Attribute Based Keyword Search Encryption Scheme

Project report submitted to
Visvesvaraya National Institute of Technology, Nagpur
in partial fulfilment of the requirements for the award of
the degree

Bachelor of Technology
In
Computer Science and Engineering

by

Ashwin Dutta (BT16CSE008) Ajay Avaghade (BT16CSE009)
Kshitij Shahade (BT16CSE044) Shubham Rokde (BT16CSE075)

under the guidance of

Dr. Syed Taqi Ali

Department of Computer Science and Engineering Visvesvaraya National Institute of Technology Nagpur 440 010 (India)

2020

Attribute Based Keyword Search Encryption Scheme

Project report submitted to
Visvesvaraya National Institute of Technology, Nagpur
in partial fulfilment of the requirements for the award of
the degree

Bachelor of Technology
In
Computer Science and Engineering

by

Ashwin Dutta (BT16CSE008) Ajay Avaghade (BT16CSE009)
Kshitij Shahade (BT16CSE044) Shubham Rokde (BT16CSE075)

under the guidance of

Dr. Syed Taqi Ali

Department of Computer Science and Engineering Visvesvaraya National Institute of Technology Nagpur 440 010 (India)

2020

Department of Computer Science and Engineering Visvesvaraya National Institute of Technology, Nagpur

Declaration

We, Ashwin Dutta, Ajay Avaghade, Kshitij Shahade and Shubham Rokde, hereby declare that this project work titled "Attribute Based Keyword Search Encryption Scheme" is carried out by us in the Department of Computer Science and Engineering of Visvesvaraya National Institute of Technology, Nagpur. The work is original and has not been submitted earlier whole or in part for the award of any degree/diploma at this or any other Institution / University.

Date:

Sr. No.	Enrollment No	Names	Signature
1.	BT16CSE008	Ashwin Dutta	
2.	BT16CSE009	Ajay Avaghade	
3.	BT16CSE044	Kshitij Shahade	
4.	BT16CSE075	Shubham Rokde	

Certificate

This is to certify that the project titled "Attribute Based Keyword Search Encryption Scheme", submitted by **Ashwin Dutta, Ajay Avaghade, Kshitij Shahade, Shubham Rokde** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering**, VNIT Nagpur. The work is comprehensive, complete and fit for final evaluation.

Dr. Syed Taqi Ali
Asst Prof., CSE, VNIT, Nagpur

Head, Department of CSE
VNIT, Nagpur
Date:

Acknowledgement

We would like to thank our guide **Dr. Syed Taqi Ali** for his invaluable guidance, support and encouragement throughout this course of work, giving us an opportunity to work at our own pace, while giving us directions whenever necessary. We would also like to thank **Dr. U. A. Deshpande**, Head of the Department for giving us full freedom to carry out our work. We extend sincere thanks to **Dr. Syed Taqi Ali** for his valuable insights on the topic as well as on the project thesis.

Last but not the least we would like to acknowledge the contribution of those who have constantly supported and motivated us thus helping us to successfully complete our final year project.

Abstract

Pairing-based cryptography has received a wide recognition and has been part of numerous researches and developments in the last few years. Researchers found these pairing very useful and applicable. Without pairings, there are many problems which cannot be solved.

In this thesis, we go through the understanding of pairings and bilinear maps and how to use them in modern cryptography. We discuss how trapdoor functions are related to pairing functions. Almost all pairing-based cryptographic system are based on elliptic curves. We explain how the elliptic curves work through example. Hyper-elliptic curves are the only known curves to produce practical and desirable pairings. We will be giving an overview on the elliptic curves.

We use Pairing based Cryptography (PBC) library for implementation of various schemes. PBC library can easily evaluate bilinear mapping and pairings. We used C language to implement the library for the simplicity of coding and better understanding. We used the PBC library to implement key agreement algorithms and digital signature schemes.

Initially, we show how PBC Library can be used to implement some basic cryptographic algorithms such as short digital signature (BLS), Identity based signatures and Key agreement protocols with their examples.

Later we focus on using PBC library in attribute-based encryption schemes and its application in encrypting files stored in clouds with distributed access control and how it is helpful in keyword search on files stored on the cloud. In chapter 1 we have given a brief introduction about PBC library and what our project is about. In chapter 2 we have discussed the literature survey associated with this project. Chapter 3 gives in detail how we applied this library to implement some cryptosystems and in chapter 4 we have discussed the results and conclusions regarding this project.

List of Figures

List of Tables

Index

CHAPTER 1: INTRODUCTION

Our major area of focus in this project is: Pairing-based cryptography which is basically a pairing among elements belonging to two cryptographic groups to a third cryptographic group which is denoted by the following mapping:

$$e: G_1 \times G_2 \rightarrow G_T$$

in order to build or analyse the cryptosystems.

1.1 Definition

The following definition is commonly used for PBC [1].

Suppose G_1, G_2 are two additive cyclic groups of prime order q, and G_T another cyclic group of order q written multiplicatively. A pairing is a map: $e: G_1 \times G_2 \rightarrow G_T$, satisfying the following properties:

Bilinearity:

$$\forall Q \in G_2, \forall P \in G_1, \forall x, y \in F_q^*: e(xQ, yP) = e(Q, P)^{xy}$$

Non-degeneracy:

e is not equal to 1

Computability:

In order to compute e, an efficient algorithm does exist.

1.2 Classification

Here, if we consider using the same groups for G_1 and G_2, then it is a symmetric pairing which maps to an element of second group from two elements of the same group.

Initialisations of pairing are categorized into these three types:

G_2 equals G_1

G_2 and G_1 are different but there is a homomorphism $G_2 \rightarrow G_1$ which can be computed.

G_2 and G_1 are different but there is no homomorphism between G_2 and G_1.

1.3 Use in Cryptography

A hard problem belonging to one group can be simplified to an easier problem in another group if both the groups are symmetric.

The approach of generalizing computational Diffie-Helman problem in some groups by using bilinear mapping like Weil or Tate pairing is not expected to be feasible but we can easily use pairing function to solve the simpler decis fie-Helman problem.

In their initial days pairings were only considered as useful in cryptanalysis, however many cryptosystems like the attribute-based encryption systems and the identity-based encryption systems make use of pairings for construction, which would not have been possible with other techniques.

We have discussed a widely used signature scheme called BLS Signature scheme to demonstrate the use of bilinear pairings.

1.4 PBC library

To perform the extensive mathematical operations involved in pairing based cryptographic systems, there is a library called GMP (GNU Multiple Precision Arithmetic) library. Built on this GMP library is the PBC (Pairing Based Cryptography) library. PBC is a free library released by Stanford University [2].

This library was developed keeping in mind the basic and core requirements while implementing a pairing based cryptographic system, and achieving portability and speed. This library is available in many languages. Functionalities such as exponentiation in groups, arithmetic in elliptic curves, generation of curves and evaluation of bilinear pairings are provided in this library.

The time taken to perform pairing operations is reasonable even though when it is written in C. We can usually get 11 milliseconds as fastest pairing time and 31 milliseconds as slowest pairing time on Pentium III 1GHz:

The library and its API are simple to use; however, the programmer needs to have a basic understanding of pairings and elliptic curve cryptography along with good command over coding in C, Python or other supported coding languages.

CHAPTER 2: LITERATURE SURVEY

2.1 Trapdoor Functions

A function which can easily be computed in one direction but very difficult to compute in opposite direction or computing its inverse, without the help of extra information commonly known as the "trapdoor", is called a trapdoor function [3]. Many cryptographic schemes and cryptosystems widely make use of trapdoor functions.

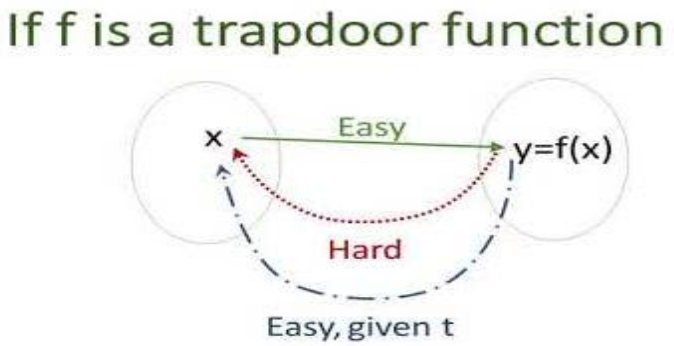


Fig. 2.1 Illustration of A Trapdoor Function

Mathematically speaking, if we have a trapdoor function *f*, then there must exist *t* which is some sort of secret or a less known information, so that when we have values of *t* and *f(x)* then finding *x* is easy. A trapdoor function typically rests at the core of modern encryption schemes as these are the mathematical operations which are simple to perform in forward direction however much harder to execute in reverse.

Example:

Let *N* be multiplication of two very large primes *q* and *p*.

N equals *q* * *p* where *q* and *p* are prime numbers

Given *N*, it is difficult to find the prime 0factorisation without knowing one of the prime numbers.

This is an example of the trapdoor function where *p* or *q* is the trapdoor.

2.2 Elliptic Curve Cryptography

Based on elliptic curves' algebraic structure which are over finite field, we can have encryption through public key cryptography. A very good approach towards this public key cryptography is the elliptic curve cryptography [4]. Elliptic curves require small sized keys whereas other non-elliptic curve cryptography requires large keys, however elliptic curve cryptography delivers equally secured encryption.

We can use elliptic curves for implementing key agreement algorithms, pseudo-random generators and digital signatures. Also, if we can combine symmetric encryption scheme with key agreement algorithms, these curves can be implicitly used for encryption. A lot of integer factorization algorithms can be implemented using elliptic curves having uses in cryptography, such as Lenstra elliptic curve factorization.

Basically, we can explain an elliptical curve as a set of points which follows this equation:

$$y^2 = x^3 + ax + b$$

The shape of the elliptic curve depends on the values assign to *b* and *a*. A secret key is created by elliptical curve cryptography over finite fields which can only be unlocked by the holder of private key. The size of the curve increases with the size of key, and hence a larger key size makes the problem harder to solve.

Following example shows how these curves work [5]:

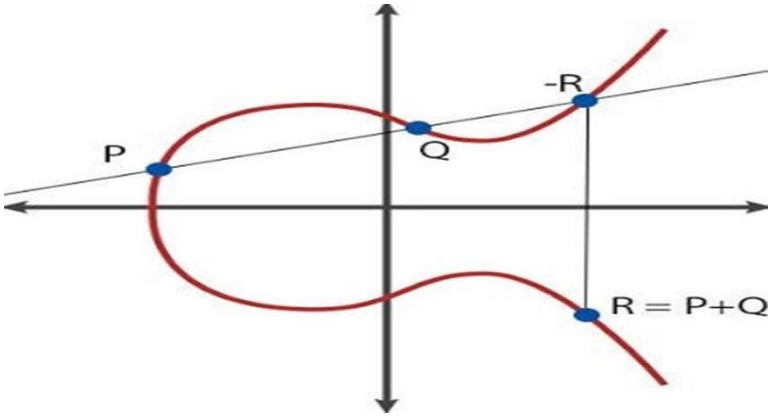


Fig. 2.2 An Elliptic Curve

First, we draw an elliptic curve by assigning specific values to *b* and *a*. Draw a line joining the points *b* to *a* which intersects the curve at some point. From that point find a reflection on the opposite side of x axis. We name this point as *c*.

Now that we have got this point *c*, draw a line joining point *a* to point *c* intersecting the curve, at another point. Just like what we did with the previous intersection point, we find the reflection of intersected point on the opposite side of the x axis and name the new point as point d and repeat this procedure for specified number of times most commonly the private key decides this.

We use the word “dot” to describe an intersection point. For example, consider initial point P then we find other points as:

P dot P = Q

P dot Q = R

P dot R = S

P dot S = T

The actual number of intersection point is not known by those who don't know the private key, and that makes it hard to recompute and evaluate how many times we dotted this equation.

An EC cryptography system can hence be defined by selecting a very large prime number, an equation for elliptic curve and an initial point on the curve. Let *priv* be a number that denotes the private key and the public point which is dotted *priv* number of times denote the public key. So now that we have the public key and if we want to compute the private key from it in this kind of cryptographic system, then it is called elliptic curve discrete logarithm function which is in fact the Trapdoor Function that we were discussing earlier.

The hard problem of elliptic curve discrete logarithm strengthens the ECC. Even after almost thirty years of research and development, mathematicians around the world still are unable to derive an algorithm to solve this problem that is efficient upon the basic approach. Roughly speaking, depending on currently applied mathematics it is difficult to find a shortcut that is minimizing the difference in a Trapdoor Function centred around this situation, unlike with factoring. This suggests that with numbers of almost the same size, factoring is significantly simpler than solving elliptic curve discrete logarithms.

As we know a more rigorous computational hard problem implies a stronger cryptography system, it thus means that elliptic curve cryptosystems are more secure as compared to Diffie-Hellman & RSA.

2.3 Bilinear Mapping

The Diffie-Hellman key exchange algorithm as well as the RSA cryptography system have given birth to a lot of public-key cryptosystems in wide use today. The RSA, although using somewhat same arithmetic operations, doesn't depend on same principles. As an example, The RSA cryptosystem uses non-cyclic groups that and it is a requirement that the group order must not be known to attacker.

In other words, bilinear maps [6] give some additional properties to cyclic groups.

2.3.1 Definition of bilinear map

Suppose G_1, G_2, and G_T be same order cyclic groups. A bilinear map $G_1 \times G_2$ giving G_T is the function $e: G_1 \times G_2 \rightarrow G_T$

$$e(u^p, v^q) = e(u, v)^{pq}.$$

where, $v \in G_2$, $u \in G_1$ and $p, q \in Z$,

These maps are known as pairings as they relate the element pairs from G_1 and G_2 to that with elements in G_T. Even the degenerate maps are admitted by this definition. These map everything to the group G_T.

2.3.2 Definition of admissible bilinear map

Suppose e $G_1 \times G_2 \rightarrow G_T$ as a bilinear map. Assume g_2 and g_1 to be generators of G_2 and G_1, resp.

If e (g_1, g_2) maps to G_T then e can be called as admissible bilinear map and e can be computed efficiently.

Admissible bilinear maps are the only bilinear maps we will be focusing on. Also, in this thesis, whenever we say bilinear map, it means admissible bilinear map.

2.3.3 Properties of Bilinear mapping function

1. G_1, G_2 are additive groups whereas G_T is a multiplicative group.

2. The bilinear pairing map $e: G_1 \times G_2 \rightarrow G_T$ satisfies:

- $e(P_2 + P_1, Q) = e(P_2, Q)\, e(P_1, Q)$
- $e(P, Q_2 + Q_1) = e(P, Q_2)\, e(P, Q_1)$ for all $Q, Q_2, Q_1 \in G_2$ and $P, P_2, P_1 \in G_1$.
- $e(xP, yQ) = e(P, Q)^{xy}$ for all $Q \in G_2$, $P \in G_1$ and $x, y \in Z_r$.
- e is efficiently computable.

2.4 Computational problems

The computation problems described below are the basis of the security of all the id-based authenticated key agreement protocols that we have discussed in this paper. Given below are some elliptic curve cryptography problems:

1.Discrete Logarithm problem (DL):

Given $Q, P \in G_1$, we have to find integer n

where $P = nQ$ in case such integer does exist.

2. Computational Diffie Hellman problem (CDH):

Given a tuple $(P, mP, nP) \in G_1$, we have to find the element mnP.

where $m, n \in Z_q^*$,

3. Decision Diffie Hellman problem (DDH):

Given the quadruple $(P, mP, nP, oP) \in G_1$, we have to prove whether $mn \bmod q = o$ or not.

where $m, n, o \in Z_q^*$

4. Bilinear Diffie Hellman problem (BDH):

Let P be a generator of G_1. Then problem which is in $< G_1, G_2, e >$ is that given $(P, mP, nP, oP) \in G_1$ for some randomly chosen m, n, o from Z_q^*, we need to evaluate $e(P, P)^{mno} \in G_2$.

2.5 Boneh-Lynn-Shacham (BLS) short signature scheme

In cryptography, a user can validate that the signer is authentic using the Boneh–Lynn–Shacham (BLS) signature scheme [7]. This scheme makes use of a bilinear pairing for confirming whether the signature is an element of an elliptic curve group.

BLS scheme consists of three functions: *generate*, *sign*, and *verify*.

2.5.1 Key Generation

A random integer x in Z_r is selected by the user who signs. The private key is x and is known only by the signer. The private key holder signer publishes the public key, g^x.

2.5.2 Signature

Some message m along with the private key x is given. The signer attains the signature by hashing the message m, as $H(m)$ where H is the hashing function. Signer outputs the signature $\sigma = h^x$.

2.5.3 Verification

A signature σ along with public key g^x is given. A person trying to verify calculates if:

$$e(\sigma, g) = e(H(m), g^x).$$

In case the above equation holds accurate, then we can say that the signature is verified.

One more thing that is worth noting is that the above signature scheme functions only because we have a bilinear pairing on a group and we are not able to find the private key x given g^x and g.

2.6 Identity Based Signatures

In order to offer integrity, authenticity and non-repudiation in asymmetric setting, digital signatures are one of the top basic primitives in cryptography. Principally, all the users of the system generate their own pair of keys, that is a public key and a secret key, and every user is distinctively recognized by other users by his/her public key. A PKI (Public-Key Infrastructure) is obligatory in order to map public keys to real-life identities. For simplifying the PKI requirements, the user's public key is contemplated as his/her identity in such an identity-based setting [8]. A reliable Key Generation Centre (KGC) circulates the user's secret key, which in turn stemmed from a master secret which is recognized only to the KGC, and who is anticipated to have a unique and secret way to confirm the identity of the user.

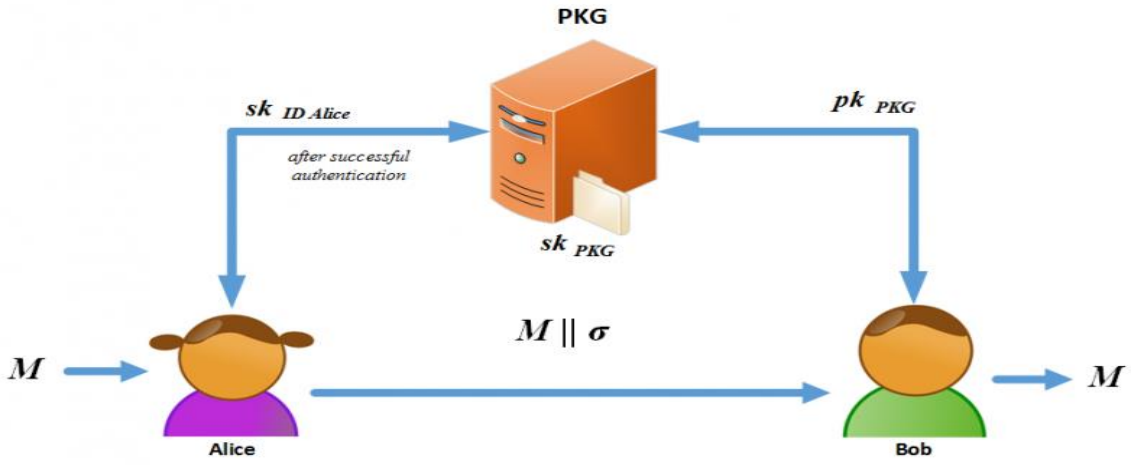


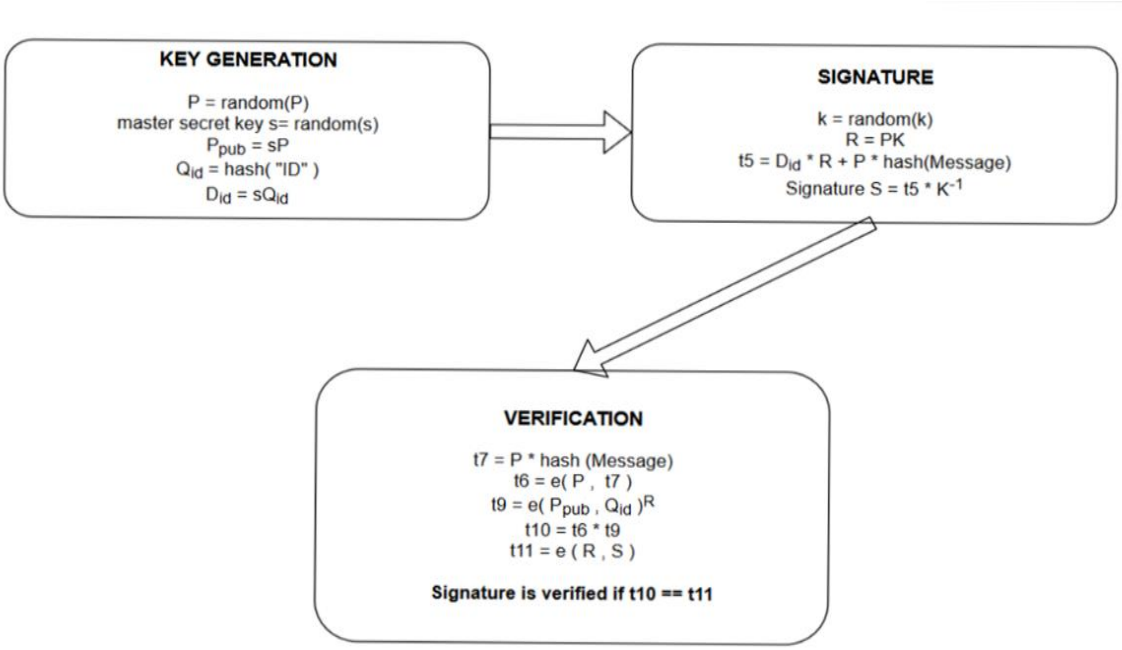
Fig. 2.3 Basic idea of Identity based signature

The three major steps involved in ID based signatures are:

- Key Generation
- Signature
- Verification

Here are some well-known Id-based signature schemes implemented using pairing based cryptography:

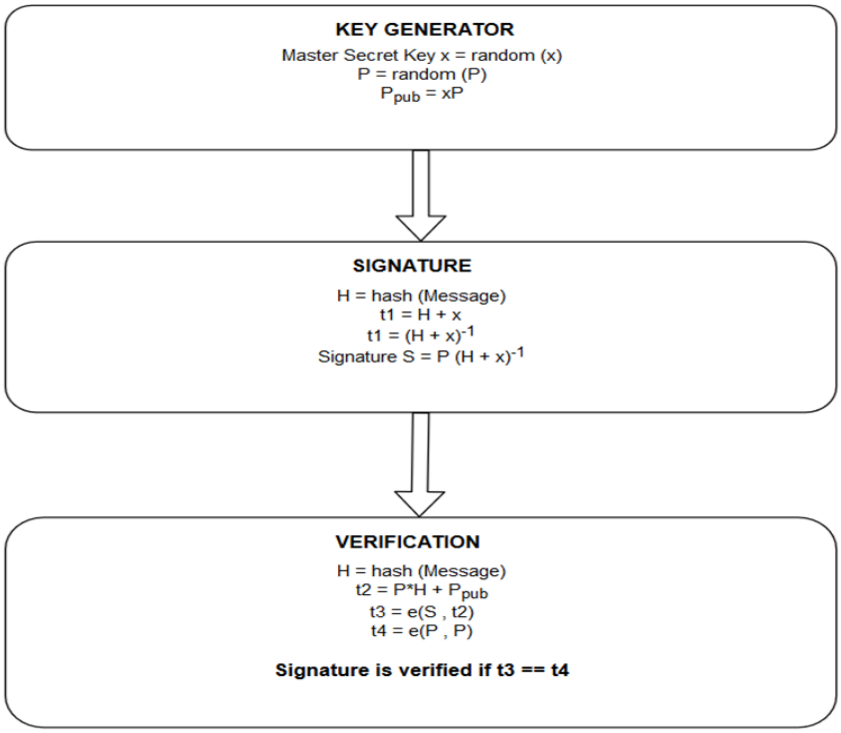
2.6.1 Paterson Id Based Signature



[15]

Fig. 2.4 Paterson ID based signature flow diagram

2.6.2 Zhang-Safavi-Naini-Susilo Signature



[14]

Fig. 2.5 Zhang-Safavi-Naini-Susilo Signature flow diagram

2.7 Key Agreement Algorithms

A key-agreement protocol [9] in cryptography is a modus operandi in which two or multiple groups can settle on a key in such a method that both of them contribute to the result. This makes it problematic for the unwanted third parties from imposing its key choice on the key-agreeing parties. Such types of protocols also do not expose the key that has been decided upon, to any spying party. There are numerous key exchange methods in which one party produces the key, and directs that key to the other party, which has no command over the key. These types of key-agreement protocols prevent some of the key circulation problems in a lot of cryptosystems.

2.7.1 Joux Tripartite Key Agreement Algorithm

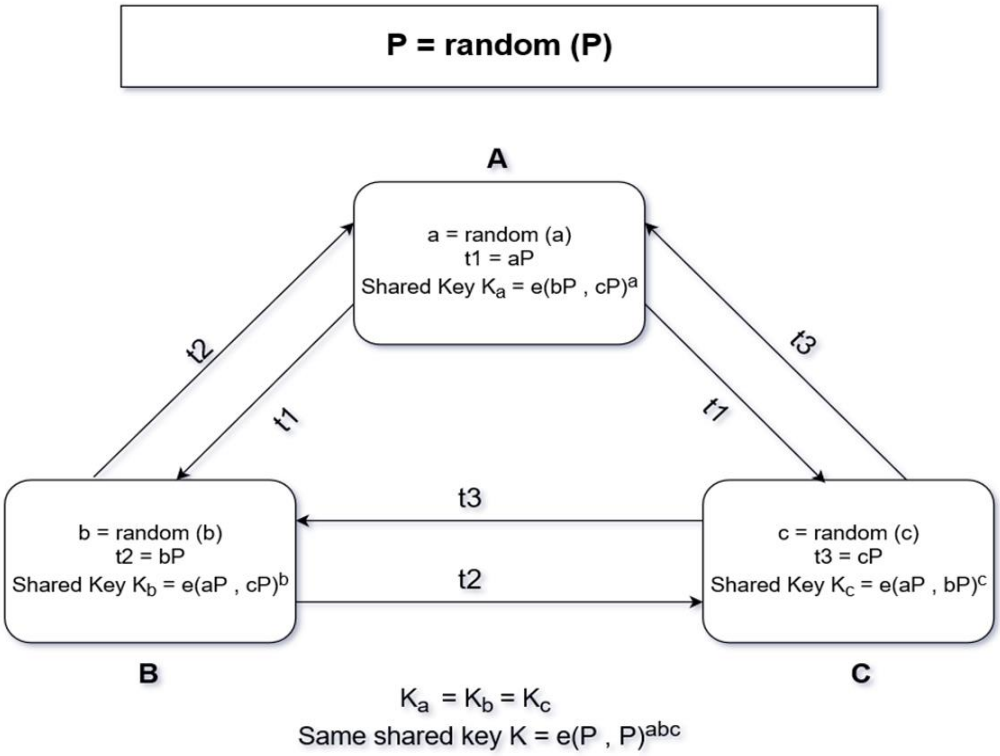


Fig. 2.6 Joux Tripartite Key Agreement Protocol flow diagram

Joux tripartite key agreement protocol is a protocol for key agreement amid three parties. A public element $P \in G_1$ is distributed to all the parties. Then each party produces an arbitrary secret key $n \in Z_r$ and multiplies it with the public element P and distributes it to the other two groups. Consequently, each party has its own private key, public key P and the keys obtained by the other two parties. Applying bilinear mapping as shown in the above figure, they all achieve the same shared key. [12]

2.7.2 Yuan-Li Id based authenticated Key Agreement Algorithm

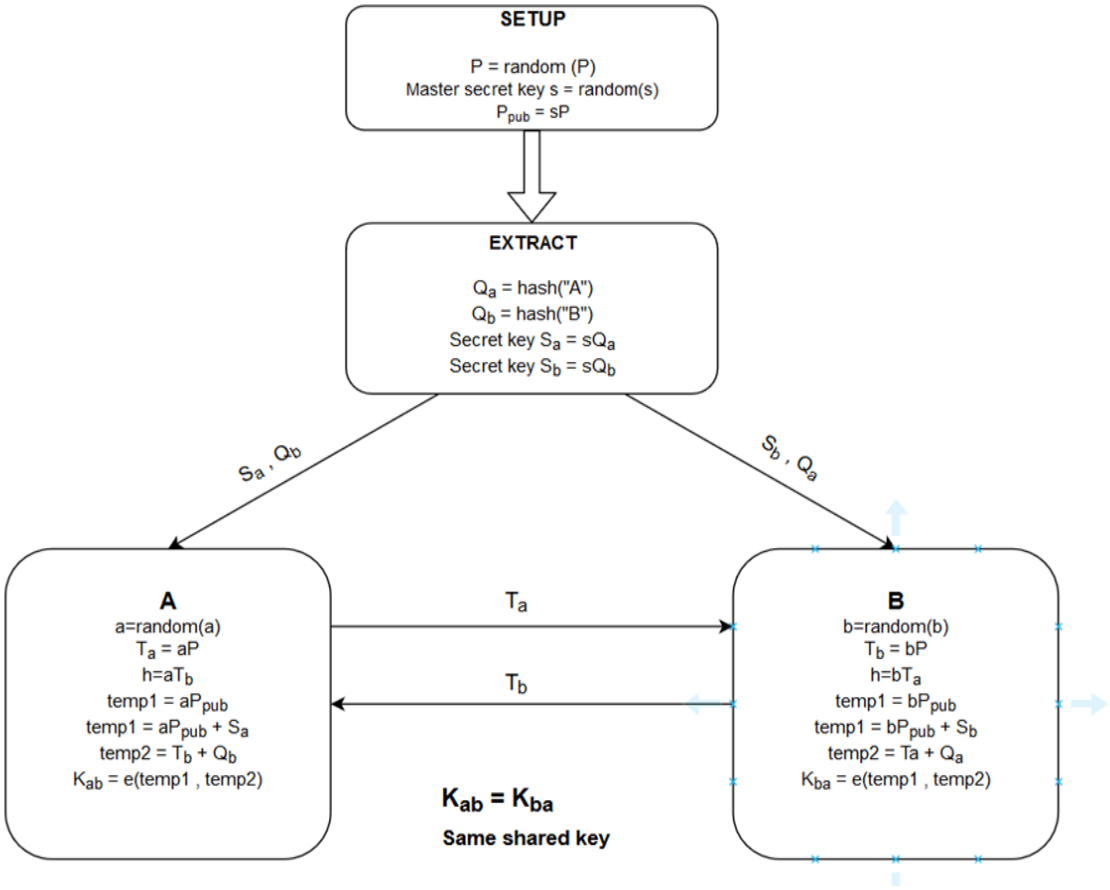


Fig. 2.7 Yuan-Li Id based authenticated Key Agreement Algorithm flow diagram

This algorithm proposes a way to use shared key between two parties. The agreement involves two users and a reliable third party known as key generator to distribute the secret keys. The above diagram depicts the way to implement the algorithm, where A and B are the users and $K_{ab} = K_{ba}$ is the shared key used by them. [13]

2.8 Distributed Access Control in Clouds (DACC)

Sushmita Ruj, Ivan Stojmenovic and Amiya Nayak who worked on this paper proposed a novel model for storing data on clouds and accessing it. In this scheme, it is not essential to store several encrypted copies of same data on the cloud. In this structure, the cloud is able to store encrypted data, but is incapable to decrypt any of it. The chief focus of this model is presenting the Key Distribution Centres (KDCs). They recommend Distributed Access Control in Clouds (DACC) algorithm [10], in which users and data owners are allocated keys by one or more KDCs. Specific fields in all records may be accessed from KDC. Thus, separate keys of owners are substituted by a single key. Specific set of attributes are appointed

to users and owners. The data is encrypted by the owner with the its acquired attributes and then uploads them over the cloud. Only the users having similar attribute set can recover this data from the cloud. ABE centred on elliptic curve bilinear pairings is utilised here. It is not possible for two users to together decrypt any data, which not any of them has individual access rights, thus the scheme is collusion secure. This methodology outcomes in lesser communication, calculation and storage expenses, when compared to prevailing schemes and models.

2.8.1 Attribute Based Encryption (ABE)

To grant rights to access to specific users and avoiding others from accessing the data is known as access control. One way of doing this is by adding a list of all authorized users to the encrypted data. However, in clouds, such types of lists can take up enormous amounts of memory and are frequently dynamic, making management of such lists very incompetent. The cloud has to look over the complete list to see if the user is authorized resulting in a vast storage and calculation cost. Another method for encryption is using the public keys of authorized users, so that by using their secret keys, only these users can decrypt data. But then we must encrypt the same data separately for every user a number of times, resulting in enormous storage costs.

Hence, an altered cryptographic technique known as Attribute Based Encryption (ABE) is utilized here for realising access control in clouds. Through ABE, data is encrypted by owners with attributes that they acquire and upload the encrypted data on the cloud. It is impossible for the cloud to decrypt the stored data. The Key Distribution Center (KDC) provide secret keys and attributes to users. Those having equivalent attribute sets are capable of decrypting the information. For instance, take into account a public health data repository. The history of the patients is included in the medical records, and may be accessed by medical experts like nurses and doctors, management authorities like government policy makers and insurance companies, and academicians and researchers. Different people may want access to different records. Attributes are assigned to every user, such as the name of hospital, profession and specialization, etc. Consider that only a neurologist or psychiatrist in Hospital X or Y is capable of decrypting the record of the medical history of a COVID-19 patient. Other users such as staff of hospital X or Y, or a gynaecologist belonging to Hospital Z, won't be able to access those encrypted records.

2.8.2 DACC Description

A new access control mechanism, DACC was proposed by them, where the attributes that users should possess are chose by the owners and decryption keys are provided to the users enabling them the access and read records that they are allowed to access. This scheme uses the Key Distribution Centers (KDCs), that dispense to users their secret keys. If there exists only one KDC for this system, then negotiating it may result in the ineffectiveness of the whole system. In order to avoid a single breakdown point, many KDCs acting as attribute authorities are used. The cloud keeps only those ciphertexts which it is incapable to decrypt. As an alternative of multiple copies, only one copy of ciphertext per record is saved in clouds. A disjoint set of attributes is possessed by every attribute authority. It is owners' verdict to select the attributes that should be possessed by the users to access the data. Every owner possesses access policy, usually in the outline of an access tree. By utilising the public keys which belongs to the policy attributes, data is encrypted which is in turn is stored on the cloud.

The choice of KDCs is application-dependent. For instance, if we wish to store health record data, there can exist multiple KDCs. Every user attains its designation from chief of medical research institute and its affiliation from the government. If two users collude, then it is ensured by the access control mechanism that they are unable to decrypt any sort of information which they are not authorized individually to access. Owner's access policy is usually in the form of a binary tree, with internal nodes in the form of AND & OR functions and attributes as leaf nodes. Lewko and Water's method is implemented here for general setting access control.

2.8.3 DACC Model and Assumptions

Consider a situation where data owners want to export their data to the cloud in the encrypted format and the users want to import this data from the cloud. In the health care scenario, users will be doctors, insurance companies and nurses whereas the patients would be the data owners. There are multiple KDCs which are assigned the task to distribute the keys to the users. KDCs can be organisations, who distribute dissimilar credentials to the users. KDCs can't be a part of the cloud as they can collude to find the users' secret keys. The figure below shows an overview of the cloud environment. Owners will encrypt the data and store it in cloud (C) and users will import the data from cloud and decrypt it. It can be noted that KDCs are not a part of the cloud. KDCs distribute the keys to the users (SK).

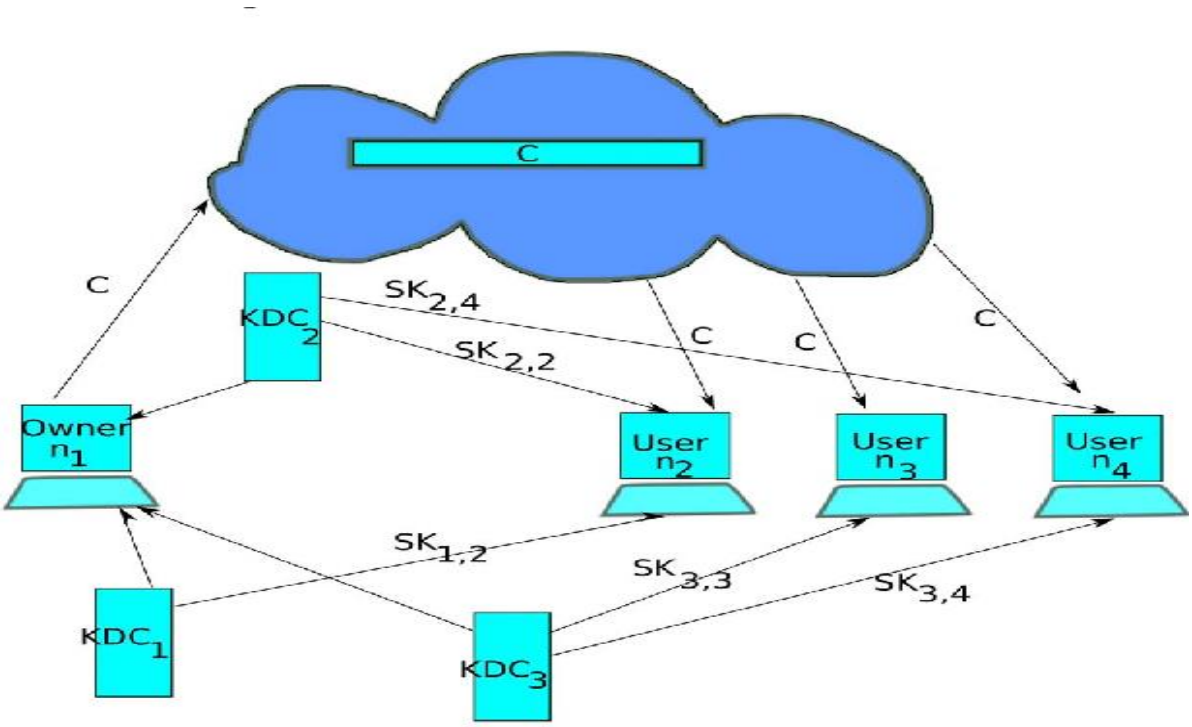


Fig. 2.8 Proposed cloud model of DACC

It is a basic assumption that users can collude to get access to the data which they should not be allowed to access, also cloud is considered to be honest to store the data and interfere with the data but also curious enough to try and find the keys of the users.

2.8.4 Format of Access Policy in DACC

Access policy can be one of the two types:

1. Boolean function

2. Linear Secret Sharing Scheme (LSSS) matrix

A Boolean function can be (x_1 or x_2 and x_3 or x_4 or x_5 and x_6), where x_1, x_2….x_6 are the attributes. The LSSS matrix can be obtained by converting a Boolean function to an access tree and then converting the access tree to the matrix. Section 3.2 describes a way to obtain the LSSS matrix using a postfix Boolean expression.

In DACC, LSSS matrix is used to represent the access policy. Consider the following example:

1. Type of job: Professor(J1), Engineer(J2), Doctor(J3)
2. Job location: India(P1) and Abroad(P2)

Boolean Expression: J1 or J2 and P1 or J3 and P2.

The above expression can be described as:

User wants a job as a professor or an engineer in India or a doctor abroad

The access policy tree is given in Figure 2.9

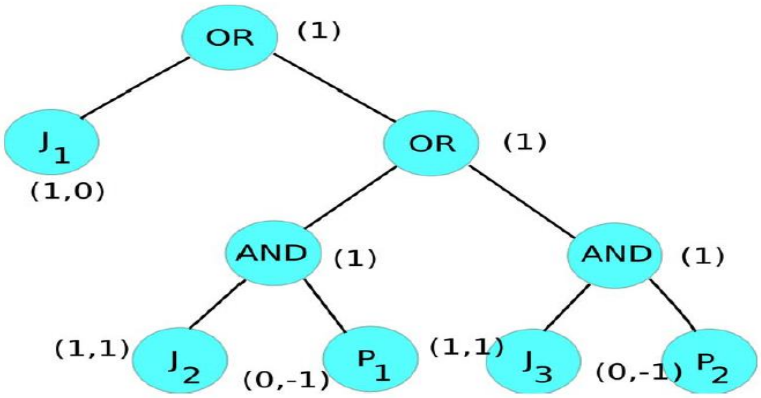


Fig. 2.9 Access policy tree for job search example; pad 0 to (1) to change it to (1, 0)

For the example taken above the R matrix obtained is:

$$R = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & -1 \\ 1 & 1 \\ 0 & -1 \end{pmatrix}.$$

Fig. 2.10 R-matrix for the job search example

2.8.5 Lewko-Waters ABE Scheme used in DACC

There are four steps in this scheme:

- Initialization of system
- Attribute and key distribution to users through KDCs
- Encryption by sender
- Decryption by receiver.

The Implementation of DACC using Lewko-Waters ABE Scheme has been discussed in Implementation Chapter 3.2.

2.9 Attribute Based Keyword Search (ABKS)

It is an important and necessary aspect to execute search over data encrypted in the cloud. But clouds are untrusted server environment and it is necessary to protect the user privacy. Thus, storing the data on the cloud without encryption can be considered a weak security in a cryptosystem. Hence must be encrypted before exporting it to cloud.

Attribute based keyword search scheme [11] discusses a way to build a cryptosystem where multiple owners can add data to the cloud servers and many data users can have access to this data and let them search over the encrypted dataset. This allows us to perform file level search operations. Each owner has to use his own access policy to implement the search authorisation.

2.9.1 ABKS Proposal

Using ABE this paper describes a way to perform keyword search on encrypted data in multiple-user multiple-owner scenario. An owner generates an index for each file he wants to share on the cloud using his access policy. This index then determines which users are able to search over the data. The user then generates a trapdoor with the help of his own attributes and the keyword he wishes to search over.

Using the user's trapdoor, the cloud server is able to search over the encrypted data and give the result only if the user's attributes satisfy access policy decided by the data owner.

2.9.2 ABKS System Model

Data owners, data users and cloud server are the 3 entities of the ABKS model. It is assumed that a trusted authority is helping with the distribution of the private keys and public keys. Using attribute-based access policy, secures indices are generated by data owners before they outsource the data to the cloud. Data user then generates a trapdoor using his own private keys and the interested keyword and submits the trapdoor to the cloud.

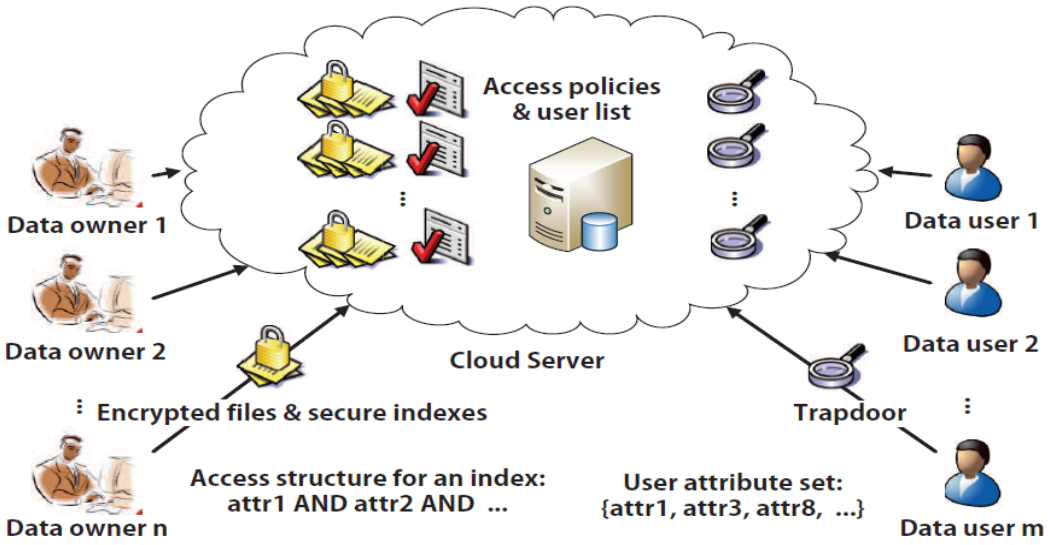


Fig. 2.11 Framework of authorized keyword search over encrypted cloud data

Th cloud server then searches for the dataset and return the result if the attributes used to create the trapdoor satisfy the access policy of the indices of the files.

2.9.3 Algorithm Definition of Authorized Keyword Search

Access policy in this scheme is represented as a sequence of AND gates. Only those users who have the attributes to satisfy the access policy are allowed to search over the datasets.

Let us consider the following situation:

N = universal attribute set {1, 2, 3, …., n}

$I \subseteq N$ = attribute set used for access policy, (Comprises of either i or $\neg i$)

Negations $\neg i$ = *not i (negation of attribute i)*

S = Attribute set for secret key of user, $S \subseteq N$

We use bilinear map and properties of bilinearity, non-degeneracy and computability.

There are 6 steps involved in the ABKS algorithm:

- Setup $(N, \lambda) \rightarrow (MK, PK)$: Generate master-key (MK) and public parameters PK using λ and N.
- CreateUL (ID, PK) $\rightarrow UL$: Generates a user list for a given dataset using the public parameters (PK) and user identities (ID).
- EncIndex (GT, PK, w) $\rightarrow D$: To encrypt the index we use the access structure (GT), public parameters (PK) and keyword $w \in W$.

- KeyGen (MK, S, PK) → SK: To generate the secret key of user we use master-key (MK), user attribute set (S) and public parameters (PK).

- GenTrapdoor (SK, w', PK) → Q: To generate the trapdoor for the keyword $w' \in W$ we use users secret keys (SK) and public parameters (PK).

- Search (D, Q, UL) → search results: This algorithm uses encrypted file index (D), user generated trapdoor (Q) and user list (UL) to return the valid search result for the authorised users.

2.9.4 Construction for ABKS

The concrete ABKS construction is based on the algorithms that we have discussed in section 2.9.4. The operations performed on system level are: System Setup, New User Enrollment, Secure Index Generation, Trapdoor Generation and Search. Also, each of these system level operations may call for one or many low-level algorithms. The implementation of Attribute Based Keyword Search has been discussed in Project work chapter 3.3.

CHAPTER 3: IMPLEMENTATION

3.1 Pairing Based Cryptography Library

3.1.1 Overview

To perform the extensive mathematical operations involved in pairing based cryptographic systems, there is a library called GMP (GNU Multiple Precision Arithmetic) library. Built on this GMP library is the PBC (Pairing Based Cryptography) library [2]. PBC is a free library released by Stanford University.

This library was developed keeping in mind the basic and core requirements while implementing a pairing based cryptographic system, and achieving portability and speed. This library is available in many languages. Functionalities such as exponentiation in groups, arithmetic in elliptic curves, generation of curves and evaluation of bilinear pairings are provided in this library.

The PBC library and its API are simple to use, however the programmer needs to have a basic understanding of pairings and elliptic curve cryptography along with good command over coding in C, Python or other supported coding languages.

More details on the PBC library can be found on the pbc library website provided by Stanford university.

3.1.2 Setup and Installation

- The PBC library has been built upon the GMP library. Hence installing the GMP library is a requirement. To install the GMP library, open the terminal in ubuntu (or any Linux distribution) and type the following command:

  - sudo apt-get install libgmp3-dev

  This will install the GMP library on your machine.

- Flex and bison are also needed for the successful installation of the PBC library. To install flex and bison, open the terminal and type the commands given below one by one:

  - sudo apt-get update
  - sudo apt-get upgrade
  - sudo apt-get install flex bison

  This will successfully install flex and bison.

- PBC library can be downloaded from downloads section of the PBC library website provide by Stanford. To install PBC library extract the contents of the downloaded file, open the terminal and change the directory to a folder where you have extracted the contents and use the commands given below to install the library on your machine:

- sudo ./configure
- sudo make
- sudo make install
- sudo ldconfig -v

PBC library is now successfully installed on your machine.

3.2 Implementation of Distributed Access Control in Clouds (DACC)

There are two steps in which encryption is done. First, we obtain the LSSS matrix from the Boolean access tree. Second, we encrypt the message and send both the message and the LSSS matrix to the cloud.

Access tree is decided by the sender. LSSS *R*-matrix can be derived as:

First construct an access tree from a postfix expression of Boolean logic for example: "EAB*CD*+AB+CD+*+*" using constructTree method which gives the pointer to root. The details of constructTree have been dropped for simplicity.

```
char postfix [] = "EAB*CD*+AB+CD+*+*";
et* root = constructTree(postfix);
```

Now construct LSSS Matrix *R* as follows:

```
root->v.push_back (1);
inorder(root);
cout << "$$" << endl;
for (int i=0; i<r.size(); i++)
{
      while(r[i].size()!= c)
      {
            r[i].push_back (0);
      }
}
```

Save the *R*-matrix in a file as:

```
myfile.open("rmatrix.txt",fstream::out);
myfile << r.size()<< " ";
myfile << r[0].size()<< " "<<endl;
for (int i=0; i<r.size(); i++)
{
      for (int j=0; j<r[i].size(); j++)
      {
```

```
                myfile << r[i][j]<< " ";
            }
            myfile<< std::endl;
        }
        myfile.close();
```

3.2.1 System Initialization

Select a large prime q, g which is a generator in G, groups G and G_T of order q, a bilinear map e: $G \times G \rightarrow G_T$, and finally the hash function H: $\{0, 1\}^* \rightarrow G$ which maps the user identities to G. The hash function we used is the default hash function in PBC.

```
        pairing_t pairing;
        pbc_demo_pairing_init (pairing, argc, argv);
```

```
        element_init_G1(g, pairing);
        element_random(g);
```

Every Key Distribution Center $A_j \in A$ maintains an attribute set L_j. These attributes are disjoint ($L_i \cap L_j = \emptyset$ for $i \neq j$). Also, each Key Distribution Center selects two random exponents y_i, $\alpha_i \in Z_q$.

```
        for (int i = 0; i < Ri; i++)
        {
                element_init_Zr(alpha[i], pairing);
                element_random(alpha[i]);
                element_init_Zr(y[i], pairing);
                element_random(y[i]);
        }
```

The secret key obtained for Key Distribution Center A_j is
$SK[j] = \{y_i , \alpha_i , i \in L_j\}$.

The public key obtained for Key Distribution Center A_j is published:
$PK[j] = \{g^{yi}, e(g, g)^{\alpha i} , i \in Lj\}$.

3.2.2 Key generation and distribution by KDCs

A data user U_u gets an attribute set $I[j, u]$ from Key Distribution Center A_j, along with a secret key $sk_{i,u}$ for every $i \in I[j, u]$

$sk_{i,u} = H(u)^{yi} g^{\alpha i}$, for $y_i, \alpha_i \in SK[j]$.

```
        element_from_hash (user, "Alice Bob Eve", 14);
        // Secret Key of user
        for (int i = 0; i < Ri; i++)
        {
                element_pow_zn (t12, user, y[i]);
                element_pow_zn (t13, g, alpha[i]);
                element_mul(sk[i], t12, t13);
```

```
}
```

Only a specified user is able to decrypt data using the secret key because all keys are sent to the data user securely by using the public key of user.

3.2.3 Encryption by Sender

Access tree is decided by the sender. We can derive the LSSS R-matrix as described at the beginning of section 3.2.

The message *M* is encrypted by the sender as:

1. Select a random element $s \in Z_q$. With s as its first entry, select a random vector $v \in Z_q^h$. h denotes number of leaves present in the access tree which is basically the number of rows in the R-matrix corresponding to the tree.

```
element_random(s);
for (int i = 0; i < Rj; i++)
{
	element_init_Zr(v[i], pairing);
}
element_set (v [0], s); // setting first element of vector as s
for (int i=1; i<Rj; i++)
{
	element_random(v2);
	element_set(v[i], v2);
}
```

2. Evaluate $\lambda_x = R_x \cdot v$, where R_x being a row of R.

```
for (int i = 0; i < Ri; i++)
{
	// Dot product of two vectors
	element_mul_mpz(lambda[i], v [0], R[i][0]);
	for (int j=1; j<Rj; j++)
	{
		element_mul_mpz (t2, v[j], R[i][j]);
		element_add(lambda[i], lambda[i], t2);
	}
}
```

3. Randomly select a vector $w \in Z_q^h$ with the first entry as 0.

```
element_set0(w [0]); // setting first element as zero
for (int i=1; i<Rj; i++)
{
	element_random(w2);
	element_set(w[i], w2);
```

```
}
```

4. Set $\omega_x = R_x \cdot w$.

```
for (int i = 0; i < Ri; i++)
{
	// Dot product of two vectors
	element_mul_mpz(omega[i], w [0], R[i][0]);
	for (int j=1; j<Rj; j++)
	{
		element_mul_mpz (t2, w[j], R[i][j]);
		element_add(omega[i], omega[i], t2);
	}
}
```

5. Corresponding to every row R_x in R, randomly select $\rho_x \in Z_q$.

```
element_random(rho);
```

6. Evaluate the following ciphertext parameters:

$C_0 = M * e(g, g)^{s.}$ In which we read text from a file and store it in a string named *str*.

```
element_from_hash (M, str, strlen(str));
element_pow_zn (gs, g, s);
element_pairing (t3, gs, g);
element_mul (C0, M, t3);
```

$C_{1,x} = e(g, g)^{\lambda x} e(g, g)^{\alpha\pi(x)\rho x}$, $\forall x$

```
for (int i = 0; i < Ri; i++)
{
	//for C1
	printf ("VALUE OF C1 FOR ROW: %d\n", i);
	element_pow_zn (t4, g, lambda[i]);
	element_pairing (t5, t4, g);
	element_mul_zn (t6, alpha[i], rho);
	element_pow_zn (t7, g, t6);
	element_pairing (t8, t7, g);
	element_mul(C1[i], t5, t8);
}
```

$C_{2,x} = g^{\rho x}$ $\forall x$

```
for (int i = 0; i < Ri; i++)
{
	//for C2
```

```
            element_pow_zn(C2[i], g, rho);
      }
```

$C_{3,x} = g^{y\pi(x)\rho x}\, g^{\omega x}\ \forall x,$

```
      for (int i = 0; i < Ri; i++)
      {
            //for C3
            element_mul_zn (t9, y[i], rho);
            element_pow_zn (t10, g, t9);
            element_pow_zn (t11, g, omega[i]);
            element_mul(C3[i], t10, t11);
      }
```

where $\pi(x)$ is mapping from R_x to the attribute i present at the corresponding leaf of the data owner's access tree.

3.2.4 Decryption by Receiver

A receiver named U_u takes ciphertext C, secret keys $\{sk_{iu}\}$, group G as inputs, and returns the message M. It receives the mapping π and access matrix R from C. It then proceeds with the following steps:

1. Receiver U_u evaluates the attribute set $\{\pi(x): x \in X\} \cap Ii$ which is common to itself as well as the access matrix where X is the set of rows of R.
2. For each of these attributes, it searches if there is a subset X' of rows of R, such that their linear combination is the vector (1, 0 . . ., 0). If such subset is not present then decryption is not possible. If it exists, it calculates constants $c_x \in Z_q$, such that $\Sigma_{x\in X'} c_x R_x = (1, 0, \ldots, 0)$.
3. The Decryption process continues as follows:

$dec(x) = (C_{1,x}\, e(H(u), C_{3,x})) / (e(sk_{\pi(x),u}, C_{2,x}))$, for every $x \in X'$

```
      for (int i = 0; i < Ri; i++)
      {
            element_pairing(t14, user, C3[i]);
            element_mul (t15, C1[i], t14);
            element_pairing (t16, sk[i], C2[i]);

            element_div (dec[i], t15, t16);
      }
```

Receiver U_u finally evaluates $M = C_0 / X_{X'}\, dec(x)$.

```
      printf ("PRODUCT OF DEC[I]s \n\n");
      element_mul (t17, dec [3], dec [4]); //Only 0,3 and 4 belong to X'
      element_mul (t17, t17, dec [0]);
      element_printf ("\n\n%B \n\n", t17);
```

```
element_div (t18, C0, t17);
```

3.3 Implementation of Attribute Based Keyword Search (ABKS)

In this section, we will discuss about the concrete ABKS construction from the perspective of system level centred on the algorithms defined in literature survey. The system level operations include the following steps:

- System Setup
- New User Enrolment
- Secure Index Generation
- Trapdoor Generation
- Search

It can be noted that one or more low level algorithms may be invoked from each individual system level operation.

3.3.1 System Setup

The Trusted Authority (TA) invokes the Setup algorithm to generate *PK* and *MK*. It selectively chooses random elements t_1, ..., t_{3n}. A collision-resistant keyed hash function H: $\{0, 1\}^* \rightarrow Z_p$ is defined, and its key is arbitrarily chosen and shared securely amid users and owners. Let $T_k = g^{tk}$ $\forall$ $k \in \{1, ..., 3n\}$ such that for $1 \leq i \leq n$, T_i are counted as *positive* attributes, T_{n+i} as *negative* ones, and T_{2n+i} as *don't care*.

```
        for (int i=0; i<3*n; i++)
        //Here n is the total number of attributes
        //g is the generator in group G
{
        element_random(t[i]);
                //element_random randomly chooses an element from Zr
        element_pow_zn(T[i], g, t[i]);
                //ti are in Zr and Ti are in G
}
```

Let Y be $e(g, g)^{y.}$

```
        element_random(y);
element_pow_zn (temp_g1, g, y); //exponentiation function in PBC
        //temp_g1 is a temporary variable in G
element_pairing (Y, g, temp_g1);
        //element_pairing is a bilinear mapping function that maps two
elements in G to an element in GT
```

The public key is PK: = < e, g, Y, T_1, ..., T_{3n} >

The master key is MK: = < y, t_1, ..., t_{3n} >. The initial version number *ver* is 1. The Trusted authority TA publishes (*ver*, *PK*) with the signature of each component of *PK*, and keeps (*ver*, *MK*).

We store these values in a text file setup.txt as:

```
FILE *fptr;
fptr = fopen ("setup.txt", "w");
ver = 1; //Initially the version is set to 1
element_fprintf (fptr, "%d\n", n);
element_fprintf (fptr, "%d\n", i_dash);
element_fprintf (fptr, "%d\n", ver);
element_fprintf (fptr, "%B\n", g);
element_fprintf (fptr, "%B\n", Y);
      element_fprintf (fptr, "%B\n", y);
element_fprintf (fptr, "%B\n", s);
for (int i=0; i<3*n; i++)
{
      element_fprintf (fptr, "%B\n", t[i]);
}
for (int i=0; i<3*n; i++)
{
      element_fprintf (fptr, "%B\n", T[i]);
}
fclose(fptr);
```

3.3.2 New User Enrolment

First, we import the required values from setup.txt

```
FILE *fptr2;
fptr2 = fopen ("setup.txt", "r");
fscanf (fptr2,"%d”, &n);
fscanf (fptr2,"%d”, &i_dash);
fscanf (fptr2,"%d”, &ver);
getline (&str, &si, fptr2);
element_set_str (g, str, 10);
getline (&str, &si, fptr2);
element_set_str (Y, str, 10);
getline (&str, &si, fptr2);
element_set_str (y, str, 10);
getline (&str, &si, fptr2);
element_set_str (s, str, 10);
for (int i=0; i<3*n; i++)
{
```

```
            getline (&str, &si, fptr2);
            element_set_str (t[i], str,10);
      }
      fclose(fptr2);
```

When a registration request is received by the TA from a new authentic user *f*, it chooses an arbitrary $x_f \in Z_p$ as a *MK* component.

```
      element_random(xf);
```

Next, the TA creates a new *PK* component $Y_f^{'} = Y^{xf}$ and makes it available for public along with its signature.

```
      element_pow_zn (Yf_, Y, xf);
```

After this, the KeyGen algorithm is executed to generate secret key *SK* for this user. The TA selects random *ri* from *Zp* for every $i \in N$, hence $r = \Sigma_{i=1 \text{ to } n}\, r_i$.

```
      for (int i=0; i<n; i++)
      {
            element_random(ri[i]);
            element_add (r, r, ri[i]); //Add function in PBC
      }
```

Set $\hat{K} = g^{y-r}$. Set $K_i = g^{ri/ti}$ for $i \in S$

For other values of *i*, set $K_i = g^{ri/tn+i}$.

```
      element_sub (temp_zr, y, r); //Subtract function in PBC
      element_pow_zn (K_hat, g, temp_zr);
      //temp_zr is a temporary variable in Z
      for (int i=0; i<n; i++)
      {
            element_invert (temp_zr, t[n+i]); //Inverse function in PBC
            element_mul_zn (temp_zr, ri[i], temp_zr); //Multiplication
            element_pow_zn(K[i], g, temp_zr);
      }
      for (int i=0; i<o; i++)//o is the number of user attributes
      {
            //S [] is the user's attribute set
            element_invert (temp_zr, t[S[i]]);
            element_mul_zn (temp_zr, ri[S[i]], temp_zr);
            element_pow_zn(K[S[i]], g, temp_zr);
      }
```

Finally, set $F_i = g^{ri/t2n+i}$.

```
      for (int i=0; i<n; i++)
```

```
	{
		element_invert (temp_zr, t [2*n + i]);
		element_mul_zn (temp_zr, ri[i], temp_zr);
		element_pow_zn (F[i], g, temp_zr);
	}
```

The secret key for the user is SK: = $<\hat{K}, x_f, ver, \{K_i, F_i\}_{i \in N}>$.

Whenever a new user joins the cloud environment, the data owner executes CeateUL algorithm to set $\bar{D}_f = Y_f^{'-s}$. A List is maintained for each dataset who are allowed to perform search actions over the dataset. This new user is then added to that list.

```
	element_neg (temp_zr, s); //Negation function in PBC
	element_pow_zn (Df_bar, Yf_,temp_zr);
```

ID_f = the identity of the new user f.

Cloud server then appends the tuple (ID_f, $\bar{D}_f$) into the user list UL upon the request of the data owner.

Finally, we create a file for the user and store the required values:

```
	char* f="username";
	char* extension = ".txt";
	char fileSpec[strlen(f)+strlen(extension)+1];
	FILE *fptr;
	//To create a file with the name of user
	snprintf (fileSpec, sizeof (fileSpec), "%s%s", f, extension);
	fptr = fopen (fileSpec, "w");
	element_fprintf (fptr, "%B\n", xf);
	element_fprintf (fptr, "%B\n", K_hat);
	for (int i=0; i<n; i++)
	{
		element_fprintf (fptr, "%B\n", K[i]);
	}
	for (int i=0; i<n; i++)
	{
		element_fprintf (fptr, "%B\n", F[i]);
	}
	element_fprintf (fptr, "%B\n", Df_bar);
	fclose(fptr);
```

3.3.3 Secure Index Generation

For generating a secure index D for a file, data owner uses the EncIndex algorithm. That will set:

$\tilde{D} = Y^s$ *and* $\hat{D} = g^s$.

```
element_pow_zn (D_hat, g, s);
element_pow_zn (D_tilde, Y, s);
```

GT (Access policy of data owner) = $\Lambda_{i \in I} \underline{i}$,

$D_i = T_i^s$ if $\underline{i} = i$,

$D_i = T^s_{n+i}$ if $\underline{i} = \neg i$ and

$D_i = T^s_{2n+i}$ for each $i \in N \backslash I$

```
for (int i=0; i<n; i++)
{
      element_pow_zn(D[i], T [2*n + i], s);
}
for (int i=0; i<m; i++)
//m is the number of attributes in data owner's access policy
{
      if(I[i]>=0) //I [] is data owner's access policy
      {
            element_pow_zn(D[I[i]], T[I[i]], s);
      }
      else
      {
            int j = (-1) *I[i];
            element_pow_zn(D[j], T [n + j], s);
      }
}
```

The data owner sets $D_{i'} = T_{i'}^{s/H(w)}$ for some attribute $i' \in N$ a keyword $w \in W$, Attribute i' is presumed to be positive without loss of generality.

```
element_from_hash (h, "hashofmessage", 13);
//element_from_hash generates a hash of message
element_invert (temp_zr, h);
element_mul_zn (temp_zr, s, temp_zr);
element_pow_zn(D[i_dash], T[i_dash], temp_zr);
```

The secure index D: = < ver, GT, $\hat{D}$, $\tilde{D}$, $\{D_i\}_{i \in N}$ >.

We store these values in a text file index.txt as:

```
FILE *fptr2;
fptr2 = fopen ("index.txt", "w");
element_fprintf (fptr2, "%B\n", D_hat);
element_fprintf (fptr2, "%B\n", D_tilde);
for (int i=0; i<n; i++)
{
```

```
            element_fprintf (fptr2, "%B\n", D[i]);
      }
      fclose(fptr2);
```

3.3.4 Trapdoor Generation

First, we import the required values from username.txt, setup.txt and index.txt as:

```
      FILE *fptr;
      fptr = fopen ("username.txt", "r");
      getline (&str, &si, fptr); //Taking input from file
      element_set_str (xf, str,10);
      getline (&str, &si, fptr);
      element_set_str (K_hat, str,10);
      for (int i=0; i<n; i++)
      {
            getline (&str, &si, fptr);
            element_set_str(K[i], str,10);
      }
      for (int i=0; i<n; i++)
      {
            getline (&str, &si, fptr);
            element_set_str(F[i], str,10);
      }
      getline (&str, &si, fptr);
      element_set_str (Df_bar, str,10);
      fclose(fptr);

      FILE *fptr2;
      fptr2 = fopen ("setup.txt", "r");
      fscanf (fptr2,"%d", &n);
      fscanf (fptr2,"%d", &i_dash);
      fclose(fptr2);

      FILE *fptr3;
      fptr3 = fopen ("index.txt", "r");
      getline (&str, &si, fptr3);
      element_set_str (D_hat, str,10);
      getline (&str, &si, fptr);
      element_set_str (D_tilde, str,10);
      for (int i=0; i<n; i++)
      {
            getline (&str, &si, fptr);
            element_set_str(D[i], str,10);
      }
      fclose(fptr3);
```

Is a user is interested in searching a keyword then he can use GenTrapdoor to generate a trapdoor. The user selects a random $u \in Z_p$.

```
element_random(u);
```

Let $\tilde{Q} = u + x_f$ and $\hat{Q} = \hat{K}^u$

```
element_add (Q_tilde, u, xf);
element_pow_zn (Q_hat, K_hat, u);
```

Also, $Q_i = K_i^u$ and $Qf_i = F_i^u$.

```
for (int i=0; i<n; i++)
{
      element_pow_zn(Q[i], K[i], u);
      element_pow_zn (Qf[i], F[i], u);
}
```

Set $Q_{i'} = K_{i'}^{H(w')\cdot u}$ and $Qf_{i'} = F_{i'}^{H(w')\cdot u}$

here i' is the same as used in the earlier step and w' is the keyword that the user wants to search.

```
element_from_hash (h_dash, "hashofmessage", 13);
element_mul_zn (temp_zr, h_dash, u);
element_pow_zn(Q[i_dash], K[i_dash], temp_zr);
element_pow_zn (Qf[i_dash], F[i_dash], temp_zr);
```

The trapdoor Q: = $<\tilde{Q}, \hat{Q}$, ver, $\{Q_i, Qfi\}_{i \in N}>$, where *ver* is the version number of *SK* for generating the trapdoor.

3.3.5 Search

The cloud receives the user identity ID_f and trapdoor Q,

1. The cloud uses the UL to find whether ID_f is a legitimate user, if not then the user is not Permitted to search.

2. Else, using the encrypted index D and the user generated trapdoor Q the cloud server executes the Search algorithm.

$e(D_{i'}, Qi_{'})$ is equal to $e(g, g)^{s\cdot u\cdot ri'}$ as well for the attribute $i' \in N$.

The user satisfies the access policy of the index and $w' = w$, if the following equation holds:

$$\tilde{D}^{\tilde{Q}} \cdot \bar{D}_f = e(\hat{D}, \hat{Q}) \cdot \Pi_{i=1 \text{ to } n}\, e(D_i, Q_i^*),$$

where $Q_i^* = Q_i$ if $i \in I$, else $Q_i^* = Qf_i$.

For LHS:

```
element_pow_zn (temp_gt, D_tilde, Q_tilde);
element_mul (lhs, temp_gt, Df_bar);
```

For RHS:

Calculating Q_i^* values:

```
for (int i=0; i<n; i++)
{
	element_set(Q_star[i], Qf[i]);
	//element_set assigns value from one variable to another
}
for (int i=0; i<m; i++)
{
	if(I[i]>=0)
	{
		element_set(Q_star[I[i]], Q[I[i]]);
	}
	else {
		int j = -I[i];
		element_set(Q_star[j], Q[j]);
	}
}
```

Calculating RHS value:

```
element_pairing (temp_gt, D [0], Q_star [0]);
element_set (rhs, temp_gt);
//temp_gt is a temporary variable in GT
for (int i=1; i<n; i++)
{
	element_pairing (temp_gt, D[i], Q_star[i]);
	element_mul (rhs, rhs, temp_gt);
}
element_pairing (temp_gt, D_hat, Q_hat);
element_mul (rhs, temp_gt, rhs);
```

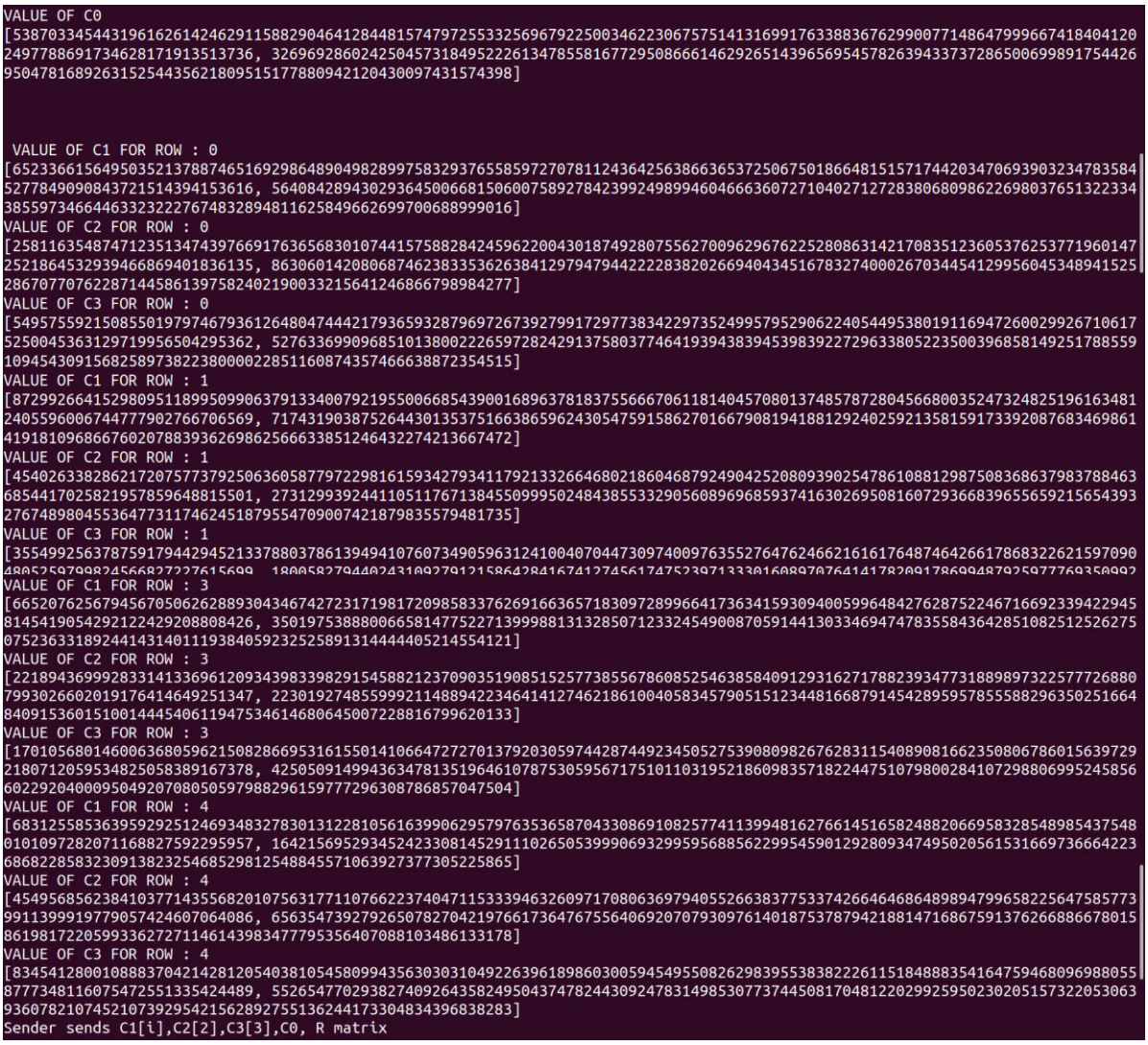
Comparing LHS and RHS of equation:

```
if (!element_cmp (lhs, rhs)) //Comparison function in PBC
{
	printf ("USER IS VERIFIED! \n\n\n\n");
}
else
{
	printf ("USER IS NOT VERIFIED\n");
}
```

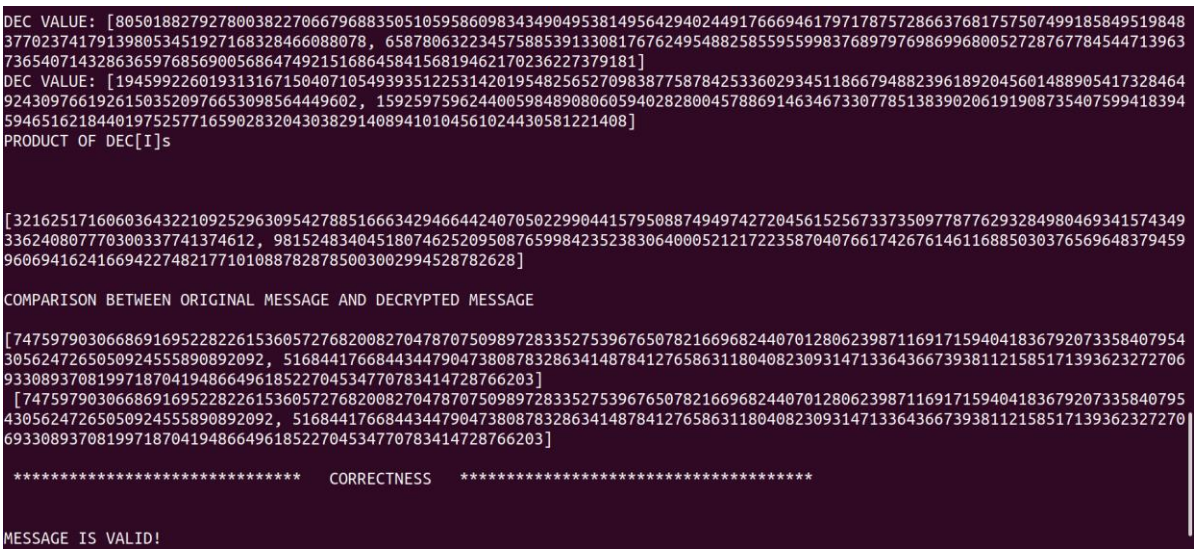
CHAPTER 4: RESULT AND CONCLUSION

After implementing the DACC algorithm, the computational costs associated with it are as follows:

To evaluate the R-matrix with *m* attributes, the time complexity is O(*m*). The time taken to check whether there exists a set of rows where CR = (1, 0, . . . , 0), is *O*(*mh*), h being the number of columns in R-matrix. The most expensive operation in DACC is bilinear pairing. In encryption part, a user U_u evaluates $e(g, g))$ one time. In order to evaluate $C_{1,x}$ it also does two scalar multiplications and one for $C_{2,x}$ and one for $C_{3,x}$ totalling to 4m scalar multiplications.



In decryption part, there are 2*m* bilinear pairings to calculate $e(H(u), C_{3,x})$. To evaluate $(e(g, g)^{\lambda x} e(H(u), g)^{\omega x})^{\sigma x}$, *m* scalar multiplications are performed.



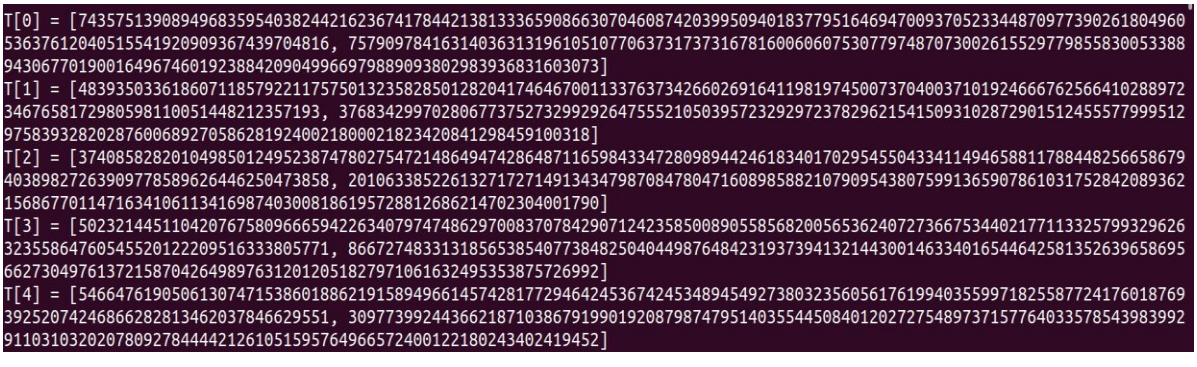
So finally the time complexity of the scheme is $(2m+1)T_p+5mT_m$, where T_p is the time taken for one bilinear pairing and T_m is the time taken for one scalar multiplication.

Access policy	Secret key size	Time complexity of user computations
A monotonic Boolean function	Does not depend on how many users exist	$O(mT_p)$

Table 4.1 Performance evaluation of DACC

After implementing the ABKS algorithm the computational costs associated with it are as follows:

Let us represent the complexity of bilinear pairing operation as P, the exponentiation in G as E, the multiplication in G as M, the exponentiation in G_1 as E_1 and the multiplication in G_1 as M_1. During the system setup, the trusted authority performs $3n$ number of exponentiations in G



and one in G_1 and one bilinear pairing.

```
Y = [231820305470433698664748668991192157534575269118714427841561096344810641123293510205676778018378015939429971591416992981634742082506793126292553368407150, 663492349746292324831460037684277602307287218451160907904176155473266348263388649672075661785152821377859036757196762410562261771344826993075042949618625 0]
```

During the New User Enrollment step, $2n + 1$ exponentiations in G are needed for secret key of new user,

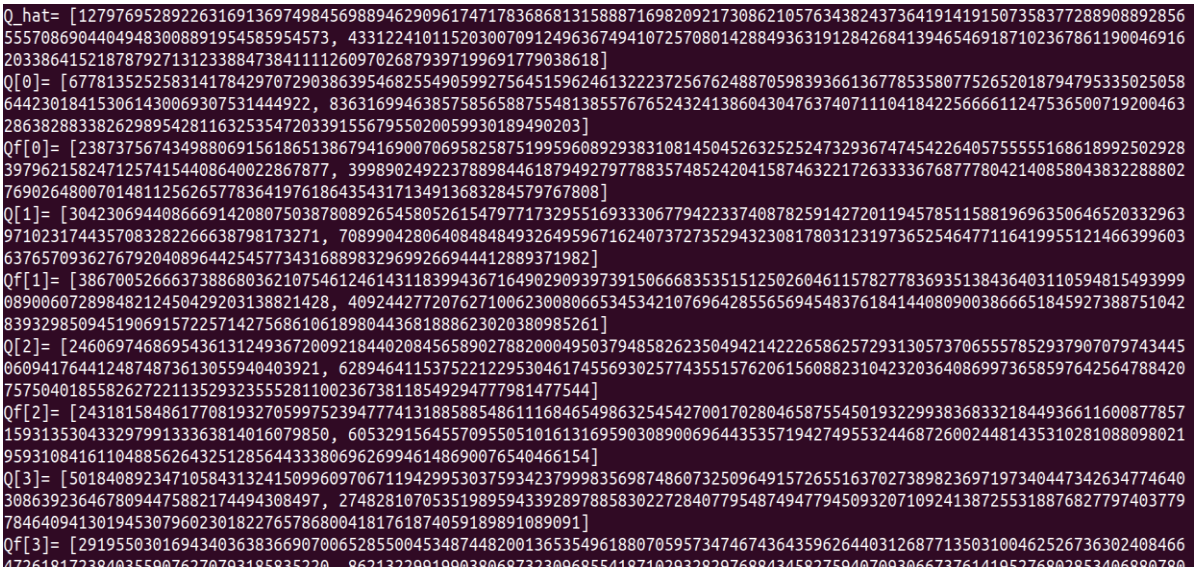
```
K_hat = [683252336072786819894684058214495354755855930157764846180942676637185457561883153627716060509196711411846847228499090760078246411296794164114461676131991, 519756683904979836869472061264623895587842016973913566769578369327425131832369633048889337644088787342686383914114933088684979260321068414784334468703628 8]
K[0] = [219053498529465383047289124739206827036766203090399245947391749187039199383499498086568279382889711692517480193175095745631380134652886089546912688417264 1, 158250184607388353359438711776153505894345897722398012980045849733759084379791539890530679799550198589908005750013055488566345681992423821981129863673894 0]
K[1] = [635762845392924111034644365215412024383244553884628348233267989402815673286325984456659014323442482637965178203997122474953579541759234603356162189887035, 347812779833573751189393434155308041091683593091672704988667000799015183029982959984464586867642455333902387974304929273654671439315366613713665784284871]
K[2] = [482202429794814932704016117555552071376748182644231805031740136648816191230691398225928995106403113652009622009272148272229338431152184564997108117791 5687, 241388767530708150100987078009423351420455260950297965842725197980003597625360117220852218978065843166204923075102120432480982200426476380814244650247437 6]
K[3] = [515583522903526530260827412733881558256484545715242421783465482086861528714940521677389720187492880305723803632204667298619245311941675935431964369278736, 708063078389622905888995360021673586089519083659103369190730957756815258510636660156392486223424297814752242422596461805672484380725057878786643327485 04]
K[4] = [143258510327980536268408641243696789417358133466515394225601952563725715166305899711980874774804626284895461069634072388226162752993823559824132210719 6521, 294633477251317098150155312602209176444561968023919007877317668233946184773145342877131625487855175877461136442715693427596785929447750495115690149352 0045]
K[5] = [732657726614951565908412205046271735802083852122386732617780998196335388006799935540317719257804916767804231923251721913013711122380833952949590596741 7247, 394515937917640745751093826487756405809920548548253973698553584247355232697442862878709920766713869646543877433485022819495768266250952003597345623587 6629]
K[6] = [604579854216340772972086431208494081400574588102989529386361027001549722732307543693898725926751323858493102590114601559779556868360194022188664133515 4593, 331633751921119998519215908402351444376446988666753863615969813652131268501427832535244910746949662552810324975708211494617347573965643331990331485164 8193]
K[7] = [608277561808472235476240038503715452872586179009359098769226422053443975239927971342210323842844141762366551646746902766616155358187241696582607822821 2417, 180164191980454859546593103577876773625761513466170396768062908412008837037713398161617013001194799288660152892680837689617089451128875546270772641032 7403]
F[0] = [678468875933791981049673035006104355565412244314505108363455821558208487320191511671298765666168596100830787335345741018221497711760058053045931291892687 6, 228452006771227672913718065528608687853207489364090510139305484443349511519726392828745093616367151957060325486613473057910260878570325200033632445682163]
F[1] = [297301436399772490913983823769770738124190897307051148316560115332700672704370341192191843658222685966970274322335700556023320269826570795894119452462213 0, 717661658447705617224976644020517553149642479174874344115262690142778262924706083713779595599147658301821513274591505790659523015810040957310819521543072495]
F[2] = [722989767886874936145686509310431544583141763537232789669239937937630288295750833859572110713304107126475850773101406136992729720484564887374315730920882 8, 393587988216707407705976693825233531735189745555168836929502387714847551210415053450620116288637023892905333410848159361254590535637447006176706352716803]
F[3] = [189687945125135886147545566924317403122802741460118461791539383117602451298556715218419919544548820999288224699998261731751955389032601582512674920222945 2, 240269570315658159048938751577152185056752111989601787320019495668605419708289836528577975332916143662396965090398270933953964826933811543786626991885186 2]
F[4] = [229047689772368558718596036416890561280880179696493148672794170962370027954877501230208941062288202751758514386861984711721431035613683768774818075248977 3, 449235868078658224827490161110307921427203871482911191757463173103849136316207100879029874576865766306895767863481863306983878300604616710307738633374486 0]
F[5] = [855928102467814772936039120134311179763069072977312715129947597673144546870841769554617781368363294794481569115896473361410405528250376908989702808335026 0, 216474238118103641665124748897176153273592303729240335019392176280043795009110967020946604629830688090844498056679229373565343021616770490547609353496382 3]
F[6] = [256138566008714466387792317570024052446177908336050178331445001855159078450794667087832821582069218652595195217900404500648626664262374209734544059762445 9, 330831035182201396877299747799267511252058526149195145566820585668584786288843132326409202948541521483630446661152095220086227700342840974030561735781427 8]
F[7] = [118970816231079188318087064240951064227783729836120902474437124262955649899937316885423547314848619571261543762372198975570437448114690166717847600589156, 255835137249581123617955492889286801314930512394572025467559657635063754898872074593897257438833037316479723705577390087986828704346698881007137119628448 5]
F[8] = [338164288352540046314346671461805134240955122629017578535461749961400242742492683263674442535789694431878858502023851257731685365520780743681760930768984 7, 283218778466190091401048114255188349005166895883803820823977741188791622836057770551651091162750208709637853486655315901594231399650600515944902633672891 8]
F[9] = [159980401906505632014461627148544176485214689257358317207598828986985865679899133562605522304232231402081927609208057083505844560117120858935388258157785, 376980335021388749355080914251457093877904797863135867459171867288210302358064610173870521405081860295191003464052466817648858971673401134071640550986727 4]
```

one G_1 exponentiation for PK and one exponentiation in G_1 for adding user in the user list.

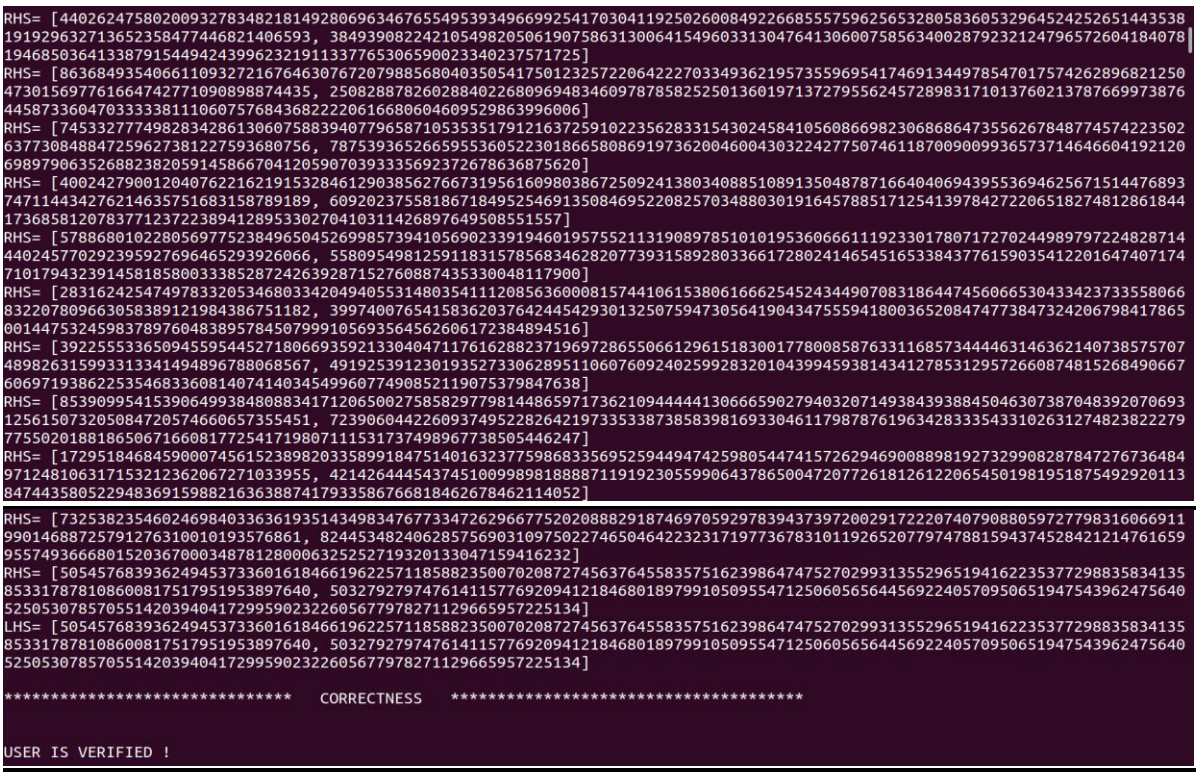
```
Yf_dash = [618881191648533107679252964312013899437975236883413652304574303479513211388477453810092135397362968088103331549210856588732052396567967370062929895761240 3, 806080740476192511238527215084500946581203314036557928962420393276137879022540321556659243560397396768938428496749349935488840479094421330522043389530184 9]
Df_bar = [238462525465098916327298221448197741718737747423206840748621139225131511834790535242717715461560365937301719098118291452851151357022750245045533052527912 9, 598368992311775958531860825686282287616404040306845131506863380183184425575961779049008960596855655606627241219053161598834951629022047259435455733552369 9]
```

During the Secure Index Generation step, almost $(n + 1)$ E + E_1 is needed for generating secure index for the file.


D_tilde = [500452883559519313751439977738974543669073529104956671352667322607863611283806775527898436337055369986316434442852325466799323070646484817131519464064769, 358648613701153932972519828208410685244347707961867566801093724479490492133202167051419250423508494118811139682733934930641331653217275462048137536163683]
D[0] = [861914840030353227832492434896013780945879187992225911700743344270103728626429675911222821675686624174790573628495642452082784081135818060750803037116488 4, 333999758780942414254831936931728367603946795018766993122352097776581597957710367143338855075685285298418895726797591636706822793240622766991965545045812 1]
D[1] = [575817082438641789541253713098441520492501415055725144652225923887250699961685888295185047201297016140580233440086650669665140708912567044657708688814928 8, 642233828540651423623084050561970492810760725143013678477189892267104108880064323923298411865321387471108309237764207035232766805160361658714210795789 1417]
D[2] = [139194727921554826632796935980602157156445493788227644248968690106779906368882226793793473876585524191189890692747933082782924610932649488609998720116258 4, 847204074461949019975278078275923321914489094289492899787991318469600793896513196646515822745351323616006142730183447302673427577568313203903170478042838 7]
D[3] = [428929801529742473270794133784456641358177564914297295465790739672530247899518106028428249529599044476362736557732076476242063376988849752436632346110927 7, 134557284802324086357049433744184746695157372309702250509170224915188046063581431070607996811379126828253679951114720590741407970034383483800466220142139 6]
D[4] = [201256484533790470294576152821060773075612031318593107969492180786201716229431160786741490529499723124868751424909180021615567927932177991705207796209741 2, 624451212564970935515629364254741226916869797880255358764741594425748301602920529340765996351473451472559004087741734373588461424704460873344727379747226 4]
D[5] = [274835838144237276464070744093403840566536709034401163468872805325378114259324046717249129449132097285949895116450874414255039100925843654860013533128832 4, 667961221572056661362668056439135235765344399077645803694953973856447861737656955229011365525988994137776062554435513479843304418493537874411731704991776]
D[6] = [261647172218253553692572485239214550073577264314900391072288804605493703290735721142418931279212879281807511554381815504170216508309277830207920667867047 0, 756995732632695262974166377317447509739663958034213789401228959919262586859812970847569826302133247747497747002958912493594442133800756291938043390435719 3]
D[7] = [277560751149720015160743290157410998524989444989339908497786473487965960444808498996885719734120308196038007229193684314298673908245621903032818977368358 4, 232802392608869936238883913955781849587260838566000795647841084392247934966460532169346024232598633469373654234497592260892242911060849854289679122664670 9]
During the Trapdoor Generation step, almost $2n + 1$ number of exponentiations in G are required when the data user creates trapdoor of any keyword to search.



During the Search operation, a computation of $n + 1$ bilinear pairing is required to search an encrypted index.



Algorithm	Cost for computing
System Setup	$3nE + E_1 + P$
New User Enrollment	$(2n + 1)\ E + 2E_1$
Secure Index Generation	$(n + 1)\ E + E_1$
Trapdoor Generation	$(2n + 1)\ E$
Search	$(n + 2)\ M_1 + (n + 1)\ P + E_1$

Table 4.2 Performance evaluation of ABKS

In our thesis we have described how we explored Stanford University's PBC library, its functioning and its applications in modern cryptosystems.

In DACC as well as in ABKS we expect the cloud to be honest. The cloud must be unable to modify data stored in it. If not, the user must take the responsibility to verify that the data is authenticated. To hide the identity of the owners and users while maintaining authenticity, we can make use of Attribute based signature scheme. ABS has proven to be more secure and authentic than other signature schemes for use in modern cryptosystems. We can also think about hiding the attributes of owners as well as users from the cloud such that only KDCs and respective users and owners can know about their attributes. Moreover, improvements need to be done for hiding access policy from the cloud.

REFERENCES

1. Wikipedia contributors. (2019, August 11). Pairing-based cryptography. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Pairing-based_cryptography&oldid=910316470

2. Ben Lynn. "PBC Library Pairing-Based-Cryptography Library" Retrieved from https://crypto.stanford.edu/pbc/

3. Wikipedia contributors. (2020, June 2). Trapdoor function. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Trapdoor_function&oldid=960263815

4. Wikipedia contributors. (2020, June 3). Elliptic-curve cryptography. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Elliptic-curve_cryptography&oldid=960452508

5. A (Relatively Easy To Understand) Primer On Elliptic Curve Cryptography from https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

6. John Bethancourt, "Intro to Bilinear Maps". From http://www.cs.cmu.edu/~bethenco/bilinear_maps.pdf

7. Wikipedia contributors. (2019, September 24). Boneh–Lynn–Shacham. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Boneh%E2%80%93Lynn%E2%80%93Shacham&oldid=917641358

8. Identity based digital signatures Revision by MigalinAS from http://cryptowiki.net/index.php?title=Identity-based_digital_signatures&oldid=21646

9. Wikipedia contributors. (2019, August 11). Key-agreement protocol. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Key-agreement_protocol&oldid=910390415

10. W. Sun, S. Yu, W. Lou, Y. T. Hou and H. Li, "Protecting your right:Attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud," IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, Toronto, ON, 2014, pp. 226-234, doi: 10.1109/INFOCOM.2014.6847943.

11. S. Ruj, A. Nayak and I. Stojmenovic, "DACC: Distributed Access Control in Clouds," 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, Changsha, 2011, pp. 91-98, doi: 10.1109/TrustCom.2011.15.

12. Joux, Antoine. "A one round protocol for tripartite Diffie–Hellman." International algorithmic number theory symposium. Springer, Berlin, Heidelberg, 2000.

13. Yuan, Q., Li, S.: A New Efficient ID-based Authenticated Key Agreement Protocol. Cryptology ePrint Archive, Report 2005/309 (2005).

14. Zhang, F., Safavi-Naini, R., Susilo, W.: An Efficient Signature Scheme from Bilinear Pairing and its Applications. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC 2004. LNCS, vol. 2947, pp. 277–290. Springer, Heidelberg (2004).

15. Paterson, K.G.: ID-based signatures from pairings on elliptic curves. Electronics Letters 38(18), 1025–1026 (2002).