

## 一、实验要求

1. 学习掌握密钥策略属性基加密(KP-ABE)算法原理。
2. 代码实现密钥策略属性基加密(KP-ABE)算法。
3. 随机指定明文数据，能够输出访问策略 Access policy、属性集 Attribute Set、密文 Ciphertext、密钥 Private key 和解密结果 Decryption results。

## 二、实验设备

主机：Windows10

工具：IntelliJ Idea

## 三、实验原理

### 1. 访问控制机制

访问控制是指对用户合法使用资源的认证和控制。

目前，对信息系统的访问控制主要采用基于角色的访问控制机制( Role Based Access Control, RBAC)及其扩展模型。RBAC 机制主要由 Sandhu 于 1996 年提出的基本模型 RBAC96 构成，其认证过程为：一个用户先由系统分配一个角色，如管理员、普通用户等；登录系统后，根据对用户角色所设置的访问策略实现对资源的访问。显然，同样的角色可以访问同样的资源。RBAC 机制是一种基于互联网的 OA 系统、银行系统、网上商店系统等的访问控制方法，是基于用户的。

对物联网而言，末端是感知网络，即可能是一个感知结点或一个物体，采用用户角色的形式进行资源控制显然不够灵活。

基于属性的访问控制(Attribute Based Access Control, ABAC)是近几年研究的热点, 若将角色映射成用户的属性, 可以构成 ABAC 与 RBAC 的对等关系, 而且属性的增加相对简单, 同时基于属性的加密算法可以使 ABAC 得以实现。ABAC 方法的问题是对较少的属性来说, 加密解密效率较高, 但随着属性数量的增加, 加密的密文长度将增加, 使算法的实用性受到限制。目前有两个发展方向, 即基于密钥策略和基于密文策略, 其目标均是改善基于属性的加密算法的性能。

## 2. 基本的属性基加密 (如 FIBE 算法)

基本的属性基加密将密文和密钥都与一组属性关联起来, 当密文与密钥之间至少有  $d$  个属性重合时, 用户就可以解密密文。虽然这个策略对于生物识别的容错加密有一定的作用, 但访问控制缺乏灵活性限制了它的应用。

## 3. 访问架构 (策略)

访问结构 (access structure) 是安全系统研究的术语, 系统的访问结构是指被授权的集合的结构。

## 4. 属性基加密系统的两种策略

KP-ABE (key policy attribute based encryption), 即密钥策略属性基加密系统。密钥策略属性基加密系统, 将策略嵌入到用户密钥中, 属性嵌入到密文中, 即密钥 (私钥) 对应于一个访问控制而密文对应于一个属性集合, 当且仅当属性集合中的属性 (主要为数据/文件属性) 能够满足此访问结构才能解密得出最

终明文。这种设计比较接近静态场景，此时密文用与其相关的属性加密存放在服务器上，当允许用户得到某些消息时，就分配一个特定的访问策略给用户，即控制“人可以访问哪类数据”。KP-ABE 的常见应用有付费视频网站、日志加密管理和审计、广播加密等。

CP-ABE (ciphertext policy attribute based encryption)，即密文策略属性基加密系统。密文策略属性基加密系统，是将策略嵌入到密文中，属性嵌入到用户密钥中，即密文对应于一个访问结构而密钥（私钥）对应于属性集合，当且仅当属性集合中的属性（主要为用户属性）能够满足此访问结构才能解密得出最终明文。这种设计比较接近于现实中的应用场景，可以假象每个用户根据自身条件或者属性从属性机构得到密钥，然后加密者来制定对消息的访问控制。这对这份数据做了一个粒度可以细化到属性级别的加密访问控制，即“有哪些属性的人可以访问数据”。CP-ABE 的常见应用有隐私数据共享等访问类应用，如云上的数据加密存储与细粒度共享等。

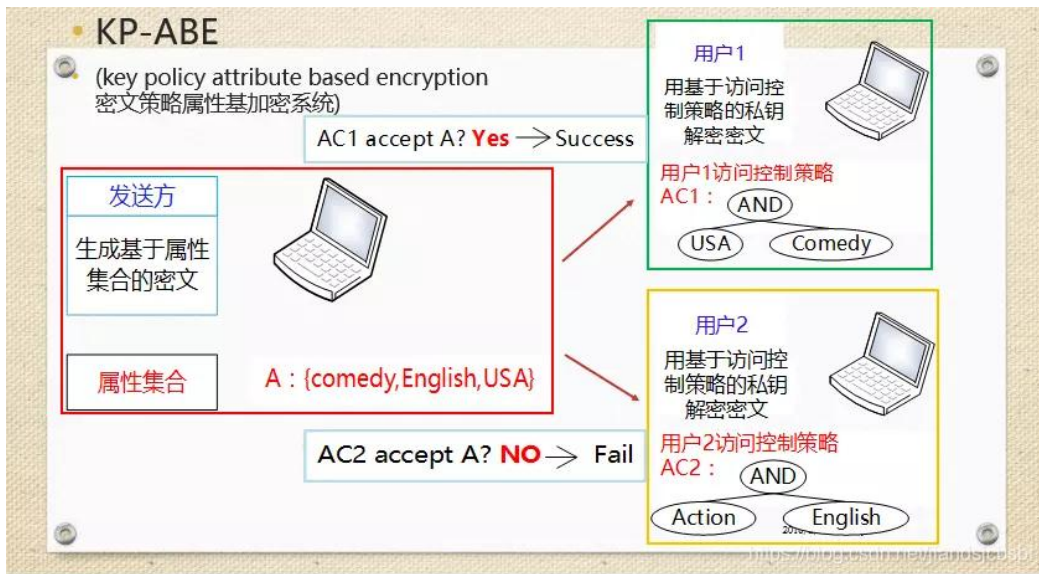
## 5. KP-ABE 算法原理

在 KP-ABE 中，与用户私钥相关联的访问结构被构造为访问树。一个访问树代表了一条解密控制策略，基于访问树的解密控制策略表述不仅支持门限方式的策略表达，也支持包含“或”、“且”逻辑运算的策略表述。

$\text{parent}(x)$ 表示树中除根节点之外的节点  $x$  的父节点。 $\text{children}(x)$ 表示除叶子结点之外的内部节点  $x$  的所有子节点。 $\text{num}(x)$ 是节点  $x$  的子节点个数。 $\text{index}(x)$ 是节点  $x$  在其所有兄弟节点中的序号也就是索引值，并且满足  $1 \leq \text{index}(x) \leq \text{num}(\text{parent}(x))$ 。 $\text{attr}(x)$ 是叶子节点  $x$  的属性。

门限方式的策略表达中，访问树的每一个内部节点  $x$  都是一个阈值门，由其子节点和阈值  $k_x$  描述。如果  $\text{num}(x)$  是其子节点的个数，则  $1 \leq k_x \leq \text{num}(x)$ 。与门和或门都可以被构造为阈值门，当  $k_x=1$  时，内部节点  $x$  就代表一个“或”门；当  $k_x=\text{num}(x)$  时，内部节点  $x$  就代表一个“与”门。叶子节点与属性相关联，由属性值和阈值  $k_x=1$  描述。也就是说对于根节点来说，其叶子节点中的  $n$  个属性只要满足  $k_x$  个即可解密，否则无法解密数据。

算法上用  $T_x$  表示以节点  $x$  为根的子树，则根为  $r$  的访问树表示为  $T_r$ ，当节点被满足时令  $T_x=1$ ，我们递归计算所有节点，当根节点满足条件时，我们可以进行解密操作，否则解密失败。



## 6. KP-ABE 具体算法流程

环境基础：  $G_1$  是一个素数阶  $p$  的乘法循环群，  $g$  是群的一个生成元，  $e: G_1 \times G_1 \rightarrow G_T$  表示双线性映射，  $k$  是一个能决定群大小的安全参数，同时定义拉格朗日系数：  $\Delta_{i,S}(x) = \prod_{j \in S, j \neq i} \frac{x-j}{i-j}$ ，其中  $i \in \mathbb{Z}_p$ （整数域上的加法循环群），  $S$  是属于  $\mathbb{Z}_p$  的成员集。所有的属性集合为  $U$ 。

### ① setup 初始化

- (1) 生成 pairing 相关公共参数  $\langle e, g, G_1, G_T, Z_p \rangle$ 。
- (2) 确定属性全集  $U$  为整数集合  $\{1, 2, \dots, |U|\}$ 。
- (3) 针对每个属性  $i$  在  $Z_p$  中随机均匀选取  $|U|$  个值  $\{t_1, t_2, \dots, t_{|U|}\}$ , 作为主密钥组件, 计算  $T_i = g^{t_i}$  ( $i \in U$ ) 作为对应的公钥组件。
- (4) 选取随机数  $y \in Z_p$ , 并计算  $Y = e(g, g)^y$ 。
- (5) 最终, 系统主密钥  $msk = \{t_1, t_2, \dots, t_{|U|}, y\}$ , 公钥  $pk = \{T_1, T_2, \dots, T_{|U|}, Y\}$ 。

### ② keygen 密钥生成 (与访问控制树相关联)

- (1) 对于访问树  $T$ , 根节点记为  $r$ , 每个非叶子节点  $x$  都有一个门限值 (节点阈值)  $k_x$ , 而每个叶子节点都对应一个属性。
- (2) 从根节点  $r$  开始, 自上而下为树中的每个非叶子节点  $x$  选择一个  $d_x = k_x - 1$  阶多项式  $q_x(x)$ 。对于根节点则选择  $q_r(0) = y$  (即拉格朗日差值法中的  $a_0$ ), 然后  $d_r$  次多项式  $q_r$  的其它点完全随机的选取。其它内部节点的多项式要满足  $q_x(0) = q_{\text{Parent}(x)}(\text{index}(x))$ , 其它点随机选取来定义  $q_x$ 。
- (3) 多项式被定义好之后, 对于叶子节点  $x$ , 我们将以下的秘密值给用户, 即用户的属性私钥从树中叶子节点抽取得到:  $D_x = g^{\frac{q_x(0)}{t_i}}$  ( $i = \text{att}(x)$ )。
- (4) 用户私钥为  $sk = \{D_i\}$  ( $i \in S$ )。

### ③ encrypt 加密

- (1) 选取随机数  $s \in Z_p$ , 针对明文消息  $M \in G_T$ , 计算  $E' = M \cdot Y^s = M \cdot e(g, g)^{ys}$ 。
- (2) 针对明文属性集合  $W$  中的每个属性  $i$ , 计算  $E_i = T_i^s$ 。
- (3) 密文为  $ct = \{E', \{E_i\}\} (i \in W)$ 。

#### ④ decrypt 解密

(1) 如果访问树存在，则可以进行递归解密操作。

(2) 如果  $x$  是叶子节点，令  $i = \text{att}(x)$ ，计算  $F = P_i = e(E_i, D_x) = e(\frac{q_x(0)}{t_i}, g^{t_i s}) = e(g, g)^{q_x(0)s} = e(g, g)^{sy}$ 。

(3) 如果  $x$  是非叶子节点，则递归进行如下操作计算：

(i) 对  $x$  的所有孩子节点  $z$ ，索引值为  $i = \text{index}$ ，令  $S_x$  为一个由  $x$  的任意  $k_x$  个子节点组成的集合，这些节点要满足对这些孩子节点进行递归解密结果不为空，也就是说密文属性（与加密前明文属性一致）与所有叶子节点属性重合数量要满足每个内部节点的门限值。如果不存在这样的集合，则解密输出为空，即解密失败，否则进行下一步运算。

(ii) 计算  $P_i = e(E_i, D_x)^{\Delta_{i, S_x}(0)} = e(g, g)^{sq_x(i)\Delta_{i, S}(0)}$ 。

(iii) 连乘运算  $F = \prod_{i \in \text{Index}(x)} P_i = e(g, g)^{s \sum_{i \in \text{Index}(x)} q_x(i)\Delta_{i, S}(0)} = e(g, g)^{sq_x(0)} = e(g, g)^{sy}$ 。

(4) 除法运算  $\frac{E'}{F} = \frac{M \cdot Y^s}{F} = \frac{M \cdot e(g, g)^{ys}}{e(g, g)^{ys}} = M$ ，得到明文消息  $M$ 。

## 四、实验流程

### 【4.1】备注

本实验将复现密钥策略属性基加密系统（KP-ABE）算法，实现过程中有几点备注如下：

1. 实现过程中所有属性均用整数表示。在实际应用中是通过索引表将每一个整数和一个字符串属性对应起来。

2. 多项式求值和拉格朗日插值在群  $Z_p$ （在 JPBC 库中表示为  $Zr$ ）上进行，

因此相应的 `int` 值在计算前要转换为 `Zr Element`。

3. 对于重复使用的值一定要记得使用 `getImmutable()` 或者 `duplicate()`。尤其是在 `for` 循环中。

4. 使用从文件 `pairingParametersFileName`（实例中为 `a.properties` 文件）中读取相关参数的方法完成对双线性群（即椭圆曲线）的初始化即在 `JPBC` 中对 `Pairing` 对象初始化。

5. 生成的公钥 `pk`、主密钥 `msk`、用户私钥 `sk`、密文 `ct` 集合也都放入对应的文件 `pk.properties`、`msk.properties`、`sk.properties`、`ct.properties` 中。

6. 本实验中的明文消息在最初选用的是随机生成为 `GT` 群上的点，这样在解密后可以还原出对应的点。后来又选用了 `JPBC` 库里的方法通过明文编码将明文字符串转换为字节数组，再由该字节数组来实例化一个 `GT` 群上的点 `Element` 类，进而继续下一步加密，这样也能够正确解密出明文消息串。缺点是：① 实例化的 `GT` 群上的点 `y` 值为 0，`x` 值的长度随明文消息串长度而改变，对明文消息串的处理安全性较低；② 解密出来的消息有冗余信息。

7. 需要加密的明文消息串从 `input.txt` 文件中读取，解密出来的结果写入到 `output.txt` 文件中。

## 【4.2】具体实现

### 1. `setup` 初始化

```

29 public static void setup(String pairingParametersFileName, int U, String pkFileName, String mskFileName) {
30     //输入为 < 相关初始化参数文件名, 属性全集U (整数), 公钥集文件名, 主密钥集文件名 >
31     Pairing bp = PairingFactory.getPairing(pairingParametersFileName); //从文件导入椭圆曲线参数, 生成Pairing实例
32     Element g = bp.getG1().newRandomElement().getImmutable(); //基于基本参数, 从G1乘法循环群中随机选取Element整数生成元g (阶数)
33
34     Properties mskProp = new Properties(); //新建Properties类以便生成对应主密钥msk封装文件
35     Properties pkProp = new Properties(); //新建Properties类以便生成对应公钥pk封装文件
36     //属性表示为1, 2, 3, ..., U
37     //对每个属性i, 选取一个随机数ti作为该属性对应的主密钥, 并计算相应公钥g^ti
38     for (int i = 1; i <= U; i++){
39         Element t = bp.getZr().newRandomElement().getImmutable(); //Element整数ti需要从加法循环群Zp中随机选取
40         Element T = g.powZn(t).getImmutable(); //Element整数Ti=g^ti
41         mskProp.setProperty("t"+i, Base64.getEncoder().withoutPadding().encodeToString(t.toBytes()));
42         pkProp.setProperty("T"+i, Base64.getEncoder().withoutPadding().encodeToString(T.toBytes()));
43         //将所得ti、Ti转换为字符串形式并进行Base64编码, 并存入对应主密钥、公钥文件中
44     }
45     //另外选取一个随机数y, 计算e(g,g)^y
46     Element y = bp.getZr().newRandomElement().getImmutable(); //Element整数y需要从加法循环群Zp中随机选取
47     Element egg_y = bp.pairing(g, g).powZn(y).getImmutable(); //基于椭圆曲线基本参数计算Y=egg_y=e(g,g)^y
48
49     //将msk和pk存储到相应的文件中
50     mskProp.setProperty("y", Base64.getEncoder().withoutPadding().encodeToString(y.toBytes()));
51     pkProp.setProperty("egg_y", Base64.getEncoder().withoutPadding().encodeToString(egg_y.toBytes()));
52     pkProp.setProperty("g", Base64.getEncoder().withoutPadding().encodeToString(g.toBytes()));
53     //将所得y、Y=egg_y=e(g,g)^y、g转换为字符串形式并进行Base64编码, 并存入对应主密钥、公钥、公钥文件中
54     //注意区分数据类型。上面写的数据类型群元素, 因此使用了Base64编码。
55
56     storePropToFile(mskProp, mskFileName); //封装进对应文件
57     storePropToFile(pkProp, pkFileName); //封装进对应文件
58
59     //输出: 系统主密钥文件 msk = { t1, t2, ..., t|U|, y }, 公钥文件 pk = { T1, T2, ..., T|U|, Y, g }
60 }

```

## 2. keygen 密钥生成

```

71 public static void keygen(String pairingParametersFileName, Node[] accessTree, String pkFileName, String mskFile
72     //输入为 < 相关初始化参数文件名, 访问控制树, 公钥文件名, 主密钥文件名, 私钥文件名 >
73     Pairing bp = PairingFactory.getPairing(pairingParametersFileName); //从文件导入椭圆曲线参数, 生成Pairing实例
74
75     Properties pkProp = loadPropFromFile(pkFileName); //从对应公钥pk封装文件载入公钥Properties实例
76     String gString = pkProp.getProperty("g"); //从对应公钥pk文件中获取生成元g
77     Element g = bp.getG1().newElementFromBytes(Base64.getDecoder().decode(gString)).getImmutable();
78     //将Base64编码后的字符串g解码转换成Element类整数
79
80     Properties mskProp = loadPropFromFile(mskFileName); //从对应主密钥msk封装文件载入主密钥Properties实例
81     String yString = mskProp.getProperty("y"); //从对应主密钥msk文件中获取随机数密钥y值
82     Element y = bp.getZr().newElementFromBytes(Base64.getDecoder().decode(yString)).getImmutable();
83     //将Base64编码后的字符串y解码转换成Element类整数
84
85     //先设置根节点要共享的秘密值
86     accessTree[0].secretShare = y;
87     //进行共享, 使得每个叶子节点获得响应的秘密分片, 使用编写的节点共享函数nodeShare()
88     nodeShare(accessTree, accessTree[0], bp);
89     //输入为 < 访问控制树的所有节点, 根节点or要分享秘密的节点, Pairing实例 >
90     //实质上是利用拉格朗日差值算法计算每个子节点的q(i)即秘密值, q1(0)=q(0)=y, 并输出 < 分享秘密后的访问控制树的所有节点 >
91     //解密时最后求出首项y=q(0)
92
93     Properties skProp = new Properties(); //新建Properties类以便生成对应私钥sk封装文件
94

```



```

94
95 //计算用户属性中每个属性对应的私钥 $Dx = g^{(qx(i)/ti)}$ ， $qx(i)$ 是多项式在该属性 $i$ 位置的值， $ti$ 是属性对应的主密钥
96 for (Node node : accessTree) {
97     if (node.isLeaf()) {
98         // 对于每个叶子节点，先获取对应的主密钥组件 $t$ ，然后计算密钥组件。
99         String tString = mskProp.getProperty("t"+node.att); //从对应主密钥msk文件中获取对应 $ti$ 的字符串
100         Element t = bp.getZr().newElementFromBytes(Base64.getDecoder().decode(tString)).getImmutable();
101         //将Base64编码后的字符串 $t$ 解码并转换成Element类整数
102         Element q = node.secretShare; //获取该节点的共享的秘数值
103         Element D = g.powZn(q.div(t)).getImmutable(); //计算Element类整数值 $Di = g^{(q(i)/ti)}$ 
104         skProp.setProperty("D"+node.att, Base64.getEncoder().withoutPadding().encodeToString(D.toBytes()));
105         //将所得 $Di$ 转换为字符串形式并进行Base64编码，并存入对应私钥文件中
106     }
107 }
108 //将用户访问树也添加到私钥中
109 //如何进行序列化和反序列化
110 //skProp.setProperty("userAttList", Arrays.toString(accessTree));
111 storePropToFile(skProp, skFileName); //封装进对应文件
112
113 //输出：私钥文件 sk = {  $Di$  } ( $i \in \text{userAttList}$ )
114 }

```

### 3. encrypt 加密

```

124 public static void encrypt(String pairingParametersFileName, Element message, int[] messageAttList, String pkFi
125 //输入为 < 相关初始化参数文件名, Element类明文信息 (GT上一点), 明文属性集, 公钥文件名, 密文文件名 >
126 Pairing bp = PairingFactory.getPairing(pairingParametersFileName); //从文件导入椭圆曲线参数, 生成Pairing实例
127
128 Properties pkProp = loadPropFromFile(pkFileName); //从对应公钥pk封装文件载入公钥Properties实例
129 String eggString = pkProp.getProperty("egg_y"); //从对应公钥pk文件中获取公钥 $egg\_y = Y = e(g, g)^y$ 
130 Element egg_y = bp.getGT().newElementFromBytes(Base64.getDecoder().decode(eggString)).getImmutable();
131 //将Base64编码后的字符串 $egg\_y$ 解码转换成Element类整数
132
133 //计算密文组件  $E' = EP = Me(g, g)^{(ys)}$ 
134 Element s = bp.getZr().newRandomElement().getImmutable(); //选取随机Element类整数 $s \in \mathbb{Z}_p$ 
135 Element EP = message.duplicate().mul(egg_y.powZn(s)).getImmutable();
136 //计算Element类整数 $E' = EP = M(Y^s) = Me(g, g)^{(ys)}$  ( $M$ 点进行双线性映射得到整数)
137
138 Properties ctProp = new Properties(); //新建Properties类以便生成对应密文ct封装文件
139 //针对每个密文属性，计算密文组件  $Ei = Ti^s$ 
140 for (int att : messageAttList) {
141     String TString = pkProp.getProperty("T"+att); //从对应公钥pk文件中获取对应 $Ti$ 的字符串
142     Element T = bp.getGT().newElementFromBytes(Base64.getDecoder().decode(TString)).getImmutable();
143     //将Base64编码后的字符串 $Ti$ 解码转换成Element类整数
144     Element E = T.powZn(s).getImmutable();
145     //针对每个明文属性，计算密文组件Element类整数  $Ei = Ti^s$ 
146
147     ctProp.setProperty("E"+att, Base64.getEncoder().withoutPadding().encodeToString(E.toBytes()));
148     //将所得密文组件 $Ei$ 转换为字符串形式并进行Base64编码，并存入对应密文文件中
149 }
150 ctProp.setProperty("EP", Base64.getEncoder().withoutPadding().encodeToString(EP.toBytes()));
151 //将Element类整数 $EP = M(Y^s) = Me(g, g)^{(ys)}$ 也转换为字符串形式并进行Base64编码，并存入对应密文文件中
152
153 //明文属性列表messageAttList也添加至密文文件中
154 ctProp.setProperty("messageAttList", Arrays.toString(messageAttList));
155 storePropToFile(ctProp, ctFileName); //封装进对应文件
156
157 //输出：密文文件 ct = <  $E' = EP$ ,  $\{Ei\} (i \in \text{messageAttList})$ , messageAttList >
158 }

```

### 4. decrypt 解密

```

169 public static Element decrypt(String pairingParametersFileName, Node[] accessTree, String pkFileName, String ctFile
170 //输入为 < 相关初始化参数文件名, 访问控制树, 公钥文件名, 密文文件名, 私钥文件名 >
171 Pairing bp = PairingFactory.getPairing(pairingParametersFileName); //从文件导入椭圆曲线参数, 生成Pairing实例
172
173 Properties pkProp = loadPropFromFile(pkFileName); //从对应公钥pk封装文件载入公钥Properties实例
174
175 Properties ctProp = loadPropFromFile(ctFileName); //从对应密文ct封装文件载入密文Properties实例
176 String messageAttListString = ctProp.getProperty("messageAttList"); //从对应密文ct文件中获取明文属性列表
177 //恢复明文消息的属性列表 int[]类型
178 int[] messageAttList = Arrays.stream(messageAttListString.substring(1, messageAttListString.length()-1).split(regex: "
179
180 Properties skProp = loadPropFromFile(skFileName); //从对应私钥sk封装文件载入私钥Properties实例
181 for (Node node : accessTree) {
182     if (node.isLeaf()) {
183         // 如果叶子节点的属性值属于属性列表, 则将属性对应的密文组件和私钥组件配对的结果作为秘密值
184         if (Arrays.stream(messageAttList).boxed().collect(Collectors.toList()).contains(node.att)){
185             String EString = ctProp.getProperty("E"+node.att); //从对应密文ct文件中获取密文组件Ei
186             Element E = bp.getG1().newElementFromBytes(Base64.getDecoder().decode(EString)).getImmutable();
187             //将Base64编码后的字符串Ei解码转换成Element类整数
188             String DString = skProp.getProperty("D"+node.att); //从对应私钥sk文件中获取用户每个属性对应的私钥Di
189             Element D = bp.getG1().newElementFromBytes(Base64.getDecoder().decode(DString)).getImmutable();
190             //将Base64编码后的字符串Di解码转换成Element类整数
191
192             // 这儿存在于密文属性列表中的叶子节点的秘密值是配对后的结果
193             //secretShare=e(Ei,Di)=e(g^(ti)*s), g^(qx(0)/(ti))=e(g,g)^(qx(0)*s)=e(g,g)^sy
194             node.secretShare = bp.pairing(E,D).getImmutable();
195         }
196     }
197 }
198
199 // 进行秘密恢复
200 boolean treeOK = nodeRecover(accessTree, accessTree[0], messageAttList, bp);
201 //
202 if (treeOK) {
203     String EPString = ctProp.getProperty("EP"); //从对应密文ct文件中获取EP=M(Y^s)=Me(g,g)^(ys)
204     Element EP = bp.getGT().newElementFromBytes(Base64.getDecoder().decode(EPString)).getImmutable();
205     //将Base64编码后的字符串Di解码转换成Element类整数
206
207     //恢复M=EP÷上述连乘结果 //恢复M=EP/denominator=M*e(g,g)^(ys)/e(g,g)^sy=M
208     Element res = EP.div(accessTree[0].secretShare);
209     return res;
210 }
211 else{
212     System.out.println("The access tree is not satisfied.");
213     return null;
214 }
215 }

```

## 5. 相关函数

### ① 随机选取多项式 $q(x)$ 的参数 $\text{coef}$ (即 $a_n$ )

```

217 //随机选取多项式参数randomP(d, a0, bp)→{ai}
218 //d-1次多项式表示为 $q(x)=\text{coef}[0] + \text{coef}[1]*x^1 + \text{coef}[2]*x^2 + \text{coef}[d-1]*x^{(d-1)}$ 
219 //多项式的系数的数据类型为 $Z_r$  Element, 从而是后续相关计算全部在 $Z_r$ 群上进行
220 //通过随机选取coef参数, 来构造d-1次多项式 $q(x)$ 。约束条件为 $q(0)=s$ 。
221 public static Element[] randomP(int d, Element s, Pairing bp) {
222     Element[] coef = new Element[d];
223     coef[0] = s;
224     for (int i = 1; i < d; i++){
225         coef[i] = bp.getZr().newRandomElement().getImmutable();
226     }
227     return coef;
228 }
229

```

### ② 计算由 $\text{coef}$ (即 $a_n$ ) 为系数确定的多项式 $q(x)$ 在点 $x$ 处的值

```

230 //计算多项式函数 $qx(x, \{a_i\}, bp) \rightarrow q(x)$ 
231 //计算由coef为系数确定的多项式qx在序号为index点处的值，注意多项式计算在群Zr上进行
232 public static Element qx(Element index, Element[] coef, Pairing bp){
233     Element res = coef[0].getImmutable();
234     for (int i = 1; i < coef.length; i++){
235         Element exp = bp.getZr().newElement(i).getImmutable();
236         //index一定要使用duplicate复制使用，因为index在每一次循环中都要使用，如果不加duplicate，index的值会发生变化
237         res = res.add(coef[i].mul(index.duplicate().powZn(exp)));
238         //q(x)=a[0]+a[1]x+a[2]x^2+...+a[n-1]x^(n-1)
239     }
240     return res;
241 }
242

```

### ③ 拉格朗日因子计算

```

243 //计算拉格朗日因子lagrange(i, S, x, bp)→δi
244 //拉格朗日因子计算 i是集合S中的某个元素，x是目标点的x值
245 public static Element lagrange(int i, int[] S, int x, Pairing bp) {
246     Element res = bp.getZr().newOneElement().getImmutable();
247     Element iElement = bp.getZr().newElement(i).getImmutable();
248     Element xElement = bp.getZr().newElement(x).getImmutable();
249     for (int j : S) {
250         if (i != j) {
251             //注意：在循环中重复使用的项一定要用duplicate复制出来使用
252             //这儿xElement和iElement重复使用，但因为前面已经getImmutable所以可以不用duplicate
253             Element numerator = xElement.sub(bp.getZr().newElement(j)); //x-xj
254             Element denominator = iElement.sub(bp.getZr().newElement(j)); //xi-xj
255             res = res.mul(numerator.div(denominator)); //(x-xj)/(xi-xj)的乘积 (i≠j)
256         }
257     }
258     return res;
259 }

```

### ④ 共享秘密

```

261 // 共享秘密
262 // nodes是整颗树的所有节点，n是要分享秘密的节点
263 //输入为 < 访问控制树的所有节点，根节点or要分享秘密的节点，Pairing实例 >
264 //根节点的q(0)=y
265 public static void nodeShare(Node[] nodes, Node n, Pairing bp){
266     // 如果是叶子节点，则不需要再分享
267     if (!n.isLeaf()){
268         // 如果不是叶子节点，则先生成一个随机多项式，多项式的常数项为当前节点的秘密值（这个值将被用于分享）
269         // 多项式的次数，由节点的gate对应的threshold即门限值t决定
270         Element[] coef = randomP(n.gate[0], n.secretShare, bp);
271         //即：由n节点的门限值为多项式阶数、n节点的秘密值为首项系数a0（=根节点的q(0)=y），随机选取多项式参数{ai}
272
273         for (int j=0; j<n.children.length; j++){
274             Node childNode = nodes[n.children[j]];
275
276             // 对于每一个子节点，以子节点的索引i为参数在整数域加法循环群Zp里随机新建对应的x，计算子节点的多项式值q(x)（也就是其对应的秘密分片）
277             // 编写了函数qx来计算由coef为系数确定的多项式qx在子节点处的值，注意多项式计算在群Zr上进行
278             childNode.secretShare = qx(bp.getZr().newElement(n.children[j]), coef, bp);
279             //q(x)=a[0]+a[1]x+a[2]x^2+...+a[n-1]x^(n-1)
280             // 递归，将该子节点的秘密继续共享下去
281             nodeShare(nodes, childNode, bp);
282         }
283     }
284     //输出为 < 访问分享了秘密后的控制树的所有节点>
285 }

```

### ⑤ 恢复秘密

```

287 // 恢复秘密
288 //nodeRecover(Nodes, n, atts, Pairing bp)→ 节点n是否可以恢复标志, 恢复秘密值为e(g,g)^sy
289 //输入为< 访问控制树, 要恢复秘密的节点, 明文属性集, Pairing实例 >
290 public static boolean nodeRecover(Node[] nodes, Node n, int[] atts, Pairing bp) {
291     //对于非叶子节点来说
292     if (!n.isLeaf()) {
293         // 对于内部节点, 维护一个子节点索引列表, 用于秘密恢复。
294         List<Integer> validChildrenList = new ArrayList<>();
295         int[] validChildren; //临时列表用来装满足条件的子节点
296         // 遍历每一个子节点看
297         for (int j=0; j<n.children.length; j++){
298             Node childNode = nodes[n.children[j]]; //子节点
299             // 存在子节点且可递归恢复, 则递归所有子节点索引列表调用, 恢复子节点的秘密值
300             if (nodeRecover(nodes, childNode, atts, bp)){
301                 System.out.println("The node with index " + n.children[j] + " is satisfied!");
302                 validChildrenList.add(n.children[j]);
303                 // 如果满足条件的子节点个数已经达到门限值, 则跳出循环, 不再计算剩余的节点
304                 if (validChildrenList.size() == n.gate[0]) {
305                     n.valid = true; //表示此节点可以恢复
306                     break;
307                 }
308             }
309             else {
310                 System.out.println("The node with index " + n.children[j] + " is not satisfied!");
311             }
312         }

```

```

313 // 如果可恢复的子节点个数等于门限值, 则利用子节点的秘密片段恢复当前节点的秘密。
314 if (validChildrenList.size() == n.gate[0]){
315     validChildren = validChildrenList.stream().mapToInt(i->i).toArray();
316     // 利用拉格朗日差值恢复秘密
317     // 注意, 此处是在指数因子上做拉格朗日差值
318     Element secret = bp.getGT().newOneElement().getImmutable();
319     for (int i : validChildren) {
320         Element delta = lagrange(i, validChildren, 0, bp); //计算对应指数因子的拉格朗日因子, 作为指数。目标值x为0, 即q1(0)的拉格朗日因子
321         secret = secret.mul(nodes[i].secretShare.duplicate().powZn(delta)); //基于拉格朗日因子进行指数运算, 然后将结果连乘
322         //计算Pi=e(Di,E1)^delta_i(0)=e(g,g)^sq(i)delta_i(0), 并将结果连乘
323         //实际上得到denominator=连乘Pi=e(g,g)^(s*(求和q(i)delta_i(0)))==e(g,g)^sq(0)=e(g,g)^sy
324     }
325     n.secretShare = secret; //e(g,g)^sy
326 }
327 }
328 else {
329     // 判断叶子节点的属性值是否属于属性列表
330     // 判断一个元素是否属于数组, 注意String类型和int类型的判断方式不同
331     if (Arrays.stream(atts).boxed().collect(Collectors.toList()).contains(n.att)){
332         n.valid = true;
333     }
334 }
335 return n.valid;
336 }
337 }

```

## ⑤ 保存和载入参数文件



```

338     public static void storePropToFile(Properties prop, String fileName){
339         try(FileOutputStream out = new FileOutputStream(fileName)){
340             prop.store(out, comments: null);
341         }
342         catch (IOException e) {
343             e.printStackTrace();
344             System.out.println(fileName + " save failed!");
345             System.exit(status: -1);
346         }
347     }
348
349     public static Properties loadPropFromFile(String fileName) {
350         Properties prop = new Properties();
351         try (FileInputStream in = new FileInputStream(fileName)){
352             prop.load(in);
353         }
354         catch (IOException e){
355             e.printStackTrace();
356             System.out.println(fileName + " load failed!");
357             System.exit(status: -1);
358         }
359         return prop;
360     }
361

```

## ⑥ 从文件中读取字符串消息和将字符串消息写入到文件中

```

363     // 从文件中读取明文字符串消息
364     private static String readPlaintextFromFile(String fileName) {
365         StringBuilder plaintext = new StringBuilder();
366         try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {
367             String line;
368             while ((line = reader.readLine()) != null) {
369                 plaintext.append(line);
370             }
371         } catch (IOException e) {
372             e.printStackTrace();
373         }
374         return plaintext.toString();
375     }
376
377     // 将明文字符串写入文件的方法
378     public static void writePlainTextToFile(String plaintext, String filePath) {
379         try {
380             // 创建一个文件写入器
381             FileWriter writer = new FileWriter(filePath);
382             // 写入明文字符串到文件
383             writer.write(plaintext);
384             // 关闭写入器
385             writer.close();
386             System.out.println("PlainText successfully written to file: " + filePath);
387         } catch (IOException e) {
388             // 捕获异常并打印错误信息
389             System.err.println("Error writing to file: " + e.getMessage());
390         }
391     }

```

## 6. 主函数

```
396 public static void main(String[] args) throws Exception {
397     int U = 20;
398     int[] messageAttList = {1, 2, 4, 5};
399     //int[] messageAttList = {1, 2};
400     Node[] accessTree = new Node[7]; //访问树大小
401     accessTree[0] = new Node(new int[]{2,3}, new int[]{1,2,3}); //构造内部根节点, gate:{门限值2, 子节点个数为3};子节点索引列表: {1,2,3}
402     accessTree[1] = new Node( att: 1); // 索引1, 叶子节点, 属性值1
403     accessTree[2] = new Node(new int[]{2,3}, new int[]{4,5,6}); //索引2, 内部节点, gate:{门限值2, 子节点个数为3};子节点索引列表: {4,5,6}
404     accessTree[3] = new Node( att: 5); // 索引3, 叶子节点, 属性值5
405     accessTree[4] = new Node( att: 2); // 索引4, 叶子节点, 属性值2
406     accessTree[5] = new Node( att: 3); // 索引5, 叶子节点, 属性值3
407     accessTree[6] = new Node( att: 4); // 索引6, 叶子节点, 属性值4
408
409     // int[] messageAttList = {1};
410     // Node[] accessTree = new Node[3];
411     // accessTree[0] = new Node(new int[]{2,2}, new int[]{1,2});
412     // accessTree[1] = new Node(1);
413     // accessTree[2] = new Node(2);
414
415     String dir = "data/";
416     String pairingParametersFileName = "a.properties";
417     String pkFileName = dir + "pk.properties";
418     String mskFileName = dir + "msk.properties";
419     String skFileName = dir + "sk.properties";
420     String ctFileName = dir + "ct.properties";
421
422     String inputFileName = dir + "input.txt"; // 输入文件名
423     String outputFileName = dir + "output.txt"; // 输入文件名
424
425
426     setup(pairingParametersFileName, U, pkFileName, mskFileName);
427
428     keygen(pairingParametersFileName, accessTree, pkFileName, mskFileName, skFileName);
429
430     //Element message = PairingFactory.getPairing(pairingParametersFileName).getGT().newRandomElement().getImmutable();
431
432     //System.out.println("明文消息:" + message);
433     //encrypt(pairingParametersFileName, message, messageAttList, pkFileName, ctFileName);
434
435     // 读取明文字符串消息
436     String plaintext = readPlaintextFromFile(inputFileName);
437     System.out.println("明文消息: " + plaintext);
438
439     // 将明文字符串转换为字节数组
440     byte[] plaintextBytes = plaintext.getBytes(StandardCharsets.UTF_8);
441     // 使用GT群上的元素来表示明文
442     Element plaintextElement = PairingFactory.getPairing(pairingParametersFileName).getGT().newElementFromBytes(plaintextBytes).getImmutable();
443     // 在加密操作中, 你可以将plaintextElement与属性相关联并进行加密
444     encrypt(pairingParametersFileName, plaintextElement, messageAttList, pkFileName, ctFileName);
445
446
447     Element res = decrypt(pairingParametersFileName, accessTree, pkFileName, ctFileName, skFileName);
448     System.out.println("解密结果:" + res);
449
450     if (res!=null) {
451         byte[] decryptedBytes = res.toBytes();
452         String decryptedPlaintext = new String(decryptedBytes, StandardCharsets.UTF_8);
453
454         // 打印解密后的明文字符串
455         System.out.println("Decrypted plaintext: " + decryptedPlaintext);
456         // 调用方法将明文字符串写入文件
457         writePlainTextToFile(decryptedPlaintext, outputFileName);
458     }
459
460     if (plaintextElement.isEqual(res)) {
461         System.out.println("成功解密!");
462         // 在解密操作中, 你可以从GT群上的点中获取字节数组, 并将其转换回明文字符串
463     }
464 }
465 }
```

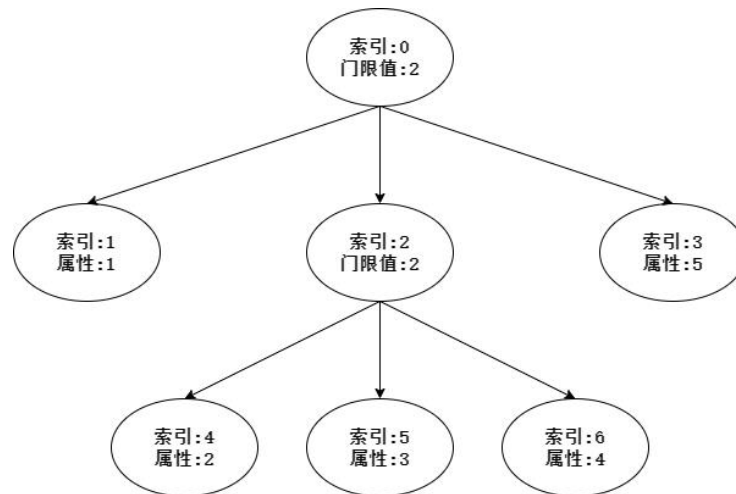
## 7. 运行检验

(1) 实验方案一：明文消息为随机生成 GT 群上的点

### ① 测试数据：

属性集为 $\{1, 2, \dots, 20\}$ ，即  $U=20$ 。

访问控制树如图：



### ② 明文属性满足访问控制树属性要求的情况

测试数据：

明文属性 =  $\{1, 2, 4, 5\}$

运行得到：

```
C:\Users\youye\.jdk\corretto-1.8.0_402\bin\java.exe ...
明文消息:{x=4845537303960927683948678459837487551040538790018195695996168595394304518848598845989804477766749998908650187076690245017225191119522593810486032493130998,y=59702824}
The node with index 1 is satisfied!
The node with index 4 is satisfied!
The node with index 5 is not satisfied!
The node with index 6 is satisfied!
The node with index 2 is satisfied!
解密结果:{x=4845537303960927683948678459837487551040538790018195695996168595394304518848598845989804477766749998908650187076690245017225191119522593810486032493130998,y=59702824}
成功解密!
```

能够成功解密得到 GT 群上的明文消息点。

### ③ 明文属性不满足访问控制树属性要求的情况

测试数据：

明文属性 = {1, 2}

运行得到：

```
C:\Users\youye\jdk\corretto-1.8.0_402\bin\java.exe ...
明文消息:{x=1218687986650924998169065295868083187703484569353314723888630778930835843091161185573898055541156898286527541163424134387004381235724036996317138478967281,y=4665
The node with index 1 is satisfied!
The node with index 4 is satisfied!
The node with index 5 is not satisfied!
The node with index 6 is not satisfied!
The node with index 2 is not satisfied!
The node with index 3 is not satisfied!
The access tree is not satisfied.
解密结果:null
```

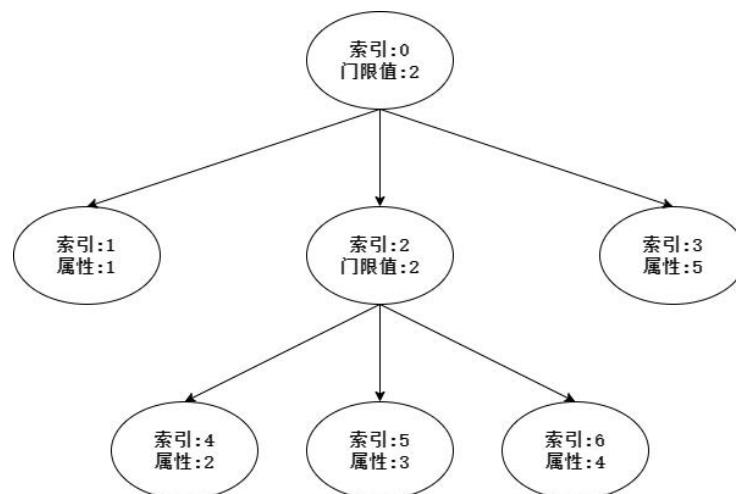
无法正确解密得到明文消息点。

## (2) 实验方案二：明文消息为 input.txt 文件内容

### ① 测试数据：

属性集为{1, 2, ..., 20}，即  $U=20$ 。

访问控制树如图：



明文消息串为 “Hello world! 3201603102!”。

### ② 明文属性满足访问控制树属性要求的情况

测试数据：

明文属性 = {1, 2, 4, 5}



运行得到:

明文消息: Hello world! 3201603102!

The node with index 1 is satisfied!

The node with index 4 is satisfied!

The node with index 5 is not satisfied!

The node with index 6 is satisfied!

The node with index 2 is satisfied!

## 解密结

果:  $\{x=1775149307253087648481863067631845197071875299811196416545, y=0\}$

Decrypted plaintext:

Hello world! 3201603102!

PlainText successfully written to file: data/output.txt

成功解密！

[illegible]

能够成功解密得到含有明文消息串的文件。

### ③ 明文属性不满足访问控制树属性要求的情况

测试数据:

明文属性 = {1, 2}

运行得到:

明文消息: Hello world! 3201603102!

The node with index 1 is sarisfied!

The node with index 4 is sarisfied!

The node with index 5 is not sarisfied!

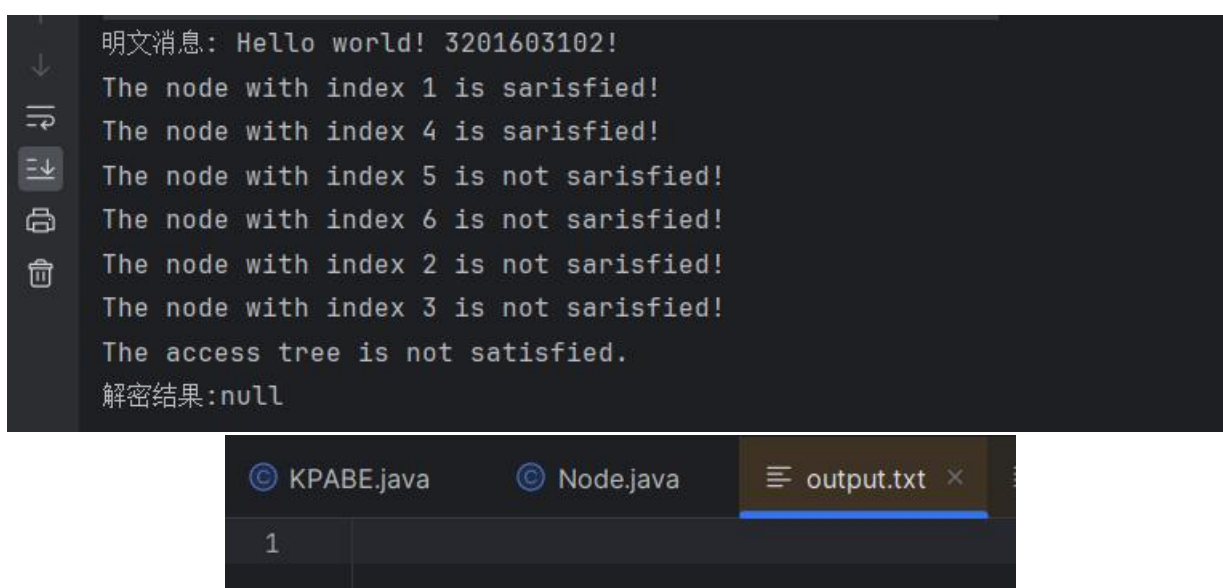
The node with index 6 is not sarisfied!

The node with index 2 is not sarisfied!

The node with index 3 is not sarisfied!

The access tree is not satisfied.

解密结果:null



```
明文消息: Hello world! 3201603102!
The node with index 1 is sarisfied!
The node with index 4 is sarisfied!
The node with index 5 is not sarisfied!
The node with index 6 is not sarisfied!
The node with index 2 is not sarisfied!
The node with index 3 is not sarisfied!
The access tree is not satisfied.
解密结果:null
```

KPABE.java   Node.java   output.txt x

1

无法正确解密得到明文消息。

## 五、实验结论

KP-ABE 的 Setup 和 Encrypt 过程和 FIBE 基本一致, 主要区别在于密钥生

成和解密阶段，将 FIBE 中的门限值替换成了细粒度的访问树控制结构，只有当密文中的属性满足密钥中嵌入的访问树时，用户才能解密该密文。根据密文中的属性和访问树的叶子节点开始匹配，层层递进直到根节点，若满足则可以恢复出根节点的秘密值  $y$ ，从而计算出  $Y^s$  的值，最终解出明文  $M$ 。

本次实验记录的 Github 地址：<https://github.com/YTR1020/KP-ABE/tree/main>

参考文献：

[1]<https://www.jianshu.com/p/8d8cf34a9aa0>.

[2]J. Li, Y. Zhang, J. Ning, X. Huang, G. S. Poh and D. Wang, "Attribute Based Encryption with Privacy Protection and Accountability for CloudIoT," in IEEE Transactions on Cloud Computing, vol. 10, no. 2, pp. 762-773, 1 April-June 2022, doi: 10.1109/TCC.2020.2975184.

[3][https://blog.csdn.net/m0\\_52322997/article/details/125013658](https://blog.csdn.net/m0_52322997/article/details/125013658).

[4]<https://blog.csdn.net/jiandsjcbsbf/article/details/113072329>.