Lab Project

Attribute-based Access Control: Healthcare Scenario

(Cryptography-based Approach)

Version 2.1

LERSAIS

School of Computing and Information

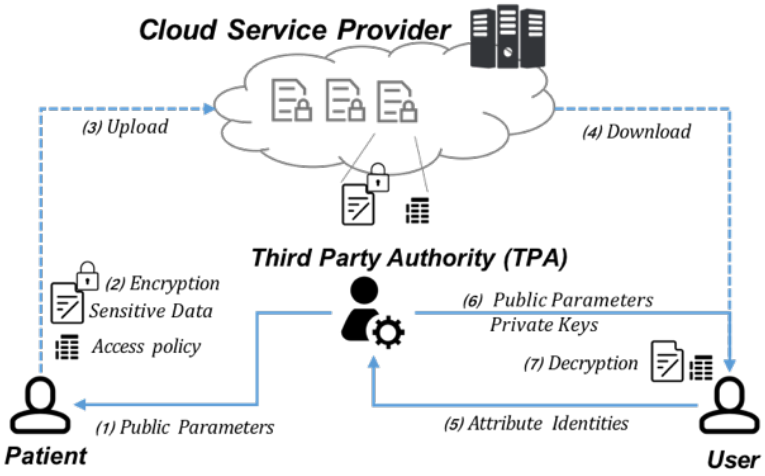
University of Pittsburg

This lab is designed by *Runhua Xu* and also part of lab module *Electronic Health Record Security* in SAHI Project

Part I: Introduction and A Motivating Scenario

Read the following scenario of healthcare application: a patient-centric health application.

Suppose that a patient/user-centric health application allows a patient/user to store and manage all of his Electronic Health Records (EHRs) by storing them in a Cloud Service Provider (CSP). The CSP is assumed to be honest-but-curious. Using an attribute-based access control (cryptography approach), a patient stores his EHRs in cloud storage. A typical usage scenario of attribute-based encryption is depicted as follows:



A patient encrypts his/her EHRs with a specified access policy and then sends to the CSP. With a provided link, anyone in the hospital can download the outsourced encrypted EHRs. However, only a user who has attributes that satisfy the access policy can access(decrypt) the encrypted EHRs.

In this lab, you will develop an advanced crypto system that supports attribute-based access control to protect the users' sensitive EHR data and provide access control features.

The Goal and Roadmap.

- Learn the usage of attribute-based encryption via a specific CP-ABE toolkit in the above scenario.
- Learn the usage of Charm-Crypto lib (How to develop a specific scheme by adopting Charm-Crypto).
- Use Charm-Crypto to develop a simple public key crypto scheme.
- Use Charm-Crypto to develop a CP-ABE scheme.

The recommended environment for this lab is Xnix OS, e.g., Unix, Linux, Mac OS. If you don't have them, try to use a virtual machine. It will help you save time dealing with environment setting/configuration.

Part II: An Overview of Attribute-based Access Control in Cryptography Approach

2.1 Environment Setting

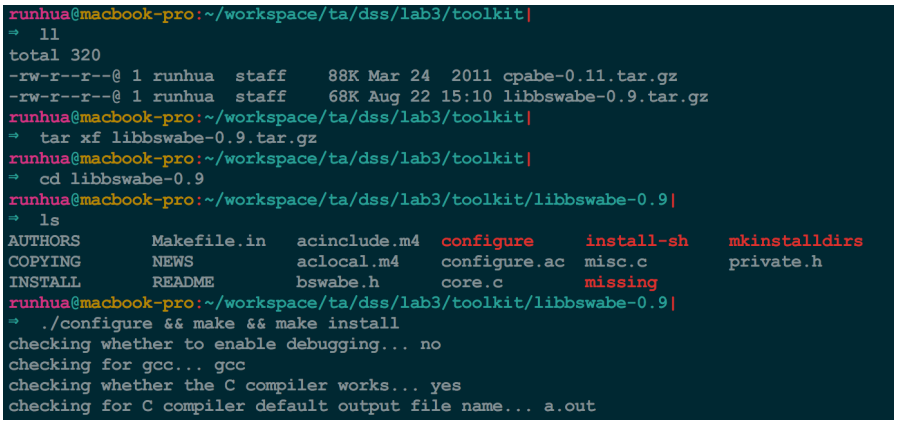
First, verify that you have installed the following dependencies:

- [GMP 5.x] (http://gmplib.org/)
- [OPENSSL](http://www.openssl.org/source/)
- [PBC](http://crypto.stanford.edu/pbc/download.html)

If your environment does not include such libs, we have provided them in the tools folder. Install GMP and OPENSSL lib first and then the PBC lib. You can download the last version from the official website. Note that the CPABE toolkit might not compile against versions of PBC older than 0.5.4.

Then, install the CP-ABE toolkit. You may follow the instructions as follows:

- Install libbswabe



- Install CP-ABE toolkit

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit|
⇒  ll
total 320
-rw-r--r--@  1 runhua  staff    88K Mar 24  2011 cpabe-0.11.tar.gz
drwxr-xr-x@ 26 runhua  staff   832B Aug 22 16:47 libbswabe-0.9
-rw-r--r--@  1 runhua  staff    68K Aug 22 15:10 libbswabe-0.9.tar.gz
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit|
⇒  tar xf cpabe-0.11.tar.gz
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit|
⇒  cd cpabe-0.11
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit/cpabe-0.11|
⇒  ls
AUTHORS               aclocal.m4            cpabe-enc.1           dec.c                 policy_lang.h
COPYING               common.c              cpabe-enc.more-man    enc.c                 policy_lang.y
INSTALL               common.h              cpabe-keygen.1        install-sh            setup.c
Makefile.in           configure             cpabe-keygen.more-man keygen.c              test-lang.c
NEWS                  configure.ac          cpabe-setup.1         missing
README                cpabe-dec.1           cpabe-setup.more-man  mkinstalldirs
acinclude.m4          cpabe-dec.more-man    cpabe.h               policy_lang.c
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit/cpabe-0.11|
⇒  ./configure && make && make install
checking whether to enable debugging... no
checking for bison... bison -y
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
```

When you install the CP-ABE toolkit successfully, you will have following four tools that can be executed in the terminal environment.

- **cpabe-setup** – generates a public key and a master secret key
- **cpabe-keygen** – generates a private key with a given set of attributes
- **cpabe-enc** – encrypts a file according to a policy, which is an expression in terms of attributes
- **cpabe-dec** – decrypts a file using a private key

To check the availability of these tools, you can check the version, e.g.,

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit|
⇒  cpabe-setup -v
cpabe-setup 0.11

Parts Copyright (C) 2006, 2007 John Bethencourt and SRI International.
This is free software released under the GPL, see the source for copying
conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.

Report bugs to John Bethencourt <bethenco@cs.berkeley.edu>.
```

To find the specific usage of these tools, check the manual, e.g.,

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/toolkit|
⇒  cpabe-enc -h
Usage: cpabe-enc [OPTION ...] PUB_KEY FILE [POLICY]

Encrypt FILE under the decryption policy POLICY using public key
PUB_KEY. The encrypted file will be written to FILE.cpabe unless
the -o option is used. The original file will be removed. If POLICY
is not specified, the policy will be read from stdin.

Mandatory arguments to long options are mandatory for short options too.
```

2.2 Adopt CP-ABE toolkit in the application scenario

Here, we first show you a demonstration of the CP-ABE toolkit, then you are required to demonstrate to use the CP-ABE toolkit in the healthcare application scenario.

Example Scenario

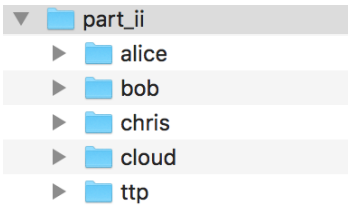
Suppose that in a company an employee Alice who wants to share a secret report to a group of employees who should satisfy one of the following conditions:

- *an employee who is the system administrator and is also in the security group*
- *an employee who is in the market group and also satisfies two of the following requirements:*
    1) *Her/His executive level should be 5*
    2) *She/He should be conjunctively enrolled in audit group*
    3) *She/He should be conjunctively enrolled in strategy group*

Try this example scenario first.

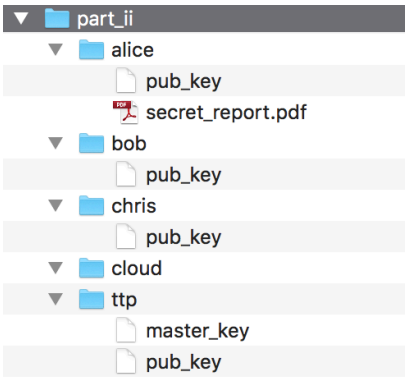
Demonstration Steps:

1) *Create five folders to represent five parties*



2) *The Third Part Authority initializes the system and delivers the public key*

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii|
⇒  cd ttp
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  cpabe-setup
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  ls
master_key pub_key
```

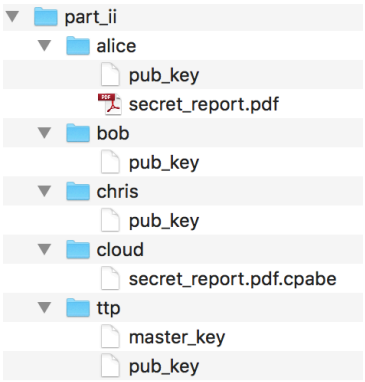


3) *Alice designs the following access policy based on the attribute identities:*

   (system_admin and security_group) or (market_group and 2 of (executive_level = 5, audit_group, strategy_group))

   Then, she encrypts the secret report and outsources it to the cloud:

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/alice|
⇒  ll
total 576
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
-rw-r--r--@ 1 runhua  staff   282K Dec  5  2017 secret_report.pdf
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/alice|
⇒  cpabe-enc pub_key secret_report.pdf -k -o ../cloud/secret_report.pdf.cpabe '(system_admin and security_group) or (market
_group and 2 of (executive_level = 5, audit_group, strategy_group))'
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/alice|
⇒  ll .
total 576
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
-rw-r--r--@ 1 runhua  staff   282K Dec  5  2017 secret_report.pdf
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/alice|
⇒  ll ../cloud
total 576
-rw-r--r--  1 runhua  staff   284K Aug 22 20:56 secret_report.pdf.cpabe
```



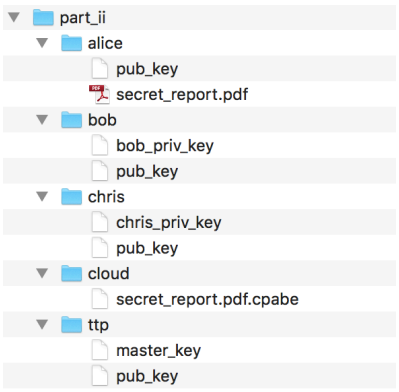
(Note that we just output the encrypted file "secret_report.pdf.cpabe" to the cloud folder instead of outsourcing to the real cloud for the purpose of simulation.)

4) *Suppose that bob has attribute identities: security_group, market_group, executive_level = 5, strategy_group; while Chris has attribute identities: system_admin, market_group, audit_group. Thus, according to our manual verification, we can learn that Bob can access the secret report while Chris cannot access the secret report. Now we verify this statement by using CP-ABE toolkit.*

5) *Bob and Chris apply their secret key from the TTP.*

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  ll
total 16
-rw-r--r--  1 runhua  staff   156B Aug 22 17:54 master_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:54 pub_key
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  cpabe-keygen -o ../bob/bob_priv_key pub_key master_key security_group market_group 'executive_level = 5' strategy_group
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  cpabe-keygen -o ../chris/chris_priv_key pub_key master_key system_admin market_group audit_group
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  ll ../bob
total 64
-rw-r--r--  1 runhua  staff    25K Aug 22 21:12 bob_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/ttp|
⇒  ll ../chris
total 16
-rw-r--r--  1 runhua  staff   966B Aug 22 21:13 chris_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
```

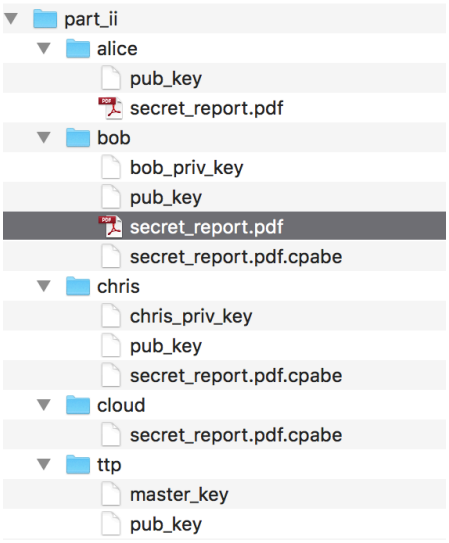
(Note that we just output the generated private key to the folder of Bob and Chris instead of sending through the secure channel.)



6) *Bob and Chris download the encrypted file from the cloud and decrypt it to access by using their private key, respectively.*

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/bob|
⇒  ll
total 64
-rw-r--r--  1 runhua  staff    25K Aug 22 21:12 bob_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/bob|
⇒  cp ../cloud/secret_report.pdf.cpabe .
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/bob|
⇒  ll
total 640
-rw-r--r--  1 runhua  staff    25K Aug 22 21:12 bob_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
-rw-r--r--  1 runhua  staff   284K Aug 22 21:17 secret_report.pdf.cpabe
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/bob|
⇒  cpabe-dec pub_key bob_priv_key secret_report.pdf.cpabe -k
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/bob|
⇒  ll
total 1208
-rw-r--r--  1 runhua  staff    25K Aug 22 21:12 bob_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
-rw-r--r--  1 runhua  staff   282K Aug 22 21:18 secret_report.pdf
-rw-r--r--  1 runhua  staff   284K Aug 22 21:17 secret_report.pdf.cpabe
```

```
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/chris|
⇒  ll
total 16
-rw-r--r--  1 runhua  staff   966B Aug 22 21:13 chris_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/chris|
⇒  cp ../cloud/secret_report.pdf.cpabe .
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/chris|
⇒  cpabe-dec pub_key chris_priv_key secret_report.pdf.cpabe -k
cannot decrypt, attributes in key do not satisfy policy
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_ii/chris|
⇒  ll
total 592
-rw-r--r--  1 runhua  staff   966B Aug 22 21:13 chris_priv_key
-rw-r--r--  1 runhua  staff   888B Aug 22 17:56 pub_key
-rw-r--r--  1 runhua  staff   284K Aug 22 21:19 secret_report.pdf.cpabe
```



Now, it is your turn to demonstrate the application in the healthcare example. Consider the following access requirements.

Suppose that a patient Alice who wants to share an EHR document to a group of staffs who work at the hospital. However, the staff should satisfy one of the following conditions:

- *a staff who is the senior physician*
- *a staff who is the junior physician but with privilege level 5*
- *a staff who is a pharmacist and also satisfies two of the following requirements:*
    1) *the pharmacist works in Pittsburgh area*
    2) *the pharmacist has privilege level 5*
    3) *the pharmacist works at UPMC*

Q1: Write similar steps as the example above. You need to abstract proper attribute identities first and then design the access policy. Then, give two test cases: one to demonstrate the access successfully, the other to demonstrates a failed access.

Part III: Develop a Simple Crypto Scheme with Charm-Crypto

In this part, you will implement a simple public key crypto scheme with Charm-Crypto.

3.1 Introduction of Charm-Crypto

Charm is a framework for rapidly prototyping advanced cryptosystems. Based on the **Python language**, it was designed from the ground up to minimize development time and code complexity while promoting the reuse of components.

Charm uses a hybrid design: performance intensive mathematical operations are implemented in native C modules, while cryptosystems themselves are written in a readable, high-level language. Charm additionally provides a number of new components to facilitate the rapid development of new schemes and protocols.

You can visit Official website via http://charm-crypto.io/ or Github via https://github.com/JHUISI/charm

3.2 Environment Setting

First, verify that you have installed the following dependencies:

- [GMP 5.x](http://gmplib.org/)
- [PBC](http://crypto.stanford.edu/pbc/download.html)
- [OPENSSL](http://www.openssl.org/source/)

You can follow the installation instructions from Github. Here is an example.

```
runhua@macbook-pro:~/workspace/ta/dss/lab3|
⇒  mkdir part_iii
runhua@macbook-pro:~/workspace/ta/dss/lab3|
⇒  cd part_iii
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_iii|
⇒  git clone https://github.com/JHUISI/charm
Cloning into 'charm'...
remote: Counting objects: 32469, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 32469 (delta 0), reused 1 (delta 0), pack-reused 32464
Receiving objects: 100% (32469/32469), 16.74 MiB | 7.88 MiB/s, done.
Resolving deltas: 100% (16522/16522), done.
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_iii|
⇒  cd charm
runhua@macbook-pro:~/workspace/ta/dss/lab3/part_iii/charm|dev
⇒  ./configure.sh --enable-darwin && make install && make test
```

Note that the option "--enable-darwin" is for Mac OS X system. The command "make install" and command "make test" may require super-user privileges of your system.

Q2: Show that you have installed charm successfully and pass the test session by attaching screenshots.

3.3 Use Charm-Crypto

The first stage of new scheme development is selecting the appropriate group to instantiate a scheme. Modern cryptographic algorithms are typically implemented on top of mathematical groups based on certain hardness assumptions (e.g., Diffie-Hellman).

At the moment, there are three cryptographic settings covered by Charm: *integergroups*, *ecgroups*, and *pairinggroups*.

- To initialize a group in the elliptic curve (EC) setting, refer to the *toolbox.eccurve* for the full set of identifiers and supported NIST approved curves (e.g., *prime192v1*).
- For EC with billinear maps (or pairings), Charm provides a set of identifiers for both symmetric and asymmetric type of curves. For example, the *'SS512'* represents a symmetric curve with a 512-bit base field and *'MNT159'* represents an asymmetric curve with 159-bit base field. Note that these curves are of prime order.
- Finally, for integer groups, typically defining large primes *p* and *q* is enough to generate an RSA group.

You can learn more about these from Charm-Crypto Documentation.

Here are detailed examples below for integer and pairing groups:

- Move to your workspace folder and create a python file to generate a random integer from *IntegerGroup* and execute it to get results, as an example below.

integer_group.py

```
1  from charm.toolbox.integergroup import IntegerGroup
2  integer_group = IntegerGroup()
3  integer_group.paramgen(1024)
4  g = integer_group.randomGen()
5  print (g) # if you use python 2.x, the print function may be different.
```

Q3: Show your generated integer from the IntegerGroup.

- Create a python file to generate a random element from *PairingGroup* and execute it to get the results, as follows.

pairing_group.py

```
1  from charm.toolbox.pairinggroup import PairingGroup,ZR,G1,G2,GT
2  pairing_group = PairingGroup('SS512')
3  g1 = pairing_group.random(G1)
4  g2 = pairing_group.random(G2)
5  g3 = pairing_group.random(ZR)
6  print (g1)
7  print (g2)
8  print (g3)
```

Q4: Show your generated elements from the PairingGroup.

3.4 Implement a Simple PKC scheme

Here you need to implement a simple public key encryption scheme proposed by Cramer-Shoup in 1998. The paper is here, if you are interested. http://knot.kaist.ac.kr/seminar/archive/46/46.pdf

The CS-PKC includes three algorithms: key generator, encryption, decryption.

Here we only provide scheme construction as follows:

Assumption: *We assume that we have a group G of prime order q, where q is large. We also assume that messages are (or can be encoded as) elements of G. We also use a universal one-way family of hash functions that map long bit strings to elements of Zq.*

Key Generation. The key generation algorithm runs as follows. Random elements $g_1, g_2 \in G$ are chosen, and random elements

$$x_1, x_2, y_1, y_2, z \in \mathbf{Z}_q$$

are also chosen. Next, the group elements

$$c = g_1^{x_1} g_2^{x_2}, \ d = g_1^{y_1} g_2^{y_2}, \ h = g_1^z$$

are computed. Next, a hash function H is chosen from the family of universal one-way hash functions. The public key is (g_1, g_2, c, d, h, H), and the private key is (x_1, x_2, y_1, y_2, z).

Encryption. Given a message $m \in G$, the encryption algorithm runs as follows. First, it chooses $r \in \mathbf{Z}_q$ at random. Then it computes

$$u_1 = g_1^r, \ u_2 = g_2^r, \ e = h^r m, \ \alpha = H(u_1, u_2, e), \ v = c^r d^{r\alpha}.$$

The ciphertext is

$$(u_1, u_2, e, v).$$

Decryption. Given a ciphertext (u_1, u_2, e, v), the decryption algorithm runs as follows. It first computes $\alpha = H(u_1, u_2, e)$, and tests if

$$u_1^{x_1+y_1\alpha} u_2^{x_2+y_2\alpha} = v.$$

If this condition does not hold, the decryption algorithm outputs "reject"; otherwise, it outputs

$$m = e/u_1^z.$$

We have provided the skeleton of such a PKC CS98 scheme (lab_pkc_cs98.py) with implementation of Key Generation algorithm. Compare the existing implementation with above algorithm descriptions to complete the *Encryption* and *Decryption* algorithms. Then run the test file (test_pkc_cs98.py) to check your work. (Note that you need to keep both scheme file and test file in the same folder)

Q5: Submit your implementation and show the screenshot of the result(s) showing that your work has passed the test case.

Part IV: Attribute based Encryption and Implementation

In this part, you will need to implement a more complex public key encryption scheme, i.e., ciphertext-policy attribute-based encryption (CP-ABE) scheme. Such a scheme could be employed as attribute-based access control framework for outsourced data in the public cloud, as in the case of the example scenario above.

Specifically, you will implement an ABE scheme proposed in the following research paper.

> *Waters, Brent. "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization." Public Key Cryptography. Vol. 6571. 2011.*

The paper is also attached. You are encouraged to read the paper. For this lab, the essential parts are as below.

Overview of Ciphertext-Policy ABE.

A ciphertext-policy attribute based encryption scheme consists of four algorithms: Setup, Encrypt, KeyGen, and Decrypt.

Setup(λ, U)**.** The setup algorithm takes security parameter and attribute universe description as input. It outputs the public parameters PK and a master key MK.

Encrypt$(\text{PK}, M, \mathbb{A})$**.** The encryption algorithm takes as input the public parameters PK, a message M, and an access structure $\mathbb{A}$ over the universe of attributes. The algorithm will encrypt M and produce a ciphertext CT such that only a user that possesses a set of attributes that satisfies the access structure will be able to decrypt the message. We will assume that the ciphertext implicitly contains $\mathbb{A}$.

Key Generation(MK, S)**.** The key generation algorithm takes as input the master key MK and a set of attributes S that describe the key. It outputs a private key SK.

Decrypt$(\text{PK}, \text{CT}, \text{SK})$**.** The decryption algorithm takes as input the public parameters PK, a ciphertext CT, which contains an access policy $\mathbb{A}$, and a private key SK, which is a private key for a set S of attributes. If the set S of attributes satisfies the access structure $\mathbb{A}$ then the algorithm will decrypt the ciphertext and return a message M.

Implementation.

Setup(U) The setup algorithm takes as input the number of attributes in the system. It then chooses a group $\mathbb{G}$ of prime order p, a generator g and U random group elements $h_1, \ldots, h_U \in \mathbb{G}$ that are associated with the U attributes in the system. In addition, it chooses random exponents $\alpha, a \in \mathbb{Z}_p$.

The public key is published as

$$\text{PK} = \quad g,\ e(g,g)^{\alpha},\ g^{a},\ h_1, \ldots, h_U.$$

The authority sets $\text{MSK} = g^{\alpha}$ as the master secret key.

KeyGen(MSK, S) The key generation algorithm takes as input the master secret key and a set S of attributes. The algorithm first chooses a random $t \in \mathbb{Z}_p$. It creates the private key as

$$K = g^{\alpha} g^{at} \quad L = g^{t} \quad \forall x \in S \; K_x = h_x^t.$$

Encrypt(PK, (M, ρ), $\mathcal{M}$) The encryption algorithm takes as input the public parameters PK and a message $\mathcal{M}$ to encrypt. In addition, it takes as input an LSSS access structure (M, ρ). The function ρ associates rows of M to attributes.

Let M be an $\ell \times n$ matrix. The algorithm first chooses a random vector $\vec{v} = (s, y_2, ..., y_n) \in \mathbb{Z}_p^n$. These values will be used to share the encryption exponent s. For $i = 1$ to ℓ, it calculates $\lambda_i = \vec{v} \cdot M_i$, where M_i is the vector corresponding to the ith row of M. In addition, the algorithm chooses random $r_1, \ldots, r_\ell \in \mathbb{Z}_p$.

The ciphertext is published as CT =

$$C = \mathcal{M} e(g,g)^{\alpha s}, \; C' = g^s, (C_1 = g^{a\lambda_1} h_{\rho(1)}^{-r_1}, \; D_1 = g^{r_1}), \ldots, (C_\ell = g^{a\lambda_\ell} h_{\rho(\ell)}^{-r_n}, \; D_\ell = g^{r_\ell})$$

along with a description of (M, ρ).

Decrypt(CT,SK) The decryption algorithm takes as input a ciphertext CT for access structure (M, ρ) and a private key for a set S. Suppose that S satisfies the access structure and let $I \subset \{1, 2, \ldots, \ell\}$ be defined as $I = \{i : \rho(i) \in S\}$. Then, let $\{\omega_i \in \mathbb{Z}_p\}_{i \in I}$ be a set of constants such that if $\{\lambda_i\}$ are valid shares of any secret s according to M, then $\sum_{i \in I} \omega_i \lambda_i = s$. (Note there could potentially be different ways of choosing the ω_i values to satisfy this.)

The decryption algorithm first computes

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

The decryption algorithm can then divide out this value from C and obtain the message $\mathcal{M}$.

We have provided the skeleton of such an ABE scheme (lab_abe.py) with implementation of the Setup algorithm. Compare the existing implementation with above algorithm descriptions to complete the *KeyGen, Encryption* and *Decryption* algorithms. Then run the test file (test_abe.py) to check your work. (Note that you need to keep both the scheme file and the test file in the same folder)

Q6: Submit your implementation and show the screenshot of result(s) showing that your work has passed the test case.