

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 pairings 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

Fig. 2.1 Illustration of A Trapdoor Function	3
Fig. 2.2 An Elliptic Curve.....	4
Fig. 2.3 Basic idea of Identity based signature	8
Fig. 2.4 Paterson ID based signature flow diagram	8
Fig. 2.5 Zhang-Safavi-Naini-Susilo Signature flow diagram	9
Fig. 2.6 Joux Tripartite Key Agreement Protocol flow diagram	10
Fig. 2.7 Yuan-Li Id based authenticated Key Agreement Algorithm flow diagram	11
Fig. 2.8 Proposed cloud model of DACC	13
Fig. 2.9 Access policy tree for job search example; pad 0 to (1) to change it to (1, 0)	14
Fig. 2.10 R-matrix for the job search example	15
Fig. 2.11 Framework of authorized keyword search over encrypted cloud data.....	16

List of Tables

Table 4.1 Performance evaluation of DACC.....	33
Table 4.2 Performance evaluation of ABKS.....	36

Index

CHAPTER 1: INTRODUCTION.....	1
1.1 Definition	1
1.2 Classification	1
1.3 Use in Cryptography	1
1.4 PBC library	2
CHAPTER 2: LITERATURE SURVEY	3
2.1 Trapdoor Functions.....	3
2.2 Elliptic Curve Cryptography.....	3
2.3 Bilinear Mapping.....	5
2.3.1 Definition of bilinear map	5
2.3.2 Definition of admissible bilinear map.....	6
2.3.3 Properties of Bilinear mapping function	6
2.4 Computational problems	6
2.5 Boneh-Lynn-Shacham (BLS) short signature scheme	7
2.5.1 Key Generation	7
2.5.2 Signature.....	7
2.5.3 Verification.....	7
2.6 Identity Based Signatures	7
2.6.1 Paterson Id Based Signature.....	8
2.6.2 Zhang-Safavi-Naini-Susilo Signature	9
2.7 Key Agreement Algorithms	9
2.7.1 Joux Tripartite Key Agreement Algorithm	10
2.7.2 Yuan-Li Id based authenticated Key Agreement Algorithm	11
2.8 Distributed Access Control in Clouds (DACC)	11
2.8.1 Attribute Based Encryption (ABE)	12
2.8.2 DACC Description	12
2.8.3 DACC Model and Assumptions.....	13
2.8.4 Format of Access Policy in DACC	14
2.8.5 Lewko-Waters ABE Scheme used in DACC	15

2.9	Attribute Based Keyword Search (ABKS).....	15
2.9.1	ABKS Proposal	15
2.9.2	ABKS System Model.....	15
2.9.3	Algorithm Definition of Authorized Keyword Search.....	16
2.9.4	Construction for ABKS	17
CHAPTER 3: IMPLEMENTATION		18
3.1	Pairing Based Cryptography Library	18
3.1.1	Overview	18
3.1.2	Setup and Installation	18
3.2	Implementation of Distributed Access Control in Clouds (DACC).....	19
3.2.1	System Initialization.....	20
3.2.2	Key generation and distribution by KDCs	20
3.2.3	Encryption by Sender	21
3.2.4	Decryption by Receiver.....	23
3.3	Implementation of Attribute Based Keyword Search (ABKS)	24
3.3.1	System Setup.....	24
3.3.2	New User Enrolment	25
3.3.3	Secure Index Generation	27
3.3.4	Trapdoor Generation	29
3.3.5	Search	30
CHAPTER 4: RESULT AND CONCLUSION.....		32
REFERENCES		37

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 discrete-Diffie-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.

If f is a trapdoor function

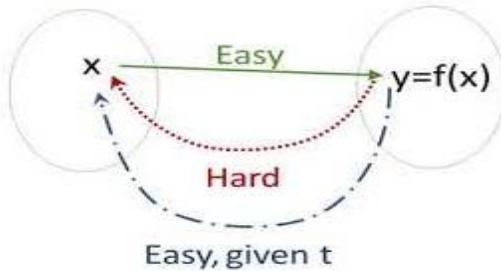


Fig. 2.1 Illustration of A Trapdoor Function

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 factorisation 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]:

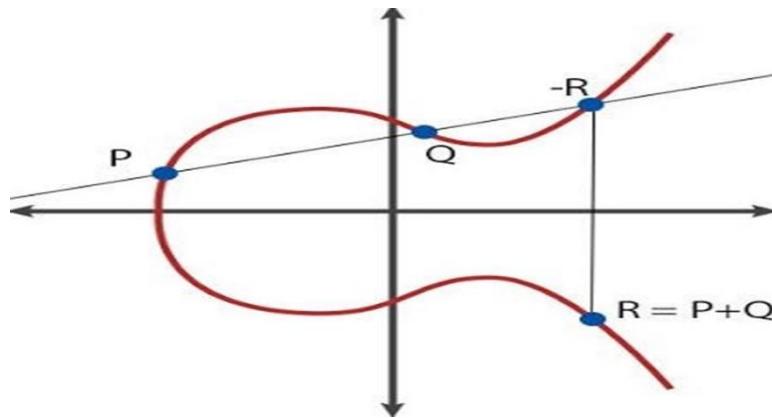


Fig. 2.2 An Elliptic Curve

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:

$$P \text{ dot } P = Q$$

$$P \text{ dot } Q = R$$

$$P \text{ dot } R = S$$

$$P \text{ dot } S = T$$

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 $p, q \in \mathbb{Z}$,

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 $x, y \in \mathbb{Z}_r$.
- 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 $m, n \in \mathbb{Z}_q^*$,

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 $m, n, o \in \mathbb{Z}_q^*$

4. Bilinear Diffie Hellman problem (BDH):

Let P be a generator of G_1 . Then problem which is in $\langle G_1, G_2, e \rangle$ is that given $(P, mP, nP, oP) \in G_1$ for some randomly chosen m, n, o from \mathbb{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.

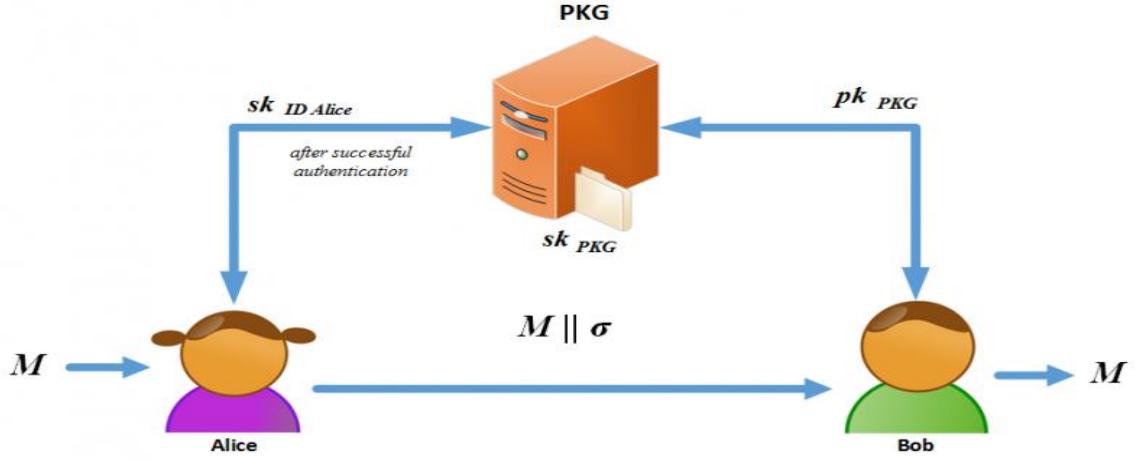


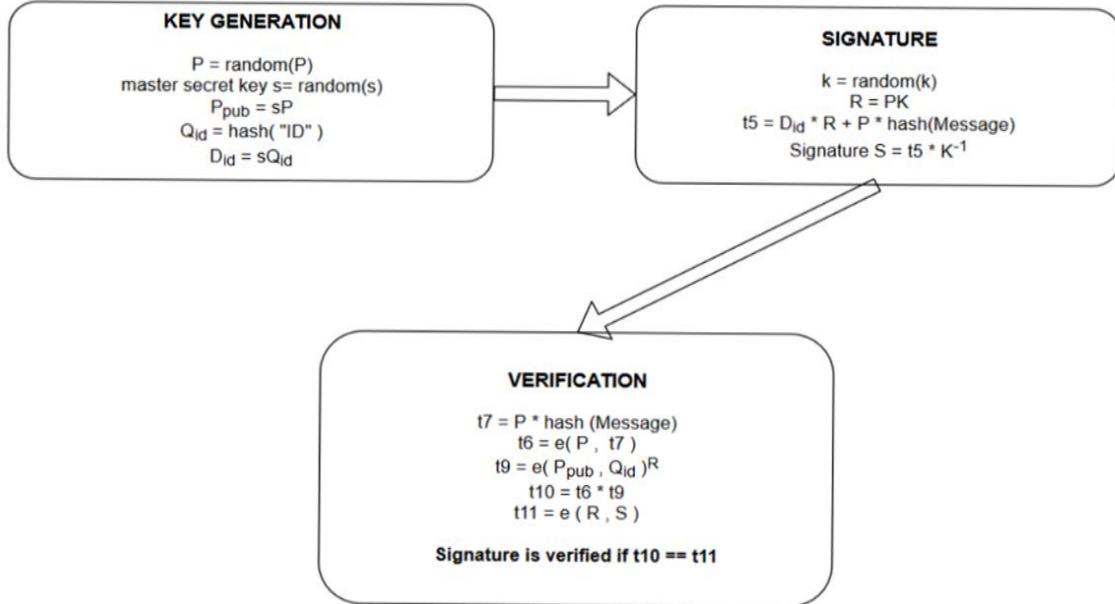
Fig. 2.3 Basic idea of Identity based signature

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:

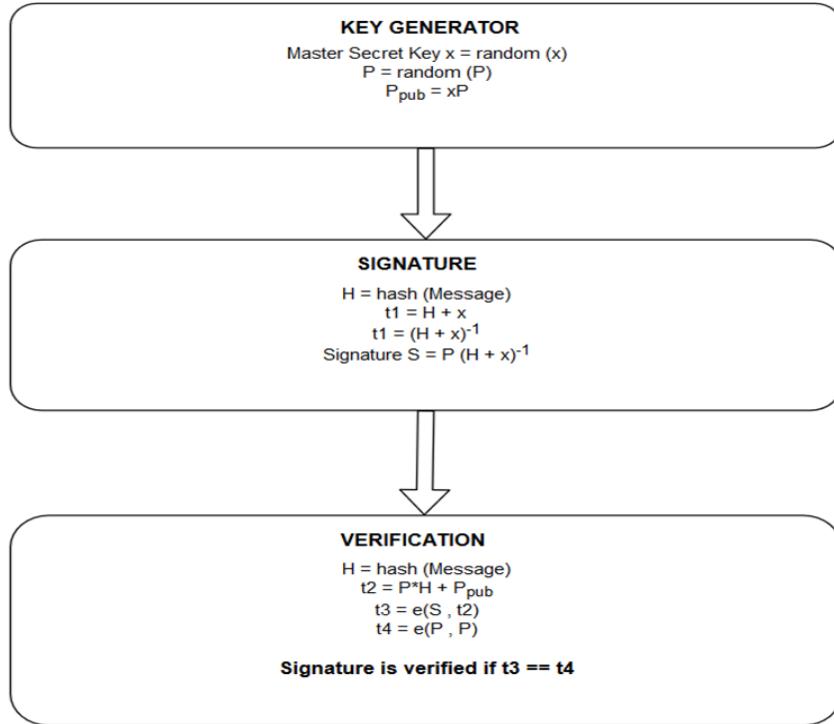
2.6.1 Paterson Id Based Signature



[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

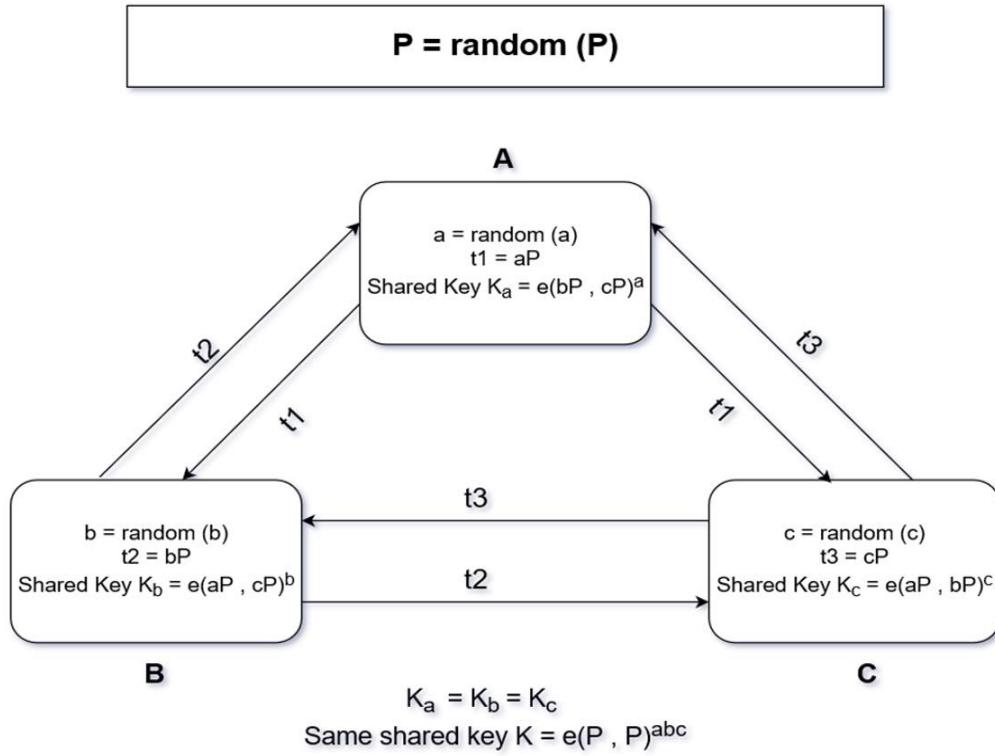


Fig. 2.6 Joux Tripartite Key Agreement Protocol flow diagram

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

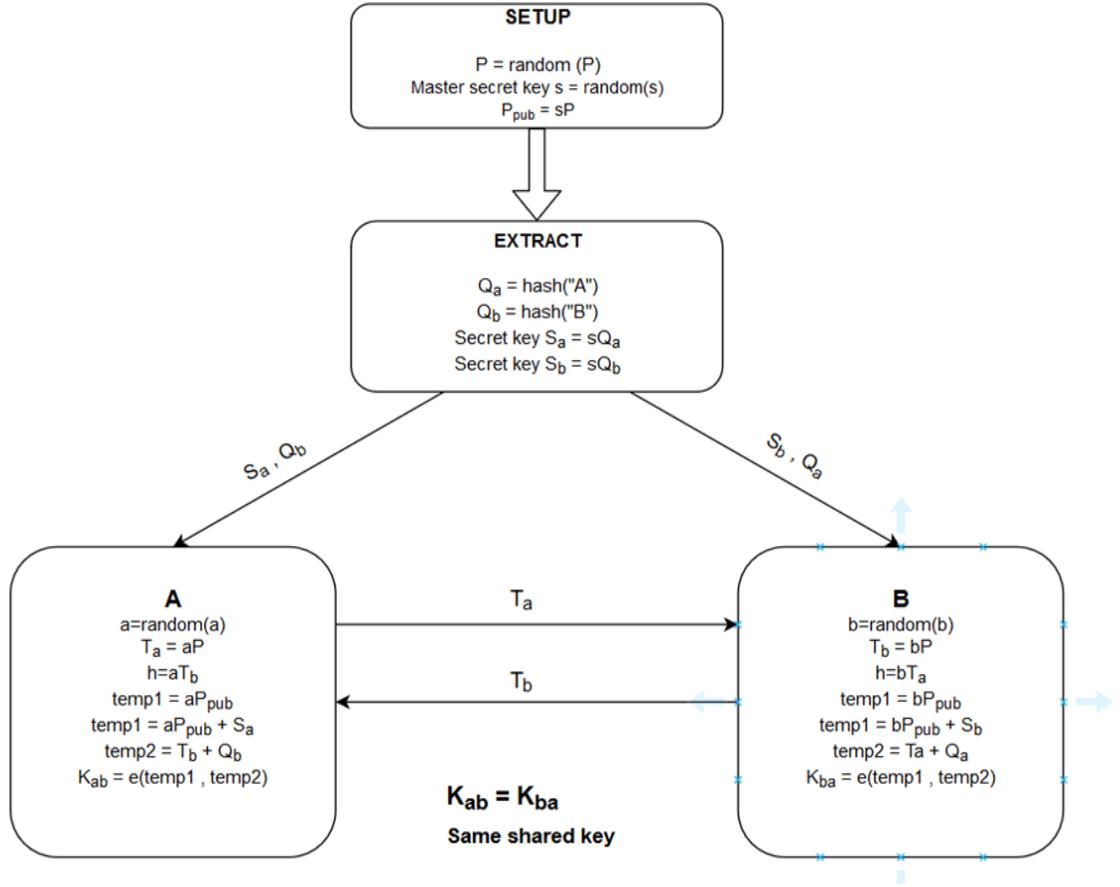


Fig. 2.7 Yuan-Li Id based authenticated Key Agreement Algorithm flow diagram

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 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).

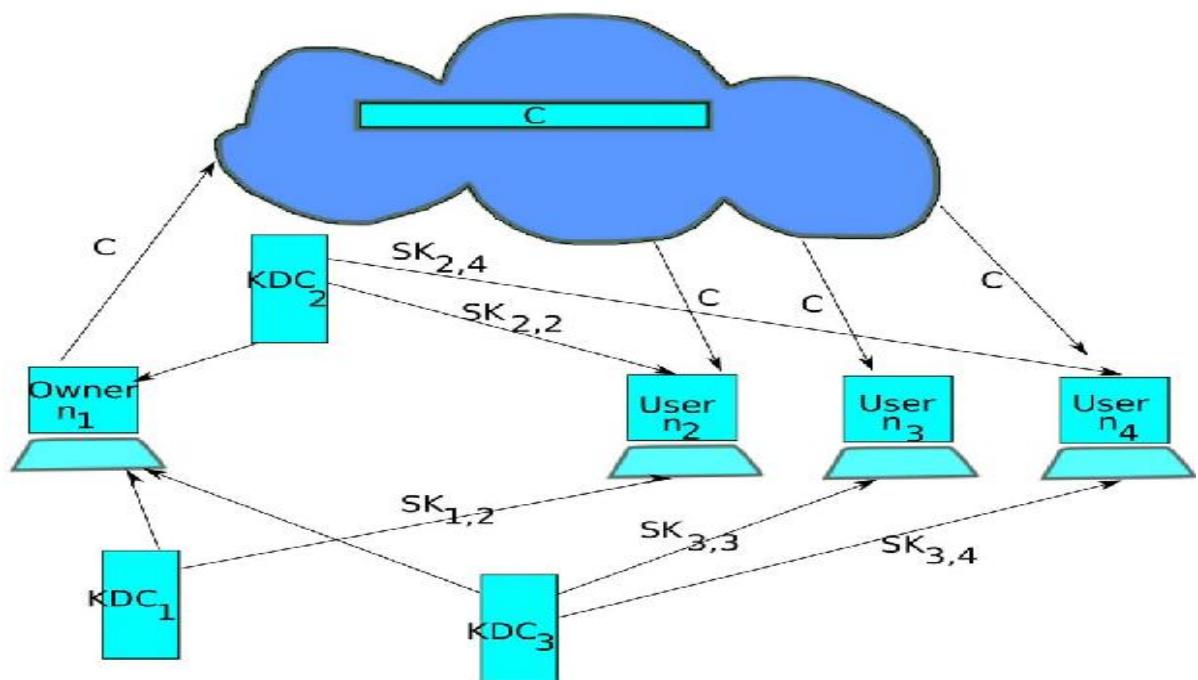


Fig. 2.8 Proposed cloud model of DACC

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 $(x_1 \text{ or } x_2 \text{ and } x_3 \text{ or } x_4 \text{ or } x_5 \text{ and } x_6)$, where x_1, x_2, \dots, 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: $J_1 \text{ or } J_2 \text{ and } P_1 \text{ or } J_3 \text{ and } P_2$.

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

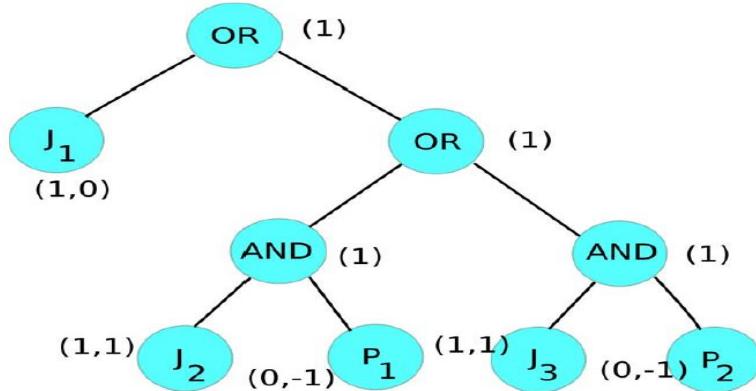


Fig. 2.9 Access policy tree for job search example; pad 0 to (1) to change it to (1, 0)

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, secure 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.

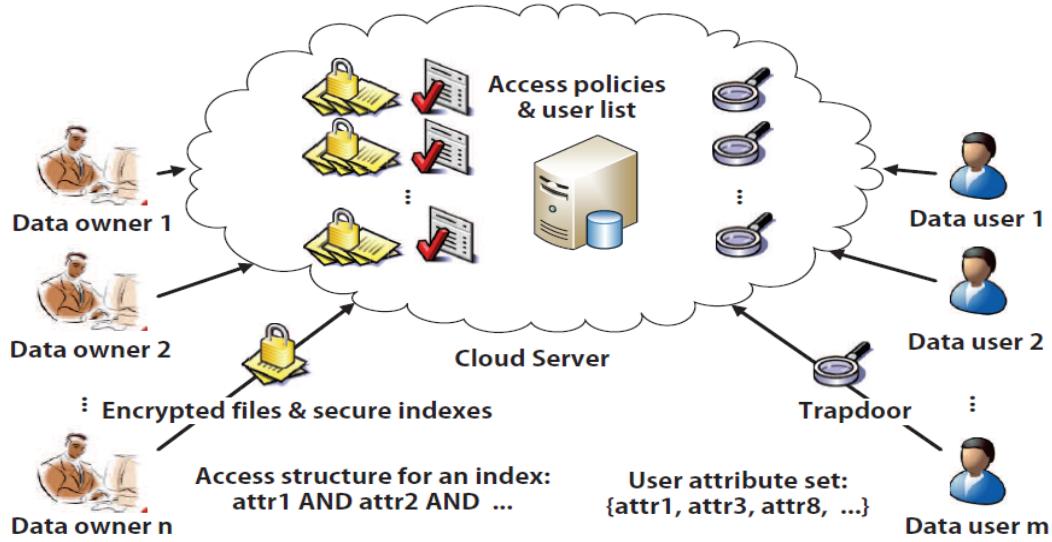


Fig. 2.11 Framework of authorized keyword search over encrypted cloud data

The 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 $\{1, 2, 3, \dots, n\}$

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

Negations $\neg i = \text{not } i$ (*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) $\rightarrow SK$: To generate the secret key of user we use master-key (MK), user attribute set (S) and public parameters (PK).
- GenTrapdoor (SK, w', PK) $\rightarrow Q$: To generate the trapdoor for the keyword $w' \in W$ we use users secret keys (SK) and public parameters (PK).
- Search (D, Q, UL) \rightarrow 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 < Rj; 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:
 $PK[j] = \{g^{y_i}, e(g, g)^{\alpha_i}, i \in L_j\}$.

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 $i \in I[j, u]$

$sk_{i,u} = H(u)^{y_i} g^{\alpha_i}$, 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 < Rj; 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 $\{\pi(x): x \in X\} \cap I_i$ 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 $(1, 0 \dots, 0)$. If such subset is not present then decryption is not possible. If it exists, it calculates constants $c_x \in Z_q$, such that $\sum_{x \in X'} c_x R_x = (1, 0, \dots, 0)$.
3. The Decryption process continues as follows:

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

```

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, \dots, 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 $T_k = g^{t_k} \forall k \in \{1, \dots, 3n\}$ 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]);
        //t_i are in Zr and T_i 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 Gr
```

The public key is PK : = $\langle e, g, Y, T_1, \dots, T_{3n} \rangle$

The master key is MK : = $\langle y, t_1, \dots, t_{3n} \rangle$. 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 $Y_f' = Y^{x_f}$ 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 r_i from Z_p for every $i \in N$, hence $r = \sum_{i=1}^n r_i$.

```

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 $K_i = g^{r_i/t_i}$ for $i \in S$

For other values of i , set $K_i = g^{r_i/t_n+i}$.

```

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 $F_i = g^{r_i/t_2n+i}$.

```
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 $SK = \langle \hat{K}, x_f, ver, \{K_i, F_i\}_{i \in N} \rangle$.

Whenever a new user joins the cloud environment, the data owner executes CeateUL algorithm to set $\tilde{D}_f = Y_f^{-s}$. 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, \tilde{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:

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

```

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

```

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

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

$D_i = T_{n+i}$ if $\underline{i} = \neg i$ and

$D_i = T_{2n+i}$ for each $i \in N \setminus I$

```

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);

```

If 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 $Q := \langle \tilde{Q}, \hat{Q}, ver, \{Q_i, Qf_i\}_{i \in N} \rangle$, 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.

$e(D_{i'}, Q_i)$ is equal to $e(g, g)^{s \cdot u \cdot r'}$ 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:

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

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 Gr
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 $CR = (1, 0, \dots, 0)$, 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.

```
VALUE OF C0
[538703345443196162614246291158829046412844815747972553325696792250034622306757514131699176338836762990077148647999667418404120
2497788691734628171913513736, 326969286024250457318495226134785581677295086661462926514396569545782639433737286500699891754426
950478168926315254435621809515177880942120430097431574398]

VALUE OF C1 FOR ROW : 0
[652336615649503521378874651692986489049828997583293765585972707811243642563866365372506750186648151571744203470693903234783584
5277849090843721514394153616, 5640842894302936450066815060075892784239924989946046663607271040271272830863142170835123605376253771960147
385597346644363232227674832894811625849662699700688999016]
VALUE OF C2 FOR ROW : 0
[258116354874712351347439766917636568301074415758828424596220043018749280755627009629676225280863142170835123605376253771960147
252186453293946689404836135, 863060142080687462383353626384129794794422283820266940434516783274000267034454129956045348941525
28670770762287144586139758240219003321564124686679894277]
VALUE OF C3 FOR ROW : 0
[54957559215085501979476936126480474442179365932879697267397299172977383422973524995795290622405449538019116947260029926710617
5250045363129719956504295362, 52763369909685101380022365972842913758037746419394383945398392279633805223500396858149251788559
10945430915682589738223800022851160874357466638872354915]
VALUE OF C1 FOR ROW : 1
[87299266415298095118995099063791334007921955006685439001689637818375566670611814057080137485787280456680035247324825196163481
2405596006744777902766706569, 71743190387526443013537516638659624305475915862701679081941881294025921358159173392087683469861
419181096866760207883936269862566633851246432274213667472]
VALUE OF C2 FOR ROW : 1
[45402633828621720757739250636058779722981615934279341179213326646802186046879249042520809390254786108812987508368637983788463
68544170258219578596488151501, 273129939244105117671384550999502484385533290560896968593741630269508160729366839655659215654393
276748980455364773117462451879554709007421879835579481735]
VALUE OF C3 FOR ROW : 1
[355499256378759179442945213378803786139494107607349059631241004070447309740097635527647624662161617648746426617868322621597099
△80△5△799824△6△827227615△9△ 1△8△5△794△4△2△1△8△2△7△1△1△8△4△2△1△7△5△6△1△7△5△7△1△2△3△0△1△8△9△7△6△1△1△7△9△2△0△1△7△8△9△4△7△9△5△7△7△6△9△9△]
VALUE OF C1 FOR ROW : 3
[6652076256794567050626288930434674272317981720985337626916636571830972899664173634159309400599648427628752246716692339422945
814541905429212242920880426, 35019753888006658147752271399881313285071233245490087059144130334694748355843642851082512526275
075236331892441431011193840592325289131444405214554121]
VALUE OF C2 FOR ROW : 3
[22189436999283314133696120934398339829154588212370903519085152577385567860852546385840912931627178823934773188997322577726880
7993026602019176414649251347, 223019274855992114889423464141274621861004058345790515123481668791454289595785558296350251664
840915360151001444540611947534614680645007228816799620133]
VALUE OF C3 FOR ROW : 3
[1760568014600636805962150828695316155014106647272701379203059744287449234505275390809826762831154089081662350806786015639729
2186712059534825058389167378, 4250509149943634781351964610787530595671751011031952186098357182244751079800284107298806995245856
602292040009504920708050597988296159777296308786857047504]
VALUE OF C1 FOR ROW : 4
[6831255836395929251246934837830131228105616399062957976353658704330869108257741139948162766145165824882066958328548985437548
010109728071168827592295957, 1642156952934524233081452911102650539990693299595688562299545901292809347495020561531669736664223
68682285832309132325468529812548845571063927377305225865]
VALUE OF C2 FOR ROW : 4
[454956856238410377143556820107563177110766223740471153339463260971708063697940552663837753374266464686489894799658225647585773
9911399919779057424607064086, 65635473927926507827042197661736476755640692070793076140187537879421881471686759137626688678015
861981722059933627271146143983477795356407088103486133178]
VALUE OF C3 FOR ROW : 4
[83454128001088837042142812054038105458099435630303104922639618986030059459508262983955383822611518488354164759468096988055
8777348116075472551335424489, 5526547702938274092643582495043747824430924783149853077374450817048122029925950230205157322053063
93667821074521073925421562892755136244173304834396838283]
Sender sends  $C1[i], C2[2], C3[3], C0$ , R matrix
```

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

```

DEC VALUE: [8050188279278003822706679688350510595860983434904953814956429402449176669461797178757286637681757507499185849519848
377023741791398053451927168328466088078, 658780632234575885391330817676249548825855959983768979769869680052728767784544713963
7365407143286359768569005686474921516864584156819462170236227379181]
DEC VALUE: [1945992260193131671504071054939351225314201954825652709838775878425336029345118667948823961892045601488905417328464
92430976619261503520976653098564449602, 159259759624400598489080665940282800457886914634673307785138390206191908735407599418394
5946516218440197525771659028320430382914089410104561024430581221408]
PRODUCT OF DEC[I]s

[321625171606036432210925296309542788516663429466442407050229904415795088749497427204561525673373509778776293284980469341574349
336240807770300337741374612, 9815248340451807462520950876599842352383064000521217235870407661742676146116885030376569648379459
9606941624166942274821771010887828785003002994528782628]

COMPARISON BETWEEN ORIGINAL MESSAGE AND DECRYPTED MESSAGE

[747597903066869169522822615360572768200827047870750989728335275396765078216696824407012806239871169171594041836792073358407954
3056247265050924555890892092, 5168441766844344790473808783286341487841276586311804082309314713364366739381121585171393623272706
933089370819971870419486649618522704534770783414728766203]
[747597903066869169522822615360572768200827047870750989728335275396765078216696824407012806239871169171594041836792073358407954
43056247265050924555890892092, 5168441766844344790473808783286341487841276586311804082309314713364366739381121585171393623272706
6933089370819971870419486649618522704534770783414728766203]

***** CORRECTNESS *****

MESSAGE IS VALID!

```

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

```

T[0] = [74357513908949683595403824421623674178442138133365908663070460874203995094018377951646947009370523344870977390261804960
53637612040515541920909367439704816, 7579097841631403631319610510770637317371678160060607530779748707300261552977985530053388
9430677019001649674601923884209049966979889093802983936831603073]
T[1] = [48393503361860711857922117575013235828501282041746467061133763734266026916411981974500737040037101924666762566410288972
3467658172980598110051448212357193, 376834299702806773527329929264755521050395723292972378296215415093102872901512455577999512
97583932802876006892705862819240021800021823428841298459100318]
T[2] = [37408582820104985012495238747802754721486494742864871165984334728098944246183401702954550433411494658811788448256658679
40389827263909778589626446250473858, 20106338522613271727149134347987084780471608985882107905438075991365907861031752842089362
1568677011471634106113416987403008186195728812686214702304001790]
T[3] = [50232144511042076758096665942263407974748629700837078429071242358500890558568200565362407273667534402177113325799329626
32355864760545520122209516333885771, 866727483313185653854077384825040449876484231937394132144300146334016544642581352639658695
662730497613721587042649897631201205182797106163249535875726992]
T[4] = [5466476190506130747153860188621915894961457428177294642453674245348945492738032356056176199403559971825587724176018769
39252074246866282813462037846629551, 309773992443662187103867919901920879874795140355445084012027275489737157764033578543983992
911031032020780927844421261051595764966572400122180243402419452]

```

and one in G_1 and one bilinear pairing.

```

Y = [2318203054704336986647486689911921575345752691187144278415610963448106411239351020567677801837801593942997159141699298163
4742082506793126292553368407150, 6634923497462923248314600376842776023072872184511609079041761554732663482633886496720756617851
52821377859036757196724105622617713448269930750429496186250]

```

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

```

K_hat = [6832523360727868198946840582144953547558559301577648461809426766371854575618831536277160605091967111411846847228499090
760078246411296794164114461676131991, 51975668390497983686947206126462389558784201697391356676957836932742513183236963304888933
7644088787342686389141149330886849792603210684174784334687036288]
K[0] = [21905349852946538304278914273920682703676620309039924594739174918703919938349949808656827938288971169251748019317509574
56313801346528860895469126884172641, 158250184607388353359438711776153505894345897722398012980045849733759084379791539890530679
79955019858990800575001305548856634568199242382198112986338940]
K[1] = [635728453929241110346443652154120243832445538846283482336798940281567328632598445665901432344248263796517820399712247
49535795417592346303356162189887035, 347812779833573751189393434155308041091683593309167270498866700079901518302998295998446458
6867642455333902387974304929273654671439315366613713665784284871]
K[2] = [4820242479748149327040161175555702137674818264423180503174013664881619123069139822592899510640311365200962200927214827
22293884311521845649971081175687, 241388767503070150819708000394235142045526095029796584275197980003597625360117220852218
9786658431662049230751021204324809822004264763808142446502474376]
K[3] = [51558352290352653026082741273388155825648454571524242178346548208686152871494052167738972018749288030572380363220466729
861924531194167593431964369278736, 708036078389622905888953600216735860895190836591033691907309577568152585106366601563924862
23424297814752242259646180562748438072505787866433274863704]
K[4] = [1432585103279805362684086412436978941735813346651539422560195256372571516630589971198087477480462628489546106963407238
82261627529938235598241322107196521, 29463347725131709815015531260220917644561968023919007877317668233946184773145342877131625
48785517587746115694271596347759678592947750844715156901493520045]
K[5] = [7326577266149515659084122050462717358028385212238673261778099819633538800679993554031771925780491676780423192325172191
3013711122380833952949590567417247, 39451593791764074575109382648775640580992054854825397369855358424735523269744286287809920
766713869646543877433485022194957682662509520035973456235876629]
K[6] = [604579854216340772972086431208498140057458810298952938636102700154972273203754369389872592675132385849310259011460155
97795568683601940221886641335154593, 31363375192111998519215968040235144437644698866753863615969813652131268501427832535244910
7469496625528103249757082114946173475739656433319903314851648193]
K[7] = [60827756180847223547624003850371545287258617900935969876922642205344397523992797134221032384284414176236655164674690276
661615358187241696582607282212417, 18016419198045493591363577876773625761513466170396768062908412008837037713398161617013
001194799288660152892680873689451128875546270772641032743]
F[0] = [67846887593379198104967303500610435556541224431450510836345582155820848732019151167129876566616859610083078733534574101
82214977117600580530459312918926876, 228452006771227672913718065528608687853207489364090510139305484443349511519726392828745093
6163671519570603254866134730579102687857030250003632445682163]
F[1] = [2973014363997724909139838237697707381241908973075114831656011533270067270437034119219184365822268596697027432233570055
60233202698265707958941194524622130, 71766165844770561722497664402051755314964247917487434115262690142778262947060837137795959
914765830182151327459150579065952301580040957310819521543072495]
F[2] = [72298767888674938614568659031403154583147635323728966923993793763028829575083385957211071330410712647585077310140613
6992792704845648873743157309208828, 393587988216707400770597669382523353173518974555516883692950238771484755121041503450620116
28863702838929053140848159361254590535637447006176706352716803]
F[3] = [189687945125135886147545569231740312280724146011846479175393831176024512985567152184199195445488209992882246999826173
17519553890326015825126749202229452, 2402695703156581590489387515771521850567521119860178732019495668605419708289836528577975
3329161436323969659098444980566792937235653430216167704905476093534963823]
F[4] = [22994768977236855871859603641689056128088017969649314867279417096237002795487750123020894106228820275175851438686198471
17214310356136837687748180752489773, 449235868078658224827490161110307921427203871482911191757463173103849136316207100879029874
5768657663068957678634818633098388306064617103077836333744860]
F[5] = [8559281024678147729360391201343117976306907297731271512994759767314454687084176955461778136836329479448156911589647336
14104055282503769089897028083350260, 216474238118103641665124748897176153273592303729240335019392176280043795009110967020946604
6298360880908444980566792937235653430216167704905476093534963823]
F[6] = [256138566008714466387723175700240524461779083365017833344500185515907845079466708783282158206921865259519521790040450
06486266642623742097345440597624459, 3308310351822013968772997477926751125205856214919514556682058566858478628884313236409202
948541521463830446661502595220086277003428409740305617357814278]
F[7] = [1189708162310791883180870642409510642277837298312099024744371242629556498999373168854235473148461957126154376237219897
5570437448114690166717847600589156, 255835137249581236179554928892868013149305123945720254675596576350637548988720745938972574
38833037131647972370557390087986828704346698881007131196284485]
F[8] = [338164283525400463143466714618051342095512269617578534617499614002427424926832636744253578969443187885850202385125
7731685365520780743681760930768947, 2832187784661900914010481142551883490051668958838038208239774118879162283605770551651091
16275983307131647972370557390087986828704346698881007131196284485]
F[9] = [1599804190650563201446162714854174685212469257358317207598828986985865789891335626052230423222314020819276902080570
83505844560117120858935388258157785, 376980335021388749355080914251457093877904797863135867459171867288210302358064610173870521
4050818602951910034640524668176488589716734011340716405509867274]

```

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

```

Yf_dash = [6188811916485310767925296431201389943797523688341365230457430347951321138847745381009213539736296808810333154921085
65887320523965679673700629298957612403, 806080740476192511238527215084500946581203314036557928962420393276137879022540321556659
2435603973967689384284967493499354888404790944213305220433895301849]
Df_bar = [238462525465098916327298221448197741718737747423206840748621139225131511834790535242717715461560365937301719098118291
452851151357022750245055330525279129, 598368992311775958318608256862822876164040403068451315068633801831844255759617790490089
605968556556066272412190531615988349516290220472594354557335523699]

```

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

```
D_tilde = [5004528835595193137514399773897454366907352910495667135266732260786336112838067755278984363370553699863164344428523
25466799323070646484817131519464064769, 3586486137011539329725198280841068524434770796186756680109372447940942133202167051419
2504235084941188111396827339349306413316532172754620481373536163683]
D[0] = [8619148400305352278324924348960137809458791879922591170074334427010372862642967591122282167568662417479057362849564245
20827840811358180607508030371164884, 33399758780942412548319369312235209777658159757710367143338855
07568528529848188957267975916367068227932406227669919655450458121]
D[1] = [5758170824386178954125371309844152049250141505572514465222592388725069996168588829518504720129701614058023344008665066
96651407089125677086888149288, 642233828540651423623084050561970492810760725143013678477189892267104108880064323923298411
8653213874711083092377642070352327668051603616587142107957891417]
D[2] = [1391947279215548266327969359806021571564454937882276442489686901067799063688822679379347387658552419118989069274793308
27829246109326494886099987201162584, 847204074461949019975278078275923321914489094289492899787991318469600793896513196646515822
74535132361600614273013434732757756831320390310740780428387]
D[3] = [4289298015297427327079413378445664135817564914297295657907396725302478995181060284282495295990447636273655773207647
62420633769888497524366323461109277, 13455728480232408635704943374418474669515737230970225059170224915188046063581431070607996
8113791268282536799511147205907414079700343834838004662201421396]
D[4] = [2012564845337904702945761528210607307561203131859310796949218078620171622943116078674149052949972312486875142490918002
16155679279321779917052077962097412, 62445121256497093551692364254741226916869797880255358764741594425748301602920529340765996
3514734514725590046877417343735884614247044608733447273797472264]
D[5] = [27483583814423727646407074409340384056653670903440116346887280532537811425932404671724912944913209728594989511645087441,
42550391009258436548600135331288324, 667961221572056661362668056439135235765344399077645803694953973856447861737656955229011365
52598899413777602554455134798433041849353787441173104991776]
D[6] = [26164717221825355369257248523921455007357726431940039187228880460549370329073572114241893127921287928180751155438181550
41762165083092778302079206678670470, 756995732632695262974166377317447509739663958034213789401228959919262586859812970847569826
3021332477474977470029589124935944421338007562919380433904357189]
D[7] = [277560751149720015160743290105741098524988444989339908497864734879659604448084989688571973412030819603800722919368431
42986739082456219030328189773683584, 2328023926808993623888913955781849587260838566000795647841084392247934966460532169346024
2325986334693736542344975922608948542896791226646709]
```

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.

```
O_hat= [12797695289226316913697498456988946290961747178368681315888716982092173086210576343824373641914191507358377288908892856
5557086904404948306889195458954573, 433122410115203067091249636749410725708014288493631912842684139465469187102367861190046916
2033864152187879271312338847384111126097026879397199691779038618]
Q[0]= [67781352528314178429707290386395468255490599275645159624613222372567624887059839366136778535807752652018794795335025058
644230184153061430069307531444922, 8363169946385758565887554813855767652432413860430476374071110418422566611247536500719200463
2863828838262989542811632535472033915567955020059930189490203]
Of[0]= [238737567434988069156186513679416900706958258751995608929383108145045263252524732936747454226405755555168618992502928
3979621582471257415440864002287877, 39989024922378989461879492797788357485242041587463221726333367687778042140858043832288802
769626480070148112562577836419761864354317134913683284579767808]
Q[1]= [304230694408666914208075038780892654580526154797717329551693330677942233740878259142720119457851158819696350646520332963
9710231744357083282266638798173271, 70899042806408484849326495967162407372735294323081780312319736525467711641995121466399603
6376570936276792040896442545773431688983269926694412889371982]
Qf[1]= [38670905266637388680362107546124613118399436716490299939739150666835351512502604611578277836935138436403110594815493999
089060607289848212450429203138821428, 40924427720762710062300806534534210769642855656945483761841440809003866651845927388751042
839329509451906915722571427568610618980443681888623020380985261]
Q[2]= [2460967468954361312493672009218420845689027882006495037948582623504942142265862572931305737065557852937907079743445
0609417644124874873613055940403921, 6289464115375221229530461745569302577435515762061560882310423203640869973658597642564788420
757504018558262722135293235528110023673811854924777981477544]
Qf[2]= [24318158486177081932705997523947741318858548611684654986325454270017028046587554501932299383683321844936611600877857
15931353043329799133363814016079850, 6053291564557095505101613169590308906964435371942749553244687260024481435310281088098021
95931084161104885626432128564433380696269946148690076540466154]
Q[3]= [5018409234710584313241509960970671194299530375934237998356987486073250964915726551637027389823697197340447342634774640
3086392364678094475882174494308497, 274828107053198959433928788583022728407795487494779450932071092413872553188768227797403779
7846094130194530796023018227657868004181761874059189891089091]
Qf[3]= [29195503016943403638366907006528550045348744820013653549618807059573474674364359626440312687713503100462526736302408466
47261817232802559076270792185935220, 9621322001900380697322006955419710202297689434592759407002066737611105276902953406990790]
```

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

```

RHS= [4402624758020093278348218149280696346765549539349669925417030411925026008492266855575962565328058360532964524252651443538
191929632713652358477446821406593, 38493908224210549820506190758631300641549603313047641306007585634002879232124796572604184078
1946850364133879154494239962321911337765306590023340237571725]
RHS= [863684935406611093272167646307672079856804035054175012325722064222703349362195735596954174691344978547017574262896821250
4730156977616647427711090898874435, 25082887826028840226809694834609787858252501360197137279556245728983171013760213787669973876
445873304703333811106075768436822220616680604609529863996006]
RHS= [7453327774982834286130607588394077965871053353179121637259102235628331543024584105608669823068686473556267848774574223502
637730848847259627381227593680756, 78753936526659553605223018665808691973620046004303224277507461187009009936573714646604192120
69897906352688238205914586670412059070393335692372678636875620]
RHS= [400242790012640762216219153284612903856276673195160980386725092413803408851089135048787166404069439553694625671514476893
747114434276214635751683158789189, 60920237558186718495254691350846952208257034880301916457885171254139784272206518274812861844
173685812078377123722389412895330270410311426897649508551557]
RHS= [5788680102280569775238496504526998573941056902339194601957552113190897851010195360666111923301780717270244989797224828714
4402457702935927696465293926066, 558095498125911831578568346282773931589280336617280241465451653384377615903512201647407174
71017943239145818580033385287242639287152760887435330048117900]
RHS= [283162425474978332053468033420494055314803541112085636000815744106153806166625452434907083186447456066530433423733558066
832207808847259627381227593680756, 399740048395784507999105693564562606172384894516]
RHS= [3922555336509455954452718066935921330404711761628823719697286550661296151830017780085876331168573444463146362140738575707
4898263159931331494896788068567, 49192539120919352733062895110667609240259928320104399459381434127853129572660874815268490667
60697193862253546833608140741403454996077490852119075379847638]
RHS= [8539099541539064993848088341712065002758582977981448659717136210944441306665902794032071493843938845046307387048392070693
1256150732050847205466057355451, 72390604226093749522864219733533873858398169330461179878761963428335433102631274823822279
77550201881865067166081772541719807111531737498967738505446247]
RHS= [1729518468459006475612362067271033955, 421426444543745100998981888711919230559906437865004720772618126122065450198195187549290113
8474435805229483691598216363887417933586766818462678462114652]
***** CORRECTNESS *****

USER IS VERIFIED !

```

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).

