NTNU – Trondheim
Norwegian University of
Science and Technology

Functional Key Exchange In A Distributed Environment

Anders Kofoed

Submission date: December 2014
Responsible professor: Colin Boyd, ITEM
Supervisor: Colin Boyd, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Functional Key Exchange In A Distributed Environment
Student: Anders Kofoed

Problem description:

Functional encryption is a new generalization of public key cryptography which allows very flexible access control to encrypted data. It is natural to consider extensions of the functional idea to other cryptographic primitives. One such primitive is key exchange. Since the scheme provides a much more dynamic and flexible way of exchanging secrets, it is natural to consider systems where users join and leave dynamically - such as chat rooms, Internet forums or similar distributed systems.

This project will explain functional key exchange and compare the technique to traditional group key exchange mechanisms, considering both efficiency and security properties. It will contain a prototype implementation of an attribute based authenticated key exchange scheme in a distributed environment, based on existing work done on attribute based key exchange - using the Charm framework and Python. Different application areas for such a system will be explored and problems arising discussed.

Responsible professor: Colin Boyd, ITEM
Supervisor: Colin Boyd, ITEM

Abstract

Write abstract

Preface

write abstract

Contents

List of Figures

Acronyms

AB-AKE attribute-based authenticated key exchange.

ABE attribute-based encryption.

AKE authenticated key exchange.

BDDH bilinear decisional Diffie-Hellman problem.

CA certificate authority.

CDH computational Diffie-Hellman problem.

CP-ABE ciphertext-policy attribute-based encryption.

DDH decisional Diffie-Hellman problem.

DEM data encapsulation mechanism.

EP-AB-KEM encapsulation policy attribute-based key encapsulation mechanism.

IB-AKE identity-based authenticated key exchange.

IBE identity-based encryption.

KEM key encapsulation mechanism.

KMS key management service.

LSSS linear secret-sharing scheme.

NTNU Norwegian University of Science and Technology.

PBKE predicate-based key exchange.

PKI public key infrastructure.

Chapter 1

Introduction

1.1 Motivation

In distributed systems users might not want to publish their identity if it is not absolutely necessary, which in most cases it is not. It is more important that the user is legit and have the correct privileges. If we could assure that all users participating in some protocol or application had the right privileges or simply the correct purpose, we could go without knowing the exact identities. Lets say you want to exchange keys and communicate securely with a group of people with the same interests as yourself, but you do not want to reveal your identity for some reason. To keep the discussion serious you would not want extraneous users to interrupt. Since nobody wants to authenticate themself using identities we need to make a decision based on something else - attributes identifying the characteristics of each user.

This can be achieved using attributes in the encryption or key exchange mechanisms. Attributes can be anything - personal attributes as birth year, gender or nationality, or affiliation to user groups providing access control, where only the group members would be allowed to communicate. Or attributes could be related to certifications or titles, this way you can be assured that the person you are communicating with has knowledge about something you need help with, without knowing the identity. This generalization can also be extended to include systems where the identity is one of, or the only, attribute. Thus achieving identity based systems without the need to generate, distribute and store huge amounts of public keys.

This project describe the term functional key exchange, covering key exchange mechanisms using the mentioned ideas to achieve dynamic group key exchange in a functional manner. Not exposing identities if not absolutely necessary is the main focus when discussing possible applications of these functional key exchange methods. There already exist several proposed schemes for both attribute- and identity-based key exchange, but there are few or none examples of systems applying the schemes in real applications. The goal of this project is to see how these schemes work in a real application environment.

1.2 Related work

The work done by Gorantla et al.[9] has a similar construction as the implementation showed in this project, they presented a generic one-round attribute-based authenticated key exchange (AB-AKE) protocol using a more complex structure, than the plain KEM showed here. Gorantle et al. [10] has shown that key exchange can be done using KEMs, which will be the approach used in this project as well. The constructions of identity-based encryption (IBE) and ABE by Waters [22, 23] are used by Charm[1] to implement these in python. The project use this ABE implementation as basis for the application demonstrated.

1.3 Scope and objectives

The project will showcase functional encryption schemes, showing how these look when implemented, and present ideas on how these can be used to provide key exchange in various applications. The focus will be on applying an attribute-based key exchange scheme in a real application, based on a construction of attribute-based encryption from the Charm framework [1]. This implementation will be extended to a functional key exchange mechanism using a hybrid encryption variant with ABE as the key encapsulation mechanism and then using the keys from the encapsulations to compute session keys.

1.4 Limitations

The project will not try to solve problems related to key distribution and how to deploy a secure and trustworthy key management service. For the proposed system it will be assumed that this kind of service exists.

1.5 Method

The methods used in this project consist of using already implemented schemes from the Charm framework, showing how these work and compare the code to the definitions from the papers from which the implementations are based. This should give insight into how the schemes work as well as how Charm looks. Based on the designs and implementations discussed, a working prototype of an application taking advantage of these will be implemented to show how modern functional key exchange schemes can be utilized in real applications. While researching the different schemes, strengths and weaknesses will be discussed, as well as arguing the decisions made during design and implementation.

1.6 Outline

The background chapter will describe all the components that are essential to both functional encryption and key exchange, from basic public key encryption to functional encryption schemes, including secret sharing, pairing-based encryption and key encapsulation. The functional encryption algorithms will be presented and discussed using code from Charm, this is logical since one of these schemes is basis for the system implemented in this project.

In the next chapter some functional key exchange schemes will be described and possible application areas discussed. Finally a prototype system will be implemented and presented, using attribute-based encryption and key encapsulation. The system will be a distributed chat using attribute-based key exchange to achieve secure communication with users joining and leaving dynamically.

Chapter 2
Background

This chapter will present and discuss principles and schemes from which the project later develops. Charm is the framework used to implement the schemes, this will be described to some extent first, before continuing on to important principles and constructions which are used in the key exchanges mechanisms later.

2.1 Charm

Charm [1] is a prototyping framework for cryptographic systems. It includes all the tools needed to implement most crypto systems, as well as keeping a collection of reusable code. The idea is to make it sufficiently easy to prototype systems which earlier only existed in research papers without actual implementations. It is of course possible to implement all schemes using other, lower level design methods, which may make for better and more efficient implementations. Most of the tools needed may be available but to combine and make good use of these is not easy.

Charm focuses on being readable and efficient to use. Several existing libraries are used to provide primitives needed, such as pairing groups and modular arithmetic. The design and implementation of Charm is described in detail in "Charm: a framework for rapidly prototyping cryptosystems"[1]. Implementations from Charm will be used in this project to implement other protocols. Schemes from the Charm library of implemented schemes will be used as examples to describe protocols in the background chapter.

2.2 Linear Secret Sharing

In many cases it may be desirable to use more than one key when encrypting, and later require a subset of these when decrypting. This concept is called secret sharing or secret splitting - a secret is "divided" into n pieces which then is distributed. To recover the message k of these n pieces needs to be present. Secret sharing is often used to store highly sensitive keys, typically private keys of a certificate authority (CA) or other keys which should not be assessible by a single user alone. For the cause of this project only linear secret-sharing scheme (LSSS) is described as this is a key component in many of the constructions used later.

A linear secret sharing scheme [5] , defined by (k, n), where n are the number of shares and k the threshold to allow recovery, are defined as following.

- **Setup** - A is a public $k \times n$ matrix with entries chosen from $F = \mathbb{Z}_m^*$ with generator m, a prime. $x = (x_1, x_2, \ldots, x_t)^K$ is a secret vector from F^k. a_{ij} is element at the ith row in the jth column of the matrix A.
- **Dealing phase** - A secret vector $x \in F^k$ where x_1 is the secret value, while the rest of the values are random from F. Each user get a share $y_i \in F$.
$$y_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ik}x_k$$
For the (k, n) threshold system there will be n such schemes and thus a $(n \times k)$ linear system $Ax = y$. The shares y_i are sent to the users, while A are made public.
- **Recovery phase** - Consist of solving a linear system of equations made of combining the users equations forming a matrix A_U where $U = u_1, \ldots, u_k$ is the users participating in the recovery. The solution is then found by solving
$$A_U x = y_U \quad ,$$
where y_U is the vector of secrets of the users. The secret is found as the first coordinate of the solution.

2.2.1 Shamir Secret Sharing

Shamir's secret sharing scheme [18] was one of the first implementations of secret sharing, and acts as an example on how these schemes may look. The main idea behind the scheme is that given k points in a plane $(x_1, y_1), ...(x_k, y_k)$ there is only one polynomial $y(x)$ of degree $k - i$ such that $y(x_i) = y_i$ for all i. We can thus chose a prime p, larger than the desired number of shares, and a random number $a < p$. Then equation 2.2.1 is the polynomial used, with the secret m. The shares are $(x, y(x), p), x = \{1, 2, ..., n\}$ of which only k is needed to recover m. With k shares we obtain a set of k equations with k unknown which is solved unambiguously.

$$y(x) = m + a_1 * x + a_2 * x^2 + ... + a_{k-1} * x^{k-1} \tag{2.1}$$

2.3 Public Key Encryption

Public key cryptography or asymmetric cryptography, as describe by ElGamal [8], allows encryption of messages without the parties sharing a secret key. Each user have a pair of keys, one public and one private key. The private key is used to decrypt and sign messages for authentication, while the public key - known publicly - is used to encrypt messages.

- **Key generation** - takes the security parameters, depending on the implementation, and outputs a public/private key pair.
- **Encryption** - takes the public key of the receiver as input together with the message, outputs a cipher text. Only the corresponding private key can decrypt.
- **Decryption** - uses the private key to decrypt the message.

This setup can also be used to achieve message authentication using digital signatures. Either by the use of the same algorithms as mentioned above or with separate ones alongside these. A common problem with public key crypto systems is how you are supposed to trust that a given public key used to sign something really belongs to the claimed owner. This issue is normally dealt with using a trusted CA issuing certificates binding a public key to a user identity. The CA has a public/private key pair, which is used to sign certificates on request. Initially the CA have to publish its public key together with a CA certificate. This is stored in the browsers of users trusting it. A user will typically send a signed certificate request to the CA, including the identity to be associated with the public key. The CA verifies the request signature and produces a certificate which is signed using the CAs own public key. When signing, the user will include this certificate as profe of his identity, which can be verified by anybody in possession of the CAs public key and CA certificate.

2.4 Hybrid Public key Encryption

A hybrid encryption scheme [16] consists of a public key encryption technique and a symmetric key encryption technique, from which the former, KEM, is used to encrypt some key K, and the latter, data encapsulation mechanism (DEM), encrypts the data. This setup can be applied using a variety of different cryptographic systems for both the KEM and DEM.

Pgp/Gpg [12, 14] is an extension to the classic public key scheme, combining the speed of symmetric key cryptography with the dynamic nature of public key systems. This is done by generating a random session key which is used to encrypt a message, this key is then encrypted using the public keys of each recipients and concatenated together with the cipher text. Pgp uses a hierarchy of trusted CAs as described in 2.3, but also what is called a "web of trust" where users can sign the public keys of eachother to assure authenticity. This way users build a net of users verifying their identity in addition to the trusted CAs.

2.4.1 Key encapsulation mechanism

KEM [17] is a technique where a random key K is generated together with its encryption C - the encapsulation. This is similar to public key encryption, except that the encryption function does not take a message as input, but only a public key. A random key is then generate, encrypted and returned from the encapsulation function. This is useful for distribution of symmetric keys that can be used again to generate session keys for two or several parties, depending on the encapsulation mechanism. A KEM consists of three algorithms:

- **Key generation** - generation of the symmetric key used by the DEM.
- **Encryption** - used to encrypt the generated key, usually using some public key.
- **Decryption** - reveals the symmetric key from the encapsulation.

Multiple KEM

Smart [19] proposed to extend the notion of KEMs to include multiple parties, he named such constructions mKEMs. Encapsulating a random generated key for multiple parties with only one encapsulkation. In other words generating one single encapsulation which several users can decrypt with their private key, retriving the symetric key.

ElGamal Key Encapsulation [13]

ElGamal encryption is a public key encryption scheme as described in 2.3. This construction can be changed slightly to become a simple KEM. Next ElGamal KEM is presented as an example of a KEM.

Key generation
: Let q be a prime and G a group of prime order q. Chose two randoms $g \in G$ and $x \in \mathbb{Z}_q$. The public key $pk = g^x$ is the public key, this is published together with $qandG$. The private secret key is $sk = x$

Encryption
: A random $r \in \mathbb{Z}_q$ is chosen. The cipher text is then $C = g^r \in G$ and the key is $k = pk^r \in G$. We say that C now encapsulates k, and C.

Decryption
: Computes k from C as $k = C^x \in G$. This is correct since $k = pk^r = g^{xr} = C^x$ and $C = g^r$.

2.5 Pairing-based encryption

Let G be a group of prime order q with generator g. Chose $x, z, y \in \mathbb{Z}_q$. The *Discrete-logarithm* problem says that it is hard to obtain x from g^x and g. Further we have the *computational Diffie-Hellman problem (CDH)* which extends this saying that it is hard to calculate g^{xy} from the tuple (g, g^x, g^y). Finally we can say that it is hard to determine if $z = xy$ given (g, g^x, g^y, g^z) - *decisional Diffie-Hellman problem (DDH)*. These assumptions are used, and relied on, in earlier crypto systems. The systems explored in this project use pairing of the traditional group assumptions. The idea behind this is to use a mapping between two cryptographic groups which allows the creation of new schemes based on the reduction of one of the problems from earlier. The most renowned pairing-based construction is the *bilinear map.* G_1 and G_2 are groups of prime order q. If $e : G_1 \times G_1 \rightarrow G_2$ then the mapping e should have the following properties to be useful:

- Bilinearity - for all $h, g \in G_1$ and all $a, b \in \mathbb{Z}_q$, then $e(h^a, g^b) = e(h, g)^{ab}$
- Non-Degeneracy - if $h \neq 0 \implies e(g, g) \neq 1$

The Weil and Tate pairing are the most used pairings where these properties hold. The pairings usually consist of one elliptic curve paired with a finite field. The point with all this is that problems in the first group may not all be hard in the pairing group. Discrete-logarithms are still hard since $e(g, g)$, $e(g, g^a) \in G_T$ is as hard as $g, g^a \in G$ where G_T is the pairing group. For DDH though, we can see that to test if $z = xy$ given g, g^x, g^y, g^z, we can just check if $e(g, g^z) = e(g^x, g^y)$. DDH is thus replaced by bilinear decisional Diffie-Hellman problem (BDDH) which is defined as - given $h, g, g^x, g^y, e(h, g)^z$ it is hard to decide if $xy = z$ [7]. These definitions made it possible to implement IBE and ABE as described in the next section.

2.6 Identity-based encryption

Public key systems rely on a trusted CA, issuing certificates assuring the binding between a public key and a claimed owner. The user generate their key pair themselves, then the public key has to be signed by a trusted certificate authority. Now the public key can be verified, assuring that nobody is forging it. Each user has to keep a large archive of keys belonging to whom he wants to communicate. More problems arise when a user wants to declare his key invalid and revoke it. All this makes for a big infrastructure of CAs and revocation mechanisms. IBE [3] introduce an approach where a users id act as the public key, this identity can typically be an e-mail address or a user name. There are no CAs, but instead each domain have a key management service (KMS) with a master public/secret key pair. The master public key can be used in conjunction with the id of the desired receiver to encrypt the message. The receiver can then decrypt the cipher text using his private key. This private key is extracted by the KMS from the identity of the specific user. The functions of a IBE scheme is as following:

Setup

Taking some security parameters a master public/secret key pair (mpk, msk) are generated.

Key delegation

Using mpk, msk and some id generating a sk_{id} for the specific user.

Encryption

Encrypts a message m using mpk and the id of the desired receiver.

Decryption

Decrypts cipher text ct using sk_{id}, obtaining m.

The removal of the CAs introduces another problem, since the users no longer generate their own key pair, a lot of power is now in the hands of the KMS. This can be compared with the power of the CA, but algorithms using a KMS have to take into consideration that this service have complete control over all keys, and can in fact generate any key. The life cycle of a system implementing IBE consist of 4 algorithms. In this section two IBE systems will be presented, one basic construction proposed by Boneh and Franklin [3], and one by Waters [22] which is also part of the Charm framework. Next follows the proposed scheme by Boneh and Franklin while figure 3.1 displays a demo of the implementation in Charm.

Boneh and Franklin [3] have proposed a scheme implementing IBE, their construction is as following:

Setup

Random prime q and groups G_1 and G_2 of prime order q are generated from the public parameters. The bilinear mapping $e : G_1 \times G_1 \to G_2$ is also defined. A random element $g \in G_1$ is chosen.
Next chose a random $s \in \mathbb{Z}_q^*$. The master public key is then $mpk = g^s$. And Master secret key $msk = s$
$H_1 : \{0,1\}^* \to G_1^*$ is a hash function. Another hash function $H_2 : G_2 \to \{0,1\}^n$ is chosen for a random n.
The system's public parameters are $q, G_1, G_2, e, n, g, mpk, H_1, H_2,$ which can be published.

Key generation

Generates a private key k_{id} from a given $id \in \{0,1\}^*$, using $msk = s$, as following. Calculate $h_{id} = H_1(id) \in G_1^*$, and set private key $sk_{id} = h_{id}^s$. This is now the private key of the user with the corresponding id.

Encryption

Encrypts $M \in \{0,1\}^n$ using the master public key and id of the desired receiver. First computes $h_{id} = H_1(id) \in G_2^*$. Next chose a random $r \in \mathbb{Z}_q^*$, the chiper text is then

$$ct = (g^r, M \oplus H_2(p_{id}^r)), \text{ with } p_{id} = e(h_{id}, mpk) \in G_2^*$$

Decrypt

Decrypts a cipher text C using a private key corresponding to the decryptors id, sk_{id}. Let $ct = (u, v)$, which yields $u = g^r$, and $v = M \oplus H_2(p_{id}^r$, as seen from the encryption function. Then the message is recovered as following:

$$v \oplus H_2(e(sk_{id}, u))$$
$$(M \oplus H_2(p_{id}^r) \oplus H_2(e(sk_{id}, g^r)) = M$$

This is correct since

$$e(sk_{id}, g^r) = e(h_{id}^s, g^r) = e(h_{id}, g)^{sr} =$$
$$e(h_{id}, g^s)^r = e(h_{id}, mpk)^r = p_{id}^r$$

and it is clear that M is actually xored with the hash of p_{id}^r in both the encryption and decryption function.

The construction of Boneh and Franklin can does the same job as the scheme by ElGamal in section 2.4.1, with the difference being that the IBE scheme uses a master public key and identities to encrypt messages. This scheme could also be changed to be a KEM, similar to the one presented in 2.4.1, by having the message M be a randomly generated key, and changing the input of the encryption function to only be an id and the mpk.

2.7 Key Exchange

A fundamental requirement in many cryptographic schemes is a way of establishing a common secret to be used later to achieve confidentiality or integrity. This is usually solved using a key exchange scheme, which can be between two are a group of parties. In this project the focus is on cases with several users must be allowed. In such group settings there are two types of environment, one when the users are known before the exchange are carried out and they stay the same throughout the life cycle, and one where users join and leave dynamically. Examples of the former are conference calls where the participants are known in advance, before setting up the call, while live chats may be the opposite. This chapter will first show how normal Diffie-Hellman can be utilized to do group key exchange. Next it will be discussed how KEMs can be used to do multi-party key exchange.

2.7.1 Group Diffie-Hellman Key Exchange

The Diffie-Hellmann key exchange algorithm in its original form allows two parties to jointly establish a common secret key which later can be used to encrypt traffic and etc. Since the introduction of the 2-party Diffie-Hellmann researchers tried to extend it to support groups of parties [20, 6]. These configurations allow several parties, typically in a multicast group or similar network, to establish a common session key.

In the 2-party Diffie-Hellman a cyclic group $\mathrm{G} = \langle x \rangle$ of prime order p is chosen carefully. Then each party chose a random number, a and b, before g^a and g^b can be exchanged and the common secret key g^{ab} computed. The group configuration of the scheme uses the same principle only with several participants as shown in Figure 2.1. The configuration is the same for n players. The scheme starts off with the first player raising g to the power of his private key and sends this value to the next player in the chain. He then raises the received value to the power of his private key and sends it, plus the intermediate values, on to the next player, this continues until the last player receives the set, he can now compute the session key $g^{x_1,x_2 \cdot x_3,\ldots,x_n}$. An attacker would have been able to see all the sent combinations, but none of these combine into the session key. New players cannot easily join or leave since all previous players would have to update their set.

It should be noted that this setup has the limitation of not being authenticated, the users participating does not authenticate themself in any way, so it is impossible to know who you are communicating with. This is why our approach later rely on a KEM providing authentication. The public key schemes presented in **??** will assure you about who you are communicating with, since only the intended user will be able to decrypt.

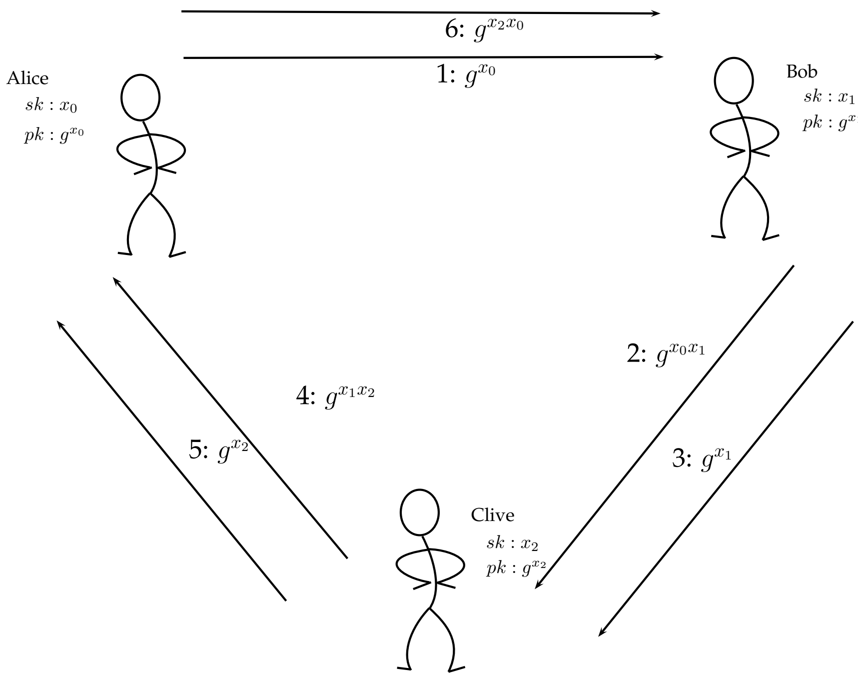

Figure 2.1: D-H group key exchange

2.7.2 Key exchange from KEMs

Gorantla et al. [10] have shown that mKEMs, can be used to establish secure shared keys. It is thus a feasible approach to use a functional encryption scheme as a mKEM to build a functional key exchange scheme. The idea is that the users exchange encapsulations which all the users will be able to decrypt, obtaining a set of symmetric keys, which in turn can be combined to a shared secret key. This secret key can then be used by the DEM to encrypt the communication between the users. The implementation presented later will use this approach, using ABE, extended to a mKEM, to exchange a secret key between multiple users.

Chapter 3 Functional Key Exchange

When communicating on the Internet it is important to control what entities have access to the messages. In most cases it is important that the users can trust that their communication cannot be stolen or eavesdropped on. Encryption is used to secure communication, to do this efficiently a shared key is usually needed. Functional key exchange is in our context defined as a set of key exchange mechanisms using some function to decided if a participant should be allowed to take part in, or be allowed access to, the key exchange. The functions will use some arguments as input and based on these decide if the session key should be output or not. This chapter will explain some proposed schemes trying to adapt this idea, then further explore possibly useful application areas and ideas. Identity-based authenticated key exchange (IB-AKE) and AB-AKE are both examples of functional key exchange, with the latter being a generalization of the former. Since AB-AKE will be used in the implementation later, this chapter will only introduce the basic ideas and principles with the implementation providing a more in depth description. AB-AKE is based on the functional encryption scheme ABE, mentioned in the previous chapter, and the key exchange uses this as the mKEM to exchange keys. Because of this it makes sense to first explain and present functional encryption which is the foundation of the work done in this project. In the section about functional encryption, ABE will also be described using the implementation from Charm which are used later. Moving on to describing thought key exchange schemes which will implemented later, then a part discussing possible applications.

The implementation of AB-AKE is created using ABE as the KEM, ABE will therefor be explained and demonstrated in the beginning of this chapter. The ABE implementation from Charm, as used later in the implementation, is described and demonstrated in the beginning of the chapter, before moving on to the key exchange schemes. The theory of ABE is first described, then the implementation of it is used to showcase how it works in practice.

3.1 Functional Encryption

Before public key cryptography was introduced, secure communication was only achieved when two parties possessed the same secret key which could be used to encrypt and decrypt messages between them. This was changed with the introduction of public key cryptography as mentioned in 2.3. Now, many years later, this is no longer sufficient to cover the new notion of the Internet - with distributed systems and cloud-based services. Standard public key encryption schemes focus on fully encrypting or decrypting plain texts - especially when decrypting you either have the correct key, which would allow you to recover the plain text, or you can not. But what if you would like to allow certain keys to decrypt only a part of the plain text, or output some information about it to anybody? It could in example be useful to allow the mail service to know some meta data, while still not allowing it to decrypt the whole thing, the private key of the server could be allowed to decrypt specific parts decideding if a e-mail is spam or not.

Functional encryption, as described by Boneh et al.[4], describes an idea where different keys produce different plain text - a function of the cipher text. One key might be used to decrypt only a specified part of the cipher text - a "function" or output a function of it. Functional schemes also make it possible for several different keys to decrypt a message if they satisfy a policy, completely or partially. This allows us to encrypt a message for a certain group of users, and later grant new users access to it without having to decrypt it. A KMS issues keys based on some characteristics, which can be used to decrypt message encrypted previously.

The term "Functional Encryption" has come to describe a wide spectrum of modern cryptography techniques, including identity-based encryption and attribute-based encryption, which will be described and demonstrated in the following sections. These schemes relates to functional encryption since it allow several independent keys to decrypt, or uses some function to decrypt without the need of *one* specific key, as in symmetric key encryption.

```
1
2 >>> from charm.toolbox.pairinggroup import PairingGroup, GT
3 >>> from charm.schemes.ibenc.ibenc_waters09 import DSE09
4 >>> group = PairingGroup('SS512')
5 >>> ibe = DSE09(group)
6 >>> mpk, msk = ibe.setup()
7 >>> ID = "student@stud.ntnu.no"
8 >>> secret_key = ibe.keygen(mpk, msk, ID)
9 >>> msg = group.random(GT)
10 >>> ct = ibe.encrypt(mpk, msg, ID)
11 >>> decrypted_msg = ibe.decrypt(ct, secret_key)
12 >>> msg == decrypted_msg
13 True
14 >>>
```

Figure 3.1: Demo run of identity-based encryption from Charm.

First the required dependencies are imported from the Charm toolbox, in this example the pairing group and the IBE class. On line 4 the pairing group is initiated with a elliptic curve with a 512-bit base - thus "SS512". Next the class object is initiated using the previously created group object. Line 6 to 13 demonstrate a IBE cycle with setup, keygen, encryption and decryption. Notice that anybody can encrypt a message for any user without having a public key stored locally, we simply use the master public key together with the identity of the recipient. This removes a lot of overhead known from public key infrastructure (PKI), we now only need one public key for each domain. IBE is a somewhat more intuitive scheme, than normal public key encryption, since the identity of the recipients is used as the public key, thus no connection between different public keys and user identities have to be stored.

This project uses a generalization of IBE called ABE as the basis of the later presented construction. ABE is in short the same is IBE with the difference being that it allows several attributes to be used as the public key, instead of only an id as in IBE. ABE will be described in dept in the next chapter.

3.1.1 Attribute-based encryption

Attribute-based encryption as explained by Goyal et al. [11] introduce an encryption scheme based on user attributes, from which the secret key is generated. This is similar to IBE, but with the possibility of more than one attribute instead of just id. Messages are encrypted using a access policy of several attributes, and only keys satisfying the access structure can decrypt the cipher text. This is typically useful in cases where the encryptor does not care about who decrypts as long as they satisfy the correct attributes or a set of them. Each user will have a private key corresponding to his set of attributes, in each domain. When encrypting, the policy is specified, this is typically a access tree where the attributes required are leaf nodes and internal nodes are "AND" or "OR"-gates. Different combinations of attributes may therefore be able to decrypt. The logical gates can be used to construct threshold requirements, where we require k out of n attributes. The encryptor can in example encrypt a message with the policy

("NTNU" and "5th year" and Telematics dpmt.) or
("Professor" and "Telematics dpmt.") or "admin"

Now a user with either the "admin"-attribute or a set including "NTNU", "5th year" and "Telematics dpmt" would be able to decrypt. A user can create access structures allowing his id or some combination of other attributes to decrypt without having the attributes himself. It is worth noticing that ABE is a generalization of IBE since an identity can be used as an attribute. The algorithm have a similar structure as the one in IBE (2.6).

- **Setup** - Taking some security parameters 1^k a master public/secret key pair (mpk, msk) are generated.
- **Key generation** - Using mpk, msk and some S describing the set of attributes - generates a sk for the specific attribute combination.
- **Encryption** - Encrypts a message m using mpk and an access structure A describing the policy of the encryption. The attributes in the access structure will define who's able to decrypt.
- **Decryption** - Decrypts cipher text ct using sk corresponding to an attribute set S, obtaining m.

Charm also include an implemented ABE scheme based on Waters[23], which can be used to describe how the algorithms work. This construction will later be used in the implementation of a prototype attribute-based key exchange application. The implementations are written in python and should be easily readable. A walkthrough of the ABE protocol using the implementation will be used to demonstrate the algorithms mentioned. The implementation and a sample run of the methods are included for each of the algorithms (the memory references are removed to make it more compact). Figure 3.2 describes the ciphertext-policy attribute-based encryption (CP-ABE) scheme of Waters[23] from which the implementation is based. The mathematical definition in figure 3.2 and the Charm descriptions in figures 3.3, 3.5, 3.7 and 3.9, can be followed in parallel and it can be seen that they in fact define the same thing. Figures 3.4, 3.6, 3.8 and 3.10 shows demonstration runs of the corresponding code blocks. The mathematical presentation might be easier to follow as the symbols are more clear, while the python code use programmatic variable names. The important point to emphasis is that the code is based directly on the mathematical definition, which is provided to make the code easier to follow.

Setup - Group G of prime order p is chosen from generator g. $\alpha, a \in \mathbb{Z}_p$ are generated at random. $H : \{0,1\} \rightarrow G$ is a hash function. The master public key is then $mpk = g, e(g,g)^\alpha, g^a$, where $e()$ is a pairing function as described in 2.2. The master secret key is $msk = g^\alpha$.
Key generation - takes msk and a set S of attributes as arguments and randomly chose $t \in \mathbb{Z}_p$. The constructed private key is then

$$K = g^\alpha g^{at} \qquad L = g^t \qquad \forall x \in S, K_x = H(x)^t \qquad .$$

Encryption - Takes mpk and a message m as arguments, together with a access structure (M, p) where M is an $l \times n$ matrix and p is a function associating rows in M with attributes. A random vector $\vec{v} = (s, y_2, ..., y_n) \in \mathbb{Z}_p^n$ is chosen, this will be used to share the encryption exponent s. Now calculate $\lambda_i = \vec{v}M_i$, for $i = 1$ to l, where M_i is the ith row of the matrix M. $r_1, \ldots, r_l \in \mathbb{Z}_p$. The cipher text is then defined as

$$C = me(g,g)^{\alpha s},\ C' = g^s,$$
$$(C_1 = g^{\alpha\lambda_1} H(p(1))^{-r_1}, D_1 = g^{r_1}, \ldots, C_n = g^{\alpha\lambda_n} H(p(n))^{-r_n}, D_n = g^{r_n}) \qquad .$$

The access structure (M, p) is attached to the cipher text as well.

Decryption - Recovers the plain text from a cipher text for a access structure, using private key corresponding to attribute set S. Let $I \subset \{1, 2, \ldots, l\}$ be defined as $I = \{i : p(i) \in S\}$. Then, chose $\{\omega \in \mathbb{Z}_p\}_{i \in I}$ so that if $\{\lambda_i\}$ are valid shares of a secret with access matrix M, then $\sum_{i \in I} \omega_i \lambda_i = s$. Now the algorithm computes

$$\frac{C', K}{\prod_{i \in I}(e(C_i, L)e(D_i, K_{p(i)}))^{\omega_i}} = \frac{e(g,g)^{\alpha s} e(g,g)^{ast}}{(\prod_{i \in I} e(g,g)^{ta\lambda_i\omega_i})} = e(g,g)^{\alpha s}$$

We have from the encryption method that $C = me(g,g)^{\alpha s}$, and m can thus be divided out.

Figure 3.2: ABE, Waters[23]

Setup

```
def setup(self):
    g1, g2 = group.random(G1), group.random(G2)
    alpha, a = group.random(), group.random()
    e_gg_alpha = pair(g1,g2) ** alpha
    msk = {'g1^alpha':g1 ** alpha, 'g2^alpha':g2 ** alpha}
    pk = {'g1':g1, 'g2':g2, 'e(gg)^alpha':e_gg_alpha,
    'g1^a':g1 ** a, 'g2^a':g2 ** a}
    return (msk, pk)
```

Figure 3.3: ABE setup function

```
>>> from charm.toolbox.pairinggroup import PairingGroup,ZR
    ,
    G1,G2,GT,pair
>>> from charm.schemes.abenc.abenc_waters09 import CPabe09
>>> groupObj = PairingGroup('SS512')
>>> cpabe = CPabe09(groupObj)
>>> (msk, mpk) = cpabe.setup()
>>> print msk
    {'g1^alpha': <pairing.Element object at 0x>,
    'g2^alpha': <pairing.Element object at 0x>}
>>> print pk
    {'g1^a': <pairing.Element object at 0x>,
    'g2^a': <pairing.Element object at 0x>,
    'g2': <pairing.Element object at 0x>,
    'g1': <pairing.Element object at 0x>,
    'e(gg)^alpha': <pairing.Element object at 0x>}
>>>
```

Figure 3.4: Demo running the setup function

The class in figure 3.3 is demonstrated in figure 3.4. The environment from which the methods are run have defined an elliptic curve with bilinear mapping. The pairing $e(g_1, g_2)$ correspond to $e(g, g)$ in 3.2. The master public and private key pair is stored locally at a server acting as a KMS. After initializing the protocol we can generate a secret key using a defined set of attributes. For this we have the key generation method as following. This will be used by the KMS to generate secret keys for users in the protocol. How these keys are distributed is a separate concern which is not dealt with in this report. It is demonstrated how the keys are generated but not how they are distributed to the correct users. It is assumed that each user can receive their key securely from the KMS, and trust it to be reliable.

Key Generation

```
def keygen(self, pk, msk, attributes):
    t = group.random()
    K = msk['g2^alpha'] * (pk['g2^a'] ** t)
    L = pk['g2'] ** t
    k_x = [group.hash(unicode(s), G1) ** t for s in attributes]

    K_x = {}
    for i in range(0, len(k_x)):
        K_x[ attributes[i] ] = k_x[i]

    attributes = [unicode(a) for a in attributes]

    key = { 'K':K, 'L':L, 'K_x':K_x, 'attributes':attributes }
    return key
```

Figure 3.5: ABE key generation function

```
>>> attr_list = ['THREE', 'ONE', 'TWO']
>>> secret_key = cpabe.keygen(pk, msk, attr_list)
>>> print secret_key
    {'K_x': {'TWO': <pairing.Element object at 0x>,
             'THREE': <pairing.Element object at 0x>,
             'ONE': <pairing.Element object at 0x>},
     'K': <pairing.Element object at 0x0>,
     'L': <pairing.Element object at 0x>,
     'attributes': [u'THREE', u'ONE', u'TWO']}
>>>
```

Figure 3.6: Demo running the key generation function

The secret key includes a list of all the attributes with a corresponding hash value raised to the power of a random value $t \in \mathbb{Z}_p$. Additionally a list of the clear text representations of the attributes are added - this will later be used when decrypting, to check if a given key comply with the policy given in the cipher text. The list of attributes for the secret key are compared with the attributes in the access structure before decrypting,this way we avoid actually trying to decrypt if the key doesn't contain the correct attributes. The public parameters in pk must be published together with the secret keys, so that each user has a key pair consisting of their personal secret key, and the master public key.

A major problem when doing ABE is preventing collusion attacks, where a group of users combine their attributes trying to satisfy a more restrictive access structure then what their individual sets of attributes allow. The construction avoids this by randomizing each key with a generated exponent t. When decrypting, each share will have this t in the exponent, which is supposed to bind the components of each key together. Combining two keys would have the value of t different so they would not work together. During decryption these shares are only relevant to the particular key used in that exact run of the decryption algorithm.

Encryption

```
def encrypt(self, pk, M, policy_str):
    # Extract the attributes as a list
    policy = util.createPolicy(policy_str)
    p_list = util.getAttributeList(policy)
    s = group.random()
    C_tilde = (pk['e(gg)^alpha'] ** s) * M
    C_0 = pk['g1'] ** s
    C, D = {}, {}
    secret = s
    shares = util.calculateSharesList(secret, policy)

    # ciphertext
    for i in range(len(p_list)):
        r = group.random()
        if shares[i][0] == p_list[i]:
           attr = shares[i][0].getAttribute()
           C[ p_list[i] ] = ((pk['g1^a'] ** shares[i][1]) *
           (group.hash(attr, G1) ** -r))
           D[ p_list[i] ] = (pk['g2'] ** r)

    return { 'C0':C_0, 'C':C, 'D':D , 'C_tilde':C_tilde,
             'policy':unicode(policy_str), 'attribute':p_list }
```

Figure 3.7: ABE encryption function

```
>>> policy = '((ONE or THREE) and (TWO or FOUR))'
>>> msg = group.random(GT)
>>> cipher_text = cpabe.encrypt(master_public_key, msg,
    policy)
>>> print msg
>>> print cipher_text
    {'C': {
            u'TWO': <pairing.Element object at 0x>,
            u'FOUR': <pairing.Element object at 0x>,
            u'THREE': <pairing.Element object at 0>,
            u'ONE': <pairing.Element object at 0x},
    'D': {
            u'TWO': <pairing.Element object at 0x>,
            u'FOUR': <pairing.Element object at 08>,
            u'THREE': <pairing.Element object at 0x>,
            u'ONE': <pairing.Element object at 0x>},
    'attribute': [u'ONE', u'THREE', u'TWO', u'FOUR'],
    'C_tilde': <pairing.Element object at 0x>,
    'policy': u'((ONE or THREE) and (TWO or FOUR))',
    'C0': <pairing.Element object at 0x>}
```

Figure 3.8: Demo running the encryption function

Before encrypting, a policy is specified, this will be the access structure used in the encryption. Since the protocol relies on pairings, only pairing elements can be used, a random message m is thus generated from the group to be used in the demonstration. If we were to encrypt some kind of readable message we would need an adapter on top, mapping messages to pairing elements. This project focus on applications where this is not needed - random group elements is sufficient for the constructions presented later, the group elements can be hashed to transform it into a random string if needed. The encryption method starts off by extracting the attributes from the policy provided, then a random group object is generated and used together with the public key and the message to calculate a internal cipher text. This secret is then split into shares using LSSS as described in 2.2. Each share is associated with one attribute and a subset of these will be require to obtain s when decrypting, according to the policy.

Decryption

```
def decrypt(self, pk, sk, ct):
    policy = util.createPolicy(ct['policy'])
    pruned = util.prune(policy, sk['attributes'])
    if pruned == False:
        return False
    coeffs = util.getCoefficients(policy)
    numerator = pair(ct['C0'], sk['K'])

    # create list for attributes in order...
    k_x, w_i = {}, {}
    for i in pruned:
        j = i.getAttributeAndIndex()
        k = i.getAttribute()
        k_x[ j ] = sk['K_x'][k]
        w_i[ j ] = coeffs[j]

    C, D = ct['C'], ct['D']
    denominator = 1
    for i in pruned:
        j = i.getAttributeAndIndex()
        denominator *= ( pair(C[j] ** w_i[j], sk['L']) *
        pair(k_x[j] ** w_i[j], D[j]) )
    return ct['C_tilde'] / (numerator / denominator)
```

Figure 3.9: ABE decryption function

```
>>> decrypted = cpabe.decrypt(master_public_key,
    secret_key, cipher_text)
>>> decrypted == msg
True
>>>
```

Figure 3.10: Demo running the decryption function

Decryption is done using the public parameters and a secret key corresponding to a set of attributes. First step in the decryption is to compare the access structure and the attributes present in the secret key. If the policy is not fulfilled the method can return straight away. The prune method performs this validation and returns a "pruned" list of attributes. This is the minimal subset of the attributes satisfying the policy - in example a set including both childes of a "OR" node would be pruned to only include one of these. Finally the secrets are combined and used to recover the message. The calculations can be recognized from the decryption method in figure 3.2.

From the scheme described it is noticeable from the encryption method that anybody can in fact encrypt for any set of attributes, as long as they have the master public key. The authentication is not mutual, the encryptor doesn't have to have any specific attributes to be able to encrypt. The protocol only provide assurances that nobody without the correct attribute set can decrypt the message, this is sufficient when used as a public key encryption mechanism, but might not hold in cases where mutual authentication is required.

3.2 Attribute-based Authenticated Key Exchange

Gorantla et al.[9] introduced the concept of AB-AKE using a attribute-based key encapsulation mechanism. In short this is a KEM with ABE as the encryption mechanism. The idea is that several users can exchange keys and thus communicate without knowing the identities of all the users. Any user satisfying the specified policy should be able to participate in the communication. AB-AKE establish a common session key between the users which can be used to communicate securely. Goyal et al. [11] introduced the notion of CP-ABE where the private key of each user are associated with attributes and the cipher text has an attached access policy. The construction by Gorantla et al.[9] uses this approach to create what they call an encapsulation policy attribute-based key encapsulation mechanism (EP-AB-KEM), where the attributes are associated to the private key of a user and the access policy is attached to the encapsulation. The encapsulation is a randomly generated symmetric key encrypted with with a master public key and a access policy. To generate the common session key each user has to upload such an encapsulation and receive encapsulations from all other users. The session key is then obtained by decapsulating and combining the symmetric keys of all participants.

AB-AKE will be described in more detail in the next chapter where a system implementing a variation of AB-AKE is presented.

3.3 Identity-based Authenticated Key Exchange

IBE as described in 2.6, can also be utilized to provide two-party mutually authenticated key exchange (AKE) [15]. The approach is based on a Diffie-Hellman key exchange using an elliptic curve. Each party chose random points a, b. The values p^a, p^b are then encrypted using the other parties public key and then exchanged in succession. B will include p^a which he recieved from a, this is done so that A can verify that B actually was able to decrypt what he sent. B actually adds to what he receives from A by decrypting and adding his contribution and then encrypting again. After decrypting, the session key is the product a^{bp}, which both can calculate. After exchanging secrets, A has to authenticate himself in the same way as B did, by sending the secret he got from B back, to show that he was able to decrypt what B sent. This technique provides mutual implicit authentication between the participants, since only the users with the correct identity can decrypt. Both parties can thus be sure that no other user than the one possessing the private key corresponding to the identity, can produce the session key. Protocol 1 shows the procedure as described by Kolesnikov et al. [15].

Protocol 1.

A - given curve and point p		B - given curve and point p
chose random point a		
	$\xrightarrow{IBEnc_B(p^a)}$	
		chose random point b
	$\xleftarrow{IBEnc_B(p^a, p^b)}$	
verify p^a after decrypting using private key		
	$\xrightarrow{IBEnc_B(p^b)}$	
		verify p^b to authenticate that A actually decrypted the message
	sk = p^{ab} sid=(p^a, p^b)	

This implementation demonstrates a scheme for key exchange between two parties with the focus on assuring authenticity of the identities of the participants. This is mostly a more effective implementation of public key crypto systems, the main difference from previously popular systems is the removal of PKI by switching from CAs to KMSs. Point being that the main idea is still to encrypt some message or symmetric key for *one* specific user. Another point in favor of IB-AKE is that it may make encryption using public key crypto more feasible for non technical users, since it is easier to understand that you can encrypt using, in example the id of the desired receiver, rather than understanding the concept of public keys.

3.4 Applications

Key exchange schemes as discussed up till this point makes it feasible to exchange secret keys, and thus allow secure communication between users without them having to reveal their identities or to simply make it more feasible to use public key encryption. Identities may also be chosen differently depending on what domain or context the communication is being carried out in. The most general and intuitive case is simply using email or some other public identifier, but there may be cases where other identities is more useful. Within a company, working titles such as CEO or CTO could be used instead, to make it more usable in large companies where not everybody necessarily know the name of all their co-workers. In this case AB-AKE would make it even more useful, since the CEO could have attributes including both his email and "CEO", allowing both identities to be used.

Being able to communicate securely even without revealing identities is clearly useful for applications where users want to stay anonymous. Typically messaging services and forums can take advantage of such characteristics, users are able to exchange keys without any previous knowledge to eachother, while still knowing enough to trust them. The Internet is full of sites where users can upload questions which then can be answered by qualified superusers, but these services has the weakness that the users have to be willing to expose their message and possibly identity to be able to get an answer. After agreeing to this, they have to trust that the administrators of the system ensure the confidentiality of your message and only allow certain users to read it. The same goes for other applications where you want only specific types of users to be able to participate.

By using functional key exchange you can specify in detail who is allowed to take part in the communication, this can range from very wide policies allowing a certain age group or gender, down to very specific characteristics such as degrees or military ranks. The most specific policy you can use is thus the identity itself as discussed earlier. This can be used in a variety of applications where access control of some degree is necessary, a good example being a room based chat system. There are several such applications where users are only allowed to join rooms if they satisfy some conditions, but usually the access control to the rooms rely on a server controlling this, so that when granted access, you have to trust that the service wont grant access to users without the correct attributes. Using functional key exchange, you as a user, would be able to ensure that nobody outside of the ongoing chat session will ever be able to read what is written. By the use of a session key relying on keying material from all participating parties. With this approach the system could inherit a hierarchy of user types, so that you have to prove your seriousness and knowledge to be able to achieve the higher rankings. This is a common way of administrating forums and chat room to avoid frivolous users who are there only to destroy the discussions. This can now be embedded directly in the encryption by adding group names as attributes in the key exchange policy. Functional key exchange can thus help prevent off-topic messaging and extraneous posting by users there with bad intentions. In the next chapter a simple version of such a chat system will be presented to show how AB-AKE can be used to keep the communication secure by the use of fresh session keys, renew every time a user joins.

Chapter 4
Design and Implementation

This chapter suggests a prototype implementation of one possible application, based on the principles and ideas in the previous two sections. The protocol structure will be displayed, together with examples from the code and test runs using the system. The chosen functional key exchange scheme used is AB-AKE as described in the previouse chapter, and it will be aplied in a distributed chat environment.

4.1 System specifications

As mentioned in 3.4 there are several scenarios where functional key exchange can provide security and privacy. This section will describe the structure and specifications for a chat system utilizing AB-AKE as described in 3.2 and supported by Wang et al.[21]. The system consists of a set of clients running a client application and a broadcast server. It could easily have been altered to support peer-to-peer, since the server only acts as a intermediate for broadcasting, caching of encapsulations and policy management. The system shown is meant to be a proof of concept for how this kind of application could look, it is thus simplified to some extent. Most of the data used are static to abstract away difficulties with administration of rooms and related policies, as well as key management. The implementation will not address key distribution, therefore a key is given to the users on connection.

The most important feature of the system is to provide encrypted communication between users who satisfy the room policy. The users should obtain a shared session key through AB-AKE. This way we assure implicit authentication of all users taking part in the conversation. A user should be able to participate in the exchange without ever having to provide an identity. It is assumed that all users have registered with some KMS prior to the key exchange - a user would typically register a set of attributes which would have to be approved by the system authority before issuing the key. When new users join, they should be able to upload their contribution and receive the rest of the keying material from the server; the users will then have to compute the new session key from the encapsulations. After exchanging keys, the users should be able to use it to encrypt the chat messages. It should be noted that anybody can in fact upload a contribution and receive encapsulations, but only the ones with the correct attributes can decapsulate and produce the session key. It might be smart for the server to challenge the new user to prove that he has the correct attributes. The server could require that new users prove that they have the correct attributes before being able to upload, to avoid flooding of the storage. This extension would simply mean the server challenging the new user with an encrypted nonce which the user would have to decrypt and send back. This mechanism was not included in this implementation as this project focus on demonstrating the key exchange process, and since this possible weakness not pose any threats to the integrity of the system it is not prioritized.

4.2 Models and construction

The high level construction of the key exchange used in the system is based on the generic one-round AB-AKE protocol presented by Gorantla et al. [9], the main differences being the encapsulation function used. The implementation described in this project is constructed based on the ABE scheme implemented in Charm, as described in 3.1.1 and by Waters et al.[23], while Gorantla et al.[9] introduce their own encapsulation policy attribute-based key encapsulation mechanism (EP-AB-KEM). This project propose a complete system utilizing the key exchange, so after obtaining a shared key, users encrypt their messages using a standard symmetric key encryption algorithm. The messages are broadcast through the same broadcast server used for key exchange.

When a new user joins, the message exchanges are paused until the new key is calculated by all users. Figure 4.1 shows the system flow when a new user connects. A client will first query the server for the room policy, before an encapsulation is generated from the received policy and the public parameters. The encapsulation is sent to the server which distributes it so that all users have all contributions.

Now the users will start using the session key to encrypt the chat messages and the server will go back to being a pure broadcast server for the now encrypted chat messages.

Figure 4.2 illustrates the encapsulation distribution process. Bob is a new user, so he uploads his contribution to the server, there are n users already in the system. Next the server broadcast Bob's encapsulation to the current users, before sending the set of active encapsulations to Bob. After obtaining all encapsulations, all users initiate the key generation procedure, as shown in figure 4.3, which consist of decapsulating the encapsulations, then using the symmetric keys with a pseudo random function on a session id, finally combining them using bitwise xor. For the pseudo random function keyed hmac from the python library is used. The session id is the concatenation of the socket addresses of each user, as for the presented implementation these addresses are sent from the server together with the encapsulations, but if the system was to be extended to a peer-to-peer configuration, each encapsulation would be sent individually from different addresses and could be obtained by the users directly.

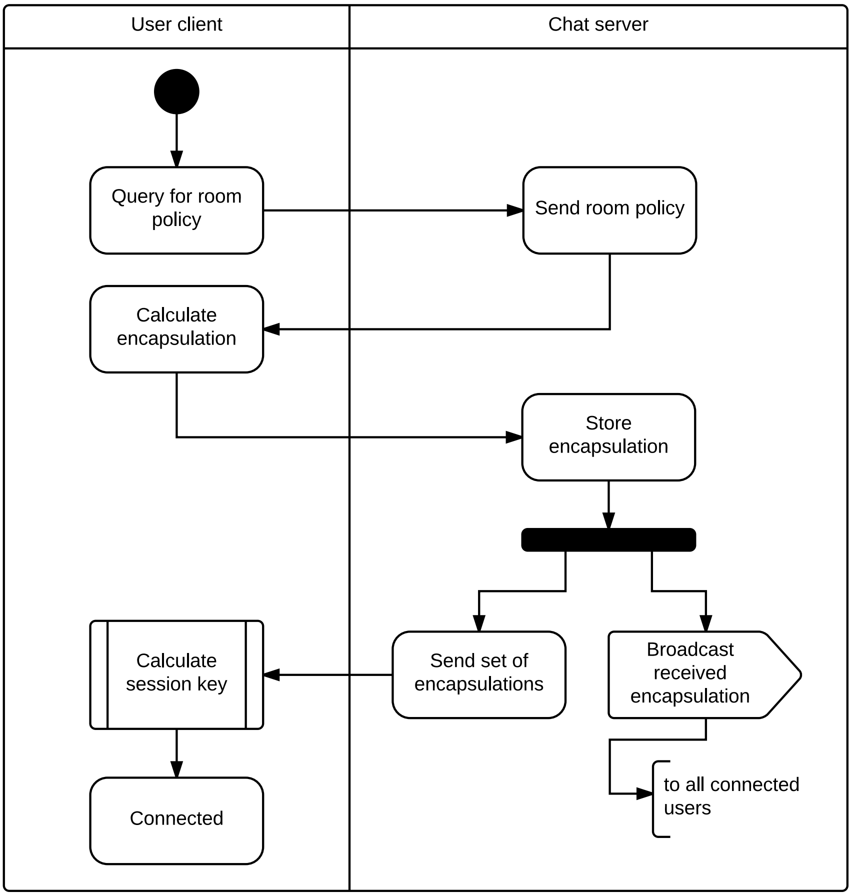

Figure 4.1: System flow.

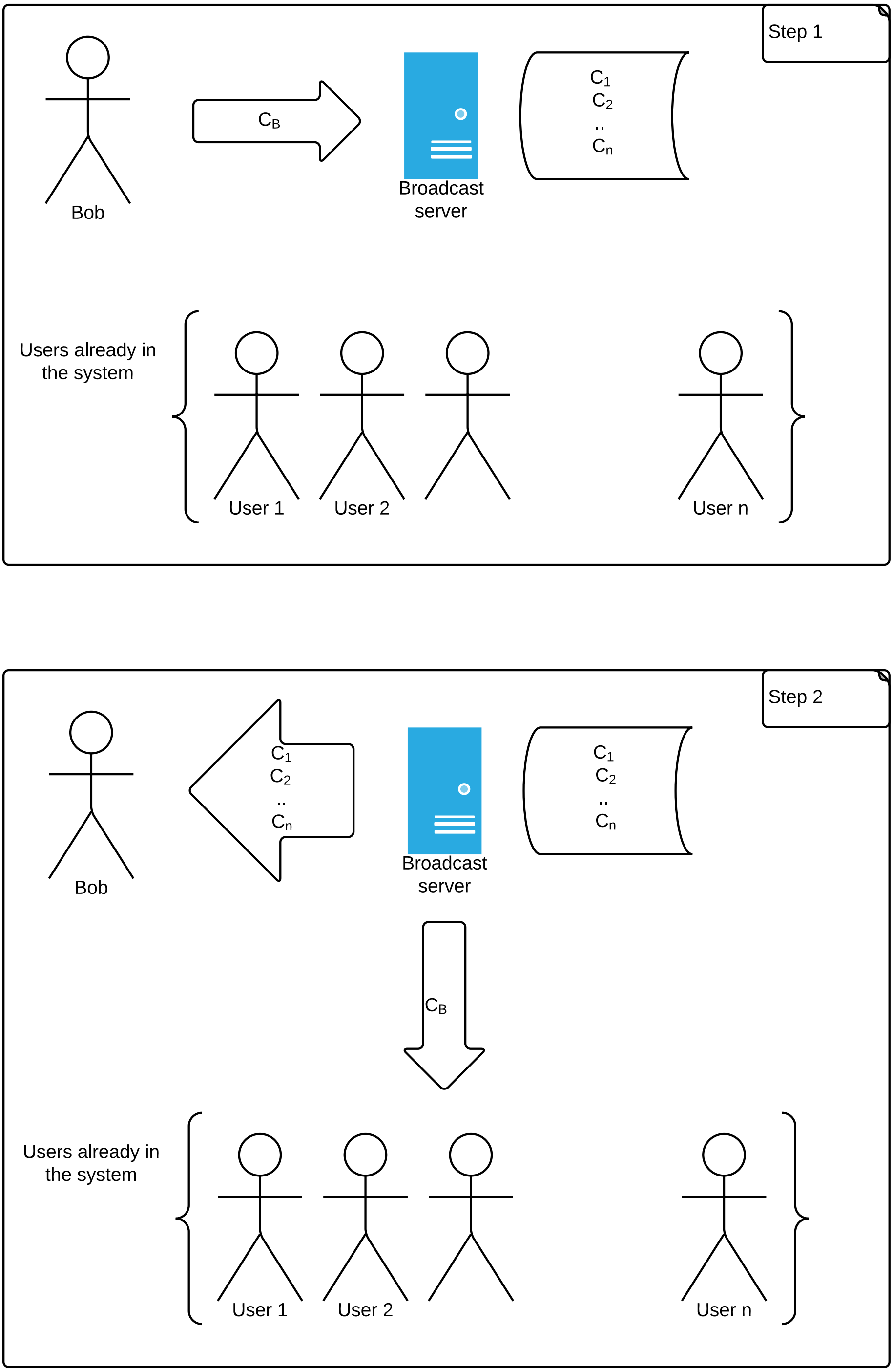

Figure 4.2: Distribution of encapsulations

Let $U_i = \{u_1, u_2, \ldots, u_n\}$ be the set of users participating in the key exchange. Sk_i is the corresponding user's secret keys. $C_j = c_1, c_2, \ldots, c_n$ is the set of encapsulations. A_j is the network address of each user.

Decapsulation - k_j is the symmetric key of encapsulation c_j. Each user, u_i, decapsulates all encapsulations as follows.

$$\text{For } j \neq i \qquad k_j \leftarrow Decapsulate(Sk_i, C_j)$$

Session id - $Sid = A_1 \parallel A_2 \parallel \cdots \parallel A_n$

Session key computation - $K = hmac_{k1}(Sid) \oplus \cdots \oplus hmac_{kn}(Sid)$

Figure 4.3: Key generation procedure

Security concerns For a system as the one presented in this project there are some security properties which should be assessed. First of all the key agreement protocol have to be secure adversaries eavesdropping on the communication channel. A seemingly legit user with access to the communication should not be able to compute the session key from the keying material exchanged without actually being a part of the session. This includes replaying packets stolen from another session. The forward and backward secrecy of the keying material as well as the session keys is a relevant problem, which may be important. Notions of forward and backwards secrecy is used differently according to the context, where the original definition assume a long-term key with a session key established using this and that future exposure of the long-term key would not reveal the session key. Alzaid et al. [2] discuss how this is not always how the term is used nowadays, and that the current use in the context of group key communication is that a session key should not be revealed if an older key gets compromised. After establishing a session key the communication channel needs to be secured using this, preferably using authenticated encryption. The last concern for systems using this kind of key exchange mechanism is how to securely issue private keys to the users, since they can't generate these themself.

The focus of this project is on the security of the session keys used to secure the communication and because these are generated from the encapsulations of the users it makes sense to consider both security of the generated session key and the encapsulations. To avoid confusion with the terminology related to forward/backward secrecy and other similar terms, a description of how the terms should be interpreted in this report are provided.

Forward secrecy Exposure of a private long-term key of a user should not reveal previous session keys established using this key.

Old encapsulation secrecy Revelation of the symmetric keys from previous key exchanges should not compromise the current session key.

Session key secrecy Compromising a session key, former or current, should not reveal any other session keys.

This system does not provide forward secrecy, since any user with the correct attributes can obtain all previous session keys by decapsulating and combining the correct set of encapsulations. The addition of forward secrecy is definitively a relevant extension, as discussed by Gorantla et al. [9]. What the system does ensure is old encapsulation secrecy, meaning that a set of encapsulations from previous sessions will not reveal the current one. New session keys are calculated every time a new user joins, the users can be sure of this since they actually provide parts of the key material themself, the new material will not be part of previous encapsulation sets. This property is important because without it a new user could never be sure that a previous key might be compromised and reveal all the messages encrypted by him. Session key secrecy is also important since new keys are generated all the time, by users joining, there will thus exist a lot of old session keys for each room. As for users leaving a decision was made not to generate new keys when users leave. If users trusted all the users currently in a room, the removal of one of these would not change the trust, and no new key needs to be generated, saving time and resources. Especially since the system presented here uses only one server doing all the work, it is good to generate new keys as infrequently as possible, since the communication is paused during key exchange. In this case some might also argue that creating a new session key on user leave would also be preferable. In addition to this, much of the security of the system rely on the KMS not going rogue, but in the applications discussed, we can assume that the users trust the KMS, this is reasonable since each system or domain, would require users to register and thus agree to the terms and conditions of the system.

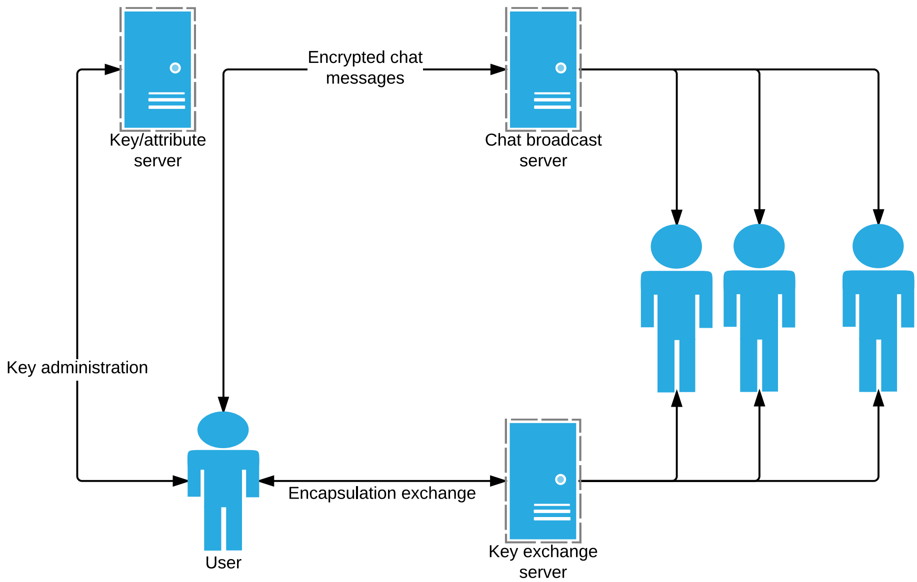

Figure 4.4: Improved system architecture

4.2.1 Possible extensions and improvements

The extended system would include a separate KMS which the users would register with to obtain their key. The best solution would probably be to have a separate virtual or physical server doing each task; attribute and key management, storage and distribution of keying material, broadcasting of chat messages/cipher texts. Figure 4.4 shows how the extended system would look on a high level. Other features like being able to create your own rooms and administrate these would also be logical, and would have to be stored and handled somewhere. The prototype presented here will be a single room with a static policy.

Another interesting extension to the system would be to remove the broadcast servers completely and apply a peer-to-peer setup, where the users distribute the encapsulations and the chat messages themself. The users would have to broadcast their encapsulations individually, the structure would look much like the one in group Diffie-Hellman as described in 2.7.1. The users would need to have some way of knowing where to send their encapsulations, this could be solved by using multicast groups, which new users subscribes to, allowing them to receive new encapsulations, this would essentially be the same as the configuration used in this project. This setup would also require the users to negotiate the policy of the room without an intermediate server keeping track of this. To avoid pauses in the communication, the key exchange could be done in parallel while still using the old key, change to the new one when finished.

4.3 Implementation

The application will consist of two components, one server class and one client class. The implementations of these follow the state diagrams 4.5 and 4.6 respectively. This section will describe how these two classes are implemented, the complete implementation including the code is attached in appendix A and B. The state diagrams represents exactly how the programs work together, the send and receive actions in the diagram represent sockets in the application, and the loop binding it together is equal to the main loops of each class. This eventually mean that a send in the figure have a corresponding socket.send() in the code.

4.4 System demonstration

This section will demonstrate how the system looks in practice, figure 4.7, 4.8 and 4.9 shows how the system acts from the perspective of the server and from the clients, respectively.

- First the server is started with a preset policy as highlighted in 4.7, then two users join in succession, it can be seen that for each new user encapsulations are broadcast. The clients that connect can be observed in 4.8 and 4.9 - when starting the client program, it will automatically try connecting to the server on address given as argument to the program.
- After connecting, the server will start by sending the room policy and setup the user with a attribute key and the public parameters, as discussed earlier this is done to abstract away the problem of key distribution. It would also be more complicated to keep the keys persistent, since it would have to be stored somewhere locally, on a deployed system these keys would be in the user data of each registered user.

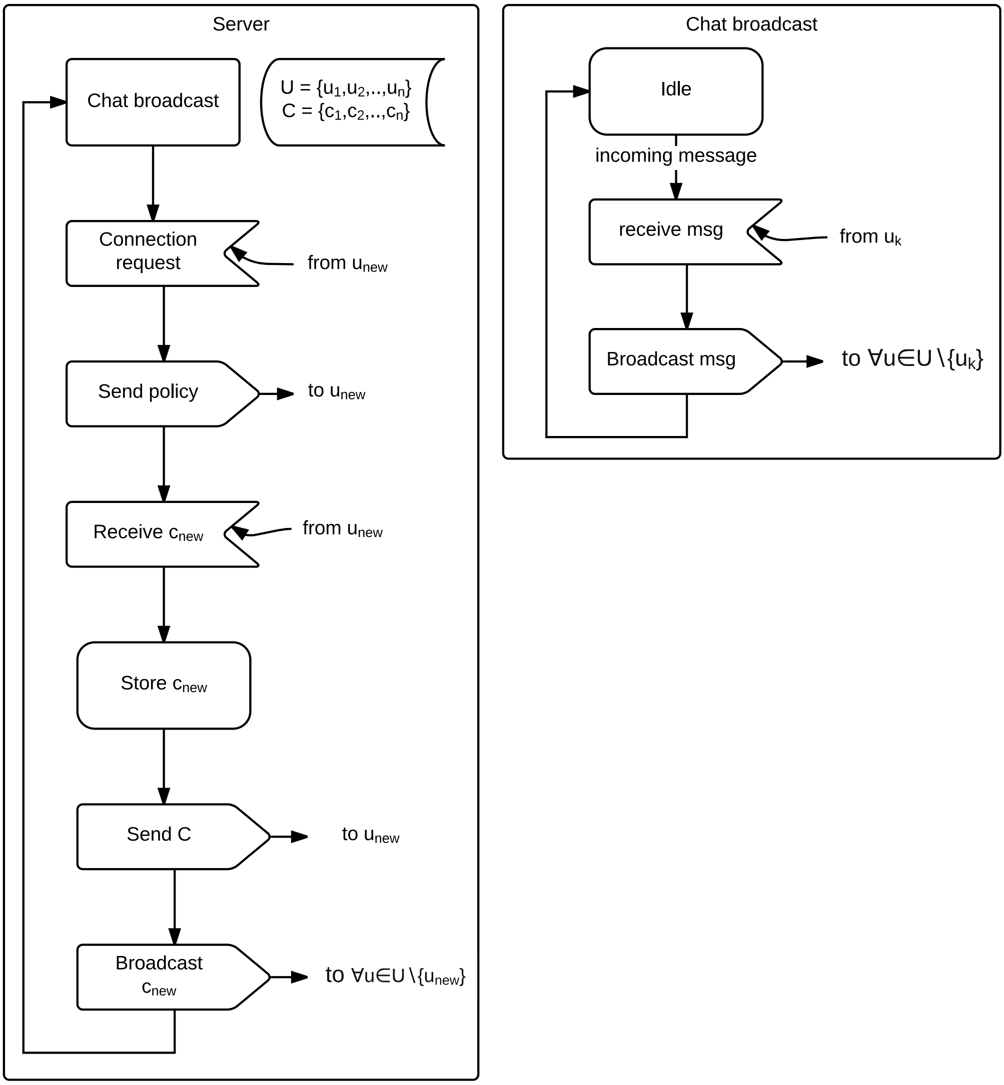

Figure 4.5: Internal flow of the server class

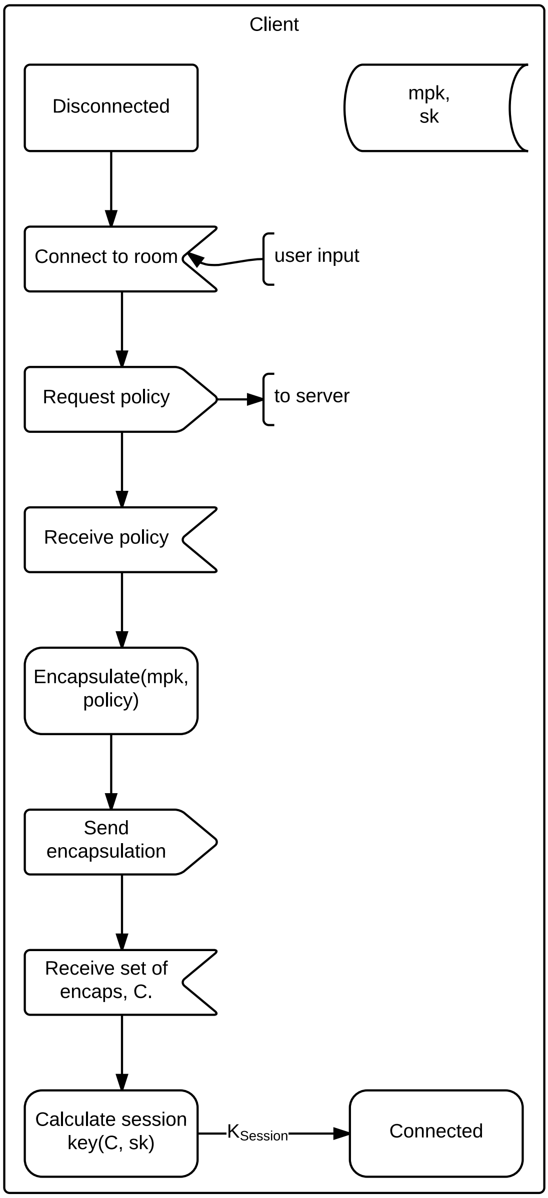

Figure 4.6: Internal flow of the client class

```
andkof@anders-s9:~/code/attr-chat$ python server.py
Chat server started on port 5000
Current room policy: ((ONE or THREE) and (TWO or FOUR))


Client Anonymous 1, on ('127.0.0.1', 43578) connected
Send encapsulation number 1 of 1
Users in the room: ['Anonymous 1', '']


Client Anonymous 2, on ('127.0.0.1', 43580) connected
Send encapsulation number 1 of 2
Send encapsulation number 2 of 2
Users in the room: ['Anonymous 2', 'Anonymous 1', '']


Client Anonymous 2 left the room.
Clients in the room: ['Anonymous 1', '']
```

Figure 4.7: Server interface

- After receiving the policy, the client uploads his encapsulation and we can see that he receives the current set from the server - which in the case of one user only is his own, which he already have.
- When the second user joins it can be seen that the first user generates a new key which is the same as the one generated at the second user. The messages are now encrypted using this key.

Usability

```
andkof@anders-s9:~/code/attr-chat$ python client.py
    localhost 5000
Chat name: (leave blank to be completely anonymous)

Received room policy from server:
((ONE or THREE) and (TWO or FOUR))
Key attributes:  [u'THREE', u'ONE', u'TWO']
Connected to remote host. Uploading and downloading
    encapsulations

Encapsulations received: 0. Generating session key.
Generated session key: 96cc1a71acf66c2c
<You>
Encapsulations received: 1. Generating session key.
Generated session key: 75f2>f141k54c7jf
<You>  Anonymous 2 entered room
<You> Hello, I'm #1.
[Anonymous 2]   Hi there, this is #2.
<You>  Client Anonymous 2 is offline
<You>  Clients in the room:  ['Anonymous 1', '']
<You>
```

Figure 4.8: Client 1 UI

```
andkof@anders-s9:~/code/attr-chat$ python client.py
    localhost 5000
Chat name: (leave blank to be completely anonymous)

Received room policy from server:
((ONE or THREE) and (TWO or FOUR))
Key attributes:  [u'THREE', u'TWO', u'FOUR']
Connected to remote host. Uploading and downloading
    encapsulations

Encapsulations received: 1. Generating session key.
Generated session key: 75f2>f141k54c7jf
[Anonymous 1]    Hello, I'm #1.
<You> Hi there, this is #2.

KeyboardInterrupt
```

Figure 4.9: Client 2 UI

Chapter 5

Conclusion

This project have presented, described and compared different functional key exchange mechanisms and showed how they can be used in different scenarios. In addition to researching the concepts of AB-AKE and IB-AKEand how these schemes are constructed, an actual application taking advantages of AB-AKE are presented with models and a demonstration. The application described uses ABE as the encapsulation mechanism so that each user can provide their own contribution, thus achieving old encapsulation secrecy - assuring that a set of compromised old encapsulations will not reveal the new session key. The system provides mutual authentication since only users with the correct attributes can obtain all the required symmetric keys to compute the session key.

finish this section, add reference to Scope and objectives in the introduction

References

[1] Joseph A Akinyele, Christina Garman, Ian Miers, Matthew W Pagano, Michael Rushanan, Matthew Green, and Aviel D Rubin. Charm: a framework for rapidly prototyping cryptosystems. *J. Cryptographic Engineering*, 3(2):111–128, 2013.

[2] Hani Alzaid, DongGook Park, Juan Manuel González Nieto, Colin Boyd, and Ernest Foo. A forward and backward secure key management in wireless sensor networks for PCS/SCADA. In *Sensor Systems and Software - First International ICST Conference, S-CUBE 2009, Pisa, Italy, September 7-9, 2009, Revised Selected Papers*, pages 66–82, 2009.

[3] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[4] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: a new vision for public-key cryptography. *Commun. ACM*, 55(11):56–64, 2012.

[5] Ilker Nadi Bozkurt, Kamer Kaya, and Ali Aydin Selçuk. Practical threshold signatures with linear secret sharing schemes. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, pages 167–178, 2009.

[6] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably secure authenticated group diffie-hellman key exchange. *ACM Trans. Inf. Syst. Secur.*, 10(3), 2007.

[7] Ratna Dutta, Rana Barua, and Palash Sarkar. Pairing-based cryptography: A survey. *Cryptology Research Group, Stat-Math and Applied Statistics Unit*, 203, 2004.

[8] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[9] M. Choudary Gorantla, Colin Boyd, and Juan Manuel González Nieto. Attribute-based authenticated key exchange. In *Information Security and Privacy - 15th Australasian Conference, ACISP 2010, Sydney, Australia, July 5-7, 2010. Proceedings*, pages 300–317, 2010.

[10] M. Choudary Gorantla, Colin Boyd, Juan Manuel González Nieto, and Mark Manulis. Generic one round group key exchange in the standard model. *IACR Cryptology ePrint Archive*, 2009:514, 2009.

[11] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 89–98, 2006.

[12] L Donnerhacke I K S GmbH H Finney D Shaw R Thayer J. Callas PGP Corporation. OpenPGP Message Format, 2007.

[13] Eike Kiltz and Krzysztof Pietrzak. Leakage resilient elgamal encryption. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 595–612, 2010.

[14] Werner Koch and Others. The GNU privacy guard. *Computer software. Available from http://www.gnupg.org*, 2003.

[15] Vladimir Kolesnikov and Ganapathy S. Sundaram. IBAKE: identity-based authenticated key exchange protocol. *IACR Cryptology ePrint Archive*, 2011:612, 2011.

[16] Kaoru Kurosawa. Hybrid Encryption. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 570–572. 2011.

[17] Kaoru Kurosawa and Le Trieu Phong. Kurosawa-desmedt key encapsulation mechanism, revisited. In *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, pages 51–68, 2014.

[18] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[19] Nigel P. Smart. Efficient key encapsulation to multiple parties. In *Security in Communication Networks, 4th International Conference, SCN 2004, Amalfi, Italy, September 8-10, 2004, Revised Selected Papers*, pages 208–219, 2004.

[20] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to group communication. In *CCS '96, Proceedings of the 3rd ACM Conference on Computer and Communications Security, New Delhi, India, March 14-16, 1996.*, pages 31–37, 1996.

[21] Hao Wang, Qiuliang Xu, Han Jiang, and Rui Li. Attribute-based authenticated key exchange protocol with general relations. In *Seventh International Conference on Computational Intelligence and Security, CIS 2011, Sanya, Hainan, China, December 3-4, 2011*, pages 900–904, 2011.

[22] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 619–636, 2009.

[23] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, pages 53–70, 2011.

Appendix A

Client.py

```
import sys, socket, select, json, time
from charm.toolbox.pairinggroup import PairingGroup,ZR,G1,G2,GT
    ,pair
from charm.schemes.abenc import abenc_waters09
from charm.core.engine.util import objectToBytes,bytesToObject


#Function to broadcast chat messages to all connected clients
def broadcast (sock, message):
    #Do not send the message to master socket and the client
        who has send us the message
    for socket in userdata.keys():
        if socket != server_socket and socket != sock :
            try :
                time.sleep(0.1)
                socket.send(message)
            except :
                # broken socket connection may be, chat client
                    pressed ctrl+c for example
                socket.close()
                if socket in userdata:
                    removeUser(sock)

def handleNewConnection():
    sockfd, addr = server_socket.accept()
    userdata[sockfd] = 'Anonymous␣{}'.format(len(userdata))
    #send the room policy
    sockfd.send(roomPolicy)
```

```
    #send pk and group
    sockfd.send(objectToBytes(pk,groupObj))
    #send key using a attr set
    sockfd.send(objectToBytes(cpabe.keygen(pk, msk, attr_dict[i
        ]),groupObj))
    #receive the encapsulation
    encap = bytesToObject(sockfd.recv(RECV_BUFFER), groupObj)
    #save it
    encapsulations.append(encap)
    entities.append(addr)
    broadcastEncapList(sockfd, addr)
    broadcast(sockfd, "[server] {} entered room\n".format(
        userdata[sockfd]))
    print 'Users in the room: {}\n\n'.format(str(userdata.
        values()))

def broadcastEncapList(sockfd, addr):
    broadcast(sock, "1000001"+str(len(encapsulations)))
    time.sleep(1)
    print "Client {}, on {} connected".format(userdata[sockfd],
         addr)
    for j in xrange(0,len(encapsulations)):
        time.sleep(0.1)
        broadcast(sock, objectToBytes(encapsulations[j],
            groupObj))
        print('Send encapsulation number {} of {}'.format(j+1,
            len(encapsulations)))

def removeUser(sock):
    broadcast(sock, "[server] Client {} is offline\n".format(
        str(userdata[sock])))
    print "Client {} left the room.".format(userdata[sock])
    del userdata[sock]
    broadcast(sock, "[server] Clients in the room:  {}\n".
        format(str(userdata.values())))
    print "Clients in the room: {}\n\n".format(str(userdata.
        values()))


def processData():
    try:
```

```
        # receiving data from the socket.
        data = sock.recv(RECV_BUFFER)
        if data:
            # there is something in the socket
            broadcast(sock, "\r" + '[' + userdata[sock] + '] '
                + data)
            return True
        else:
            # remove the socket that's broken
            if sock in userdata.keys():
                removeUser(sock)
            return True
    except:
        return False

def disconnected():
    print "Unexpected error:", sys.exc_info()[0]
    print 'Users in the room: {}'.format(str(userdata.values())
        )
    broadcast(sock, "[server] Client {} is offline\n".format(
        userdata[sock]))

if __name__ == '__main__':
    attr_dict = [
                [ 'THREE', 'ONE', 'TWO'],
                [ 'THREE', 'TWO', 'FOUR'],
                [ 'ONE', 'THREE', 'FOUR'],
                [ 'ONE', 'TWO', 'FIVE']]
    i=0
    groupObj = PairingGroup('SS512')
    #Init the abe class
    cpabe = abenc_waters09.CPabe09(groupObj)
    (msk, pk) = cpabe.setup()
    #Run setup method
    roomPolicy = '((ONE or THREE) and (TWO or FOUR))'

    userdata = {} #list of sockets and chat names.
    encapsulations = []
    entities = []
    RECV_BUFFER = 4096
    PORT = 5000
```

```
server_socket = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.
    SO_REUSEADDR, 1)
server_socket.bind(("0.0.0.0", PORT))
server_socket.listen(10)

# Add server socket to the list of readable connections
userdata[server_socket] = ''

print "Chat␣server␣started␣on␣port␣" + str(PORT)
print "Current␣room␣policy:␣{}\n\n".format(roomPolicy)

#Start the server main loop
while 1:
    read_sockets,write_sockets,error_sockets = select.
        select(userdata,[],[])
    for sock in read_sockets:
        if sock == server_socket: #New connection
            handleNewConnection()
            i+=1
        else:  #Incoming message from a client
            processData() or disconnected()
server_socket.close()
```

Appendix B
Server.py

```
import socket, select, string, sys, os, base64
from charm.toolbox.pairinggroup import PairingGroup,ZR,G1,G2,GT
    ,pair
from charm.schemes.abenc import abenc_waters09
from charm.core.engine.util import objectToBytes,bytesToObject
from Crypto.Cipher import AES
from Crypto import Random
from charm.core.math.pairing import hashPair as sha1

def prompt() :
    sys.stdout.write('<You> ')
    sys.stdout.flush()

def encapsulate(cpkey):
    key = groupObj.random(GT)
    return key, cpabe.encrypt(pk, key, policy)

def xorString(s1,s2):
    return ''.join(chr(ord(a) ^ ord(b)) for a,b in zip(s1,s2))

def generateSessionKey(encaps):
    key=sha1(cpabe.decrypt(pk, cpkey, encaps.pop()))
    for encap in encaps:
        key = xorString(key,sha1(cpabe.decrypt(pk, cpkey, encap
            )))
    return key

#padding functions for the encryption.
```

```
BS = 16
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(s[-1])]

#AES encryption functions
def encrypt(key, raw ):
    raw = pad(raw)
    iv = Random.new().read( AES.block_size )
    cipher = AES.new(key, AES.MODE_CBC, iv )
    return base64.b64encode( iv + cipher.encrypt( raw ) )

def decrypt( self, enc ):
    enc = base64.b64decode(enc)
    iv = enc[:16]
    cipher = AES.new(key, AES.MODE_CBC, iv )
    return unpad(cipher.decrypt( enc[16:] ))

def connect(server, port):
    try :
        server.connect((host, port))
        policy = server.recv(4096)
        print('Received room policy from server: {}'.format(
            policy))
        pk = bytesToObject(server.recv(4096), groupObj)
        cpkey = bytesToObject(server.recv(4096), groupObj)
        print('Key attributes:  {}'.format(cpkey['attributes'])
            )
        return pk, cpkey, policy
    except :
        print 'Unable to connect'
        sys.exit()

def receiveEncapsulations() :
    #sys.stdout.write("\n\nReceived encapsulations ")
    encapsulations = []
    for i in xrange(0,int(data[7])):
        try:
            encapsulations.append(bytesToObject(server.recv
                (4096), groupObj))
            #sys.stdout.write(str(i+1) + ", ")
        except:
```

```
            print("RECEIVE ENCAP EXCEPTION ON RUN # {}, 
                RETRYING ONCE".format(i))
            encapsulations.append(bytesToObject(server.recv
                (4096), groupObj))
            continue
    print('\nEncapsulations received: {}. Generating session 
        key.'.format(len(encapsulations)-1))
    return encapsulations

def printMessage(data):
    sender = data.split("]",1)[0] + "]"
    msg = data.split("]",1)[1]
    if sender!='[server]':
        msg = sender + "   " + decrypt(key, msg)
    sys.stdout.write(msg)
    sys.stdout.flush()
    prompt()

#main function
if __name__ == '__main__':
    groupObj = PairingGroup('SS512')
    cpabe = abenc_waters09.CPabe09(groupObj)
    policy=''
    if(len(sys.argv) < 3) :
        print 'Usage : python telnet.py hostname port'
        sys.exit()

    host = sys.argv[1]
    port = int(sys.argv[2])

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.settimeout(2)
    print("Chat name: (leave blank to be completely anonymous) 
        ")
    nickName = sys.stdin.readline().strip()

    # connect to remote host
    pk, cpkey, policy = connect(server, port)
    print 'Connected to remote host. Uploading and downloading 
        encapsulations'
    key, encap = encapsulate(cpkey)
```

```
server.send(objectToBytes(encap, groupObj))

key = None
AESobj = None

#Start main loop
while 1:
    socket_list = [sys.stdin, server]
    read_sockets, write_sockets, error_sockets = select.
        select(socket_list , [], [])
    for sock in read_sockets:
        #incoming message from remote server
        if sock == server:
            data = sock.recv(4096)
            if not data :
                print '\nDisconnected from chat server'
                sys.exit()
            elif data[:7] == "1000001": #unique tag
                notifying about a new client
                encapsulations = receiveEncapsulations()
                key = generateSessionKey(encapsulations)
                    [-16:]
                print('Generated session key: {}'.format(
                    key))
                AESobj = AES.new(key, AES.MODE_CBC, 'This 
                    is an IV456')
                prompt()
            else :
                printMessage(data)

        #user entered a message
        else :
            msg = sys.stdin.readline()
            server.send(encrypt(key, msg.encode('utf-8')))
            prompt()
```