JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 401-417 (2018)
DOI: 10.6688/JISE.201803_34(2).0006

Phrase Search for Encrypted Cloud Storage*

YEN-CHUNG CHEN[1,2], YU-SUNG WU[1] AND WEN-GUEY TZENG[1]
[1]*Department of Computer Science*
National Chiao Tung University
Hsinchu, 300 Taiwan
[2]*Information and Communication Security Laboratory*
Chunghwa Telecom Laboratories
Taoyuan, 326 Taiwan
E-mail: {yenchung; ysw; wgtzeng}@cs.nctu.edu.tw

With the growth in the popularity of cloud storage service (CSS), the accumulation of private data on the cloud requires the use of data encryption to prevent leakage of sensitive information to untrusted third parties. However, as the amount of data kept on the cloud storage is increasing, the use of data encryption makes it difficult or even impossible to locate the data of interest efficiently and securely.

In this paper, we present a framework for CSS to support queries in encrypted form so that the data on cloud storage can be located efficiently and securely. At the core of the framework is a novel indexing structure, called the bloom filter encrypted search tree (BFEST). The BFEST supports queries in the form of phrase keywords. Client-side encryption, using secret keys that are unknown to the cloud service provider, protects the queries and the retrieved data.

We implemented a prototype by extending the hicloud S3 [13] CSS with the proposed framework. The experimental results indicate that the framework can ensure query privacy for encrypted data with an acceptable performance overhead in a practical setting.

Keywords: cloud storage, privacy, encrypted search, phrase search, bloom filter

1. INTRODUCTION

Cloud computing has not only led the evolution of system architecture but it has also brought benefits and changed the way people interact with applications. However, the resulting security issues should also be considered. Consolidation in cloud computing implies the sharing of underlying resources. If the security isolation mechanism fails because of accidents or malicious attacks, no physical boundary at the infrastructure level can deter the attack propagation. In addition, the cloud service provider may not be fully trusted either. This is a great concern for users who would like to store sensitive data.

Data encryption is a practical way to protect data residing on a cloud. The secret keys used to encrypt each user's data can be stored locally by individual users [1] or remotely by a storage service provider [1, 10]. Assuming that the client-side does not contain backdoor programs or malware, keeping the encryption key locally can protect the data from security attacks on the cloud. However, as the amount of data stored on the cloud increases, we will need a search mechanism to ensure that the data of interest can be located and retrieved efficiently. Most existing cloud storage services (CSSs) [1, 12,

Received August 2, 2016; revised October 6 & November 12, 2016; accepted January 6, 2017.
Communicated by Xiaohong Jiang.
* This research supported in part by MOST project 104-2221-E-009-112-MY3 and 104-2221-E-009-104-MY3, Taiwan.

13] do not implement a search mechanism on the server-side. Instead, the search mechanism and the underlying index have to be maintained by the client-side. One reason why the server-side search mechanism is not commonly used is because users may want to encrypt the data on the cloud, which includes encrypting the index of the data. A server-side search mechanism would have to support searches over an encrypted index, which is unintuitive and difficult. The previous studies of Private Information Retrieval (PIR) allow users to retrieve data from the servers without revealing for what data they are looking. However, PIR requires a large volume of network transmission between the client and the servers, and it is not yet suitable for practical applications. Additionally, many studies focus on individual aspects, such as a conjunctive search, query hiding by an extra user-trusted layer, or a two-round phrase search.

Our study extends the existing CSS hicloud S3 [13] to support phrase search on the server-side. To ensure the service provider cannot read the user's data, the data kept on the cloud is encrypted, and the encryption key is maintained by the user. We design a secure searchable index named BFEST (bloom filter encrypted search tree) that allows the service provider to perform a search without compromising data confidentiality. The user can use the extended APIs (Application Program Interfaces) to locate data objects that satisfy the given query.

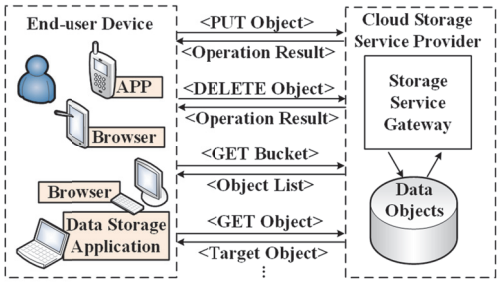

Fig. 1. Baseline cloud storage.

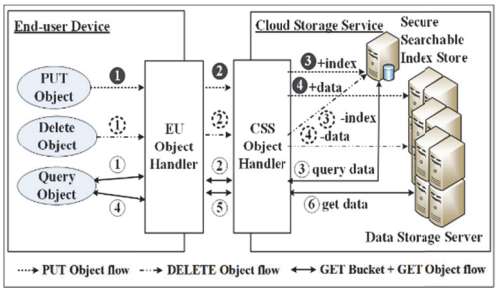

Fig. 2. The architecture of hicloud S3 security.

2. CLOUD STORAGE SERVICE

2.1 Baseline Architecture

CSSs such as Amazon S3 and Google Cloud Storage let the users store and access the data in the cloud through the APIs. They mostly follow the usage scenarios shown in Fig. 1, where an end-user (EU) can upload data via the PUT object API and download data via the GET object API. To protect user data privacy, in general, data objects are encrypted before uploading. For user convenience, the provider could maintain a searchable index, so the user can conduct a full-text search for the data of interest. However, one main issue with the providers is that the queries sent to them leak private information. In the following section, we formalize the threat model of a cloud-based storage service (Section 2.2) and present a framework to ensure user query privacy (Section 3).

2.2 Threat Model

We assume that an encrypted channel is available for securing the communication between the CSS and the EU device; the EU is assumed to be fully trusted. However, the storage service provider may be untrusted, even though it could guarantee data integrity and data availability.

While the user can maintain the secret keys for data encryption and the secure searchable index respectively, the data and the index can be encrypted in the EU device. Therefore, it is secure for the data at rest in the cloud. However, if the query mechanism is enabled, the query criteria contain what can be used to derive private information about the user [7, 29]. Thus, the user's privacy will be considered compromised. The act of sending a query is not considered part of user privacy. The user can mix true queries with meaningless queries randomly to mask the events of sending queries if needed.

3. SYSTEM DESIGN

Existing CSSs, as mentioned earlier, have limited support for encrypted query and protection of user privacy. In this paper, we propose a cloud storage system called "hicloud S3 security" to address the limitations. hicloud S3 security supports the same PUT/DELETE/GET operations as in representative CSSs such as Amazon S3 [1]. However, the data objects and the index are encrypted by secret keys that are only known by the EU. In addition, hicloud S3 security allows users to specify encrypted phrase criteria in the GET operation so that only data objects that match the query need to be returned to the EU. Under hicloud S3 security, neither the user query nor the user data is exposed to the CSS, and all the secret keys are only known to the EU. Fig. 2 shows the architecture of hicloud S3 security.

3.1 Secure Searchable Index

The secure searchable index is structured by BFEST, as shown in Fig. 3. It has two parts: a document tree (DT) and a number of phrase trees (PTs). The EU uploads a PT for each data object to the CSS via the PUT Object API. The CSS inserts the PT received from the EU into the secure searchable index as a DT leaf node.

Every PT node in a PT contains a bloom filter (BF) *bf* and a location list *loc*. Fig. 4 shows an example of a PT, and we take the leaf node K_1 for instance. As each leaf node represents a distinct keyword, K_1 stores all information including the bloom filter values for all possible k'_1 (encrypted format of k_1) and the location information for the keyword. During the construction of a PT, the bloom filter of a non-leaf node forms the bitwise-OR of the bloom filters of its child nodes. For example, the bloom filter of BF_lv4_1 is the bitwise-OR of the bloom filters of K_1 and K_2.

Conversely, each DT node contains a counting bloom filter (CBF) *cbf* and an identifier *fid*. If a DT node is a leaf node, its *fid* will serve as the reference to the associated object and its *cbf* will record the keywords in that object. If a DT node is a non-leaf node, its *cbf* value will be the element-wise summation of the *cbf* values from its child nodes.

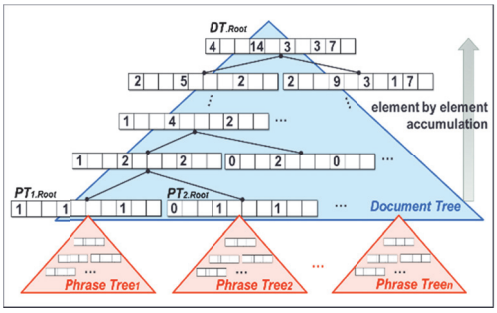

Fig. 3. Secure searchable index (BFEST).

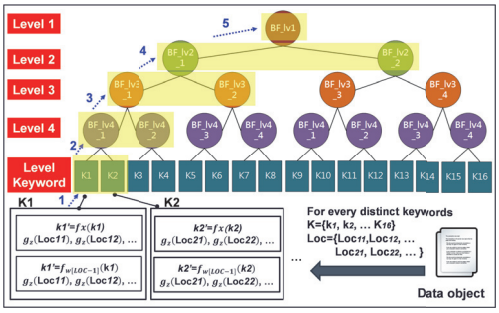

Fig. 4. Example of a phrase tree (PT).

3.2 PUT Object

Once we use the PUT Object API to upload a data object, the EU Object Handler at first creates the PT for the data object based on the index key. It then encrypts the data object with the data key. The encrypted data object and the PT are uploaded to the CSS, where the PT is integrated into the secure searchable index.

Fig. 7 shows the pseudo code of the protocol described above, and Table 1 presents the notations used in those functions. In step 1, the EU, which is demoted as U_{id}, plans to save the data object F and its PT into the cloud storage via the EU Object Handler. To make the data secure and searchable, the EU generates an index key SK_x, a data key SK_y, and a location key SK_z for F in step 2.

In step 3, the EU creates the PT called PT_F for F by *PHRASE_TREE_BUILDER* (SK_x, SK_z, F) shown in Fig. 5. The *PHRASE_TREE_BUILDER* function finds each word in F from Line 3 and uses two index keys to encrypt the word w_i at Line 4. The first key is the generated key SK_x, and the second key is generated based on its preceding word, *i.e.*, w_{i-1}. We generate only one encrypted word by using SK_x if the word is the leading word in the document. The second key is needed to support the phrase search because it guarantees the correct order between a keyword and its preceding keyword in the phrase. Once the encrypted words w_i' and w_i'' are prepared, they are pushed into the bloom filter, which represents the encrypted format of w_i. We also use the location key SK_z to encrypt the location information for w_i. The location value is useful for the EU to validate whether a phrase query is matched after the operation of GET Bucket with a privacy-preserving query. When each word in F is processed, the final step is to calculate each BF value in each node of PT_F.

The data object is encrypted to F' in EU in step 4, and PT_F and F' are sent to the CSS over the established encrypted communication channel in step 5. After the CSS Object Handler receives PT_F and F' in step 6, the DT DT_{Uid} and PT list *ptList* are updated by *DOCUMENT_TREE_BUILDER*(U_{id}, F_{id}, PT_F), as shown in Fig. 6. In *DOCUMENT_TREE_BUILDER* , we first copy the BF values in the root node of PT_F to the newly created leaf node of DT_{Uid} and save identifier F_{id} to indicate that this leaf node represents the data object F. Additionally, after calculating every CBF value for the nodes in each level in Line 5, we put the PT PT_F, LOC_{LN}, which corresponds to the location number in DT_{Uid}, and the data object identifier F into *ptList*. *ptList* is used when updating or deleting the data object, which already exists in the CSS.

The encrypted data F' is stored to the Data Storage Server in step 7, and, as the final step, the EU is notified that the PUT operation is complete.

Table 1. Notations used in the protocol of data object operations.

Notation	Meaning	Notation	Meaning
F	Data object owned by EU	SK_z	The location key for EU
F'	Data object encrypted from F	U_{id}	The identifier for EU
F_{id}	The identifier for F	DT_{Uid}	The document tree for U_{id}
SK_x	The index key for EU	PT_F	The phrase tree for F
SK_y	The data key for EU	Q	The query data

```
01  PHRASE_TREE_BUILDER(SK_x, SK_z, F){
02    Create a phrase tree PT_F   /* PT_F is a binary tree for data object F */
03    For each word w_i in F
04      Encrypt w_i with SK_x and with w_{i-1} if i > 1, i.e., w'_i=f_{SKx}(w_i) and w''_i=f_{wi-1}(w_i)
05      Create a leaf node LN for w_i, and add the BF values of w'_i and w''_i to LN
06      Encrypt the location of w_i with SK_z, i.e., l_i=g_{SKz}(loc(w_i)), and tag it to LN
07    From the leaf nodes in PT_F, repeatedly OR the BF value with its sibling, or directly copy the BF
        value if no sibling exists to the parent node in each level of PT_F until the root node is calculated
08    Return PT_F }
```

Fig. 5. PT builder function.

```
01  DOCUMENT_TREE_BUILDER(U_id, F_id, PT_F){
02    If DT_Uid does not exist for owner U_id, create DT_Uid and its leaf node LN
03    Else, create a leaf node LN for DT_Uid
04    Set the CBF value of LN as the BF value of PT_F.root and tag F_id to LN
05    From the leaf node LN in DT_Uid, repeatedly sum up each sibling node of the CBF values or directly
        copy the CBF value, if the node does not have its sibling node, to the parent node in each level
        of DT_Uid until the root node is calculated
06    Add the value {F_id, LOC_LN, PT_F} to ptList /* ptList is a collection including all added PT */
07    Return {DT_Uid, ptList} } /* LOC_LN is the leaf node location number for LN in DT_Uid */
```

Fig. 6. DT builder function.

PUT Object and BFEST

1. **(EU)** is identified as U_{id}, which owns a data object F, and plans to save F and its index into the cloud storage in encrypted format through **EU Object Handler**.
2. **(EU Object Handler)** generates the secret keys SK_x, SK_y, and $SK_z \leftarrow \{0,1\}^m$ as the index key, data key, and location key, respectively, if the keys do not exist.
3. **(EU Object Handler)** uses the index key SK_x and the data object F to build the PT by $PT_F = PHRASE\_TREE\_BUILDER(SK_x, SK_z, F)$.
4. **(EU Object Handler)** uses the data key SK_y to produce the encrypted data $F' = f_{SKy}(F)$.
5. **(EU Object Handler)** regards the object ID of F as F_{id}, and sends $\{F_{id}, PT_F, F'\}$ for U_{id} through the PUT Object operation to **CSS Object Handler** over the established encrypted communication channel.
6. **(CSS Object Handler)** receives $\{U_{id}, F_{id}, PT_F\}$. It then updates the index by $\{DT_{Uid}, ptList\} = DOCUMENT\_TREE\_BUILDER(U_{id}, F_{id}, PT_F)$.
7. **(CSS Object Handler)** stores the encrypted data F' to the cloud file system in **Data Storage Server**.
8. **(CSS)** notifies **EU** that the BFEST and the encrypted data are processed and saved properly through the response of PUT Object operation.

Fig. 7. Protocol for PUT object.

3.3 DELETE Object

Similar to the case of PUT object, we rely on the baseline DELETE object function to delete the data object from the cloud storage. However, the PT and DT corresponding to the data object in BFEST have to be updated. Fig. 9 shows the pseudo code for deleting the data object.

In step 1, the EU deletes the data object in the CSS, so the EU Object Handler uses the Delete Object API to send object ID F_{id} to the CSS Object Handler in step 2. In order to update BFEST and delete F', respectively, the CSS Object Handler uses the method *BFEST_FILE_REMOVER*(U_{id}, F_{id}) shown in Fig. 8. In step 3, PT_F and the location of the leaf node LN in DT_{Uid} for F can be found by seeking *ptList*. To remove information related to F in DT_{Uid}, we subtract the CBF value of LN from the CBF value for every node, respectively, in the shortest path from LN to the root of DT_{Uid}. F' is also deleted by referencing F_{id}. After removing all the related data for F in the CSS, the EU is notified about the status of the operation.

```
01  BFEST_FILE_REMOVER(Uid, Fid){
02     Load the existing document tree DTUid and phrase tree list ptList for owner Uid
03     Find the phrase tree PTF and LOCLN for Fid from ptList
04     Read LOCLN and find out the corresponding leaf node LN of DTUid
05     Subtract a CBF value of LN from each CBF value for the nodes from LN to the root of DTUid
06     Mark LN as unused and remove PTF from ptList
07     Remove the encrypted data F' from the cloud file system according to Fid }
```

Fig. 8. Data object removing function.

DELETE Object and BFEST

1. **(EU)** whose identifier is U_{id} has already put a data object F in the **CSS** and plans to delete F and its index from it through **EU Object Handler**.
2. **(EU Object Handler)** sends the object ID F_{id} through the baseline Delete Object operation to **CSS Object Handler** over the established encrypted communication channel.
3. **(CSS Object Handler)** receives F_{id}, and uses *BFEST_FILE_REMOVER*(U_{id}, F_{id}) to update DT_{Uid} and *ptList*, and then delete F'.
4. **(CSS)** notifies the index, and the encrypted data are deleted for the **EU** through the response of the Delete Object operation.

Fig. 9. Protocol for DELETE object.

3.4 GET Bucket with Privacy Preserving Object Query

The baseline GET Bucket API simply lists the data objects contained in a bucket. To support the query ability over encrypted data, we make the search function available by adding the query data to the request header. Thus, GET Bucket returns the list of data objects, which may contain the query. However, if no query data is set, GET Bucket will return a list all the data objects in the bucket. Fig. 14 shows the pseudo code for the search feature.

In step 1, the EU plans to find the data objects that contain query data Q by using the revised GET Bucket. EU Object Handler then uses the existing index key SK_x to generate the query using the method *QUERY_GENERATOR*(K, Q, *TYPE*) shown in step 2 of

Fig. 10. The method creates a pair of data: a bloom filter *qbf* and the set Q', which represents the encrypted words from Q. There are actually two types of query modes: phrase query and conjunctive query. If the EU wants to query a phrase, the first word in Q is encrypted by SK_x and the following words are encrypted by the preceding word in Q. However, if the EU wants to perform a conjunctive query, all words in Q would be encrypted by SK_x.

Each encrypted keyword is collected into Q' and added into the bloom filter *qbf*. CSS Object Handler receives the query from the revised GET Bucket sent by the EU Object Handler in step 3. Once the CSS receives the query data, it calls *BFEST_TRAVERSER*(U_{id}, *qbf*, Q') to collect the documents that match the query data in set D in step 4. This method, which is shown in Fig. 11, comprises two phases. In the first phase, we start from the root of DT and find the documents that may contain the query data by *BFEST_DT_TRAVERSE*(*dtNode*, *qbf*) in Line 6. Fig. 12 describes the traversing method for DT. We would recursively call this the traversing method if the current node indicates that *qbf* is at least in one of its child nodes. When traversing to the leaf nodes, we collect the document IDs in α as the candidates of matched documents. In the second phase, we traverse each PT, which represents the documents in α by *BFEST_PT_TRAVERSE* (*ptNode*, *keyword*) described in Fig. 13. After the matching process is completed, D is sent back to the EU. If the phrase query mode is selected in the initial step, the EU confirms which documents in D contain the query phrase. The returned location values are decrypted and checked to ascertain whether those values are adjacent and in an incremental order.

```
01  QUERY_GENERATOR(K, Q, TYPE){
02    Create a bloom filter qbf
03    For each word w_i in Q
04      If TYPE is equal to "phrase" then encrypt w_i with K if i=1, or encrypt w_i with w_{i-1},
          i.e., w_1'=f_K(w_1) or w_i'=f_{wi-1}(w_i)
05      If TYPE is equal to "conjunctive" then encrypt w_i with K, i.e., w_i'=f_K(w_i)
06      Add the BF value of w_i' to qbf
07      Add w_i' to the word set Q'
08    Return {qbf, Q'} }
```

Fig. 10. Query data generating function.

```
01  BFEST_TRAVERSER(U_id, qbf, Q'){
02    /* DT.root: The root node of document tree for owner U_id
03      PT_IDn.root: The root node of phrase tree for document n
04      Q' = {Q'_1, Q'_2,..., Q'_m}: a sequence of m encrypted query keywords
05      α: A document set */
06    Find and collect the document IDs which contain each keyword in the phrase query by
        α = BFEST_DT_TRAVERSE(DT.root, qbf)
07    If α is not NULL, for each document ID in α={ID_1, ID_2,..., ID_n}
08      Allocate the corresponding phrase tree PT_IDn and find the encrypted location for each
        keyword in Q' by LOC'_{IDn_Q'm} = BFEST_PT_TRAVERSE(PT_IDn.root, Q'_m)
09      Collect each encrypted location to the location set LOC'_IDn
10    Return res = {<ID, LOC'_IDi>} |1 ≤ i ≤ n}
11    Else
12      No document matches the query keywords, and return res = NULL }
```

Fig. 11. BFEST query processing function.

While the EU figures out which document contains the query data, it uses GET Object to retrieve the data object from the CSS. The way it operates is very similar to the baseline GET Object API, except that the returned data object needs to be decrypted. In fact, the CSS only needs to send back the encrypted data object. The EU can complete the decryption simply by using the decryption function.

```
01  BFEST_DT_TRAVERSE(dtNode, qbf){
02    With respect to the elements with value 1 in qbf, if all the values of the corresponding locations
        in dtNode are greater than 0, it implies some documents are matched to the query keywords
03    If dtNode is NOT a leaf node of DT
04      Call BFEST_DT_TRAVERSE(dtNode.L, qbf) if the left child of dtNode exists
05      Call BFEST_DT_TRAVERSE(dtNode.R, qbf) if the right child of dtNode exists
06    Else /* dtNode is a leaf node of DT. α represent the document ID set */
07      Add the document ID stored in the dtNode to α, and return α
08    Else, no document is matched, return NULL}
```

Fig. 12. DT traversing function.

```
01  BFEST_PT_TRAVERSE(ptNode, keyword){
02    If the bloom filter represented by ptNode contains keyword
03      If ptNode is not a leaf node of PT
04        Call BFEST_PT_TRAVERSE(ptNode.L, keyword) if the left child of ptNode exists
05        If LOC is still NULL /* LOC is used to save location values */
06          Call BFEST_PT_TRAVERSE(ptNode.R, keyword) if the right child of ptNode exists
07      Else /* ptNode is a leaf node of PT */
08        Add the encrypted location value(s) to LOC
09        Return LOC
10    Else /* keyword is not in the ptNode and its belonging nodes */
11      Return NULL }
```

Fig. 13. PT traversing function.

GET Bucket with Privacy Preserving Query

1. **(EU)** is identified as U_{id}, and plans to find all the objects which contain with query data Q in the bucket through **EU Object Handler**.
2. **(EU Object Handler)** uses the existed index key SK_x to build the query data by $\{qbf, Q'\}$ = *QUERY_GENERATOR*(SK_x, Q, "*conjunctive*"|"*phrase*").
3. **(Query Generator)** sends the query data $\{qbf, Q'\}$ through the GET Bucket operation to **CSS Object Handler** over the established encrypted communication channel.
4. **(CSS Object Handler)** receives the query data and collects the matched document IDs into the set D by *BFEST_TRAVERSER*(U_{id}, qbf, Q'). D is sent back to EU.
5. **(EU)** For each ID in D, the EU decrypts each location value in the corresponding LOC'_{IDn} and the document is considered to contain the phrase if the location values are adjacent and in incremental order.

Fig. 14. Protocol for GET Bucket with privacy preserving query.

3.5 Privacy Analysis for Operations

In the PUT Object operation, the index and data keys are used for the PT and the data object, respectively; thus, the CSS cannot attack the user's query privacy by deriving user queries from the data object. The EU builds the PT for the data object, and the CSS only regards the root of PT as the leaf node of DT. Therefore, the CSS can only guess the

existence of some identical words from the value distributions of the bloom filters among each PT. However, in the DELETE Object operation, the EU only sends the identifier F_{id}. The CSS then removes its PT and updates the CBF values for each related node in DT accordingly. The CSS does not learn any query privacy in these actions. Similarly, in the GET Object operation, the EU simply sends the identifier F_{id} of the required data object and decrypts the received data object from the CSS. The CSS cannot learn any further information.

The main privacy issue in our framework is in the GET bucket operation, which involves the query strings, as the CSS manipulates PT and DT during the operation. Each EU query is $\{qbf, Q'\}$, which consists of a bloom filter and encrypted query keywords. For simplicity in the analysis, we assume that the distribution of user queries follows a uniform distribution. Under the threat model given in Section 2.2, there are several different attacks against the EU's query privacy. The first attack is an attack by the CSS on the user's query privacy by deriving the distribution of user queries from the returned data sets. While the CSS could replay the previously received query, this would result in the same data set being returned by the protocol. The CSS does not acquire any additional information regarding the distribution of user queries. As BFEST in the CSS is stored in ciphertext, once we query something, the CSS can only know the association between the data objects, the encrypted keywords, and what data objects the user is interested in.

The CSS may simulate user queries by generating random queries and observe the returned data sets. The CSS could compare the returned data sets from the various simulated user queries and distinguish which encrypted keywords denote the same data set. In practice, we use the 128 or 256 bits long secret key (or even longer) as the index key; the key space is relatively large compared to the real world plaintext we use. As a result, the random queries may easily generate a number of nonexistent encrypted keywords and make the CSS hard to learn from simulation.

The last attack refers to the leakage of the location values residing in the PT leaf nodes. Since the PT contains each location value for the words in the data object, the CSS may infer the meanings encrypted by SK_z. If a PT only contains one keyword, the CSS knows the encrypted value means 1. Once the CSS can read a number of PTs, which contain various numbers of keywords, the location value can be identified one by one. To solve this issue, we can add a random value to the location values for each data object before encrypting with SK_z or use an individual location key for each data object.

As the false positive feature for the bloom filter brings some ambiguity, we hide our query intention by setting the false positive rate to obscure the CSS, even though it increases the amount of returned data object from the CSS to the EU.

3.6 Performance Analysis for Operations

To support our query feature, a number of jobs have to be added into the APIs. In the operation of PUT Object, we build the DT in the CSS and the PTs in the EU. The extra workload for building the PT of the data object is about $O(\overline{w}\log\overline{w})$, where $\overline{w}$ is the average distinct word count for the data object. However, after PT is put into the CSS, the CSS uses the BF value in the root node of PT to update those related CBF values of the related nodes in DT. This update operation costs about $O(\log m)$, where m indicates the number of documents stored in the CSS. When removing the data object from the

CSS by the DELETE Object operation, we not only delete the PT, but also update the CBF values of the related nodes in DT, which also costs about $O(\log m)$.

In GET Bucket, *qbf* in the query is used to determine how many data objects may contain the query keywords with the root of DT in $O(1)$. Besides, it costs $O(m' \cdot \log m)$ to decide which data objects contain the keywords in DT, where m' indicates the number of matched documents. Moreover, it costs $O(m'n \cdot \log\overline{w})$ to retrieve the detail including the location values in the corresponding PTs, where n indicates the number of query keywords. GET Object is not directly involved in query issues; it is only required to perform a decryption after retrieving the data object from the CSS.

Nevertheless, the search feature introduces an extra space usage issue. The CSS stores the DT and a number of PTs. The space is about $O(m\log m/2 \cdot Length(CBF) + m\overline{w}\log\overline{w}/2 \cdot Length(BF))$, where $Length(CBF)$ and $Length(BF)$ mean the bit lengths of counting the bloom filters used in the DT and in the PTs.

4. EXPERIMENTS

We carried out experiments to understand how the selection of BFEST parameters affects the operation of hicloud S3 security. Specifically, we focused on the false positive rate of the query operation and the size of BFEST index. We also studied the performance overhead incurred by BFEST. Our experiment testbed consists of two machines (one as the EU and the other as the CSS) on a local area network. The EU machine is equipped with an Intel i7 3.2 GHz 64-bit processor and 1 GB of DDR3-800 RAM. The CSS machine is equipped with an Intel i7 3.2 GHz 64-bit processor and 4 GB of DDR3-800 RAM. For the EU client program, we modified JetS3 [14], which is compatible with the Amazon S3 and hicloud S3 security APIs, to include the encrypted phrase search functionality. The CSS was running the prototype of hicloud S3 security. The symmetric encryption keys used by the prototype are all 256-bits.

We stored the event logs collected from various network devices (firewalls, proxies, e-mail servers, *etc.*) in a mid-sized corporate environment over a one-hour period to hicloud S3 security. The encrypted phrase search allows system administrators to query for events that match certain conditions (*i.e.*, events with specific dates, IP addresses, error codes). A total of 609,239 event log records are grouped into 200 data objects and stored on hicloud S3 security. The event logs collected from devices with different brands were preprocessed to a standardized format, so they could be explored in the same manner. The event log comprises a number of data fields including date and time, source IP (we used xx.yy to mask the actual digits), destination, and bytes sent and received. Each data field was indexed into the PT.

4.1 BFEST False Positives

As mentioned in Section 3.5, hicloud S3 security takes advantage of the inherent false positives in the bloom filter queries to disguise the access pattern of the cloud storage. It is critical to select the appropriate parameters for bloom filters to ensure optimal security and performance. The bloom filters used by the PTs and DTs are controlled by two parameters: *the number of hash functions* and *the size of the bloom filter*. The false

positive rate of the bloom filters can be derived from these two parameters [6]. Assume each data object has 1,000 distinct words. If we vary the number of hash functions and the length of the bloom filter, the expected false positive (FP) rate will be as shown in Table 2. When increasing the number of hash functions and the size of the bloom filter, the FP for BFEST will drop accordingly.

To validate that the BFEST implementation in the prototype follows the theoretical FP rate during the query operation, we set up the BFEST with different numbers of hash functions. For each setting, we queried the prototype system with six different types of queries as shown in Table 3 and measured the FP rates. The query types include single keywords (log entry ID, IP address, device name, and protocol type) and also phrases (message texts). In addition, we also made queries for nonexistent random strings.

Fig. 15 shows the FP rates of the query for each number of hash functions and query type. In all cases, the FP rate for each query type decreases as the number of hash functions increases. The observations showed that the real FP rate is equal to or smaller than our expectation in the first half cases. However, it has limited effectiveness when we use more hash functions and a longer bloom filter. In the following experiments, we chose to use the setting of three hash functions and a 4328-bit bloom filter to maintain the FP rate at about 10%.

Table 2. BF parameters and the expected FP rate.

# of bloom filter hash functions	Bloom filter length (bits)	Expected BFEST false positive rate
1	1443	50.00%
2	2885	25.00%
3	4328	12.50%
4	5771	6.25%
5	7213	3.13%
6	8656	1.56%
7	10099	0.78%
8	11542	0.39%
9	12984	0.20%
10	14427	0.10%

Table 3. Types of queries.

Query type	Query type description	Keyword
A	Log entry ID	dqmTpKiaoJdzrg==
B	IP address	192.168.xxx.xx
C	Device name	netscreen
D	Protocol type	tcp
E	Message text	system clock
F	Nonexistent	tcpp

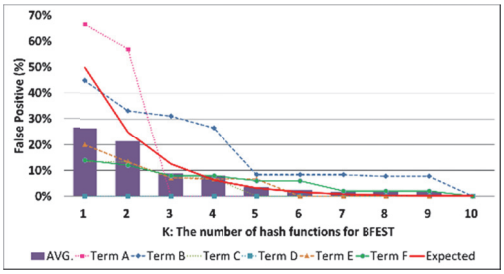

Fig. 15. BFEST false positive rates.

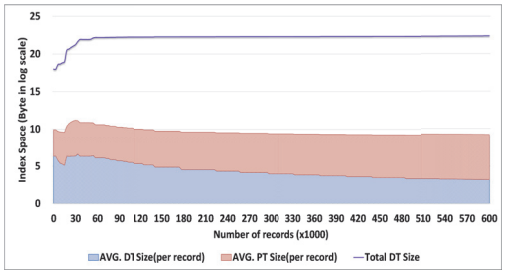

Fig. 16. BFEST index space.

4.2 BFEST Index Space

The parameters of the bloom filter not only control BFEST FP rate, but also affect the length of the bloom filter used in the DT and the PTs. To understand how much index space BFEST used, we calculate the space usage in detail with the parameters that bring less than the 10% average FP rate mentioned in the previous section (3 hash functions/ 4328-bit long bloom filter). In this experiment, we again put all data files into the CSS.

Fig. 16 shows the average DT size for each event log record, the average PT size for each event log record, and the total DT size in logarithm scale. Whenever a new PT is put into the BFEST index space, the total DT size grows logarithmically. Therefore, with respect to the average PT size for each event log record, the average DT size would gradually decrease. As we expected, the overall BFEST index space in the CSS increases in a reasonable manner; the rate of the BFEST index space growth does not go beyond the logarithmic scale.

4.3 PUT Object Operation Processing Time

To evaluate the processing time of PUT Object, we measure the upload time for all event log records to the CSS. The processing time of PUT Object includes the network transmission time and the BFEST update time at the CSS. We do not include the PT creation time because the PT is created by the EU and the creation time is negligible.

We uploaded each successively generated event log file as a data object to the CSS via the PUT Object. The numbers of records in each event log file in fact vary from 1 to 50,000 and in a random order. We sort the processing times by the number of records shown in Fig. 17. The round-shaped points correspond to the PUT object operation times, and the diamond-shaped points correspond to the times for updating the BFEST. We can see that the PUT Object operation time is dominated by the network transmission time as updating the BFEST only consumes a small portion of the overall time.

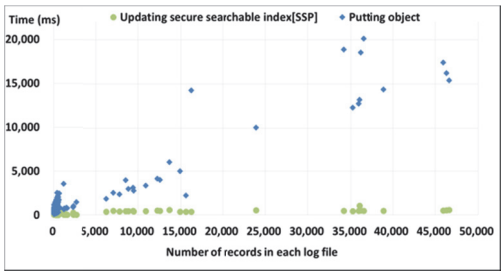

Fig. 17. Performance of PUT object operation.

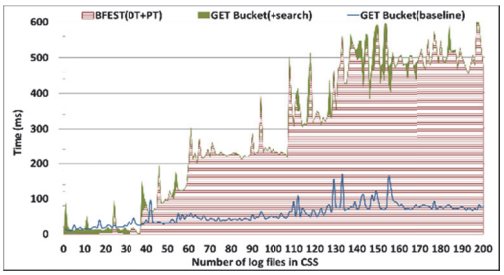

Fig. 18. Performance of GET Bucket operation.

4.4 GET Bucket Operation Processing Time

To evaluate the processing time of GET Bucket, we invoked the GET Bucket API against hicloud S3 security by using IP addresses in the logs as the query strings. As the size of BFEST affects the processing time of the GET Bucket, we also varied the number of log files stored at the CSS and measured the respective GET Bucket processing time.

Fig. 18 shows the experimental result. The Get Bucket (baseline) corresponds to the time taken to list all the log files without any search filtering (*i.e.*, listing the content from all the buckets). BFEST(DT+PT) corresponds to the time taken for searching the DT and PTs. The GET Bucket(+search) corresponds to the overall processing time of the GET Bucket operation on hicloud S3 security (including the time taken by BFEST(DT+PT) at the CSS and the time for handling the returned result at the EU).

We can also see that the time it takes to list all the bucket information by GET Bucket (baseline) is insensitive to the number of log files and remains at about 56.36ms. Nevertheless, the processing time for GET Bucket(+search) and for BFEST(DT+PT) increases with the number of log files stored in the CSS, and unsurprisingly, the manipulation of BFEST occupies the most usage time.

5. RELATED WORK

In terms of privacy-preserving query to an "honest-but-curious" (HBC) cloud storage provider, an encrypted search is a prerequisite option to satisfy the requirement. There are two types of encrypted search schemes: deterministic encrypted search schemes [3, 5, 9, 19, 24, 30] and probabilistic encrypted search schemes [23]. Liu *et al.* [19] introduced a deterministic encrypted search scheme that allows an EU to retrieve an encrypted file owned by a content provider from a cloud storage provider without revealing the query keywords. The primary limitation of the scheme is that it requires a key server to maintain the corresponding relation between files and encryption keys. Bellare *et al.* [3], Sun *et al.* [24] and Zheng *et al.* [30] all proposed public-key based schemes to the encrypted search problem. The public-key based schemes require extra key management and higher computation cost. Song *et al.* [23] proposed a probabilistic scheme to search over encrypted data and hide user query information from the CSS. However, they only encrypt the data, and it takes linear time to search every document to locate the document with the matching keyword. In comparison, the BFEST search index allows the user to determine the presence of given keywords in $O(1)$ time and retrieve the matched documents in $O(\log n)$ time.

The bloom filter has been widely used to implement the search functionality [4, 11, 20, 22] such as checking the presence of keywords in a document. The bloom filter can also be adapted to support advanced search features. Pal *et al.* [20] supported case sensitive and approximate search features with synonyms. The BFEST in our work is an extension of the bloom filter that allows encrypted phrase searches.

Some of the encrypted search schemes employ an additional layer to hide user queries from the data owner [2, 18]. EUs have to fully trust this layer, or the protection of EU query privacy cannot be achieved. Our framework does not require the additional layer, so the EU query privacy can be fully ensured.

Yavuz *et al.* [27] proposed a dynamic searchable encryption that supports single keyword search. In addition, there are a number of works supporting more sophisticated search schemes including verifiable multi-keyword search [8, 25], multi-keyword ranked search [16] and similarity search [28]. However, none of those work supports phrase search.

Many phrase searches, which first appeared in [26, 31], involve two-phase protocols

[15, 21, 26, 31]. Zittrower *et al.* [31] used a trusted server to accomplish an encrypted phrase search. Tang *et al.* [26] and Poon *et al.* [21] used a dictionary to map keywords to unique words in the client side. Our framework can perform phrase search in a single phase and does not require an extra trusted server or a dictionary on the client-side. Li *et al.* [17] proposed a lightweight phrase encrypted search scheme, which uses a lookup table and an array with a number of linked lists for each distinct keyword in documents. The index update cost in the BFEST involves a replacement for each individual PT and an update operation for the DT. By contrast, an update in [17] requires the user to search for every word of the document in the lookup table to locate the places in the linked list for updates. Thus, the overall time complexity is higher than ours.

6. CONCLUSION

We propose a privacy-preserving query framework for encrypted cloud storage. The framework adopts symmetric-key encryption and a tree-based search structure to maintain query performance and ensure query privacy. The secure searchable index (BFEST) in the framework is jointly operated by the EU and the CSS to reduce computation and network communication costs of the EU.

In terms of query format, we support queries in the form of phrases. The framework is flexible enough to suit real-world applications, such as supporting searches in encrypted corporate event logs. The experimental results indicate that the framework can effectively protect the user data and the privacy of user queries. The computation overhead on the EU is negligible, and the communication overhead can be minimized by tuning BFEST parameters to limit the number of candidate data objects returned by the CSS.

REFERENCES

1. Amazon S3, https://aws.amazon.com/tw/s3.
2. S. Artzi, A. Kiezun, C. Newport, and D. Schultz, "Encrypted keyword search in a distributed storage system," Technical Report, MIT-CSAIL-TR-2006-010, Computer Science and Artificial Intelligence Laboratory, MIT, February 2006.
3. M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and efficiently searchable encryption," in *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2007, pp. 535-552.
4. S. M. Bellovin and W. R. Cheswick, "Privacy-enhanced searches using encrypted bloom filters," Technical Report, CUCS-034-07, Department of Computer Science, Columbia University, September 2007.
5. R. Bost, "Forward secure searchable encryption," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, 2016, pp. 1143-1154.
6. A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, Vol. 1, 2004, pp. 485-509.
7. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 668-679.

8. R. Cheng, J. Yan, C. Guan, F. Zhang, and K. Ren, "Verifiable searchable symmetric encryption from indistinguishability obfuscation," in *Proceedings of the 10th ACM Symposium on Information*, *Computer and Communications Security*, 2015, pp. 621-626.
9. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM Cnference on Computer and Communications Security*, 2006, pp. 79-88.
10. Dropbox, https://www.dropbox.com.
11. E.-J. Goh, "Secure indexes," Technical Report, 2003/216, IACR ePrint Cryptology Archive, March 2004, Google Cloud Storage, https://cloud.google.com/storage.
12. Google Cloud Storage, https://cloud.google.com/storage.
13. hicloud S3, http://hicloud.hinet.net/s3.
14. JetS3, http://www.jets3t.org.
15. Z. A. Kissel and J. Wang, "Verifiable phrase search over encrypted data secure against a semi-honest-but-curious adversary," in *Proceedings of IEEE International Conference on Distributed Computing Systems Workshops*, 2013, pp. 126-131.
16. H. Li, D. Liu, Y. Dai, T. H. Luan, and X. S. Shen, "Enabling efficient multi-keyword ranked search over encrypted mobile cloud data through blind storage," *IEEE Transactions on Emerging Topics in Computing*, Vol. 3, 2015, pp. 127-138.
17. M. Li, W. Jia, C. Guo, W. Sun, and X. Tan, "LPSSE: Lightweight phrase search with symmetric searchable encryption in cloud storage," in *Proceedings of International Conference on Information Technology: New Generations*, 2015, pp. 174-178.
18. M. Li, S. Yu, N. Cao, and W. Lou, "Authorized private keyword search over encrypted data in cloud computing," in *Proceedings of International Conference on Distributed Computing Systems*, 2011, pp. 383-392.
19. H.-X. Liu, J.-X. Dai, and C. Jiang, "Research on privacy preserving keyword search in cloud storage," in *Proceedings of IEEE International Conference on Computer Science and Information Technology*, 2010, pp. 444-446.
20. S. K. Pal, P. Sardana, and A. Sardana, "Efficient search on encrypted data using bloom filter," in *Proceedings of International Conference on Computing for Sustainable Global Development*, 2014, pp. 412-416.
21. H. T. Poon and A. Miri, "An efficient conjunctive keyword and phase search scheme for encrypted cloud storage systems," in *Proceedings of IEEE International Conference on Cloud Computing*, 2015, pp. 508-515.
22. M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin, "Secure anonymous database search," in *Proceedings of ACM Workshop on Cloud Computing Security*, 2009, pp. 115-126.
23. D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of IEEE Symposium on Security and Privacy*, 2000, pp. 44-55.
24. S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for Boolean queries," in *Proceedings of the 21st European Symposium on Research in Computer Security*, Vol. 9878, 2016, pp. 154-172.
25. W. Sun, X. Liu, W. Lou, Y. T. Hou, and H. Li, "Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data," in

Proceedings of IEEE Conference on Computer Communications, 2015, pp. 2110-2118.
26. Y. Tang, D. Gu, N. Ding, and H. Lu, "Phrase search over encrypted data with symmetric encryption scheme," in *Proceedings of International Conference on Distributed Computing Systems Workshops*, 2012, pp. 471-480.
27. A. A. Yavuz and J. Guajardo, "Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware," in *Proceedings of the 22nd International Conference on Selected Areas in Cryptography*, Vol. 9566, 2016, pp. 241-259.
28. X. Yuan, H. Cui, X. Wang, and C. Wang, "Enabling privacy-assured similarity retrieval over millions of encrypted records," in *Proceedings of the 20th European Symposium on Research in Computer Security*, Vol. 9327, 2015, pp. 40-60.
29. Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 707-720.
30. Q. Zheng, S. Xu, and G. Ateniese, "VABKS: Verifiable attribute-based keyword search over outsourced encrypted data," in *Proceedings of IEEE Conference on Computer Communications*, 2014, pp. 522-530.
31. S. Zittrower and C. C. Zou, "Encrypted phrase searching in the cloud," in *Proceedings of IEEE Global Communications Conference*, 2012, pp. 764-770.

Yen-Chung Chen (陳彥仲) received his B.S. degree in Computer Science and Information Engineering from Fu Jen Catholic University, Taipei County, Taiwan in 2001; and M.S. degree in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan in 2003. He joined Chunghwa Telecom Laboratories in 2004, and is currently working at Information and Communication Security Laboratory and pursuing Ph.D. degree in the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. His research interests include information security, cloud computing and communication networks.

Yu-Sung Wu (吳育松) received B.S. in Electrical Engineering from National Tsing Hua University, Hsinchu, Taiwan in 2002, M.S. and Ph.D. in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana in 2004 and 2009. In 2009, he joined National Chiao Tung University in Hsinchu, Taiwan, where he is currently an Associate Professor of the Computer Science Department and Director of Laboratory of Security and Systems. Previously, he had worked at Purdue CERIAS research center conducting research on the design of automated response system for distributed applications. He had also worked at Avaya Labs in New Jersey developing prototypes of intrusion detection system for VoIP environment. Prof. Yu-Sung

Wu is a member of IEEE and ACM. He had served on the committees of several conferences including DSN, ICDCS, SERE, and APNOMS.

Wen-Guey Tzeng (曾文貴) received his B.S. degree in Computer Science and Information Engineering from National Taiwan University, Taiwan, 1985; and M.S. and Ph.D. degrees in Computer Science from the State University of New York at Stony Brook, USA, in 1987 and 1991, respectively. He joined the Department of Computer Science, National Chiao Tung University, Taiwan, in 1991. His current research interests include security data analytics, cryptology, information security and network security.