: Sort the list of keyword indexes in alphabetical order.
3. Construct q-gram: Construct the desire q-gram for a sorted word, using:

   $if(keyWordSize > qGramSize)$
   $newWord := w >> (keyWordSize - qGramSize);$
   $elseif(keyWordSize < qGramSize)$
   $newWord := worQGRAMSZ\_ZERO\_C;$

4. Generate Pointers to Store or Retrieve Bloom Filters: Now, choose the q-gram size for word to generate pointers to find the Bloom filter. Same pointer will be used to store the Bloom filter for each word. The Bloom filter found using this pointer would be used to build codeword for that particular word. For example, if we choose q-gram segmentation size of three bytes to generate pointers for store or retrieve Bloom filters that mean we will have the 2^{24} Bloom filters available to build secure index.

   $qGramPointer := newWord >> (qGramSize - qGramptrSz);$

5. Select Filter: Get the respective Bloom filter using the pointer generated in step 4. Bloom filter found for the word will be used to build the codeword for that particular word.

   $bFilter := read(qGramPointer)$

6. Split Secret key: Split the secret key. For example, when key is generated from SHA-512 then split the key into 32 keys.

   $(k_1, \ldots, k_r) \leftarrow K;$

7. Construct Trapdoor: Construct the trapdoor using the q-gram word, keys and a hash function f. For example, we are using the 256-bit q-gram, and secret key is generated from SHA-512 that is further split into 32 subset keys, so we can split the q-gram into 32-bytes where each byte is padded with zeros to desire length of subset keys

   $trapdoor := SHA256(\{(newWordByte_1 \; xor \; k_1), \ldots, (newWordByte_{16} \; xor \; k_{32})\})$;

8. Construct Codeword: A codeword is then constructed using the trapdoor. This simply takes each byte of trapdoor and hashes it with the q-gram filter pointer.

   $cw := \{f_{Qptr}(trapdoorByte_1), \ldots, f_{Qptr}(trapdoorByte_{32})$;

9. Adding into Filter: The codeword can then be added to the Bloom filter.

   $newFilter := bFilter(cw);$

10. Store the Filter: Store the filter using the using the pointer.

$write(qGramPointer, newFilter)$;

11. Output: Perform CPA-secure private-key encryption scheme on document to generate ciphertext$c$. Repeat steps 4 to 10 for each word in the list and output the codeword d_{cw} document and ciphertext c.

Algorithm 3: To construct Layers Indexing

$\boldsymbol{Y \leftarrow BuildIndex(d_{cw})}$**:**

1. Construct the master list: Using the sequence of codeword documents $d = (d_{\text{cw}\,1}, \ldots, d_{cwn})$ generated using the $DocEnc$ algorithm, construct the master list of codewords.

$count := totalDocuments$;
$while(count)$
$begin$
$newFile := readNextDoc(d_{cw})$;

$while(newFile\ != eof)$
$begin$
$newCodeword := readCodeword(newFile)$;
$docId := assignDocId(newFile)$;
$docSize := assignDocSize(newFile)$;
$nodeEntry := \{newCodeword, docId, docSize\}$;
$insertNewNodes(newDocFile, nodeEntry)$;
end

$masterFileList := readAllCodeword(newDocFile)$;
$count - -$;
end

Each list item is made of these tuples *(codeword, doc/class id, doc/class size*).

2. Convert into binary code: Convert the codewords master file list into the equivalent binary codewords.

$masterBinaryFile := convertToBinary(masterFileList)$;

3. Sorting: Sort the codeword binary code. Each binary codeword is itself is the address.

$masterBinaryFile := convertToBinary(masterFileList)$;

a) If sorted list has the identical addresses, (collision) they will combine into the same address. Collision is resolved either by user selection of result or intersection and joining multiple word queries.

4. Count: Count the sorted binary codeword.

$while(masterBinaryFile\ != eof)$
totalBinary$Codeword++$;

5. Construct the bottom layer: Using the sorted binary codeword, construct the list L_i. The list is built in layers, see the Fig. 1. The structure of the bottom layer (L_1) is *(binary codeword, collision, doc ID/class pointer, doc ID/class size)* and shown in Fig. 1. Every page has multiple nodes (layer node) in it. These nodes are in the form of the array up to the page size. Page size varies from 4KB to 64 MB and it will be a user defined constant.

$for(i = 0; i \leq \text{totalBinary}Codeword; i++)$
$begin$
$for(j = 0; j \leq L_1 PAGESIZE\_C; j++)$
$pageL1[i][j] := readNewNode(masterFileList)$;
end

a) Selected page size is dependent on multiple factors i.e., external memory size, number of client queries supported at a time, the underlying storage system (non-volatile memory versus disk cylinders), number of desired layers for pointer search and space requirements for lookup tables. The preference is to select the smaller L_1 page size (i.e., 4KB). As codeword is generated through random process and multiple client query is also random so the concept of locality seldom exists.

b) The doc ID pointer that is used in above structure will point to doc ID structure, shown in Fig. 1. The structure

of doc ID node is *(doc ID, encrypted document address, identifying text, word frequency, delete, next doc tier address)*. There are multiple doc ID structures (say 10) in a node (leaf node) and every page has multiple nodes in it. Multiple tiers of doc ID nodes exist in the form of a linked list, see the Fig. 1

c) If the classes are used (presently not describe in this paper) then the class pointer structure will be used. The node structure of class ID is *(class ID, class vector, class address offset, class size)*. In every page there will be multiple nodes of these structures.

6. Construct the top layers: Pages will be added in the L_1until the complete list of binary codeword reaches to the count that is defined in step 4. When the next page is started$, we$ will save the previous page q-gram starting and ending page offsets in layer-2 (L_2) using the structure *(starting offset of codeword, ending offset of codeword, destination codeword page address)*. Where,

$startingOffsetCodewordSize := startingAddressCodeword \gg L_1PAGESIZE\_C;$
$endingingOffsetCodewordSize := endingingAddressCodeword \gg L_1PAGESIZE\_C;$
$destinationCodewordPageAddressSize := numberofPagesInL_1;$

7. Layers hierarchy and page size: Layers exists in hierarchical form, for example, L_2 layer lies on the top of L1 layer, see the Fig. 1. The L_2 layer page size varies from 16MB to 512MB. Every node of L_2 contains a pointer for L_1 pages and the structure for L_2 is *(starting offset of codeword, ending offset of codeword, destination codeword page address)*.

8. Repeat step 6 and step 7: Step 6 and step 7 continue until we reach to the count calculated in step 4. The process of combining the multiple pages into the above layer is continued for upward layers until all the pointers will be summed up in a single page. This means, for example, if L_2 requires one page to define all the pages used in L_1 then we will stop at L_2 and no further upward layers will be constructed.

Algorithm 4: To construct search pattern for a given keyword

$\boldsymbol{cw_s \leftarrow SrchCodeword(K, \quad w)}$:

1. Construct q-gram: Using the client query we need to defined q-grams.

$if(keyWordSize > qGramSize)$
$\quad newWord := w >> (keyWordSize - qGramSize);$
$elseif(keyWordSize < qGramSize)$
$\quad newWord := worQGRAMSZ\_ZERO\_C;$

2. Generate Pointers to Store or Retrieve Bloom Filters: Now, choose the q-gram size for word to generate pointers to find the Bloom filter. Same pointer will be used to store the Bloom filter for each word. The Bloom filter found using this pointer would be used to build codeword for that particular word. For example, if we choose q-gram segmentation size of three bytes to generate pointers for store or retrieve Bloom filters that mean we will have the 2^{24}Bloom filters available to build secure index.

$qGramPointer := newWord >> (qGramSize - qGramptrSz);$

3. Select Filter: Get the respective Bloom filter using the pointer generated in step 4. Bloom filter found for the word will be used to build the codeword for that particular word.

$bFilter := read(qGramPointer)$

4. Split Secret key: Split the secret key. For example, when key is generated from SHA-512 then split the key into 32 keys.

$(k_1, \ldots, k_r) \leftarrow K;$

5. Construct Trapdoor: Construct the trapdoor using the q-gram word, keys and a hash function f. For example, we are using the 256-bit q-gram, and secret key is generated from SHA-512 that is further split into 32 subset keys, so we can split the q-gram into 32-bytes where each byte is padded with zeros to desire length of subset keys

$trapdoor := SHA256(\{(newWordByte_1 \; xor \; k_1), \ldots, (newWordByte_{32} \; xor \; k_{32})\})$;

6. Construct Codeword: A codeword is then constructed using the trapdoor. This simply takes each byte of trapdoor and hashes it with the q-gram filter pointer.

$cw := \{f_{Qptr}(trapdoorByte_1), \ldots, f_{Qptr}(trapdoorByte_{32})$;

Algorithm 5: To construct node for an addition operation

$\boldsymbol{nodeAdd_{cw} \leftarrow AddCodeword(d_{cw}, codewordIndex):}$

1. Construct the node: Construct the node that has tuple *(codeword, doc/class id, doc/class size)*

   $newCodeword := readNextDoc(d_{cw}, codewordIndex);$
   $docId := assignDocId(d_{cw});$
   $docSize := assignDocSize(d_{cw});$
   $nodeAdd_{cw} := \{newCodeword, docId, docSize\};$

2. Output: Output the $nodeAdd_{cw}$.

Algorithm 6: To construct node for deletion operation

$\boldsymbol{nodeDel_{cw} \leftarrow DelCodeword(d_{cw}, codewordIndex):}$

1. Extract the codeword: Extract the codeword from a document

   $cw_d := readNextDoc(d_{cw}, codewordIndex);$
   $docId := assignDocId(d_{cw});$
   $docSize := assignDocSize(d_{cw});$
   $nodeDel_{cw} := \{newCodeword, docId, docSize\};$

2. Output: Output the $nodeDel_{cw}$

Algorithm 7: To perform searches on Layers Indexing

$\boldsymbol{LeafNode \leftarrow Search(\Upsilon,\ cw_s,\ tierEnable,\ tierAdress):}$

1. If the *tierEnable*bit is set then goto step 5 otherwise continue from step 2.

2. Top layer search: Codewordcw_ssearch will start from the top most layer of the index Υ, for, example, layer l_2. The searches are performed using the interval halving method. The algorithm is given below:
   a) Setup variable: Let $a := startingPage\ Offset;\ b := endingPageOffset; s := nodeSize; x_m = (a + b)/(s * 2)$;Where$x_m$ is the page offset, $f(x_1)$ is the codeword that needed to be searched.

   b) Read the codeword: Read the address from the node pointed by x_m. There will be two values for $f(x_m)$ that is $f(x_m - low)$ and$f(x_m - high)$, where $f(x_m - low)$ is the starting offset of a codeword and $f(x_m - high)$ is the ending offset of codeword.

   c) Select the lower range: If $f(x_1) < f(x_m - high)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie beyondx_m. Therefore, set $a := startingPage\ Offset;\ b := x_m$; else goto step e

   d) Select the upper range: If $f(x_1) > f(x_m)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie lower thenx_m. Therefore, set $a := x_m;\ b := ending\ page\ offset$; else goto step e

   e) Result or repeat: If $f(x_1)$ lie in the range $f(x_m - low, x_m - high)$ then terminate goto f else continue from step a

   f) Output: Address of the lower layer page, for example, layer-1.

3. Bottom layer search: Using the address found in $step\ 1$ repeat $step\ 1_a\ to\ 1_f$ for pointer of the doc Id $tier\ 1$ page.

4. Read $tier\ 1$ leaf node: Read the $tier\ 1$ leaf node.

5. Output: Output the node based on leaf node found in step 4 or using the input tierAddress.

Algorithm 8: To perform addition or update on Layer Indexing

$\boldsymbol{\Upsilon' \leftarrow Add(\Upsilon, R_{cw}, nodeAdd_{cw})}$:

1. Top layer search: Using codeword$cw_a$from $nodeAdd_{cw}$search starts from the top most layer of the index Υ, for, example, layer l_2. The searches are performed using the interval halving method. The algorithm is given below:
   a) Setup variable: Let $a := startingPage\ Offset;\ b := endingPageOffset; s := nodeSize; x_m = (a + b)/(s * 2)$;Where$x_m$ is the page offset, $f(x_1)$ is the codeword that needed to be searched.

b) Read the codeword: Read the address from the node pointed by x_m. There will be two values for $f(x_m)$ that is $f(x_m - low)$ and$f(x_m - high)$, where $f(x_m - low)$ is the starting offset of a codeword and $f(x_m - high)$ is the ending offset of codeword.

c) Select the lower range: If $f(x_1) < f(x_m - high)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie beyondx_m. Therefore, set $a := startingPage\ Offset$; $b := x_m$; else goto step e

d) Select the upper range: If $f(x_1) > f(x_m)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie lower thenx_m. Therefore, set $a := x_m$; $b := ending\ page\ offset$; else goto step e

e) Result or repeat: If $f(x_1)$ lie in the range $f(x_m - low, x_m - high)$ then terminate goto f else continue from step a

f) Output: Address of the lower layer page, for example, layer l_1.

2. Bottom layer search: Using the address found in $step\ 1$ repeat $step\ 1_a\ to\ 1_f$ for codeword cw_athat needed to be added or updated.

3. Page read: If the codeword$cw_a$is found in the search address then goto step 5, otherwise using the address that is the output of step 1 read rest of the page of layer l_1.

4. Insert node and write back: Insert the $nodeAdd_{cw}$ node and write back the remaining page into the layer l_1.

5. Update or addition on tier layers: Read all the leaf tier layers $(t_1, \ldots, t_j)$one at a time and based on ranking number of R_{cw} of codeword write back the leaf tier layers with updated or added leaf node.

6. Output: After insertion of new node in tier layers, the new index is Y'

Algorithm 9: To perform deletion on Layers Indexing

$node_{Del} \leftarrow Del(Y,\ nodeDel_{cw})$:

1. Top layer search: Using codeword$cw_d$from node $nodeDel_{cw}$search starts from the top most layer of the index Y, for, example, layer l_2. The searches are performed using the interval halving method. The algorithm is given below:

g) Setup variable: Let $a := startingPage\ Offset$; $b := endingPageOffset$; $s := nodeSize$; $x_m = (a + b)/(s * 2)$;Wherex_m is the page offset, $f(x_1)$ is the codeword that needed to be searched.

h) Read the codeword: Read the address from the node pointed by x_m. There will be two values for $f(x_m)$ that is $f(x_m - low)$ and$f(x_m - high)$, where $f(x_m - low)$ is the starting offset of a codeword and $f(x_m - high)$ is the ending offset of codeword.

i) Select the lower range: If $f(x_1) < f(x_m - high)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie beyondx_m. Therefore, set $a := startingPage\ Offset$; $b := x_m$; else goto step e

j) Select the upper range: If $f(x_1) > f(x_m)$ and $f(x_1)$ does not lie in the range $f(x_m - low, x_m - high)$ then region cannot lie lower thenx_m. Therefore, set $a := x_m$; $b := ending\ page\ offset$; else goto step e

k) Result or repeat: If $f(x_1)$ lie in the range $f(x_m - low, x_m - high)$ then terminate goto f else goto continue from step a

l) Output: Address of the lower layer page, for example, layer l_1.

2. Bottom layer search: Using the address found in $step\ 1$ repeat $step\ 1_a\ to\ 1_f$ for pointer of the doc Id $tier\ 1$ page.

3. Search $tier\ nodes$: Search the $tier\ nodes$ and output the delete node.

```
{tierPage, docId, docSize} := nodeDel_cw;
  while(j != docSize/STRUCTSZ_C)
  begin
  i := 0;
  newNode := readNode(tierPage);

  while(i != NUMOFSTRUC_C)
  begin
                        newStruc := readNextDecryptedStruc(newNode);
  {(tDocId, encDocAddress, idText, rankParameter, del, nextDoctierAddress)}:  = newStruc;
  if(docId == tDocId)
  begin
                              docAddress := encDocAddress;
   tierPageSave := tierPage;
   strucOffset := i;
   strucFound := 1;
                    nodeDel := {tierPageSave, docAddress,    strucOffset};
   break;
  end
  i := i + 1;
  end

  tierPage := nextDoctierAddress;
  j := j + 1;
  end
```

4. Output: Output the $nodeDel$.

Algorithm 10: To rank the documents for a keyword
$\boldsymbol{R_{cw} \leftarrow Rank(n, df_t, tf_{t,d})}$**:** Calculate rank and output R_{cw} using $tf_{t,d} * log\ n/df_t$.

Algorithm 11: To de-cipher the document
d ← Dec(K, c): Using CPA-secure private-key decryption scheme and a ciphertext c and outputs a document d.

Table 2: Dynamic SSE scheme using layers indexing

Theorem (Security): *Layers Indexing scheme, is DSSE scheme that is based on eleven algorithms as given in the Definition 1 is secure in semi-honest model. Consider the following probabilistic experiment,Ais an adversary, S is the Simulator and L_1, L_2 and L_3 are the leakage algorithms as defined in Definition 2, 3 and 4. We will simulate the functionality of the Layer Indexing scheme such that it performs interactions with a real-world semi-honest Server A.*

Proof: The proof involves in demonstrating that client interact with Simulator *S* using the ideal functionality *F* that can simulate the real world functionality $\prod_F$ with adversary A. This is achieved using the given leakages L_1, L_2 and L_3. Also, this is the adaptive nature of security such that the input to the clients at any time can be influenced by the view of the server until that time.

- Simulating Building Index (Υ):

We describe a simulator S that interacts with adversary A in an execution of an ideal world experiment using the leakage definitions described in definition 2. Given the leakage $L_1(L_{sz}, n, sz)$ it constructs the c and Υ as follows. It simulates using the simulator S, the cipher text documents $c = (c_1, \ldots, c_n)$ and the sequence of codeword in the documents $d = (d_{\text{cw}\,1}, \ldots, d_{cwn})$together with number files n and size of the each file that are described in the leakage function L_1. To simulate the Layers Indexing Υ using the sequence of codeword documents $\mathrm{d} = (\mathrm{d}_{\text{cw}\,1}, \ldots, \mathrm{d}_{\text{cwn}})$ generated using the DocEnc algorithm construct the master list of codewords. Each list item in the master list is made of these *tuples (codeword, doc/class id, doc/class size)*. Simulator then converts every codeword into equivalent binary codewords and performs sorting on binary codewords. Simulator counts the total number of binary codewords to build multiple layers.

Simulator starts building the Layers Indexing from the bottom layer. Each layer has the multiple pages in it where each page consists of nodes in the form of the array. Each node in a page has these tuples *(binary codeword, collision, doc ID/class pointer, doc ID/class size).* Pages will be added in the bottom layer L_1 until the complete list of binary codeword reaches to the total count. While building each page in the bottom layer L_1, the simulator saves the previous page q-gram starting and ending page offsets in layer-2 (L_2) using the structure *(starting offset of codeword, ending offset of codeword, destination codeword page address)*. The process of combining the multiple pages into the above layer is continued for upward layers until all the pointers will be summed up in a single page. Finally, the simulator outputs the layers indexing Υ to the adversary.

- Simulating Search ($LeafNode$):

Simulator simulates the search pattern cw_s for given keyword and Key K. Simulator performs searches for the codeword cw_s in the Layers Indexing Υ. There are two cases as Layers Indexing Υ arranges the rank documents in multiple tiers, at the beginning of search $tierEnable$ bit de-assertedso that $tierAdress$ is not utilized. Simulator starts the search pattern cw_s searches from the topmost layer of the Layer Indexing Υ, for example, layer l_2. Simulator performs searches using interval halving method. Simulator first finds the address range for cw_s in layer l_2 and then simulator further performs searches on the destination layer l_1 page. The address found in layer l_1 page for leaf tier t_1 is used to output $LeafNode$ that contains all the structures for tier t_1 node. After sending the first output of the searches, simulator is busy in pre-fetching the next list of rank outputs from the next leaf node where $tierEnable$ bit is asserted and the $tierAdress$ is set from the *next doc tier address*.

- Simulating Addition (Υ'):

Suppose now the simulator wishes to add a document d containing the keyword w in the Layers Indexing Υ. There are two cases, firstly the keyword is not present in the Layers Indexing Υ, and this case is not discussed to save space. Secondly, given keyword w already exists in the Layers Indexing Υ. Simulator adds the new document structure in the leaf tier layers $(t_1, \ldots, t_j)$ based on ranking R_{cw}. To add a document that has the single keyword, simulator first call $DocEnc(K, d)$ to get codeword cw. Simulator then using $nodeAdd_{cw}$ generates the layer node that needs to be inserted in the layer l_1. Simulator then adds the leaf node in leaf tier layers $(t_1, \ldots, t_j)$. The leaf node has the storage location pointer to store document d in the storage array. This pointer is calculated from the storage array A_s.

- Simulating Deletion ($node_{Del}$):

Suppose now the simulator delete a document d containing the keyword w from the Layers Indexing Υ. Let's assume that there are several documents exist for keyword w for a corresponding codeword cw. Simulator deletes the document d from the leaf tier layers $(t_1, \ldots, t_j)$. To delete a document that has the single keyword, simulator first call $DocEnc(K, d)$ to get the codeword cw. Simulator using $\boldsymbol{nodeDel_{cw}}$ searches the leaf node that is deleted from the leaf tiers. Simulator then deletes a leaf node from a respective leaf tier layers $(t_1, \ldots, t_j)$. Finally, simulator mark up the bit in the storage array A_s to represent it as a free storage space.

- Simulating Ranking (R_{cw}):

Simulator calculates rank and output R_{cw} using $tf_{t,d} * log\ n/df_t$.

It is important to show that ciphertext documents from a Real $\boldsymbol{(\prod_F A)}$ and from an Ideal $\boldsymbol{(F, S)}$ experiment are negligibly close. This is because, the keys used are indistinguishable from real keys since they are constructed pseudo-random function f such as SHA-512, which is indistinguishable. Therefore, the CPA-security guarantees that adversary cannot distinguish between the real and simulated encryptions of the documents. This concludes our proof.

Theorem 2 (*search time*): *Let W denotes the universe of words. If $d = (w_1, \ldots, w_m) \in W^n$ then these words are represented in equivalent secure form as codeword $d_{cw} = (cw_1, \ldots, cw_m) \in$. Then dynamic hierarchical Layers Indexing exists for query searches such that the search time for the codeword is $O(log(\mathcal{W}/\mathfrak{P}))$, n is the size of the document collection, $\mathcal{W} = \sum \mathfrak{w}$ number of words in n collection of documents and $\mathfrak{P}$ is the number of nodes in a page.*

Proof: As we have discussed before our Layers Indexing Υ defined as:

i. If l is the list then l_k denotes total number of layers that contains the list.

ii. The array is presented through A and total number of cells in the array is presented through$\sum A$. A Layers *Indexing*Υ consists of multiple layers $(l_1, \ldots, l_k)$ and leaf tiers$(t_1, \ldots, t_j)$. Layers $(l_1, \ldots, l_k)$ are in the external memory and leaf tiers $(t_1, \ldots, t_j)$are in the disk.

iii. Each layer of $(l_1, \ldots, l_k)$ is made of multiple pages, where each page size is equal to the array size$\sum A$.

iv. We further define a *layer node* in a page that store structure at location i ∈ $[\sum A]$ is $A[i]$ and $A[i] := s$ represent the operation that stores s at location i in A.

v. $\mathfrak{Y}$is the number of nodes in a page.

We are using the Interval Having Method within page so its search complexity in literature [3] defined as O(log W) and as we divided the memory into pages where each page has $\mathfrak{Y}$ number of nodes therefore our search complexity becomes O(log (W/ $\mathfrak{Y}$). The searches are performed in memory in parallel time and if P denote the number of processors to perform searches then overall search complexity become O(1/P log(W/ $\mathfrak{Y}$). This completes the proof.

4.0 ARCHITECTURE, ANALYSIS AND PERFORMANCE RESULTS

Architecture: The system level architecture for the proposed Layers Indexing is as shown in the Fig. 2. The client side supports multi-user environment that communicate with the cloud servers. The cloud servers use the Layers Indexing consists of multiple layers $(l_1, \ldots, l_k)$that are divided over many servers to accommodate address space in the server memories. This is possible because our scheme uses sorted linear address space with free space for addition operations. These servers then communicate over the LAN with the storage nodes that has leaf tiers $(t_1, \ldots, t_j)$.

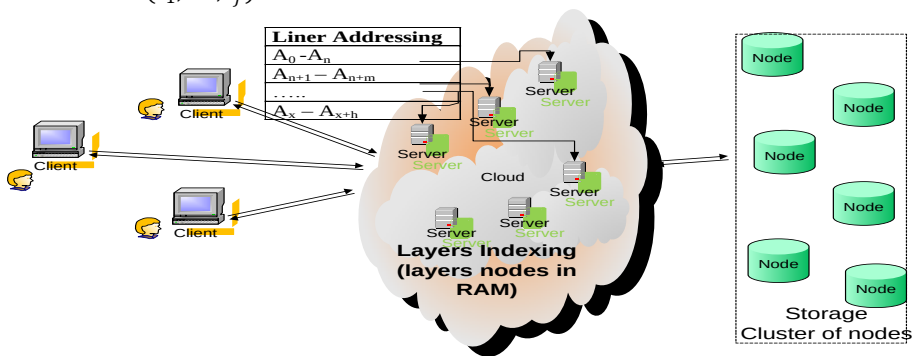

Fig. 2: System Level Architecture

Analysis parameters: The analysis is performed for layers indexing scheme against real data sets to see how many memory layers are required, the number of blade server with memory and storage nodes required and to verify that overall search time performance is consistent with $O(1/p \log(\mathcal{W}/\mathfrak{Y}))$. To achieve this, we have used the collection statistics from Reuters-RCV1collection [3] and very large collection [3]. The Reuters-RCV1 collection has 800,000 documents with average number of tokens per document of 200. The average byte per token without spaces or punctuation is 7.5 bytes. The number of distinct words in the collection is 400,000. This results in 80 million codewords for our searches. The very large collection has the 1,000,000,000 documents with average token per document of 1,000. The number of distinct words in the collection is 44,000,000. This results in 44 billion codewords for our searches. The document storage requirement for large size collection is 10 TB (rounded to next number). We further assume that for large collection, we have 200 documents for every codeword to rank. We are using 100 B for each tier structure in the leaf node, this gives us the overall leaf tiers $(t_1, \ldots, t_j)$space of 890 TB. To further analyze our layers indexing scheme, we define the intermediate collection that has the 900 million codewords for searches.

In today's computing servers have up to 768 GB of DDR3 memory using the LRDIMM. Our scheme analysis is based on the Blade server with 512 GB of DDR3-2000 memory and 1 TB of disk. The Blade server uses the Intel Xeon processor with 20M cache and the 3.1 GHz CPU clock.

Table 3: Memory Layers and Storage Nodes

Number of documents (Millions)	Codewords (Millions)	Node Size (Bytes)	Overall memory size with 20 % free space (GB)	Memory layers (64 MB Page)	Blade server with 512 GB memory	Number of storage nodes (1 TB)
0.8	80	32	96	2	1	
0.8	80	40	120	2	1	
0.8	80	48	144	2	1	
9	900	32	1080	3	1	
9	900	40	1350	3	2	
9	900	48	1620	3	3	
1000	44000	32	52800	3	104	890
1000	44000	40	66000	3	129	890
1000	44000	48	79200	3	155	890

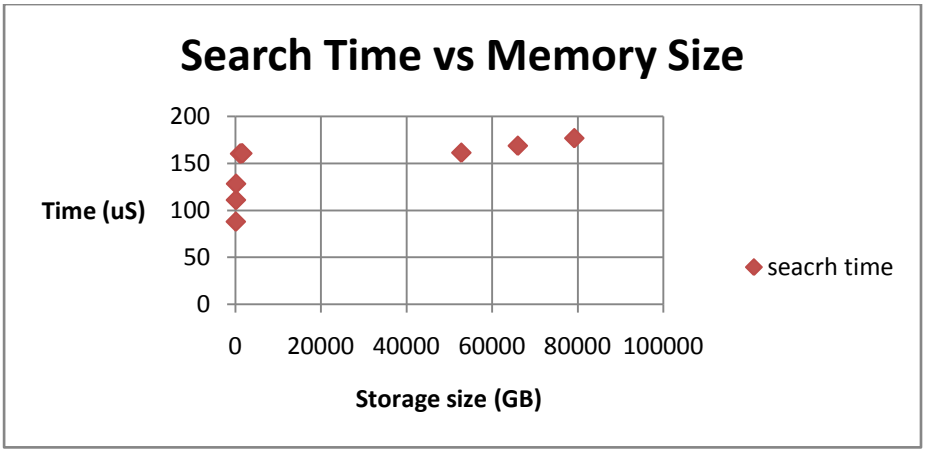

Fig. 3: Search Time versus Memory Size

Analysis: Table 3 shows the analysis for our in-memory layers indexing using the page size of 64 MB for all memory layers. The memory page sizes can vary from 4MB to 64MB. We analyze the layers indexing against three layer node sizes 32B, 40B and 48B to see its effect on memory layers for index construction. We use the 20% free space for additions or updates to support dynamic operations. We concluded that layers indexing memory layers increases from two to three, when the number of codeword increases from 80 million to 44 billion.

Secondly, we are interested in the implementation of our scheme using the Blade server, as shown in table 3 that number of storage nodes and Blade server required for our in-memory layers indexing. Our scheme for 32B node size uses 52 Blade servers and 890 nodes for storage using 1 TB disk. In scenario's, where construction requires multiple Blade servers, dictionary address space is divided across servers, see Fig 2. This is possible because our scheme uses sorted linear address space with free space for addition operations.

Thirdly, we perform the analysis to see the impact of collection sizes against the search time. Table 1 shows overall search time $O(1/p \log(W/2))$ in multi-processor environment. Now, we look more closely to search time using the system parameters specified for analysis. We have this equation for 64 MB pages:

$$T_s = l_k * (15 * T_c + 8 * T_{mavg}) + T_{daccs} \qquad (3)$$

Where T_s is the overall search time using the scheme described in Table 2, T_c is the CPU cycle time using cache, l_k is the number of layers used in layers indexing, T_{mavg} is the average access time for DDR3 memory and the T_{daccs} is the disk access time for leaf layer. We are using 256 KB blocks for CPU processing. We have plotted search times against the memory sizes in Fig. 3.

5.0 CONCLUSIONS

In this paper, we provided DSSE scheme to enable fast searches and to perform searches over large collections of unstructured documents. We have provided hierarchical in-memoryLayersIndexing dictionary construction scheme for fast rank searches over encrypted unstructured documents.

REFERENCES

[1] MadihaWaris, Dr. Shoab Ahmad Khan, "Indexing of unstructured data for searchable encryption in cloud environment", Accepted in *IEEE Technically Cosponsored Science and Information Conference 2013*, London (To be held 7-9 October, 2013)

[2] Muhammad Zaman Fakhar, Dr. Shoab Ahmad Khan, MadihaWaris, " position based sentence search for encrypted unstructured data in a cloud environment", *Proceedings of International Conference on Cloud Computing and eGovernance* 2013

[3] Christopher D. Manning, PrabhakarRaghavan and HinrichSchütze, "Introduction to Information Retrieval", *Cambridge University Press*. 2008

[4] Ferragina, Paolo, and RossanoVenturini. "The Compressed Permuterm Index", *ACM Transactions on Algorithms* (TALG), 7.1 (2010): 1-21

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422-426, 1970

[6] C. Wang, N. Cao, K Ren and W. Lou, "Enabling secure and Efficient Ranked Keyword Search over Outsourced Cloud Data", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 8, August 2012

[7] Zittrower, S., Zou, C.C., "Encrypted Phrase Searching in the Cloud", *Global Communications Conference (GLOBECOM)*, 764-770, December 2012

[8] Burden, Richard L. Faires, J. Douglas (1985), "2.1 The Bisection Algorithm", *Numerical Analysis (3rd ed.)*, PWS Publishers, ISBN 0-87150-857-5

[9] Prosenjit Bose,Karim Douïeb, Stefan Langerman, "Dynamic optimality for skip lists and B-trees", *Proceeding 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1106–1114, 2008

[10] Paolo Ferragina, Rodrigo Gonza´ lez, Gonzalo Navarro, RossanoVenturini, "Compressed Text Indexes: From Theory to Practice", *ACM Journal of Experimental Algorithmic*, Vol. 13 Article 1.12, December 2008

[11] Manber, U. Andwu, S., "Glimpse: a tool to search through entire file systems", *In Proceedings of the USENIX Winter 1994 Technical Conference*. USENIX Association, 4–4

[12] Ricardo Baeza-Yates, Gonzalo Navarro, "Block addressing indices for approximate text retrieval", *CIKM '97 Proceedings of the sixth international conference on Information and knowledge management*

[13] P. Ferragina, R. Grossi., "The String B-tree: a new data structure for string search in external memory and its applications", *Journal of the ACM*, 46(2): 236-280, 1999

[14] D. Song, D. Wagner, and A. Perrig, “Practical techniques for searches on encrypted data”, in *Proc. of S&P*, 2000.

[15] E.-J. Goh, “Secure indexes”, *Cryptology ePrint Archive*, 2003, http://eprint.iacr.org/2003/216.

[16] M. A. Shayegan, S. Aghabozorgi, and R. G. Raj, “A Novel Two-Stage Spectrum-Based Approach for Dimensionality Reduction: A Case Study on the Recognition of Handwritten Numerals,” Journal of Applied Mathematics, vol. 2014, Article ID 654787, 14 pages, 2014. doi:10.1155/2014/654787.

[17] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient constructions", in *Proc. of ACM CCS*, 2006.

[18] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption", in *Financial Cryptography (FC)*, 2013.

[19] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption", in *ACM Conference on Computer and Communications Security*, pages 965–976, 2012.

[20] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker, "Computationally efficient searchable symmetric encryption", in *Secure Data Management*, pages 87–100, 2010.

[21] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs", *J. ACM*, 1996.

[22] Moohebat, M., Raj, R.G. , Kareem, S.B.A., Thorleuchter, D., “Identifying ISI-indexed articles by their lexical usage: A text analysis approach”, Journal of the Association for Information Science and Technology, Vol. 66, No. 3, pp. 501–511. doi: 10.1002/asi.23194.

[23] K. Kurosawa and Y. Ohtaki, "UC-secure searchable symmetric encryption", in *Financial Cryptography (FC)*, pages 285–298, 2012.

[24] Practical Dynamic Searchable Symmetric Encryption with Small Leakage, "Emil Stefanov, Charalampos Papamanthou, and Elaine Shi", in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[25] AshwinSwaminathan, Yinian Mao, Guan-Ming Su, Hongmei Gou, Avinash L. Varna, Shan He, Min Wu, and Douglas W. Oard, "Confidentiality-preserving rank-ordered search", In *StorageSS ’07:Proceedings of the 2007 ACM workshop on Storage security and survivability*, pages 7–12, New York,NY, USA, 2007. ACM.

[26] Luhn, Hans Peter., “A statistical approach to mechanized encoding and searching of literary information.”, *IBM Journal of Research and Development* 1(4):309–317. 133, 527 1957

[27] Luhn, Hans Peter.. “The automatic creation of literature abstracts”, *IBM Journal of Research and Development* 2(2):159–165, 317. 133, 527 1958

[28] Spärck Jones, Karen, “A statistical interpretation of term specificity and its application in retrieval”, *Journal of Documentation* 28(1):11–21. 133, 525, 1972

[29] O. Goldreich, “Foundations of Cryptography: Volume 2, Basic Applications”, *Cambridge University Press*, New York, NY, USA, 2004.

[30] J. Katz and Y. Lindell, “Introduction to Modern Cryptography”, *Chapman & Hall/CRC*, 2008

[31] M. Naveed, M. Prabhakaran and C. A. Gunter, "Dynamic searchable encryption via blind storage", *Proc. IEEE Symp. Secur. Privacy*, pp.639 -654

[32] T. Suga, T. Nishide, and K. Sakurai, "Secure keyword search using bloom filter with specified character positions," in *Provable Security, ser. Lecture Notes in Computer Science*, T. Takagi, G. Wang, Z. Qin, S. Jiang, and Y. Yu, Eds. Springer Berlin Heidelberg, 2012, vol. 7496, pp. 235-252