

BIG DATA PRIVACY PROTECTION USING CRYPTOGRAPHIC CLOUD STORAGE

A PROJECT REPORT

Submitted by

**PATHANGE S MADHUKIRAN [Reg No: 1031310185]
NIKHIL GRANDHI [Reg No: 1031310203]**

Under the guidance of

R. Vidhya

(M.E, Assistant Professor, Department of Computer Science & Engineering)

*in partial fulfillment for the award of the degree
of*

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



S.R.M. Nagar, Kattankulathur, Kancheepuram District

MAY 2017

SRM UNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled “**BIG DATA PRIVACY PROTECTION USING CRYPTOGRAPHIC CLOUD STORAGE**” is the bonafide work of “**PATHANGE S MADHUKIRAN [Reg No: 1031310185], NIKHIL GRANDHI [Reg No: 1031310203]**” who has carried out the project work under my guidance. Further certified, that to the best of my knowledge of work reported here in does not form any other project report or dissertation on the basics of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

R. Vidhya
GUIDE
M.E, Assistant Professor
Dept. of Computer Science & Engineering

Signature of the Internal Examiner

SIGNATURE

Dr. B.Amutha
HEAD OF THE DEPARTMENT
Dept. of Computer Science & Engineering

Signature of the External Examiner

ABSTRACT

The world today generates a huge amount of data every day. It includes click stream data, browsing history, social media activity, opinion polls etc. This massive amount of data, Big Data, is collected regularly by companies and governments. Big data now exists in almost every application relating to human behaviour and interactions and as such it becomes too vital to ensure its security and privacy. In the past, large amounts of data were hosted on proprietary infrastructure and private networks. Today, it is possible for any organization of any size to cheaply store and access Big Data via public cloud infrastructure.

ACKNOWLEDGEMENTS

Our sincere thanks to the following teachers who helped in the preparation of project. We would like to thank our director **Dr.C.MUTHAMIZHCELVAN**, Faculty of Engineering and Technology,SRM University, Chennai.

We express our deep sense of gratitude to **Dr.B.AMUTHA**, professor,Head of department of computer science and engineering, SRM University,Chennai, for giving us the opportunity to take up this project

We would sincerely thank our project Co-ordinator **Dr.E.Poovammal**, professor, department of computer science and engineering, SRM University, Chennai, for the valuable inputs for this project

I would like to express my deepest gratitude to my guide, **R.Vidhya** for her valuable suggestions and guidance, personal caring, encouragement, timely help and providing us with the best atmosphere for doing research. Even through the work,inspite of her heavy schedule, she has maintained cheerful and support to us for completing this research work.

Several other people also contributed their maximum to the project.We sincerely acknowledge all their efforts which was indispensable for the completion of project

Pathange S MadhuKiran

Nikhil Grandhi

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	vi
LIST OF FIGURES	vii
ABBREVIATIONS	viii
1 INTRODUCTION	1
2 LITERATURE SURVEY	3
3 SYMMETRIC SEARCHABLE ENCRYPTION	5
3.1 Key Generation	8
3.2 Keyword Index creation	8
3.3 Encryption	9
3.4 Search	9
3.5 Decryption	10
3.6 Adding Documents	10
4 Coding, Testing, Outputs	11
4.1 Key Generation	11
4.2 Encryption	16
4.3 Decryption	27
4.4 Search	31
5 Conclusion	38
6 Future Enhancement	39

LIST OF TABLES

3.1	Keyword Index 1	8
3.2	Keyword Index 2	9
3.3	Inverted Index	10

LIST OF FIGURES

3.1	Single Party Architecture	6
3.2	Multiple Party Architecture	7
4.1	Key Generation	15
4.2	Files to be uploaded to the cloud	24
4.3	Encyption files	25
4.4	One of the Encrypted files	26
4.5	Decrypted Files	30
4.6	Encrypted files retrived from cloud by searching	37

ABBREVIATIONS

SE	Searchable encryption
SSE	Symmetric searchable encryption
CSP	Cloud Service Provider

CHAPTER 1

INTRODUCTION

From the development in Storage technologies and cloud computing, users are increasingly storing and accessing their data on cheap public cloud servers. The benefits of public cloud storage is that users need not foot the cost of building and getting a private cloud structure and can instead pay a Cloud Service Provider (CSP) an amount corresponding to its storage needs. The availability and reliability of the data stored at a public cloud server come at the cost of significant security and confidentiality-concerns. These concerned obstacles are the most glaring to the huge usage of cloud storage as stated in Kamara and Lauter (2010).

The solution to the above problem would be to develop a virtual private storage service which has the security and assurance in a cloud, the availability and the reliability of a public cloud. This can be achieved via the latest achievements in cryptographic techniques namely, Searchable encryption (SE) Kamara and Lauter (2010). Through searchable encryption one can encrypt at client side, sending it to the server and then searching on the data that is encrypted without revealing anything about the data in plaintext. Searchable encryption is better than traditional encryption as it does not require downloading of all the data and decrypting it in order to use it. It only retrieves the specific data which one requires through a keyword search on the encrypted data. The keyword in question would also have to be encrypted in order to prevent leakage of information to the server.

To enable the user by keyword usage of keyword functionality gets into the race. Now considering a cloud application which has of a CSP, with the users who store information in the cloud.

The purpose of this paper is to demonstrate a feasible solution to the problem of Big

Data privacy on the cloud and to discuss a possible implementation of symmetric searchable encryption. In the following sections the paper will go into the architecture of the cryptographic cloud followed by the working of the symmetric searchable encryption.

CHAPTER 2

LITERATURE SURVEY

In Kamara and Lauter (2010) the authors present the notion of a cryptographic cloud storage architecture. This paper illustrates the use of using the cloud as a medium of storage in a manner both secure and reliable. It shows the various ways in which cryptography can be used to provide safety and assurance to the end user. The authors also expand upon their premise and also present techniques to prove data possession.

Goh. (2009) demonstrates the use of a secure index. It is here that the introduction of indexes to secure data first emerged. The concept of utilizing keywords to secure data materialized in this paper. It formally defines a secure index and formulates a security model for indexes known as semantic security against chosen keyword attack. Its implementation uses z-inx using pseudo-random functions and Bloom filters.

D. Song and Perrig. (2000) introduces the idea of searchable encryption. The algorithms it presents are $O(n)$, where n is the average size of each document. While this paper demonstrates a new concept the technique is too slow compared to ordinary insecure storage. It has however provided a new concept to the field of storage and security.

The paper R. Curtmola and Ostrovsky. (2008) provides improved security definitions and constructions. While papers prior only considered single party enquires, this paper illustrates multi-party enquires. The two schemes it presents are adaptable and non-adaptable searchable encryption schemes. These schemes have proved to be easier to implement and run faster than any prior work on searchable encryption.

Md Iftexhar Salam and Yap. (2012) provide and analysis of all the methods and schemes to protect cloud data. They demonstrate how to implement an adaptive searchable scheme in a mobile setting and also provide several future enhancements. Their implementation makes use of an entire document for keywords and they contrast this with taking only a few select keywords.

The paper Chiang et al. (2010) discusses the authenticity of data stored on the cloud. Through the techniques discussed users can effectively determine whether their data has been tampered or modified. As it does this via sampling the documents the overhead is minute compared to the benefits it offers. It is therefore an effective addition to one's cloud data.

Betti (2008) demonstrates a way of encryption without the use of encryption. This technique protects against both history based attacks and also future queries to an effective degree. Through this paper users can either choose more than one privacy technique which provides more diverse security guarantees and make it difficult to exploit the user's data.

CHAPTER 3

SYMMETRIC SEARCHABLE ENCRYPTION

Symmetric searchable encryption (SSE) is an encryption scheme which enables one to search over encrypted data using tokens over an encrypted search index. This helps one in creating a cryptographic cloud as it is only through the ability to search that one is freed from having to download and decrypt the entire cloud data in order to obtain any particular file.

Considering a search index for a file collection which maps keywords to the files which contain them. In a searchable encryption scheme, the search index is encrypted such that one can make use of it only with the help of tokens or trapdoors. A token generated for a keyword retrieves from the search index all the encrypted files which contain that keyword. Moreover, one can create a token only with the secret key which was used to create the search index. This ensures that only users with the knowledge of the secret key can access the search index and retrieve files.

Despite all the safety guarantees the searchable encryption scheme provides information may be leaked to the cloud provider. This can happen if the cloud provider analyses the search and retrieval pattern of the user. The cloud provider could then make a guess at what keywords are being used and what all documents have those keywords in common. But this leakage of information is because of the retrieval of files and not because the failure of the encryption scheme and is inevitable. One way to prevent this would be to introduce false positives and then sort them out at the users machine. However this leads to more complexity. SSE schemes were first introduced in D. Song and Perrig. (2000). They were subsequently improved and given new security definitions in R. Curtmola and Ostrovsky. (2008), Goh. (2009), Chiang et al. (2010)

Architecture

The architecture would consist of three parties: Party A that stores its data on the cloud, Party B which needs to access the cloud data and the cloud service provider. Now consider the scenario where Party A stores and retrieves its data from the cloud. To do this it first creates a key which stores on A local system. A will add appropriate metadata to upload data to the cloud, encrypt the data and metadata with the searchable encryption scheme and finally upload the encrypted data to the cloud. Whenever A wants to download data it first generates a token. This token is created with the help of a search query and the cryptographic key stored in A. This token will be then transfered to the cloud provider where it is used to search for the appropriate encrypted files and send them to the user. The usage of cryptographic key to decrypt the files. Thus, data is stored and retrieved without revealing anything about the data to the cloud. We can observe this scenario in 3.1

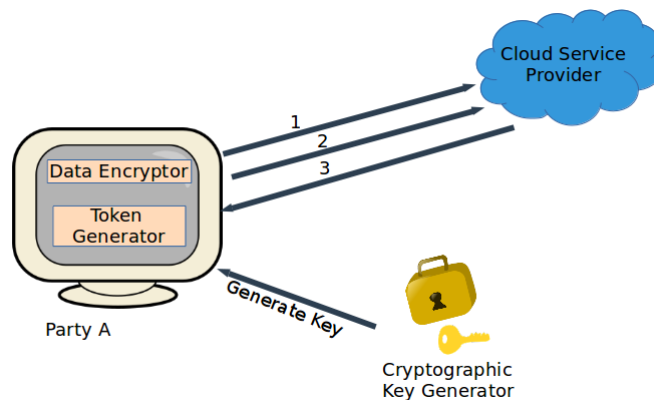


Figure 3.1: Single Party Architecture

In Figure 3.1 the steps undergoes are as (1) Process the data, encrypt it and send it to the cloud, (2) Token is generated based on search query and sent to the cloud service provider, (3) Cloud searches for the appropriate encrypted files and sends them back to Party A.

Now consider the case where Party B wants to access the data stored by A on the cloud. When Party B requests data, Party A generates a token and a credential for B. They are then delivered to Party B. Party B transfers the token to the cloud, which utilizes it to retrieve the encrypted files and sends them to B. These files are then decrypted by B using its credential. Thus, data is confidentially shared between the two parties through the cloud. We can observe this scenario in 3.2.

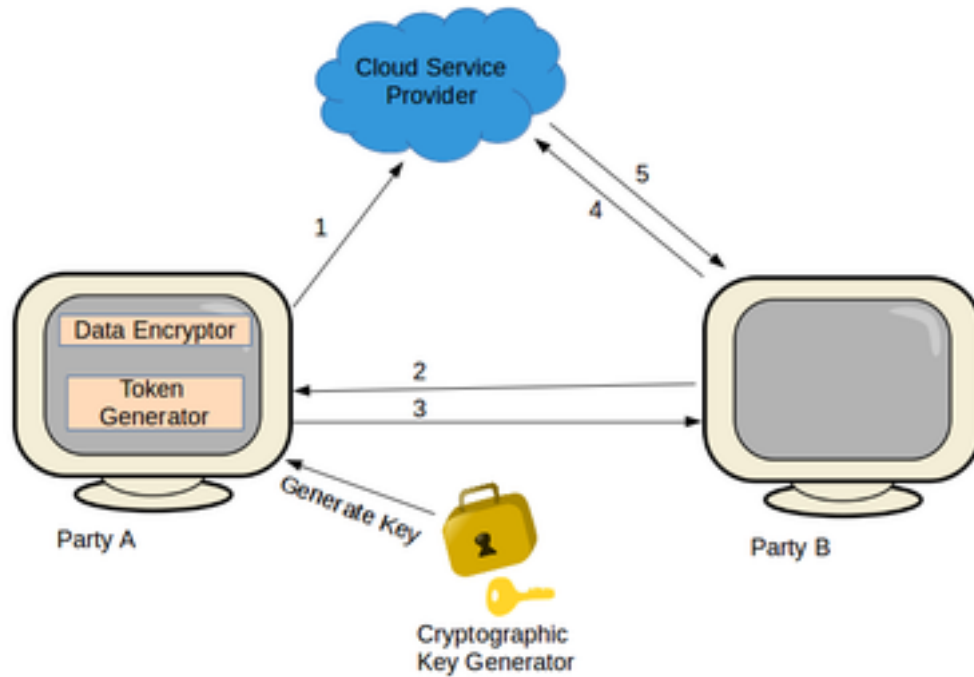


Figure 3.2: Multiple Party Architecture

(1) Process the data, encrypt it and send it to the cloud, (2) Party B requests Party A to find a keyword (3) A token and credential are generated and sent to Party B (4) Party B sent the token to the cloud (5) Cloud searches for the appropriate encrypted files and sends them back to Party B

The Symmetric Searchable Encryption Scheme consists of the following steps:-

1. Key Generation
2. Key word Index Creation
3. Encryption

4. Search
5. Decryption
6. Adding Documents

3.1 Key Generation

Two symmetric keys K1 and K2 are generated and stored at the local machine of the user. They are used in encrypting the search index contents and the documents respectively. As they are symmetric keys.

Table 3.1: Keyword Index 1

Document	Content ID
D1	1
D2	2
D3	3
D4	4
D5	5
D6	6

3.2 Keyword Index creation

As per Md Iftekhar Salam and Yap. (2012), Betti (2008) In this step the Keyword to Document index is created. Let D be the set of all documents to be uploaded to the cloud. First, assign each document in D a unique content id. After that decide upon the keywords to be used and build an index which maps the documents content ids to the corresponding keywords they contain as in Table 3.2. Using this index that is created an inverted index maps the keywords to the corresponding content ids of the files. The table inverted index is shown in Table 3.3.

Before the encryption of files, a specified action is needed and start the encryption schema. User will process the file set and fetches out the word to create an index file.

Table 3.2: Keyword Index 2

Content ID	Keyword
1	w1,w3,w5
2	w2,w3,w7
3	w1,w4,w9
4	w6,w8,w9
5	w8,w10
6	w10

For example: Let there is a set of five files named D, where $D=D1,D2,D3,D4,D5$ to be uploaded in cloud. Prior to this step, User process these set of files and creates an index

3.3 Encryption

The keyword index and the documents are encrypted which are using the generated keys in the Key generation. The key $K1$ is used to construct the encrypted keyword index. The keywords are encrypted as $ENCK1(wi||n)$, where $ENCK1$ represents encryption with key $K1$, $||$ represents concatenation, wi is the i th keyword in the keyword index created in Keyword Index Creation step, and n is the n th time of occurrence of the keyword wi in D . These encrypted keywords are then mapped to their corresponding content id of the document the keyword is located in. This is shown in Table 3.3. The key $K2$ is used to encrypt the documents in D . The encrypted documents are renamed to their corresponding content id. The encrypted documents and index are then sent to the cloud.

3.4 Search

In this step a token is generated to get the files from the cloud. It takes a keyword as a input and result in the token as a block of $ENCK1(w||i)$, and where $ENCK1$ defines encryption with the key- $K1$, w is the keyword, $||$ is concatenation and $0 \leq i \leq n$, and n represents the total number of documents.

This is used when, a user wanted to search a file containing a particular keyword which provides the user with the function that generates a search token which is to be

Table 3.3: Inverted Index

Keyword	Content ID
W1	1,4
W2	2
W3	1,2
W4	3
W5	1
W6	4
W7	2
W8	4,5
W9	3,4
W10	5,6

used at the server. To carry out this search operation, user needs to input the keyword to be searched and then compute for the search token.

3.5 Decryption

The encrypted files that the user receives are decrypt using the key K_2 . This is represented as $DECK_2(D_i)$ which represents decryption of document D_i with key K_2 .

The module of decryption is situated at the user device which provides the decryption.

3.6 Adding Documents

With regards to adding more documents to D , instead of adding them directly to the existing index they are added as a separate group say document group G . On G the same operations as those on D are applied and its index is added to the cloud. Search is then possible on both D and G on the cloud.

CHAPTER 4

CODING, TESTING, OUTPUTS

4.1 Key Generation

```
import javax.crypto.*;

import java.util.Base64;

import java.io.Serializable;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

class KeyGen implements Serializable{

    private static final long serialVersionUID = 1L;

    private SecretKey key1 = null;

    private SecretKey key2 = null;
```

```

        public SecretKey Key1() {

return key1;

        }

        public SecretKey Key2() {

return key2;

        }

        public void generate() throws Exception {

KeyGenerator generator = KeyGenerator.getInstance("AES");

generator.init(128);

key1 = generator.generateKey();

key2 = generator.generateKey();

        }

        public void store(String address) throws Exception {

FileOutputStream fos = new FileOutputStream(address);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(this);

```

```

oos.close();

    }

    public static KeyGen read(String address) throws
        Exception {

        FileInputStream fis = new FileInputStream(address);

        ObjectInputStream ois = new ObjectInputStream(fis);

        return (KeyGen) ois.readObject();

    }

    public static void main(String[] args) throws
        Exception{

        KeyGen gen = new KeyGen();

        gen.generate();

        gen.store("/home/nikhil/Desktop/Project localmachine/
            ser/keys.ser");

        KeyGen result = KeyGen.read("/home/nikhil/Desktop/
            Project localmachine/ser/keys.ser");

        String encodedKey1 =
        Base64.getEncoder().encodeToString(result.key1.getEncoded());

        String encodedKey2 =

```

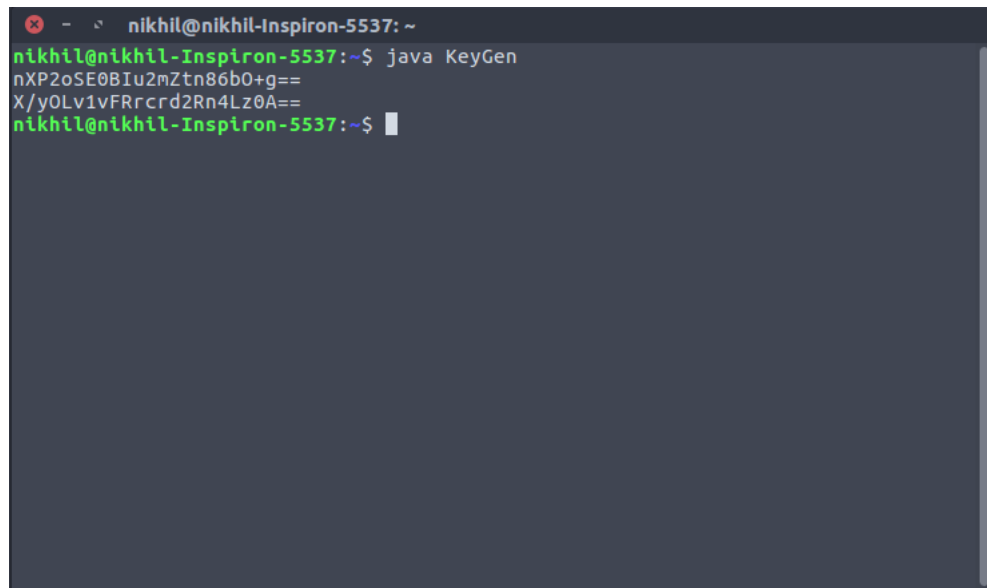
```
        Base64.getEncoder().encodeToString(result.key2.getEncoded());

    System.out.println(encodedKey1);

    System.out.println(encodedKey2);

    }

}
```



```
nikhil@nikhil-Inspiron-5537: ~  
nikhil@nikhil-Inspiron-5537:~$ java KeyGen  
nXP2oSE0BIu2mZtn86b0+g==  
X/y0Lv1vFRrcrd2Rn4Lz0A==  
nikhil@nikhil-Inspiron-5537:~$
```

Figure 4.1: Key Generation

In this Key generation process javax.crypto library is used to generate the AES Keys. There are two 128 bit AES keys- Keyword encryption keys i.e., K1 and document encryption keys i.e., K2. These keys are generated and stored. This key generation has three methods, They are :

- 1) Generate() : Helps in generating two 128 bit AES keys
- 2) Store() : Helps in converting encoded generated keys into strings writing key objects and storing it into a location of user device.
- 3) Read() : This functionality provides reading keys from the user. Here it takes the input as key object to read the key values following this it converts key string to an array construction secret key from the array

4.2 Encryption

```
import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

import java.util.Scanner;

import java.util.HashMap;

import java.util.LinkedList;

import javax.crypto.*;

public class Encryption {

    /* The encryption class takes the users data as input

    it then creates a map of the keywords to the files

    Following that it creates the encrypted index while

    simultaneously encrypting the data. The data is then

    uploaded to the cloud.*/

    static HashMap<Integer, String> idfilemap =

        new HashMap<Integer, String>();
```



```

static HashMap<String, Integer> fileidmap =
    new HashMap<String, Integer>();

public static void init(final File folder) throws
    Exception {

    /* init() is used to create the filename to contentid
    and the contentid to filename maps*/

    int i = 1;

    for (final File fileEntry : folder.listFiles()) {

        idfilemap.put(i, fileEntry.getName());

        fileidmap.put(fileEntry.getName(), i);

        i++;

    }

}

//1. encrypt files and store on server

//2. Extract keywords from each file and store
    map(keyword, LinkedList of files containing the keyword)

//3. Create emap(encrypted keyword, content id of file)

```

```

    public static void encrypt(final File folder)
        throws Exception {

        // read keys from the localmachine's memory

        KeyGen kg = KeyGen.read("/home/nikhil/Desktop/Project
            localmachine/ser/keys.ser");

        HashMap<String, LinkedList<String>> map =
            new HashMap<String, LinkedList<String>>();

        /* Access each file in the directory using the
        for each loop */

        for (final File fileEntry : folder.listFiles()) {

            if (fileEntry.isDirectory()) continue;

            // encrypting files and storing on the server

            FileInputStream fis = new FileInputStream(fileEntry);

            byte[] plainText = new byte[(int)fileEntry.length()];

            fis.read(plainText);

            Cipher aesCipher = Cipher.getInstance("AES");

            aesCipher.init(Cipher.ENCRYPT_MODE, kg.Key2());

            byte[] byteCipherText = aesCipher.doFinal(plainText);

```

```

File file = new File("/home/nikhil/Desktop/Project server/"
    + fileidmap.get(fileEntry.getName()));

FileOutputStream fos = new FileOutputStream(file);

fos.write(byteCipherText);

fis.close();

fos.close();

/*extracting keywords from each file and storing in
map(keyword,LinkedList of files containing the keyword)*/

Scanner scn = new Scanner(fileEntry);

for(int i = 0; i < 10 && scn.hasNext(); i++) {

String s = scn.next();

if(map.containsKey(s)) {

    map.get(s).add(fileEntry.getName());

}

else {

LinkedList<String> l = new LinkedList<String>();
l.add(fileEntry.getName());

```

```

        map.put(s,l);

    }

    }

}

/* create map(encrypted keyword, content id of file in
which keyword is present)*/

HashMap<String, Integer> emap =
    new HashMap<String, Integer>();

String buff = null;

for(HashMap.Entry m : map.entrySet()) {

    /* map.entrySet gives us an iterator which lets us access
    each key, value pair in the map. The for each loop iterates
    through each entry. Using m.getKey() we get the key and similarl
    using m.getValue() we get the corresponding value.*/

    int j = 1;

    LinkedList<String> ls = (LinkedList<String>) m.getValue();

    // encrypt the keywords

```

```

        for (String s: ls) {

            // using the list iterator

String w = (String)m.getKey() + j;

j++;

Cipher aesCipher = Cipher.getInstance("AES");

aesCipher.init(Cipher.ENCRYPT_MODE, kg.Key1());

byte[] plainText = w.getBytes();

byte[] byteCipherText = aesCipher.doFinal(plainText);

emap.put(new String(byteCipherText), fileidmap.get(s));

/*if( w.equals("I1")) {

    System.out.println(new String(byteCipherText));

    buff = new String(byteCipherText);*/

}

}

}

// write the encrypted index to memory

```

```

FileOutputStream f =
    new FileOutputStream("/home/nikhil/Desktop/Project
        server/ser/index.ser");

ObjectOutputStream o = new ObjectOutputStream(f);

o.writeObject(emap);

o.close();

System.out.println(emap.get(buff));

    }

    public static void writeDocIDFileNameMapping()
        throws Exception {

        // write the content id to filename mapping to memory

FileOutputStream fos = new
    FileOutputStream("/home/nikhil/Desktop/Project
        localmachine/ser/idfilemap.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(idfilemap);

oos.close();

    }

```

```
        public static void main(String[] args)
            throws Exception{

Encryption.init(
    new File("/home/nikhil/Desktop/Project files"));

Encryption.encrypt(
    new File("/home/nikhil/Desktop/Project files"));

Encryption.writeDocIDFileNameMapping();

    }
}
```

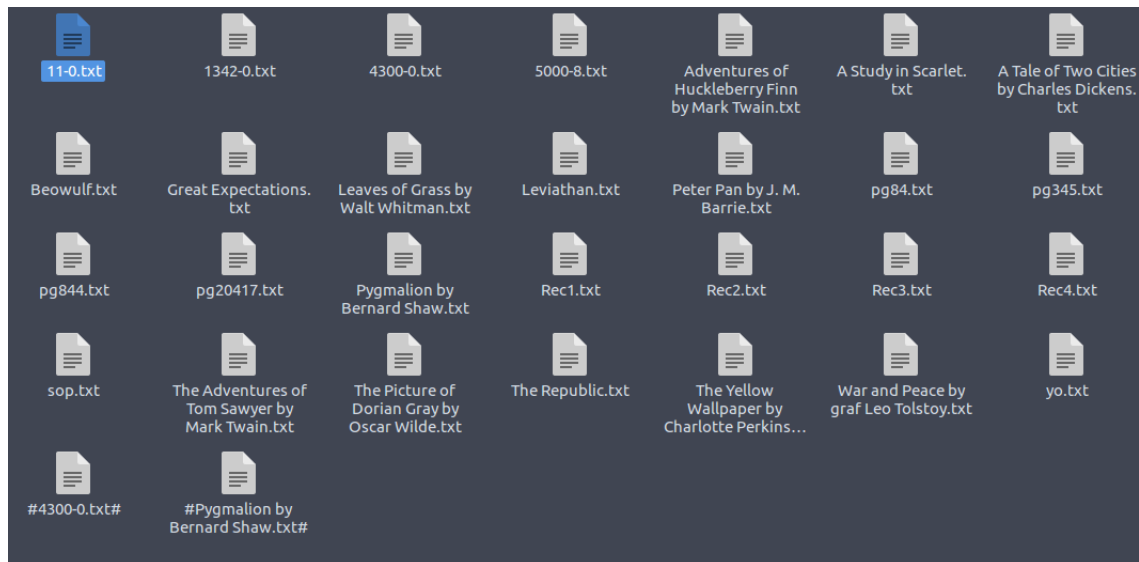


Figure 4.2: Files to be uploaded to the cloud

Encryption follows the pre-processing the module of encryption which results in encrypted documents. AES is used for index encryption which consists of two methods. encrypt() method writeDocIDFileNameMapping()

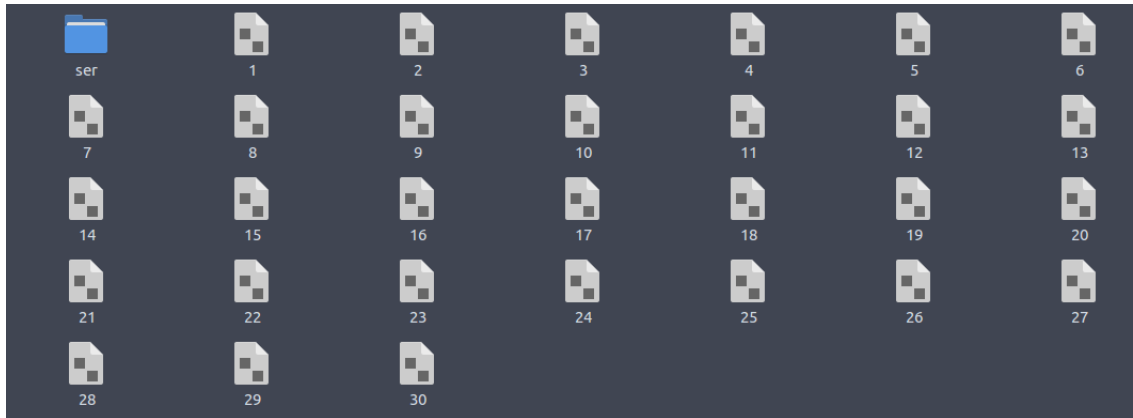


Figure 4.3: Encyption files

After completion of pre-processing the encryption module results in the encrypted documents. AES was used for the index and document encrytion. The class of encryption consists of two methods: encrypt() witeDocIDFileNameMapping().

4.3 Decryption

```
import java.io.*;

import javax.crypto.*;

import java.util.*;

public class Decryption {

    /* This class is used to decrypt the files received

    from the server. It uses the output of keygen to do so.

    After decryption it renames the files to their original

    names using the contentid-filename map*/

    public static void decrypt() throws Exception{

        KeyGen kg = KeyGen.read("/home/nikhil/Desktop/

        Project localmachine/ser/keys.ser");

        File folder = new File("/home/nikhil/Desktop/

        Project localmachine/encrypted files");

        FileInputStream fis = new

        FileInputStream("/home/nikhil/Desktop/

        Project localmachine/ser/idfilemap.ser");

        ObjectInputStream ois = new ObjectInputStream(fis);
```

```

        // creating the HashMap for idfile map

HashMap<Integer, String> idfilemap =
    new HashMap<Integer, String>();

idfilemap = (HashMap<Integer, String>) ois.readObject();

// Decrypting the retrieved files in the localmachine

for (final File fileEntry : folder.listFiles()) {

    if (fileEntry.isDirectory()) continue;

    Cipher aesCipher2 = Cipher.getInstance("AES");

    aesCipher2.init(Cipher.DECRYPT_MODE, kg.Key2());

    FileInputStream fin = new FileInputStream(fileEntry);

    // renaming the files

    FileOutputStream fout = new
        FileOutputStream("/home/nikhil/Desktop/
        Project localmachine/" +
        idfilemap.get(Integer.parseInt(
        fileEntry.getName())));

    byte[] byteCipherText = new byte[(int)fileEntry.length()];

    fin.read(byteCipherText);

    byte[] bytePlainText = aesCipher2.doFinal(byteCipherText);

```

```
fout.write(bytePlainText);  
}  
  
}  
  
public static void main(String[] args) throws Exception{  
  
    Decryption.decrypt();  
  
}  
}
```

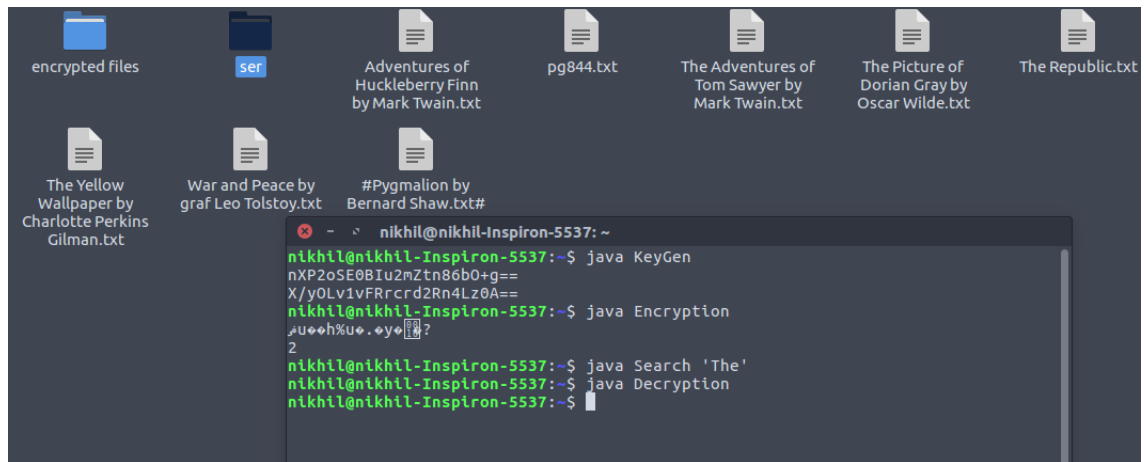


Figure 4.5: Decrypted Files

Once the encrypted files arrive at the local machine from the server the decrypt() method is initiated.

This method uses the symmetric key generated during the key generation phase for the decryption of files.

It reads in each encrypted file, decrypts it and then writes the plaintext to memory.

4.4 Search

```
import javax.crypto.*;

import java.io.*;

import java.util.*;

public class Search {
    /* This class is used to search and retrieve the
    encrypted files from the cloud

    this is done via the functions searchToken() and search()

    they create a token and retrieve the encrypted files respectively*/

    public static void searchToken(String keyword) throws Exception
    {
        //Read in the keys from the local machine

        KeyGen kg = KeyGen.read
        ("/Projectlocalmachine/ser/keys.ser");

        //Creating input and output file streams

        //then create a linked list of strings

        //To this list add the encrypted keyword concatenated with the 1

        //where n is the number of documents in D
```

```

FileInputStream fis = new FileInputStream("/home/nikhil/Desktop/Proj

ObjectInputStream ois =
new ObjectInputStream(fis);

HashMap<Integer, String> m =
(HashMap<Integer, String>) ois.readObject();

LinkedList<String> l =
new LinkedList<String>();

ois.close();

fis = new FileInputStream
("/home/nikhil/Desktop/Project server/ser/index.ser");

ois = new ObjectInputStream(fis);

HashMap<String, Integer> emap = (HashMap<String, Integer>)

ois.readObject();

ois.close();

// Creating linked list of encrypted keywords to send to the server

for(int i = 1; i <= m.size(); i++) {

    String s = keyword + i;

    Cipher aesCipher = Cipher.getInstance("AES");

```



```

        aesCipher.init(Cipher.ENCRYPT_MODE, kg.Key1());

        byte[] byteCipherText = aesCipher.doFinal(s.getBytes());

        l.add(new String(byteCipherText));

    }

    FileOutputStream fos = new FileOutputStream
    ("/home/nikhil/Desktop/Project server/ser/searchToken.ser");

    /* send the linked list to the server

    the server uses it to find the documents containing the keywords */

    ObjectOutputStream oos = new ObjectOutputStream(fos);

    oos.writeObject(l);

    /* System.out.println("emap:\n");

    for(HashMap.Entry mm : emap.entrySet()) {

        Cipher aesCipher2 = Cipher.getInstance("AES");

        aesCipher2.init(Cipher.DECRYPT_MODE, kg.Key1());

        byte[] bytePlainText = aesCipher2.doFinal((byte[])mm.getKey());

        if (new String(bytePlainText).equals("I1"))

            System.out.println(new String((byte[])mm.getKey()));
    }

```

```

        //      System.out.println(new String(bytePlainText));

    }*/

    // for( byte[] b : l) {

    }

    /* the search function gets the token from the user and

    uses it to retrieve the files from the server */

    public static void search() throws Exception {

FileInputStream fis = new FileInputStream("/home/nikhil/Desktop/Proj

ObjectInputStream ois = new ObjectInputStream(fis);

LinkedList<String> tokens = (LinkedList<String>) ois.readObject();

// System.out.println(tokens);

ois.close();

fis = new FileInputStream("/home/nikhil/Desktop/Project server/ser/i

ois = new ObjectInputStream(fis);

HashMap<String,Integer> emap = (HashMap<String,Integer>)

ois.readObject();

```

```

ois.close();

//send files which are found using searchToken to the localmachine

for( String b : tokens) {

    if (!emap.containsKey(b)) break;

    int i = emap.get(b);

    //System.out.println(emap.get(b));

    File file = new File
    ("/Project server/" + i);

    fis = new FileInputStream(file);

    File file2 = new File
    ("/encrypted files/" + i);

    FileOutputStream fos =
    new FileOutputStream(file2);

    byte[] buffer = new byte[(int)file.length()];

    fis.read(buffer);

    fos.write(buffer);

}

}

```

```
        public static void main(String[] args)
            throws Exception {

    /* Take argument as keyword and generate the token

then send it to the server to retrieve the encrypted files*/

Search.searchToken(args[0]);

Search.search();

    }

}
```

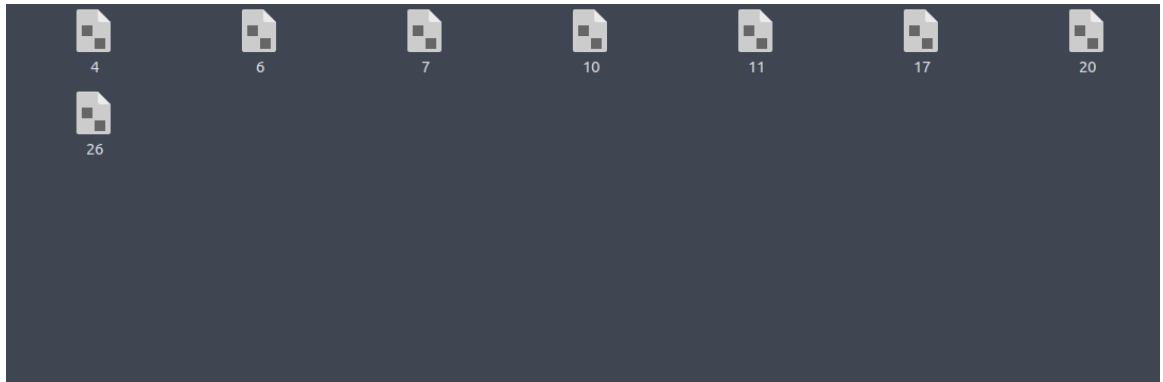


Figure 4.6: Encrypted files retrived from cloud by searching

Search class: Operation starts at the user device which performs at the server. This operation has two methods. They are: 1) Search Token(): Inputs are document ID and keywords to the token at client side. This is used to generate the value of user search and results in the token which is to be fetched.

2) Search(): When search request is sent by the user, the server receives the request. This server takes the token and the database of user as input and fetches the IDs which consists the search token.

CHAPTER 5

CONCLUSION

This paper shows how the SSE scheme can be used to guarantee privacy on a public cloud. The paper discussed the architecture of the cryptographic cloud and also the working of SSE. This can be used to fulfil the security of many a cloud based service and assure the users of the confidentiality of their data. Such a scheme enables users to build a secure storage over the infrastructure of the cloud provider.

Cryptographic cloud storage ensures that the storage provider learns nothing about the data, does not tamper with the data, keeps it available at all times to any authorized device, provides a public cloud services advantage of backing up data, retrieves the data efficiently and enables sharing of the data. As technology keeps growing so does the amount of data used by it. Storage of this data in a private cloud is secure but infeasible, storage in a public cloud is feasible but insecure. Cryptographic cloud provides a happy marriage of the best features of both private and public clouds.

CHAPTER 6

FUTURE ENHANCEMENT

The construction of the SSE discussed in the paper deals only with text documents. It can be enhanced to deal with other formats such as spreadsheets, images, videos, etc. This would require extracting meaningful keywords from these files in order to index them. In this manner the cloud can be enabled to handle multiple formats and thus boost its usability. The cloud service provider despite not being able to read the encrypted data may attempt to tamper it or add files of its own to the user's data. This causes loss or corruption of the user data which the cloud service provider can easily escape blame by accusing the fault to lie with the user. This can be prevented with the help of integrity checks. Integrity checks allow users to detect whether changes have been made to the data. This can be done without having to retrieve the data.

REFERENCES

1. Betti, R. (2008). "Implementation of encryption techniques." *SSE in data security*, A. Morassi and F. Vestroni, eds., Vol. 499 of *CISM Courses and Lectures*, 67–93.
2. Chiang, D.-Y., Lin, C.-S., and Su, F.-H. (2010). "Cloud cryptography and data security." *Journal of Intelligent engineering services*, 42(2), 79–86.
3. D. Song, D. W. and Perrig., A. (2000). "Practical techniques for searching on encrypted data. in ieee symposium on research in security and privacy." *Journal of Engineering*, 121, 45–55.
4. Goh., E.-J. (2009). "Technical report 2003/216, iacr eprint cryptography archive." *Advances in Engineering Software*, 40, 883–891.
5. Kamara, S. and Lauter, K. (2010). "Cryptographic cloud storage,â€ financial cryptography and data security." *Cryptography and Data security*, 8(6), 443–461.
6. Md Iftexhar Salam, Wei-Chuen Yau, J.-J. C. S.-H. H. H.-C. L. R. C.-W. P. G. S. P. S.-Y. T. and Yap., W.-S. (2012). "Implementation of searchable symmetric encryption for privacy-preserving keyword search on cloud storage." *Journal of SSE*, 215(1), 17–34.
7. R. Curtmola, J. Garay, S. K. and Ostrovsky., R. (2008). "Searchable symmetric encryption: Improved definitions and efficient constructions. in a. juels, r. wright, and s. de capitani di vimercati, editors, acm conference on computer and communications security." *Scientia Iranica*, 15(3), 79–89.