A Practical Framework for Executing Complex Queries over Encrypted Multimedia Data

Fahad Shaon and Murat Kantarcioglu

The University of Texas at Dallas, Richardson, TX 75080, USA
{fahad.shaon,muratk}@utdallas.edu

Abstract. Over the last few years, data storage in cloud based services has been very popular due to easy management and monetary advantages of cloud computing. Recent developments showed that such data could be leaked due to various attacks. To address some of these attacks, encrypting sensitive data before sending to cloud emerged as an important protection mechanism. If the data is encrypted with traditional techniques, selective retrieval of encrypted data becomes challenging. To address this challenge, efficient searchable encryption schemes have been developed over the years. Almost all of the existing searchable encryption schemes are developed for keyword searches and require running some code on the cloud servers. However, many of the existing cloud storage services (e.g., Dropbox[1], Box[2], Google Drive[3], etc.) only allow simple data object retrieval and do not provide computational support needed to realize most of the searchable encryption schemes.
In this paper, we address the problem of efficient execution of complex search queries over wide range of encrypted data types (e.g., image files) without requiring customized computational support from the cloud servers. To this end, we provide an extensible framework for supporting complex search queries over encrypted multimedia data. Before any data is uploaded to the cloud, important features are extracted to support different query types (e.g., extracting facial features to support face recognition queries) and complex queries are converted to series of object retrieval tasks for cloud service. Our results show that this framework may support wide range of image retrieval queries on encrypted data with little overhead and without any change to underlying data storage services.

1 Introduction

Cloud computing is being adopted by organizations and individuals to address various types of computation needs including file storage, archiving, etc. However, there have been several incidents of data leak in popular cloud storage service providers [1,32]. To ensure the security of the sensitive data and prevent any unauthorized access, users may need to encrypt data before uploading

[1] https://www.dropbox.com
[2] https://www.box.com/
[3] http://drive.google.com/

to cloud. If data uploaded to cloud is encrypted using traditional encryption techniques, executing search queries on the stored data become infeasible. To alleviate this situation many searchable encryption techniques have been proposed [12,20,6,16,7,24,25,14,5]. Among those approaches, searchable symmetric encryption (SSE) [12,20,6,16,7,24,5] emerges as an efficient alternative for cloud based storage systems due to minimal storage overhead, low performance overhead, and relatively good security.

However, almost all searchable encryption techniques require executing some code on the cloud servers to enable efficient processing. On the other hand, popular commercial personal cloud storage providers[1] [2] [3] only support basic file operations like read and write file that makes it infeasible to apply traditional SSE techniques. Furthermore, complex queries on multimedia data may require running different and expensive cryptographic operations. These limitations create a significant problem for wide adoption of SSE techniques. Therefore, developing SSE schemes that can run on the existing cloud storage systems without requiring the cloud service providers cooperation emerges as an important and urgent need. To our knowledge, only [24] considered a setup without computational support from the cloud storage but the proposed solution does not support efficient complex querying over encrypted data.

Even though, one can wish that an alternative SSE as a service could be offered in the near future by the cloud service providers, due to network effects, many of the existing users may not want to switch their cloud service providers. Therefore, any new "secure" cloud storage with SSE providers may have a hard time in getting significant traction. So supporting SSE on the existing cloud storage platforms without requiring any support from the cloud storage service providers is a critical need.

In addition, adoption of multimedia (e.g., image, music, video, etc.) data for social communication is increasing day by day. KPCB analyst Mary Meeker's 2014 annual Internet Trends report[4] states *1.8 billion* photos shared *each day*. However, indexing multimedia data is harder compared to text data. A significant pre-processing is required to convert raw multimedia data to a searchable format and queries made on multimedia data are complex as well. So building efficient cryptographic storage system that can easily handle multimedia content is a very important problem.

To address these challenges, in this paper, we propose an efficient searchable encryption scheme framework that can work on existing cloud storage services and can easily handle multimedia data. Our proposed framework only requires file storage and retrieval support from cloud storage services. Furthermore, by leveraging the extensible extract, transform and load operations provided by our framework, very complex queries can be executed on the encrypted data. As an example, we show how our framework could be used to run face recognition queries on encrypted images. To our knowledge, this is the first system that can support *complex queries* on encrypted multimedia data *without significant computational support* from the cloud service provider (i.e., without running cus-

[4] `http://www.kpcb.com/blog/2014-internet-trends`

tomized code in the cloud). *Main contributions* of this work can be summarized as follows:

- We propose a generic outsourcing framework that enables secure and efficient querying on any data. Our framework supports complex querying on any encrypted data by allowing queries to be represented as series of simple equality queries using the features extracted from the data. Later on, these extracted features are transformed into encrypted indexes and these indexes are loaded to cloud and leveraged for efficient encrypted query processing.
- We prove that our system satisfies adaptive semantic security for dynamic SSE.
- We show the applicability of our framework by applying it to state-of-the-art image querying algorithms (e.g., face recognition) on encrypted data.
- We implement a prototype of our system and empirically evaluate the efficiency under various query types using real world cloud services. Our results show that our system introduces very little overhead, which makes it remarkably efficient and applicable to real-world problems.

The rest of the paper is organized as follows: Section 2 discusses previous related works, Section 3 provides the general setup and threat model of our system, Section 4 describes internal details of each phases, Section 5 extends our initial framework making it dynamic, in Section 6 we formally prove the security of our system, Section 7 shows an application of our proposed framework, Section 8 shows the experimentations, and in Section 9 we conclude our work.

2 Related Work

Currently there are few ways to build encrypted cloud storage with content based search. Searchable symmetric encryption(SSE) is one of those, which allows users to encrypt data in a fashion that can be searched later on. Different aspects of SSE has been studied extensively as shown in an extensive survey of provably secure searchable encryption by Bösch at el. in [5]. Curtmola at el. [12] provided simple construction for SSE with practical security definitions, which was then adopted and extended by several others in subsequent work. Few works also looked into dynamic construction of SSE [20,6,16,17] so that new documents can be added after SSE construction.

Another branch of study related to SSE is supporting conjunctive boolean query. Cash at el. [7] proposed such a construction, where authors used multi-round protocol for doing boolean query with reasonable information leakage. In the process they also claimed to build the most efficient SSE in terms of time and storage. Kuzu at el. [18] proposed an efficient SSE construction for similarity search, where they used locality sensitive hashing to convert similarity search to equality search. There are also work towards supporting efficient range query, substring matching query, etc. [13], where a rich query is converted to an exact matching query. However, these constructions require specialized server. Importantly, we can easily adopt such a conversion technique in our framework.

Naveed at el. [24] proposed a dynamic searchable encryption schema with simple storage server similar to our setup. The system also hides certain level of access pattern. However, authors did not consider complex query problem in their work, which is one of the major challenges that we solved in this work.

Another way of querying encrypted database is oblivious RAM (ORAM) described by Ostrovsky [25] and Goldreich at el. [14], which also hides search access pattern and much secure. Despite recent developments [28,34,33], traditional ORAM remains inefficient for practical usage in cloud storage system as described in [4,23]. Furthermore, our proposed system converts complex operations into sequence of key value read and write operations, which can easily be combined with ORAM technique to hide the access pattern.

Qin at el. [29] proposed an efficient privacy preserving cloud based secure image feature extraction and comparison technique. Similar construction for ranked image retrieval is proposed by [39,21,30]. These systems depend on highly capable cloud server for preforming image similarity query.

Finally, there are few commercial secure cloud storage systems, e.g., SpiderOak[5], BoxCryptor[6],Wuala[7], etc. Even though these systems are easy to use and provide reliable security, these systems provide neither server based search nor complex query support. All these systems depend on either operating system or local indices to provide search functionalities. As a result, to provide search functionalities these systems need to download and decrypt all the data stored in cloud server, which might not be efficient solution in all circumstances.

3 Background and Threat Model

Searchable Symmetric Encryption (SSE) is one of the many mechanisms to enable search over encrypted data. In a SSE schema, we not only encrypt the input dataset, but also we create an encrypted inverted index. The index contains mapping of encrypted version of keywords (called trapdoors) to list of document ids that contains corresponding plain text keywords. Formally, a SSE schema is defined as collection of 5 algorithms $SSE = (Gen, Enc, Trpdr, Search, Dec)$ Given security parameter Gen generates a master symmetric key, Enc generates the encrypted inverted index and encrypted data sets from the input dataset. $Trpdr$ algorithm takes keywords as input and outputs the trapdoor, which is used by $Search$ algorithm to find list of documents associated with input keywords. Finally, the Dec algorithm decrypts the encrypted document given the id and proper key. We refer the reader to [12] for further discussion of SSE. Furthermore, in a typical SSE settings, Gen, Enc, $Trpdr$, and Dec are performed in a client device and the $Search$ algorithm is performed in a cloud server. For this reason, we need a server with custom computational support to run a SSE based system. Here, we focus on building a framework that enables us to build SSE alike schema with complex query processing capabilities using file storage servers that does not have custom computation support.

[5] `https://spideroak.com/`
[6] `https://www.boxcryptor.com/`
[7] `https://www.wuala.com/`

Threat Model. In this study, we consider a setup, where a user owns a set of documents, which includes multimedia documents. User wants to store these documents into a cloud storage server in encrypted form. User also wants to perform complex search queries over the encrypted data. Most importantly, user wants to utilize existing cloud storage service, which is not capable of executing any custom code provided by user. Formally cloud storage server $\mathcal{Z}$ can *only* preform *read* and *write* operations. This simple requirement of cloud storage server makes the system easily adoptable in several real world scenario. On the other hand, user have devices with sufficient computation power that can perform modern symmetric cryptography algorithms and are called clients.

In our system, the communication between server and client is done over encrypted channel, such as https. So eavesdroppers can not learn any meaning full information about the documents capturing the communication, apart form existence of such communication. We also assume that the cloud storage server $\mathcal{Z}$ is managed by Bob, who is semi-honest. As such, he follows the protocol as it is define but he may try to infer private information about the document he hosts. Furthermore, the system does not hide search access pattern, meaning Bob can observe the trapdoors in search query. Based on the encrypted file accesses after subsequent search queries Bob also can figure out trapdoor to document ids assignments. However, Bob can not observe the plain text keyword of trapdoors.

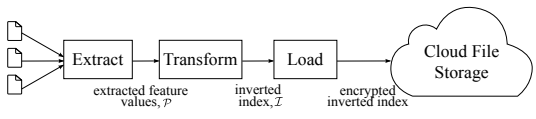
4 The Proposed System

Our main motivation is to build encrypted cloud storage that can support complex search query with support of simple file storage server. We generalize the required computations into a five phase *Extract, Transform, Load, Query, Post-Process (ETLQP)* framework. These five phases represent chronological order of operations required to create, store encrypted index, and perform complex operations. Figure 1(a) and 1(b) illustrates an overview of different phases in our system.

4.1 Extract

In this phase we extract necessary features from a dataset. Let, $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ be a set of documents, $id(d_i)$ be the identifier of document d_i, $\Theta = \{\theta_1, \theta_2, ..., \theta_m\}$ be a set of m feature extractor functions. Functions in Θ can extract set of feature and value pairs (f, v) from documents. We build list U_i with all the feature value pairs extracted from d_i. For all the feature extractors $\theta_j \in \Theta$ we compute $(f, v) \leftarrow \theta_j(d_i)$ and store (f, v) in U_i. Finally we organize the result in $\mathcal{P}$, such that $\mathcal{P}[id(d_i)] \leftarrow U_i$. Such an example $\mathcal{P}$ is illustrated in Figure 1(c). Here, we have four documents $\{D_1, ..D_4\}$. D_1 has feature value pairs $U_1 = \{(f_a, v_\alpha), (f_b, v_\beta), (f_b, v_\gamma)\}$, etc.

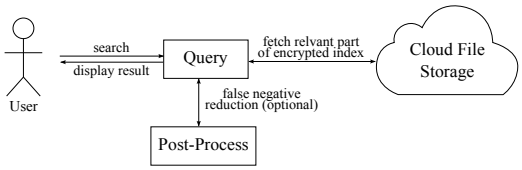
To make the process more concrete, let us assume that, we want to build an encrypted image storage application that can preform location based query over the encrypted images. In other word, the system is capable of answering queries, such as, *find images taken in Italy.* To support such a query, we implement a



(a) Index creation, encryption and upload

Document ID	Feature Value pairs
$id(D_1)$	$(f_a, v_\alpha), (f_b, v_\beta), (f_b, v_\gamma)$
$id(D_2)$	$(f_a, v_\sigma), (f_b, v_\beta)$
$id(D_3)$	$(f_a, v_\alpha), (f_b, v_\gamma)$
$id(D_4)$	$(f_a, v_\delta), (f_b, v_\beta), (f_b, v_\gamma)$

(c) Extracted feature-values, $\mathcal{P}$



(b) Query and post-process phase to search content

Search Signature	Document ID List
s_1	$id(D_1), id(D_3)$
s_2	$id(D_1), id(D_2), id(D_4)$
s_3	$id(D_1), id(D_3), id(D_4)$
s_4	$id(D_2)$
s_5	$id(D_4)$

(d) Inverted index, $\mathcal{I}$

Fig. 1. Overall workflow of our proposed system and important data structures. (a) Index creation consists of extract, transform and load phases. (b) Search consists of query and post-process phases. (c) $\mathcal{P}$, output of extract phase that maps document ids to feature value pairs, (d) Inverted index $\mathcal{I}$, that maps search signatures to document ids.

feature extractor function θ_l, where θ_l extracts location information from image meta data. Output of θ_l is defined as a feature value pair (*"LOCATION", "longitude and latitude of image"*). We define as many feature extractor necessary based on application need. However, all feature extractor functions returns values in similar format. In Section 7 we discuss in details how we defined more feature extractors and use those to answer much more complicated queries.

4.2 Transform

In this phase we transform the extracted feature values into much simpler form so that complex search operations can be expressed as series of equality searches. We compute search signatures s form feature-value pairs and associate corresponding documents with s. This association at query stage can be used to infer existence of a feature-value pair in a document. Essentially here we define sets of transform functions $\mathcal{T} = \{t_1, .., t_p\}$, where each transform function is designed to generate search signatures from a feature value pair (f, v) and $\mathcal{T}_f$ defines subset of transformation functions that can be applied to feature f.

With these transform functions $\mathcal{T}$, we generate an inverted index $\mathcal{I}$ that is indexed by search signatures and contains list of document ids. For all the feature value pairs in $\mathcal{P}$, we generate search signature $s^t_{f,v} \leftarrow t(f, v)$ where $t \in \mathcal{T}_f$. We build document id list V_s for all the unique search signature s that contains $id(D_i)$ if and only if there exists a feature value pair (f, v) that is in U_i and at least one transformation function t that generates search signature s. Finally we fill the inverted index $\mathcal{I}$ such that $\mathcal{I}[s] \leftarrow V_s$. In Figure 1(d) we show such an example $\mathcal{I}$, which is created from $\mathcal{P}$ of Figure 1(c). Here, search signature s_1, s_2, s_3, s_4, s_5 are generated from feature value pairs (f_a, v_α), (f_b, v_β), (f_b, v_γ), (f_a, v_σ), (f_a, v_δ) accordingly.

Similarly, in our encrypted image storage application example, we define a transform function t_l that takes geographic location and document id as input,

converts the location information to mailing address using reverse address lookup service, takes the country information and document id to construct a search signature using a collision resistant hash function.

Using such extract transform model has several benefits over adhoc model. The proposed model helps us to organize the necessary computation into modules, which intern increase development efficiency. The feature extractor functions can be reused in other project.

Algorithm 1 describes *extract* and *transform* phases for building inverted index.

Algorithm 1 Extract and transform algorithm for building inverted index

Extract
Require: $\mathcal{D}$ = Document set, Θ = Feature extractor function set.
$\mathcal{P} \leftarrow$ empty hash table.
for all document d in $\mathcal{D}$ **do**
    $U \leftarrow$ empty list
    for all feature extractor θ in Θ **do**
        $(f, v) \leftarrow \theta(d)$ and add to list U
    end for
    $\mathcal{P}[id(d)] \leftarrow U$
end for
return $\mathcal{P}$

Transform
Require: $\mathcal{P}$ = Extracted feature-value hash table, $\mathcal{T}$ = Transform function set
$\mathcal{I} \leftarrow$ empty hash table
for all document id $id(d)$ in $\mathcal{P}$ **do**
    for all feature-value pair (f, v) in $\mathcal{P}[id(d)]$ **do**
        for all transformation function t in $\mathcal{T}_f$ **do**
            $s \leftarrow t(f, v)$ and add $id(d)$ to $\mathcal{I}[s]$
        end for
    end for
end for
return $\mathcal{I}$

4.3 Load

In this phase we setup our encryption schema, encrypt the inverted index, and upload the encrypted version into a file storage server $\mathcal{Z}$. We initialize a master encryption key K, three random constants C_1, C_2, C_3, a secure pseudo random permutation function φ, and a keyed pseudo random function H. Given a key, φ encrypts data, φ^{-1} decrypts corresponding result, and H generates authentication code of messages. The pseudo random permutation φ takes an encryption key and an arbitrary length binary string as input and outputs a cipher text. Given output cipher text and corresponding encryption key the inverse φ^{-1} will output the original message back. We are also assuming that, output of φ

is indistinguishable under non-adaptive and adaptive chosen ciphertext attack (IND-CCA1, IND-CCA2). The keyed pseudo random function H also takes an encryption key and an arbitrary length binary string as input and outputs a fixed length binary string. In addition, we define a small synchronized cache $\mathcal{C}$ and an encryption key K_C for encrypting the cache. $\mathcal{C}$ is always synchronized with storage server $\mathcal{Z}$. Synchronization is achieved by updating the server's version after any change in client's version and before updating the cache locally most recent version is downloaded from the server first. In $\mathcal{C}$, we store document id list size of all search signatures of $\mathcal{I}$, which is notated by $\mathcal{C}.freq$. Later, we also use this cache to store information related to individual files to make the query phase easier.

We divide all the document id lists in $\mathcal{I}$ into b length blocks and add padding to last block if needed. The value of b is determined by defining and minimizing a cost function (described in Subsection 4.6). We generate trapdoors $T_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_1)$ and $K_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_2)$ for j^{th} block of document list of $\mathcal{I}[s]$. We use K_j^s to encrypt block contents and T_j^s as the key for encrypted inverted index $\mathcal{E}$. So $\mathcal{E}[T_j^s] \leftarrow \varphi(K_j^s,\ j^{th}\ block\ of\ \mathcal{I}[s])$. To query the inverted index later on, our system will regenerate these two trapdoors and perform inverse operations to build the original document id list. In addition, we store number of documents associated with a signature s in $\mathcal{C}.freq[s]$, then encrypt and upload the cache. Algorithm 2 describes the operations necessary for *load* phase.

4.4 Query

In previous phases we have created an encrypted inverted index and uploaded into file storage server $\mathcal{Z}$. Query and post-process phases are dedicated for querying the index and returning proper output to user. First, given a user query q, we extract and transform it to a set of search signatures $\mathcal{Q}$. We use number of

Algorithm 2 Load encrypted index

Require: K = Master key, $\mathcal{I}$ = Inverted index of search signatures, $\mathcal{C}$ = Synchronized cache, K_C = encryption key for cache, $\mathcal{Z}$ = File storage server.
$b \leftarrow optimize(\mathcal{I})$
for all signature s in $\mathcal{I}$ **do**
    $blocks_s \leftarrow \lceil \frac{|\mathcal{I}[s]|}{b} \rceil$
    for $j = 1 \rightarrow blocks_s$ **do**
        $T_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_1)$, $K_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_2)$
        $sub \leftarrow \mathcal{I}[s].slice((j-1) \times b, j \times b)$
        $\mathcal{E}[T_j^s] \leftarrow \varphi(K_j^s, pad(sub))$
    end for
    $\mathcal{C}.freq[s] \leftarrow |\mathcal{I}[s]|$
end for
for all trapdoor t in $\mathcal{E}$ **do**
    $\mathcal{Z}.write(t, \mathcal{E}[t])$
end for
$C_{sig} \leftarrow H(K_C \mid\mid C_3, 1)$
$\mathcal{Z}.write(C_{sig}, \varphi(K_C, \mathcal{C}))$

document ids per block, stored in $\mathcal{C}.freq$, to compute block counts, which in turn used to compute trapdoors K_j^s and T_j^s for each block of search signatures. Using these trapdoors we retrieve and decrypt document ids. Finally, the result is organized into a hash table $\mathcal{R}$ such that $\mathcal{R}[s] = \mathcal{I}[s]$ for all $s \in \mathcal{Q}$. Algorithm 3 contains the detail operations of query phase.

Algorithm 3 Query

Require: K = Master key, q = Query, b = block size, $\mathcal{Z}$ = File storage server
$\mathcal{Q} \leftarrow$ Extract and Transform q
for all search signatures s in $\mathcal{Q}$ **do**
    $blocks_s \leftarrow \lceil \frac{\mathcal{C}.freq[s]}{b} \rceil$
    for $i = 1 \rightarrow blocks_s$ **do**
        $T_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_1)$, $K_j^s \leftarrow H(K, s \mid\mid j \mid\mid C_2)$
        $L \leftarrow \mathcal{Z}.read(T_j^s)$
        add $\varphi^{-1}(K_j^s, L)$ in $\mathcal{R}[s]$
    end for
end for
return $\mathcal{R}$

4.5 Post-process

In this step we further process the result of query phase to remove false positive entries. Given result set $\mathcal{R}$ from query phase for query q, we remove id of document that does not match the original query. Therefore, $\mathcal{R}.remove(id(d))$ if $q(d) \neq True$. Query that only contains exact search features, this phase is optional.

4.6 Optimal block size analysis

Block size has a direct impact on performance of our proposed system. Larger block size implies waste of space for padding and smaller block size implies many blocks to process. So we need to find an optimal value of block size b that keeps the over all cost to minimal. In our construction for each block we have a fixed cost and a dynamic cost that is related to block length. We define fixed cost as α and co-efficient of dynamic cost β. Cost can be in terms of time and size. Both linearly depends on block size in our construction. So cost for a b length block is $(\alpha + \beta \times b)$. Let, $\mathcal{J}(s)$ is $|\mathcal{I}[s]|$ meaning document id list size for search signature s and total cost $\mathcal{G}(b)$ for blocking and encrypting given inverted index $\mathcal{I}$ for block length b then

$$\mathcal{G}(b) = \sum_{s \in \mathcal{I}} \left\lceil \frac{\mathcal{J}(s)}{b} \right\rceil (\alpha + \beta \times b)$$

We want to minimize the above function for b. However, if contains a ceiling function, which can not be minimize by taking derivatives and equating to zero. So we approximate the probability distribution of $\mathcal{J}$, i.e., lengths of document id list in $\mathcal{I}$. We assumed that, distribution is Pareto distribution [3], which is defined

by probability density function (PDF) $f(x|\gamma, x_m) = \frac{\gamma x_m^\gamma}{x^{(\gamma+1)}}$, and cumulative distribution function (CDF) $F(x|\gamma, x_m) = 1 - (\frac{x_m}{x})^\gamma$, where x is the random variable, γ is distribution parameter, and x_m is minimum value of x.

In our total cost analysis for each $\mathcal{J}(s)$ smaller or equal b cost is exactly $(\alpha+\beta\times b)$ and number of elements where $\mathcal{J}(s) \leq b$ is equal to $F(b)$. For elements where $\mathcal{J}(s) > b$ we can approximate the total cost using expected value of $\mathcal{J}(s)$. Finally, the cost function

$$\mathcal{G}(b) = (\alpha + \beta b)F(b) + E[\mathcal{J}(s)]_{\mathcal{J}(s)>b}(\frac{\alpha + \beta b}{b})$$

where E is expectation of probability distribution. Now we can compute the expectation by integration.

$$\mathcal{G}(b) = (\alpha + \beta b)(1 - (\frac{x_m}{b})^\gamma) + (\frac{\alpha + \beta b}{b}) \int_b^\infty \frac{\gamma x_m^\gamma x}{x^{\gamma+1}} dx$$

After preforming integration and several algebraic simplification we get the final form

$$\mathcal{G}(b) = (\alpha + \beta b) - (\alpha + \beta b)x_m^\gamma b^{-\gamma} + (\gamma x_m^\gamma \frac{b^{-\gamma} + 1}{\gamma - 1})(\frac{\alpha}{b} + \beta)$$

And the first order derivative is

$$\mathcal{G}'(b) = \beta - x_m^\gamma \beta b^{-\gamma} + (\alpha + \beta b)x_m^\gamma \gamma b^{-\gamma-1} - \gamma x_m^\gamma b^{-\gamma}(\frac{\alpha}{b} + \beta) - \frac{\gamma x_m^\gamma}{\gamma - 1} b^{-\gamma-1}\alpha$$

Now we minimize b by setting $\mathcal{G}'(b) = 0$ and solving the equation for b. In experimentation we observe that method of moments estimation for x_m and γ gives almost correct value.

5 Dynamic Document Addition

Here we are going to improve our algorithms to support dynamic addition of documents. Given a new document set D' for addition, we first perform extract and transform to build an inverted index $\mathcal{I}'$. Now we download and decrypt the cache $\mathcal{C}$ and compute number of blocks x, number of empty spaces in last block y from $\mathcal{C}.freq$ information for signatures that are already in inverted index $\mathcal{I}$. On the other hand assign zero to x and y for search signatures that we have not seen yet. If there is empty space meaning $y > 0$ then we fill the last block with new document ids. Rest of the document ids are divided into b length blocks and encrypted with appropriate key. Algorithm 4 describes dynamic document addition in details.

5.1 Bandwidth Requirement Analysis

One might argue that, since we are performing all the complex operations on client side, so encrypt the inverted index $\mathcal{I}$ like another document; then download, decrypt, and search in the local inverted index in time of query to avoid all

Algorithm 4 Dynamic document addition

Require: D' = Documents to add, K = Master key, C_1, C_2, C_3 = Constants, b = block size, K_C = Encryption key for cache, Z = File storage server, Θ = Feature extractor function set, $\mathcal{T}$ = Transform function set.
$\mathcal{I}' \leftarrow Transform(Extract(D', \Theta), \mathcal{T})$
$C_{sig} \leftarrow H(K_C \mid\mid C_3, 1)$
$\mathcal{C} \leftarrow \varphi^{-1}(K_C, \mathcal{Z}.read(C_{sig}))$ // download and decrypt
for all signature s in $\mathcal{I}'$ **do**
  if s in $\mathcal{C}.freq$ **then**
    $x \leftarrow \lceil \frac{\mathcal{C}.freq[s]}{b} \rceil$, $y \leftarrow x \times b - \mathcal{C}.freq[s]$
  else $x \leftarrow 0$, $y \leftarrow 0$
  end if
  if $y > 0$ **then**
    $T_x^s \leftarrow H(K, s \mid\mid x \mid\mid C_1)$, $K_x^s \leftarrow H(K, s \mid\mid x \mid\mid C_2)$
    $L \leftarrow \varphi^{-1}(K_x^s, \mathcal{Z}.read(T_x^s))$
    Fill empty spaces in L
    $\mathcal{Z}.write(T_x^s, \varphi(K_x^s, L))$
  end if
  for $j = 1 \rightarrow \lceil \frac{|\mathcal{I}'[s]| - y}{b} \rceil$ **do**
    $k \leftarrow j \ + \ x$
    $T_k^s \leftarrow H(K, s \mid\mid k \mid\mid C_1)$, $K_k^s \leftarrow H(K, s \mid\mid k \mid\mid C_2)$
    $sub \leftarrow \mathcal{I}'[s].slice((k-1) \times b + y, j \times b + y)$
    $\mathcal{Z}.write(T_k^s, \varphi(K_k^s, pad(sub)))$
  end for
  $\mathcal{C}.freq[s] \leftarrow \mathcal{C}.freq[s] \ + \ |\mathcal{I}'[s]|$
  $\mathcal{Z}.write(C_{sig}, \varphi(K_C, \mathcal{C}))$ // encrypt and upload
end for

the complexities. However, such approach will increase bandwidth consumption for dynamically updating index.

Let, $\{q_1,...,q_\varrho\}$ be ϱ consecutive queries that user like to perform on a dynamically updating index, (i.e., new documents are added in between each query), $|q_i|$ be the length of query q_i, $|\mathcal{E}(q_i)|$ be the size of blocks returned by query q_i, $|\mathcal{Y}|$ be the maximum among $\{|\mathcal{E}(q_1)|,..., |\mathcal{E}(q_\varrho)|\}$, $|\mathcal{I}|$ be the size of inverted index, $|\mathcal{C}.freq|$ be the size of cache required for storing frequency of all the buckets. Total bandwidth cost for preforming n queries ignoring the addition cost

$$\varrho|\mathcal{C}.freq| + \sum_{i=1}^{\varrho}(|\mathcal{E}(q_i)| + |q_i|) \leq \varrho(|\mathcal{C}.freq| + |\mathcal{Y}| + |q_i|)$$

On the other hand, if we keep a local inverted index the bandwidth cost would be simply $\varrho|\mathcal{I}|$. Since after each update index is updated and we need to download the recent version. In practice $|\mathcal{C}.freq| + |\mathcal{Y}| + |q_i| \ll |\mathcal{I}|$. Also if we consider the addition cost, our system will out perform. Because during addition we are only adding new blocks not updating the whole index. In contrast, complete local inverted index needs to be sent to server after any addition. So building encrypted inverted index always saves bandwidth. However, the amount of savings depends on the dataset and query load.

6 Security

In this section, we formally prove the security of our proposed system. The cloud service is managed by semi-honest Bob, who follows the defined protocol but may try to infer private information about the document he hosts. Over the years, many security definitions have been proposed for searchable encryption for semi-honest model. Among those simulation based adaptive semantic security definition by Curtmola, at el., [12] is widely used in literature. Later it is customized to work under random oracle model by Kamara, at el., in [17]. We adapt this definition to prove our security model.

In our proposed static model, we are leaking encrypted document size, block length, number of total blocks, trapdoor of blocks related to a search query information. In dynamic model, in addition to these information, we are leaking length of newly added encrypted documents, and associated search signatures. Also note that the cache $\mathcal{C}$ can be considered as a document that is updated with a new length in every addition operation. This *does not leak any additional information* because in the cache we are storing (1) document id lists length information, which is some constant times number of search signatures and (2) few internal information about documents, which is some constant times number of documents. All of these atomic information are already leaked due to index.

We will first define necessary patterns, history, trace, and view for our schema then prove this schema satisfies adaptive semantic security.

Search Signature Pattern (μ_p): Suppose $\{o_1, o_2, ..., o_\eta\}$ is a set of η consecutive operations on the encryption collection such that o_i is a search or addition request. Each operation o_i has a set of associated search signatures denoted as o_i^s. Specifically, if o_i is a search instance, it involves a single search signature $o_i^s = \{s_{i_1}\}$, If o_i is an addition, it involves a set of search signatures that are included in the whole dataset of new documents such that $o_i^s = \{s_{i_1}, ..., s_{i_\varsigma}\}$. Then μ_p is a function such that $\mu_p((i,\rho),(j,\ell)) = 1$ if $s_{i_\rho} = s_{j_\ell}$ and $\mu_p((i,\rho),(j,\ell)) = 0$ otherwise, for $1 \leq i, j \leq \eta$, $1 \leq \rho \leq |o_i^s|$, and $1 \leq \ell \leq |o_j^s|$

Search pattern ($\mathcal{N}_p$): Suppose o_i is a search request, $cnt(s_{i_1})$ be the number of times s_{i_1} occurs in the dataset. Then, $\mathcal{N}_p(o_i) = (cnt(s_{i_1}))$. Note that we are assuming that adversary can infer this count. However, we are not disclosing this information directly.

Addition Pattern ($\mathcal{A}_p$): Suppose o_i is an addition request for a document collection $\{D_\iota, ..., D_\rho\}$, $|C_x|$ denotes the bit-length for the encrypted form of D_x, $\{s_{j_1}, ..., s_{j_\varsigma}\}$ is set of search signatures that are included in a new corpus, and $cnt(s_{j_\iota})$ denotes the number of documents associated with s_{j_ι} in modified dataset. Then $\mathcal{A}_p(o_j) = (\{|C_\iota|, ..., |C_\rho|\}, \{cnt(s_{j_1}), ..., cnt(s_{j_\varsigma})\})$

History ($\mathcal{H}_\eta$): Let $\mathcal{D}$ be the document collection and $OP = \{o_1, ..., o_\eta\}$ be the consecutive search or addition requests that are issued by user. Then $\mathcal{H}_\eta = (\mathcal{D}, OP)$ is defined as η query history.

Trace (λ): Let $C = \{C_1, ..., C_n\}$ be the collection of encrypted data items, $|C_i|$ be the size of C_i, $\mu_p(\mathcal{H}_\eta)$, $\mathcal{N}_p(\mathcal{H}_\eta)$, $\mathcal{A}_p(\mathcal{H}_\eta)$, b be the search signature, search, addition pattern for $\mathcal{H}_\eta$, length of each block in encrypted inverted index respectively. Then $\lambda(\mathcal{H}_\eta) = \{(|C_1|, ...,|C_n|), \mu_p(\mathcal{H}_\eta), \mathcal{A}_p(\mathcal{H}_\eta), \mathcal{N}_p(\mathcal{H}_\eta), b\}$ is

defined as the trace of $\mathcal{H}_\eta$. Trace can be considered as the maximum amount of information that a data owner allows its leakage to an adversary.

View (v): Let $C = \{C_1, ..., C_n\}$ be the collection of encrypted data items, $\mathcal{E}$ be the encrypted inverted index, and $\Pi = \{\Pi_{o_1}, ..., \Pi_{o_\eta}\}$ be the trapdoors and encrypted values for η consecutive requests in $\mathcal{H}_\eta$. Then, $v(\mathcal{H}_\eta) = \{C, \mathcal{E}, \Pi\}$ is defined as the view of $\mathcal{H}_\eta$. View is the information that is accessible to an adversary.

Adaptive Semantic Security for Dynamic SSE: SSE schema satisfies adaptive semantic security in random oracle model, if there exists a probabilistic polynomial time simulator $\mathcal{S}$ that can adaptively simulate the adversary's view of the history from the trace with probability negligibly close to 1 through interaction with random oracle. Intuitively, this definition implies that all the information that is accessible to the adversary can be constructed from the trace. Formally, let $\mathcal{H}_\eta$ be a random history from all possible history, $v(\mathcal{H}_\eta)$ be the view, $\lambda(\mathcal{H}_\eta)$ be the trace of $\mathcal{H}_\eta$. Then, scheme satisfies the security definition in random oracle model if one can define a simulator $\mathcal{S}$ such that for all the polynomial size distinguishers $Dist$, for all polynomial $ploy$ and a large Λ:

$$Pr[Dist(v(\mathcal{H}_\eta)) = 1] - Pr[Dist(\mathcal{S}(\lambda(\mathcal{H}_\eta))) = 1] < \frac{1}{poly(\Lambda)}$$

where probabilities are taken over $\mathcal{H}_\eta$ and the internal coins of key generation and encryption.

Theorem 1. *Proposed scheme satisfies the adaptive semantic security.*

Proof. We will show the existence of polynomial size simulator $\mathcal{S}$ such that the simulated view $v_S(\mathcal{H}_\eta)$ and the real view $v_R(\mathcal{H}_\eta)$ of history $\mathcal{H}_\eta$ are computationally indistinguishable. Let $v_R(\mathcal{H}_\eta) = \{C, \mathcal{E}, \Pi\}$ be the real view. Then $\mathcal{S}$ adaptively generates the simulated view $v_S = \{C^*, \mathcal{E}^*, \Pi^*\}$

$\mathcal{S}$ chooses n random values $\{C_1^*, ..., C_n^*\}$ such that $|C_1^*| = |C_1|, ..., |C_n^*| = |C_n|$. In this setting, C_i is output of a secure encryption scheme. By the pseudo-randomness of the applied encryption, C_i is computationally indistinguishable from C_i^*.

Given the documents per search signature (e.g., $cnt(s)$) and block length b, $\mathcal{S}$ computes number of entries in $\mathcal{E}$ and generates that many (k_i^*, v_i^*). Note that for every (k_i, v_i) in real encrypted inverted index $\mathcal{E}$ there exists a (k_i^*, v_i^*) in simulated encrypted inverted index $\mathcal{E}^*$. Here length of k_i and k_i^* is equal to the output length of pseudo random function H. Similarly, length of v_i and v_i^* is equal to b. Here, encrypted keys and blocks are computationally indistinguishable from random values by pseudo-randomness of the applied encryption.

$\mathcal{S}$ simulates requests $\Pi_{o_1}, ..., \Pi_{o_\eta}$ according to their types

1) Π_{o_i} is a search request: We define $\mathcal{X}_s = \{\pi_{s^1}, ..., \pi_{s^{\lceil \frac{cnt(s)}{b} \rceil}}\}$ be the trapdoors generated for search signature s. Suppose, $\Pi_{o_i} = (\mathcal{X}_{s_{i_1}}, cnt(s_{i_1}))$ is a search request. Then $\mathcal{S}$ copies $cnt(s_{i_1})$ from $\mathcal{N}_p(o_i)$ to $cnt(s_{i_1})^*$. Then if $\mu_p((i,1), (j,\ell)) = 1$ for any $1 \leq j < i$ and $1 \leq \ell \leq |o_j|$ then $\mathcal{X}^*_{s_{i_1}} = \mathcal{X}^*_{s_{j_\ell}}$. Otherwise $\mathcal{X}^*_{s_{i_1}}$ is set to $\lceil \frac{cnt(s_{i_1})}{b} \rceil$ number of random row-key from simulated encrypted inverted

index $\mathcal{E}^*$ such that those was not previously selected during the simulation. In this setting, components of simulated and real requests are computationally indistinguishable by the pseudo-randomness of the applied encryption. Hence Π_{o_i} and $\Pi^*_{o_i}$ are computationally indistinguishable.

2) Π_{o_i} is an addition request: Suppose, $\Pi_{o_i} = ((\mathcal{X}_{s_{i_1}}, cnt(s_{i_1}))$, ..., $(\mathcal{X}_{s_{i_\varsigma}}, cnt(s_{i_\varsigma})))$ is an addition pattern, $|\Pi_{o_i}|$ be the number of pairs.

For each of the pair individually (iterated with ρ, where $1 \leq \rho \leq |o^s_i|$) simulator $\mathcal{S}$ does the following. First copy $cnt(s_{i_\rho})$ from $\mathcal{A}_p(o_i)$ to $cnt(s_{i_\rho})^*$. Next, if $\mu_p((i,\rho),(j,\ell)) = 0$, for all $1 \leq j < i$ and $1 \leq \ell \leq |o^s_j|$ meaning new search signature so copy $\lceil \frac{cnt(s_{i_\rho})}{b} \rceil$ new random row keys and values from $\mathcal{E}^*$ to $\mathcal{X}^*_{s_{i_\rho}}$ such that those was not used earlier. However, things get little complicated when there is at least one $\mu_p((i,\rho),(j,\ell)) = 1$ meaning this search signature has been seen earlier. Note that during addition phase client only needs to update last block and add more blocks if necessary. For this simulator $\mathcal{S}$ needs to search in revers find the largest j that is smaller than i where $\mu_p((i,\rho),(j,\ell)) = 1$ That is the place where s_{i_ρ} was last used. Now find $cnt(s_{j_\ell})$ either from $\mathcal{A}_p(o_j)$ or $\mathcal{N}_p(o_j)$ depending on the j^{th} operation. Now $cnt(s_{i_\rho}) - cnt(s_{j_\ell})$ is the number of new documents that have search signature s_{i_ρ} and $\mathcal{S}$ assigns $\lfloor \frac{cnt(s_{i_\rho}) - cnt(s_{j_\ell})}{b} \rfloor$ new k^* from $\mathcal{E}*$ that are not already used and corresponding random v^*. Also $\mathcal{S}$ has to add one more row key-value pair (k^*, v^*) to for the last block that's being updated. $\mathcal{S}$ picks last element of $\mathcal{X}^*_{s_{j_\ell}}$ from so far generated Π^* and randomly generate a new value $v*$ for that element too. In this setting, the constructed $\Pi^*_{o_i}$ is computationally indistinguishable from Π_{o_i}.

Since each component of $v_R(\mathcal{H}_\eta)$ and $v_S(\mathcal{H}_\eta)$ are computationally indistinguishable, we can conclude that the proposed schema satisfies the security definition.

7 Application of ETLQP framework

As an application of our ETLQP framework we built an image storage system that saves encrypted images in cloud storage and built an encrypted index to search later on. Before going into further detail of our ETLQP framework implementation we briefly describe Fuzzy Color and Texture Histogram (FCTH) [8], Eigenface [36], Locality Sensitive Hashing(LSH) [15], and range query to exact query conversion mechanism [13]. FCTH and Eigenface are used for image similarity search and face recognition respectively and LSH is used for dimension reduction. Finally, as the name suggests range query to exact query conversion mechanism is used to convert a range query in a defined ranged to sequence of matching query. These concepts are vital to the development of our system.

Fuzzy Color Texture Histogram (FCTH) [8] is an histogram of image that combines texture and color information. It is widely used in content based image retrieval systems (CBIR) [22,9,40,10,11]. In FCTH the texture information is represented by an eight-bin histogram derived via the fuzzy system that uses the high-frequency bands of the Haar Wavelet transform. The color is represented by a 24-bin color histogram computed like in the CEDD descriptor. Overall, the

final histogram include 192 regions. Each of the 1600 image blocks is processed and assigned to a region as in the CEDD. The final 192-bin histogram is also normalized and quantized such that each bin value is an integer between 0 to 7 inclusive. FCTH of an image can be considered as a vector with 192 dimensions and distance between FCTH vector of images can be used to determine similarity among images.

Eigenface [36] is a very well studied, effective yet simple technique for face recognition using static 2D face image. It consists of three major operations - finding eigenvectors of faces, finding weights of each faces, and recognition tasks.

Finding Eigenvectors. We start with M face centered upright frontal images that are represented as $N \times N$ square matrices. Let, $\{\Gamma_1, \ ... \ , \ \Gamma_M\}$ are $N^2 \times 1$ vector representation of these square matrices, $\Psi = \frac{1}{M}\sum_{i=1}^{M} \Gamma_i$ is the average of these vectors, and $\Phi_i = \Gamma_i - \Psi$ is computed by subtracting average Ψ from ith image vector.

Now eigenvectors u_i of co-variance matrix $C = AA^T$, where $A = [\Phi_1 \, \Phi_2 \ldots \Phi_M]$, can be used to approximate the faces. However, there are N^2 eigenvectors for C. In practice N^2 can be a very large number, thus computing eigenvectors of C can be very difficult. So instead of AA^T matrix we compute eigenvectors of A^TA and take top K vectors for approximating eigenvectors u_i, where $\left\|u_i\right\| = 1$. The selection of these eigenvectors is done *heuristically*.

Finding Weights. Φ_i can be represented as a linear combination of these eigenvectors $\Phi_i = \sum_{j=1}^{K} w_j u_j$ and weights can be calculated as $w_j = u_j^T \Phi_i$. Each normalized image is represented in this basis as a vector $\Omega_i = \begin{bmatrix} w_1 & w_2 & \ldots & w_k \end{bmatrix}^T$ for $i = 1, 2, \ldots M$. This is essentially projecting face images into new eigenspace (the collection of eigenvectors).

Recognition Task. Given a probe image Γ, we first normalize $\Phi = \Gamma - \Psi$ then project into eigenspace such that $\Omega = \begin{bmatrix} w_1 & w_2 & \ldots & w_K \end{bmatrix}^T$, where $w_i = u_i^T \Phi$. Now we need to find out nearest faces in this eigenspace by $e_r = min \left\|\Omega - \Omega_i\right\|$. If $e_r <$ a threshold chosen heuristically, then we can say that the probe image is recognized as the image with which it gives the lowest score.

In summary, face images are considered as a point in a high dimensional space. An eigenspace consisting few significant eigen vectors are computed for approximating faces in a training face dataset. Next, test face images are projected into the computed eigenspace. Distances of test face images and all training faces images are computed. If any distance is bellow a pre-determined threshold then those faces are considered a match for associated test face. A detail formal explanation of eigen-face schema is presented in full version [31].

Locality sensitive hashing is a technique widely used to reduce dimensions. Core concept of LSH is to define a family of hash functions such that similar items belong to same bucket with high probability. More specifically we utilized LSH in euclidean space and adopted widely accepted projection over random line technique described in [2]. Let, r be a random projection vectors, v be an input vector, o be a random number used as offset, and w be bucket length parameter fixed by user. The bucket id is computed by $Round(\frac{v.r+o}{w})$ function. Finally, several such projection vectors are used to generate several bucket ids

for a single input vector. In this setting, nearby items will share at least a same bucket with very high probability. In practice value of w and number of random projections are controlled to achieve required success rate.

Range query to exact query conversion. We adopt the range query mechanism described in [13]. Let, a be a discrete feature that has value ranging from 0 to 2^{t-1}, meaning it requires t bits to represent in binary. We first create binary tree of t depth representing the complete range. Each leaf node (at depth t) represent an element in the range and we level all left edge as 0 and right edge as 1. So, the path from the root to a leaf node essentially represent the binary encoding of that leaf. In transform phase, we convert an input value of the range to t feature-value tuples, where the feature is concatenation of field name, depth i and value is binary encoding of inner node at depth i. During the query phase given a range we first find the cover as described in [13], create the corresponding search signatures and perform the query.

7.1 ETLQP for image storage

To build an application using ETLQP framework described system section, programmer has to define proper extract and transformation functions. Load, Query and Post-Process phases remain the same. For our image storage software we consider four features *location* - where the picture was take, *time* - when the picture was taken, *texture and color* - for searching similar pictures, and *faces* - for face recognition. In our implemented system queries of first two features are equality search and later two are similarity search. Similarity searches are difficult to perform since result not only contains exact matches but also contains results that are similar. So, we need to have a similarity measure for the feature in question. To accomplish such a similarity queries we utilize LSH, which essentially helps us to convert the query to sequence of equality search. In addition, result of LSH can contain false positives. We need extra post processing to remove those.

Extract. Location and time data are extracted from Exif[8] meta-data. Exif is a very popular standard for attaching image meta-data into image used by all popular camera manufacturers. Camera with Global Positioning System (GPS) module can store longitude and latitude of a picture taken into Exif data, which can be extracted easily using available libraries[9]. We use FCTH for similarity analysis and used a open source implementation of FCTH analyzer [22]. Finally, for face recognition using Eigenface, we extract frontal faces from images using haar cascade [37,19] frontal face pattern classifier.

Transform. Now we define appropriate transformations for extracted features. Main idea behind the definition of transformation functions is to make the query easier later on. So definition of transformation functions is mainly guided by the query demand.

[8] `http://www.cipa.jp/std/documents/e/DC-008-2012_E.pdf`
[9] `https://drewnoakes.com/code/exif`

- **Location.** Location information in terms of longitude and latitude is difficult to use in practice. We use OpenStreetMap's reverse geolocation service[10] to determine address of latitude and longitude associated with the image. To make query easier later, we generate search signatures of six sub-features of the address - full address, city, county, country, state, and zip.
- **Time.** Similarly we break created date of an image into five sub-features - complete date, year, month, day of month, and day of week. We generate search signatures based on these sub-features. In addition, to support range query based on date we convert the time into unix time stamp that essentially represents seconds passed from 1 *January* 1970 without considering the leap second. Then we divide the time stamp by number of seconds in a day (86400), that gives us the number of days passed from epoch. Finally, we build the range query binary tree with depth 20, which essentially is capable of covering dates till year 4840. Then we create the feature value list as described earlier.
- **Texture and Color.** In the extract phase we extracted FCTH of provided image, which is a 192 dimensional vector. We can treat each dimension as different sub features but that will make it difficult to perform similarity search later on. Instead we define an euclidean LSH schema that put near elements into same bucket and use the bucket ids to generate search signatures.
- **Face.** We built an eigenface schema with extracted face images. Again to preserve similarity we built an euclidean LSH schema with weight vectors of faces and store the eigenspace related information into synchronized cache $\mathcal{C}$. In particular we store the average face, selected top eigenfaces, and weights of all faces. Storing such information is the major reason of defining the cache $\mathcal{C}$.

Query and Post-Process. With previously defined extract and transform functions client can perform *time queries*, such as find images that are taken on specific year, month, day of week, day of moth, or in a rang of dates, etc. Client can also perform *location queries*, such as find pictures taken in a country, state, city, etc. In both of these cases, we transform a query into encrypted search signatures and retrieve associated encrypted document ids from the cloud storage server. Finally we decrypt and display the result directly to the user. On the other hand, for face recognition and image-similarity query, we extract appropriate feature values from a query image and transform these values into LSH bucket ids of previously defined LSH schema. We generate encrypted search signature, retrieve encrypted document ids, and decrypt the result like date and time queries. However, before showing results to user we remove false positive results introduced by the LSH schema.

8 Experimental Evaluation

Setup. In our proposed design we have two components client and server. Client processes images, performs cryptographic operations, and produces encrypted inverted index that is stored in server. In query phase client retrieves partial index from the server based on user-query.

[10] `http://wiki.openstreetmap.org/wiki/Nominatim`

ETQLP client is written in Java using several other libraries for image feature extraction. Cryptographic operations are performed using Java Cryptographic Extension (JCE) implementation. During our experimentation, we execute the client program in a computer with *Intel(R) Core(TM) i7-4770 3.40GHz* CPU, *16GB* RAM running *Ubuntu 14.04.4 LTS*. Our implemented client can store encrypted inverted index into different types of servers.

- File storage server in local network. We developed a very simple web based storage service that has two end points file read and file write. Our server is written in Python (v2.7.6) using Flask (v0.10.1) microframework and files are stored in a MongoDB (v3.2.0). We deployed our local storage server in a machine with *Intel(R) Xeon(R) CPU E5420 2.50GHz* CPU, *30GB* of RAM running *CentOS 6.4*. In addition, our client computer is also in the same network. Here, file path is search signature of encrypted inverted index $\mathcal{E}$ and file content is encrypted document id list.

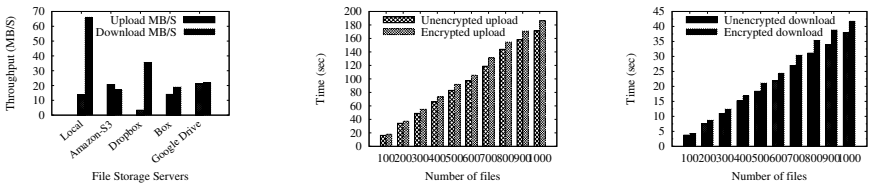
- Amazon S3[11]is very popular commercial object and file storage system, which provides easy to use representational state transfer (REST) application program interface (API) for storing, retrieving and managing arbitrary binary data or file. Amazon also provides very extensive software development kit (SDK) for building applications to utilize it's services. In our implementation, search signatures of encrypted inverted index $\mathcal{E}$ are keys of S3 objects and content of the objects are associated encrypted document id list.

- Personal file storage services. Among the popular commercial personal file storage services, we implemented capabilities to store inverted index into Dropbox[1], Box[2], and Google Drive[3] because of available open source SKD on these platforms. Here, each entry in encrypted inverted index $\mathcal{E}$ is saved as separate file, where file name is encrypted search signature and file content is encrypted document id list. *Due to rate limitations* [12] [13] [14] *we could not perform extensive analysis on these platforms.* However, a typical user adding images time to time will not have any trouble using any of these platforms as cloud file storage server. We reached the rate limit due to repeated nature of our experiment.

In 2(a) we illustrate the throughput of each of the servers. We compute the system throughput by upload and downloading 100MB in the storage servers. We observe that local server performs extremely well in case of download because of MongoDB's advanced cache management, which keeps the recently used data in RAM to improved performance. In addition, in our smaller scale experiments we observed that performance of personal storage server scales according to this throughput ratio.

Dataset. Thomee at el. presented the Yahoo Flickr Creative Commons 100 Million Dataset (YFCC100M) in [35], which contains basic information of 100 million media objects, of which approximately 99.2 million are photos and 0.8 million are videos, all of which carry permissive creative commons license[15]. We have randomly selected 20109 images and downloaded the original version of

[11] `https://aws.amazon.com/s3/`
[12] `https://www.dropbox.com/developers/core/bestpractices`
[13] `https://developers.box.com/docs/#rate-limiting`
[14] `https://developers.google.com/drive/v3/web/handle-errors`



(a) Server Throughput (b) Overhead Upload (c) Overhead Download

Fig. 2. Server throughput, overhead of encryption decryption upon upload and download.

these images. Size of this random dataset is 42.3GB, average file size is 2.15MB, number of faces detected 7027, and 4102 images have latitude and longitude embedded in EXIF data.

Face Detection Accuracy. Our constructed dataset is randomly selected and unlabeled. As a consequence the correctness of our face detection system remains unmeasured. So we perform face detection on two know face datasets *Caltech Faces* [38] and *Color FERET* [27,26]. *Caltech Faces* dataset contains 450 frontal face images each containing picture of an individual. Our system detected 431 of those correctly, yielding a 95.78% accuracy. We also observed that most of the failed images are too dark to detect any face. *Color FERET* dataset contains a total of 11338 facial images, which were collected by photographing 994 subjects at various angles. Since our face detection system detects frontal faces only, we extract frontal face images with `fa` and `fb` suffixes. We found that there are total 2722 such images. Our face detection system successfully detected 2459 images, yielding 90.33% accuracy. Here, most of the failed cases has glass or similar face obstructing additions.

Experiments. Before performing any experiment for the proposed ETLQP framework, we first compare performance of an encrypted image storage system with an unencrypted one to observe overhead of encryption. We randomly select few images, encrypt and upload those images. Then we download, decrypt and save those files again and measure the performance. We perform this experiment with local storage server and in the client we used a thread pool with 2 threads to parallelize the operations. Encrypted files are slightly larger than the unencrypted version because we padded the input file and added a 256-bit message authentication code. So overall size over head is very negligible. We observe on average 10.09% increment in execution time for encrypted upload compare to unencrypted upload, illustrated in 2(b). Similarly we observe on average 13.06% increment in execution time for downloading encrypted file and decrypt, compare to unencrypted download, illustrated in 2(c). So we conclude that encryption *does not* add significant overhead for an efficiently implemented client.

Now, we measure performance of different phases of our framework for varying number of randomly selected images from above dataset. We measure performance of different phases of our framework for varying number of randomly

[15] `http://creativecommons.org/licenses/`

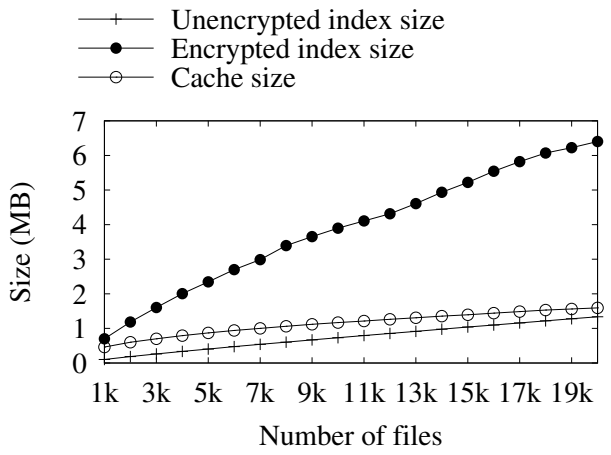
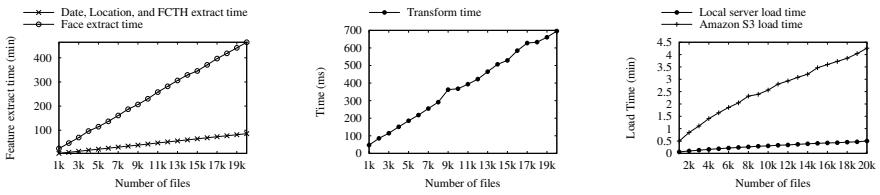


Fig. 3. Size of unencrypted index, encrypted index and cache vs. number of files

selected images from above dataset. Horizontal axis of most of the reported graphs is number of randomly selected images used to build the index and vertical axis is the observation. We repeat each experiment for at least 3 times and report the average observation value.

We extracted four features of the images *created date*, *location*, *FCTH vector*, and *faces*. For *created date* feature we generated search signatures of *day of week*, *day of month*, *month*, *year*, *week of week year*, and a combination of *year, month, and date*. Also we generated range query related signature to perform arbitrary range query on date. For *location* feature, we first reverse looked up the address of latitude, longitude extracted form images using open street map[10] Next we created search signatures based on *city*, *state*, *county*, *country*, *zip code* and *full address*. *FCTH vector* is extracted from all the images with Lucene image retrieval [22] implementation. We detected *frontal faces* using OpenCV implementation of haar cascade classifier, converted all the face images to median face size, built eigenface classifier on the detected faces, and store the computed weight vector of all the faces as image feature. Figure 3 illustrates size growth of unencrypted inverted index, encrypted inverted index, and synchronized cache. The growth is linear, which implies index size increment is proportional to the number of files added. Moreover, in our experiment we observed that for 20000 images encrypted inverted index size is only 7.05MB, which is about four average size images in our dataset. So size over head of our proposed system is very low.

We also observe that feature extraction is the most time consuming phase of our system. Figure 4(a) illustrates required time for extracting features. We observe that face detection and extraction time is the dominating factor in this phase. It requires 464.54 minutes to detect and extract faces from 20000 images in sequential manner, averaging about 1.39 seconds per image. In addition, other three features takes 85.87 minutes for 20000 images, averaging 0.26 seconds per images. Even though it looks like a long time for a lot of images but time required for individual image is very little. Furthermore, these experiments are done in



(a) Extract time (b) Transform time (c) Load time

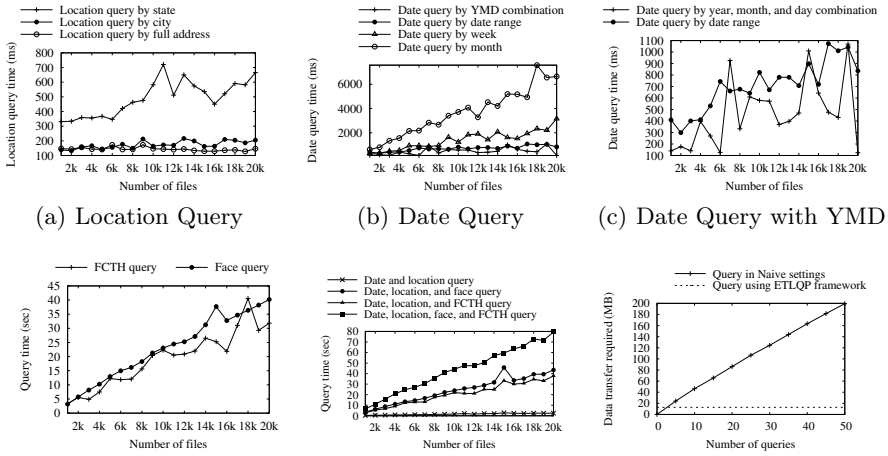
Fig. 4. Time required in different phases of building and uploading the index for different number of images.

sequential manner. A multi-threaded implementation will certainly reduce the over all time. In addition, in this prototype we implemented a separate program to call native OpenCV API to detect faces and communicate the results back to the main process, which added extra overhead. In contrast, transform phase is one of the fastest phase in our implementation. Here, extracted feature values are transformed into inverted index of search signatures and document ids. We observed that the growth is almost linear and for 20000 images it only requires 696 milliseconds, shown in Figure 4(b).

Next phase in our framework is load, where we encrypt and load the inverted index into a cloud storage server. In our experiments, we load the encrypted index into (1) Local server and (2) Amazon S3. Figure 4(c) shows the time required for encrypting and loading inverted index into local and Amazon S3 server. For 20000 images it requires 20.52 seconds to encrypt and load the entire inverted index into local storage server and 5.65 minutes to complete in Amazon S3 server. Furthermore, the time growth is linear due to the linear growth of index size.

After loading the data into cloud storage server we perform queries on extracted features. For location feature we perform query with five randomly selected states, cities, and full addresses. Figure 5(a) shows the performance of location queries on different number of randomly selected images from the dataset. It is interesting to observer that query by full address performs fastest among all three query categories. Query by state takes longest to finish and query by city performs in between. This is because time require to finish a query is proportional to the number to blocks fetched and processed. Very few images are like to have same full address however more images likely to have common state or city. As a consequence we observe the above performance. Similarly for date feature we randomly select five year, month, date(YMD) combinations, date range, months, and weeks. Query by year, month, date combination and range query by date takes smallest amount time. In contrast, query by month takes longest and query by week in between. Figure 5(b) and 5(c) illustrates performance of different types of date query vs number of files.

For FCTH feature, we randomly select five images among input images, get FCTH vectors, then perform same euclidean LSH transformation defined in transform phase to get the search signatures. For face feature we randomly select five faces and compute weights with eigen vector information stored in cache, then perform euclidean LSH transformation and get search signatures.



(a) Location Query (b) Date Query (c) Date Query with YMD

(d) FCTH and Face Query (e) Combination Queries (f) Naive Query Size

Fig. 5. Time required for different type of queries vs number of files.

Using these search signatures we get matching images. Finally we remove images that are too far from the query image. Determining the accuracy of our proposed system for this two features is difficult since the dataset is unlabeled. However, we can estimate performance with experimentation as shown in Figure 5(d). FCTH and face query both takes significantly longer than location and date query, this is due to the nature of these features, extra LSH transformations, and result post processing. In our experiment we setup an euclidean LSH schema with 4 random projections. For FCTH feature, each random projection line is divided into 20 unit length buckets and during query time we search we query for images that has distance less than 8 unites. For 20000 images we observed 78.4% precision and 16.6% recall. LSH parameters can be adjusted according to the needs of application. Our experiments gives an general idea of performance overhead of different types of complex queries.

We also perform four conjunctive combination of queries. We perform different types of queries individually then intersects the result to get the final result. First combination is date and location query combination, where we combine location queries with date queries. Second combination is date, location, and FCTH query, where we combine three types of queries together. Next combination is date, location, and face query, which is also three type queries. Fourth combination is date, location, FCTH and face query, which combines all the features our system can extract. As shown in Figure 5(e) fourth combination takes longer to preform and first combination takes the smallest amount of time. This is because location and date queries are individually very fast compared to other two types of queries.

Finally, we compared performance of our framework with a naive implementation. In the naive implementation the extract and transform phases remains the same. However, the load and query phase is different. In naive implementation we encrypt and upload the inverted index as a single file. During query

phase we download and decrypt the whole encrypted index to preform queries. Figure 5(f) illustrates data transfer required to perform year-month-date(YMD) query using our proposed framework and naive implementation. As the number of query increases data transfer requirement increases liner to the index size. On the other hand, in our framework initial index loading phase requires loading the encrypted inverted index then on subsequent query the data transfer is very little.

9 Conclusion

In this study, we addressed the problem of searchable encryption with simple server that can support complex queries with multimedia data type. We made several contributions including an extensible general framework with security proof and its implementation. Our defined extract, transform, load, query and post-process (ETLQP) framework can build efficient searchable encryption scheme for complex data types (e.g, images). With this framework we can perform very sophisticated queries, such as face recognition, without needing cryptographic computational support from the server. Our implementation shows small overhead for building encrypted search index and performing such complex queries. In addition, we also show that overhead of general cryptographic operations is negligible compared to other necessary operations of a cloud based file storage system.

Acknowledgments. The research reported herein was supported in part by NIH awards 1R0-1LM009989 & 1R01HG006844, NSF CNS-1111529, CNS-1228198, & CICI-1547324.

References

1. Agarwal, A.: Web vulnerability affecting shared links. `https://blogs.dropbox.com/dropbox/2014/05/web-vulnerability-affecting-shared-links/`
2. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM 51(1), 117–122 (Jan 2008), `http://doi.acm.org/10.1145/1327452.1327494`
3. Arnold, B.C.: Pareto distribution. Wiley Online Library (1985)
4. Bindschaedler, V., Naveed, M., Pan, X., Wang, X., Huang, Y.: Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 837–849. ACM (2015)
5. Bösch, C., Hartel, P., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. ACM Computing Surveys (CSUR) 47(2), 18 (2015)
6. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: Network and Distributed System Security Symposium, NDSS. vol. 14 (2014)
7. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. Cryptology ePrint Archive, Report 2013/169 (2013), `http://eprint.iacr.org/`
8. Chatzichristofis, S., Boutalis, Y.: Fcth: Fuzzy color and texture histogram - a low level feature for accurate image retrieval. In: Image Analysis for Multimedia Interactive Services, 2008. WIAMIS '08. Ninth International Workshop on. pp. 191–196 (May 2008)

9. Chatzichristofis, S., Boutalis, Y., Lux, M.: Img(rummager): An interactive content based image retrieval system. In: Similarity Search and Applications, 2009. SISAP '09. Second International Workshop on. pp. 151–153 (Aug 2009)
10. Chatzichristofis, S.A., Zagoris, K., Boutalis, Y.S., Papamarkos, N.: Accurate image retrieval based on compact composite descriptors and relevance feedback information. International Journal of Pattern Recognition and Artificial Intelligence 24(02), 207–244 (2010)
11. Chatzichristofis, S., Boutalis, Y.: Content based radiology image retrieval using a fuzzy rule based scalable composite descriptor. Multimedia Tools and Applications 46(2-3), 493–519 (2010), http://dx.doi.org/10.1007/s11042-009-0349-x
12. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 79–88. ACM (2006)
13. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II, chap. Rich Queries on Encrypted Data: Beyond Exact Matches, pp. 123–145. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-24177-7_7
14. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM 43(3), 431–473 (May 1996), http://doi.acm.org/10.1145/233551.233553
15. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. pp. 604–613. STOC '98, ACM, New York, NY, USA (1998), http://doi.acm.org/10.1145/276698.276876
16. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security, pp. 258–274. Springer (2013)
17. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 965–976. CCS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2382196.2382298
18. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on. pp. 1156–1167. IEEE (2012)
19. Lienhart, R., Maydt, J.: An extended set of haar-like features for rapid object detection. In: Image Processing. 2002. Proceedings. 2002 International Conference on. vol. 1, pp. I–900. IEEE (2002)
20. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Jonker, W., PetkoviÄĞ, M. (eds.) Secure Data Management, Lecture Notes in Computer Science, vol. 6358, pp. 87–100. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15546-8_7
21. Lu, W., Swaminathan, A., Varna, A.L., Wu, M.: Enabling search over encrypted multimedia databases. In: IS&T/SPIE Electronic Imaging. pp. 725418–725418. International Society for Optics and Photonics (2009)
22. Lux, M., Chatzichristofis, S.A.: Lire: Lucene image retrieval: An extensible java cbir library. In: Proceedings of the 16th ACM International Conference on Multimedia.

pp. 1085–1088. MM '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1459359.1459577
23. Naveed, M.: The fallacy of composition of oblivious ram and searchable encryption. Tech. rep., Cryptology ePrint Archive, Report 2015/668 (2015)
24. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 639–654. IEEE (2014)
25. Ostrovsky, R.: Efficient computation on oblivious rams. In: Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing. pp. 514–523. STOC '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/100216.100289
26. Phillips, P.J., Moon, H., Rizvi, S., Rauss, P.J., et al.: The feret evaluation methodology for face-recognition algorithms. Pattern Analysis and Machine Intelligence, IEEE Transactions on 22(10), 1090–1104 (2000)
27. Phillips, P.J., Wechsler, H., Huang, J., Rauss, P.J.: The feret database and evaluation procedure for face-recognition algorithms. Image and vision computing 16(5), 295–306 (1998)
28. Pinkas, B., Reinman, T.: Oblivious ram revisited. In: Advances in Cryptology–CRYPTO 2010, pp. 502–519. Springer (2010)
29. Qin, Z., Yan, J., Ren, K., Chen, C.W., Wang, C.: Towards efficient privacy-preserving image feature extraction in cloud computing. In: Proceedings of the ACM International Conference on Multimedia. pp. 497–506. ACM (2014)
30. Raval, N., Pillutla, M.R., Bansal, P., Srinathan, K., Jawahar, C.: Efficient content similarity search on encrypted data using hierarchical index structures
31. Shaon, F., Kantarcioglu, M.: A Practical Framework for Executing Complex Queries over Encrypted Multimedia Data. https://www.utdallas.edu/~fahad.shaon/complex-query-framework-full.pdf
32. Stadmeyer, K.: Google drive update to protect to shared links. https://security.googleblog.com/2014/06/google-drive-update-to-protect-to.html (2014)
33. Stefanov, E., Shi, E.: Oblivistore: High performance oblivious cloud storage. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 253–267. IEEE (2013)
34. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310. ACM (2013)
35. Thomee, B., Shamma, D.A., Friedland, G., Elizalde, B., Ni, K., Poland, D., Borth, D., Li, L.J.: The new data and new challenges in multimedia research. arXiv preprint arXiv:1503.01817 (2015)
36. Turk, M., Pentland, A.: Eigenfaces for recognition. Cognitive Neuroscience, Journal of 3(1), 71–86 (Jan 1991)
37. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. vol. 1, pp. I–511. IEEE (2001)
38. Weber, M.: Frontal face dataset. http://www.vision.caltech.edu/html-files/archive.html
39. Xia, Z., Zhu, Y., Sun, X., Wang, J.: A similarity search scheme over encrypted cloud images based on secure transformation. International Journal of Future Generation Communication and Networking 6(6), 71–80 (2013)
40. Yang, Z., Kamata, S., Ahrary, A.: Nir: Content based image retrieval on cloud computing. In: Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on. vol. 3, pp. 556–559 (Nov 2009)