身份基加密 (Identity based Encryption)算法

论文 Identity-Based Encryption from the Weil Pairing

Setup

1. 生成pairing相关公共参数 $< e, G_1, G_T, Z_r >$
2. 选取随机数$x \in Z_r$ 作为系统主密钥msk
3. 选取随机元素$g \in G_1$作为生成元，计算公共参数g^x。因此，有系统公钥$pk =< g, g^x >$
4. 选取公共哈希函数 $H_1 : \{0,1\}^* \rightarrow G_1^*$，$H_2 : G_T \rightarrow \{0,1\}^n$

KeyGen

1. 给定用户身份 $ID \in \{0,1\}^*$，将其映射为群G_1上的元素。即计算 $Q_{ID} = H_1(ID)$
2. 由系统主密钥x计算此ID对应的私钥为$sk = Q_{ID}^x$

Encrypt

1. 针对目标用户身份$ID \in \{0,1\}^*$，计算 $Q_{ID} = H_1(ID)$
2. 选取随机数$r \in Z_r$，计算密文组件$C_1 = g^r$
3. 计算$g_{ID} = e(Q_{ID}, g^x)^r$
4. 计算密文组件$C_2 = M \oplus H_2(g_{ID})$，其中$M \in \{0,1\}^n$是明文数据
5. 最终的密文为$< C_1, C_2 >$

Decrypt

1. 解密的关键在于恢复g_{ID}
2. $e(sk, C_1) = e(Q_{ID}^x, g^r) = e(Q_{ID}, g)^{xr} = g_{ID}$
3. 恢复明文 $M = C_2 \oplus H_2(e(sk, C_1))$

代码实现注意事项

1. 选择用Properties保存是因为支持键值读取，比如密文可能包含多个组件，方便分别读取每个组件。
2. SetProperties的第二个参数必须为String类型，因此需要先将要保存的元素转换为String类型后存储。

```
//写入文件
Properties pkProp = new Properties();
//pkProp.setProperty("g", new String(g.toBytes()));  //可以用这种方式将g转换为字符串后写入，但文件中会显示乱码
//为了避免乱码问题，统一采用Base64编码为可读字符串形式
pkProp.setProperty("g", Base64.getEncoder().encodeToString(g.toBytes()));
pkProp.setProperty("gx", Base64.getEncoder().encodeToString(gx.toBytes()));
storePropToFile(pkProp, pkFileName);

//从文件读取
```

```
Properties pkProp = loadPropFromFile(pkFileName);
String gString = pkProp.getProperty("g");
Element g =
bp.getG1().newElementFromBytes(Base64.getDecoder().decode(gString)).getImmutable();
String gxString = pkProp.getProperty("gx");
Element gx =
bp.getG1().newElementFromBytes(Base64.getDecoder().decode(gxString)).getImmutable()
;
```

3. `bp.getG1().newElementFromBytes()`和`bp.getG1().newElementFromHash()`用法。前者一般用于将一个恢复一个通过bytes到处的元素，比如上图的代码中用于恢复`g`和`gx`。后者用于从一个hash得到的字节数组中构造一个新元素，如下代码中用于将ID映射为群元素。

```
byte[] idHash = sha1(id);
Element QID = bp.getG1().newElementFromHash(idHash, 0,
idHash.length).getImmutable();
```

4. 算法实现过程中，遵从原文，密文组件的计算采用了异或方式，即$C_2 = M \oplus H_2(g_{ID})$。在后面的ABE方案中，一般都是直接计算$C_2 = M \cdot g_{ID}$，这样仿真起来会更简单一些。