密码学综合实验第 6 次实验

--非对称可搜索加密

姓　　名：柳致远

学　　号：2113683

专　　业：密码科学技术

指导老师：邓琮弋

目录

一、 实验内容.....3
1、 算法背景.....3
2、 算法描述.....4
3、 算法实验方案.....4
二、 实验代码分析.....5
1、 生成公私钥.....5
2、 加密函数.....5
3、 生成陷门.....6
4、 测试函数.....6
三、 实验步骤及截图.....7
1、 实验环境配置.....7
2、 实验过程截图.....7
四、 实验结果分析.....7
五、 附加题.....8
1、 实验背景：.....8
2、 实验要求.....8
3、 实验分析.....8
4、 代码分析.....9
5、 实验结果及截图.....10
6、 实验结果分析及感想.....10

一、实验内容

1、算法背景

非对称可搜索加密（Public Key Encryption with Keyword Search，PEKS）是一种密码学技术，旨在保护数据的隐私性同时允许对加密数据进行搜索操作。这种技术的背景主要包括对数据隐私性和数据搜索功能的需求。

在传统的加密方案中，通常使用对称加密算法来保护数据的隐私性，但这种方式通常不能满足数据搜索的需求。当数据被加密后，即使是数据的所有者也无法有效地对加密数据进行搜索操作，除非他们拥有解密密钥。然而，如果数据的所有者需要与其他人共享加密数据，并且这些人需要对数据进行搜索操作，那么传统的加密方法就会受到限制。

非对称可搜索加密技术的出现就是为了解决这个问题。它允许数据的所有者使用公钥将数据加密，并将加密数据共享给其他人，同时保护数据的隐私性。同时，其他人可以使用一个特殊的搜索密钥来对加密数据进行搜索操作，而不需要解密数据。这样一来，数据的所有者可以将加密数据共享给其他人，同时保持数据的隐私性，并允许他们对加密数据进行搜索操作，从而满足了数据共享和搜索的需求。

比如：Bob 使用 Alice 的公钥pk加密邮件和相关关键词，并将形如$(PKE.Encrypt(pk, MSG),\ PEKS.Encrypt(pk, W_1),\ \cdots, PEKS.Encrypt(pk, W_n))$的密文发送至邮件服务器。这里，$PKE.Encrypt$为公钥密码加密算法，$MSG$为邮件内容，$W_1, \cdots, W_n$为与$MSG$关联的关键词。Alice 将$T_{"urgent"}$或$T_{"lunch"}$长驻服务器，新邮件到来时，服务器自动对其关联的关键词执行与$T_{"urgent"}$或$T_{"lunch"}$相关的$Test$算法，如果输出 1，便将该邮件转发至 Alice 的手机或个人电脑。

2、算法描述

定义（非对称可搜索加密，public key encryption with keyword search, PEKS）非对称密码体制下可搜索加密算法可描述为$PEKS = (KeyGen, Encrypt, Trapdoor, Test)$：

- ✧ $(pk, sk) = KeyGen(\lambda)$：输入安全参数$\lambda$，输出公钥$pk$和私钥$sk$；
- ✧ $C_W = Encrypt(pk, W)$：输入公钥pk和关键词W，输出关键词密文C_W；
- ✧ $T_W = Trapdoor(sk, W)$：输入私钥sk和关键词W，输出陷门T_W；
- ✧ $b = Test(pk, C_W, T_{W'})$：输入公钥$pk$、陷门$T_{W'}$和关键词密文$C_W$，根据$W$与$W'$的匹配结果，输出判定值$b \in \{0,1\}$。

3、算法实验方案

2004 年，Boneh 提出了第一个非对称的可搜索加密方案，具体构造如下：

令$e: G_1 \times G_1 \to G_2$为双线性对，函数$H_1: \{0,1\}^* \to G_1$和$H_2: G_2 \to \{0,1\}^{\log p}$为哈希函数。

- Keygen：输入安全参数，该安全参数决定群G_1和G_2的阶 p，随机挑选$\alpha \leftarrow Z_p^*$和G_1的生成元 g，输出$pk \coloneqq (g, h = g^{\alpha})$和$sk \coloneqq \alpha$。
- Encrypt：输入公钥和关键词，随机选择$r \leftarrow Z_p^*$，计算$t \coloneqq e(H_1(w), h^r)$，输出$c \coloneqq (g^r, H_2(t))$。
- TrapDoor：输入私钥和关键词，输出$td \coloneqq H_1(w)^{\alpha}$。
- Test：输入陷门和密文，记密文为$c = (c_1, c_2)$，若$H_2(e(td, c_1)) = c_2$，输出 1，否则输出 0。

正确性：

$$\mathrm{e}(td, \mathrm{c}_1) = \mathrm{e}(\mathsf{H}_1(\mathrm{w})^{\alpha}, \mathrm{g}^{\mathrm{r}}) = \mathrm{e}(\mathsf{H}_1(\mathrm{w}), \mathrm{g}^{\alpha \mathrm{r}}) = \mathrm{e}(\mathsf{H}_1(\mathrm{w}), \mathrm{h}^{\mathrm{r}}),$$

$$\mathsf{H}_2(\mathrm{e}(td, c_1)) = c_2$$

二、实验代码分析

1、生成公私钥

```
def Setup(param_id='SS512'):
    group = PairingGroup(param_id)
    g = group.random(G1)
    alpha = group.random(ZR)
    sk = group.serialize(alpha)
    pk = [group.serialize(g), group.serialize(g ** alpha)]
    return [sk, pk]
```

Setup 函数接收一个参数 param_ip，该参数用于选择用于创建加密方案的参数集。陆续随机生成群元素 g 和 alpha，分别用于公钥和私钥的生成。随后分别用上述生成的随机数生成公钥和私钥并返回公私钥对。

2、加密函数

```
def Enc(pk, w, param_id='SS512'):
    group = PairingGroup(param_id)
    g, h = group.deserialize(pk[0]), group.deserialize(pk[1])
    r = group.random(ZR)
    t = pair(Hash1(w), h ** r)
    c1 = g ** r
    c2 = t
    print("The serialize value of pair(Hash1(w) , h ** r) is:"group.serialize(c2))
    return [group.serialize(c1), Hash2(group.serialize(c2)).hexdigest()]
```

加密函数 **Enc**，用于对消息 **w** 使用给定的公钥 **pk** 进行加密。首先，将公钥 **pk** 中的两个部分进行反序列化，得到两个群元素 **g** 和 **h**，随机生成随机数 r，利用消息 w 的 Hash1（md5）和公钥中的第 2 个部分 **h ** r** ，计算了一个双线性映射值 **t**，利用公钥的第 1 个部分 **g** 计算加密消息的第一个部分 **c1 = g ** r**，最后将双线性映射值 t 分配给加密后的消息的第二个部分 c2。

3、生成陷门

```
def TdGen(sk, w, param_id='SS512'):
    group = PairingGroup(param_id)
    sk = group.deserialize(sk)
    td = Hash1(w) ** sk
    return group.serialize(td)
```

首先将传入的私钥 **sk** 反序列化，以便在后续的计算中使用。后计算关键词 **w** 的哈希值并将其提升至私钥 **sk** 的幂次方，得到陷门 **td**，将陷门 **td** 序列化并返回其序列化形式。在 Hash1 函数中除了对消息 **w** 进行 **md5** 哈希运算还将 hash 结果打印。

4、测试函数

```
def Test(td, c, param_id='SS512'):
    group = PairingGroup(param_id)
    c1 = group.deserialize(c[0])
    c2 = c[1]
    print("The value of c[1] is:", c2)
    td = group.deserialize(td)
    return Hash2(group.serialize(pair(td, c1))).hexdigest() == c2
```

从密文 **c** 中反序列化出第一个部分 **c1**，这是一个群元素，紧接着提取密文 c 的第二个部分 c2，它应该是一个哈希值的字符串并打印。并从陷门 **td** 中反序列化出陷门的值，准备用于后续的计算。最后，将哈希值与密文的第二个部分 **c2** 进行比较，如果相等则返回 True，否则返回 False。

三、实验步骤及截图

1、实验环境配置

本次实验需要使用到一个实用的 python 密码工具库 charm-crypto，这个工具库在 windows 系统下的配置过程较为复杂，故选择在虚拟机环境下使用 Ubuntu 系统下进行配置，配置过程详情见 Charm-Crypto 开发框架保姆级搭建教程 - 知乎 (zhihu.com)，这个教程是我目前找到的比较容易方便的一个。读者可自行挑选其他配置教程。

2、实验过程截图

四、实验结果分析

可以通过实验过程截图看出，该程序一共调用了 4 次 Enc 函数并进行测试，而在每次测试中都会进行 assert 函数断言测试，若通过，即 **Hash2(group.serialize(pair(td, c1))).hexdigest() == c2** 或不等时测试 assert not,通过时才会进行下一次测试。由每次测试输出的 Hash1(w)的值相同和测试次数与输出结果个数相同可知该实验结果成功。

五、附加题

1、实验背景：

PEKS 本身定义存在严重的安全隐患：关键词猜测攻击。关键词猜测攻击是由于关键词空间远小于密钥空间，而且用户通常使用常用关键词进行检索，这就给攻击者提供了只需采用字典攻击就能达到目的的“捷径”。

导致关键词猜测攻击的原因可归结为：①关键词空间较小，且用户集中于使用常用词汇，给攻击者提供了遍历关键词空间的可能；②PEKS 算法一致性约束，使攻击者拥有对本次攻击是否成功的预先判定：执行 Test 算法，返回 1 说明本次攻击成功，否则可以再继续猜测。

为抵御关键词猜测攻击，很多方案提出，比如可以在服务器端进行模糊陷门测试，过滤大部分不相关邮件，最后在本地精确匹配，得到检索结果。这种方法通过引入模糊陷门，一定程度降低了接收者外部 PEKS 算法一致性，使其能够抵御关键词猜测攻击，但增加了客户服务器通信量和用户端计算量。

2、实验要求

基于 Boneh 的第一个 PEKS 方案，简述抵御关键词猜测攻击的办法。如果可以，请通过代码实现。

3、实验分析

通过加盐哈希方式模糊关键词处理，消息所有者直接存储加盐哈希之后的结果并使服务器用这个结果制造陷门，此时制造陷门用的关键词没有任何实际语义故无法通过字典遍历的方式破解攻击。此时的关键词空间就基本等同于密钥空间，具有良好的安全性。本次实验中我采随机生成整数加盐后用 sha-256 的方式来哈希运算实现模糊关键词操作。

4、代码分析

增加加盐哈希函数

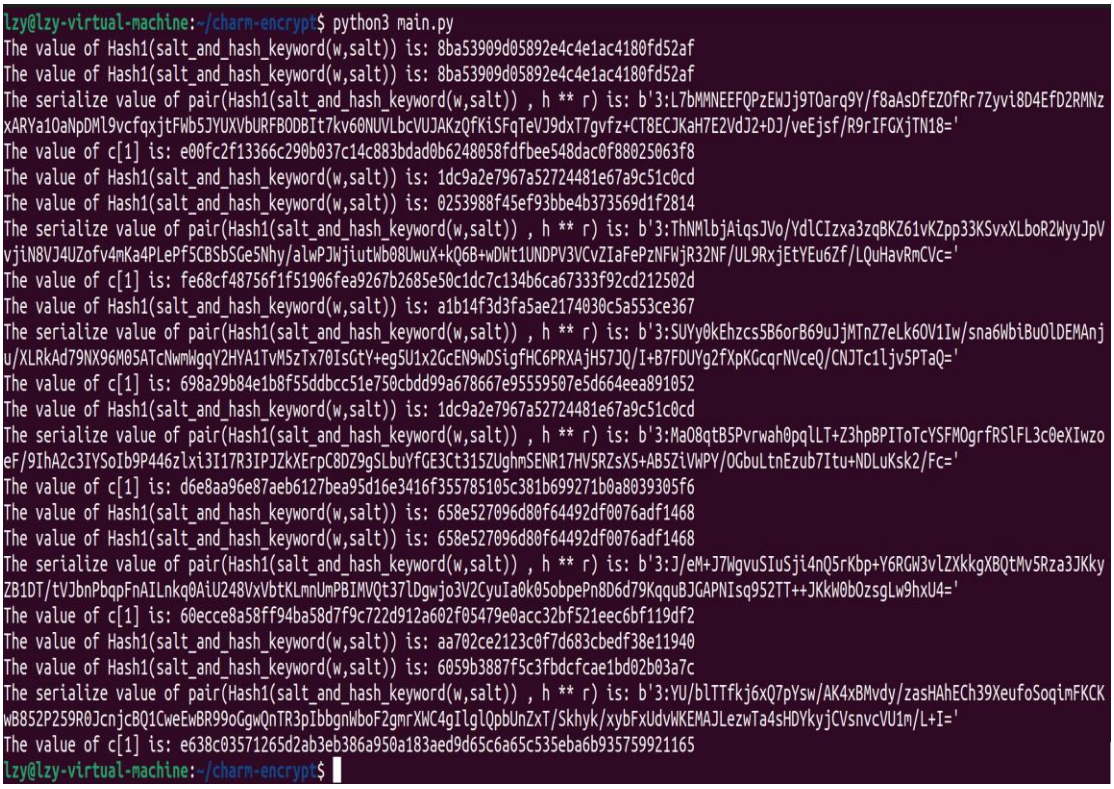
```
def salt_and_hash_keyword(w,salt):
    # 加盐
    salted_w = w + str(salt)
    # 哈希处理
    hashed_w = hash_function(salted_w)
    return hashed_w

def hash_function(input_string):
    # 在这里实现哈希函数的具体逻辑，例如使用 hashlib 库进行哈希计算
    hashed_result = hashlib.sha256(input_string.encode()).hexdigest()
    return hashed_result
```

盐值 salt 在 TdGen 函数中随机生成，TdGen 增加一行代码：salt = group.random(ZR)。这样在每次调用 TdGen 函数用户生成陷门时就会随机生成一个 salt 并对加盐后的关键词 w 进行 sha-256 哈希运算，用户将此结果保存，以后需要检索时需要输入该结果而不是原关键词，以此抵抗内部敌手的关键词攻击。

```
if __name__ == '__main__':
    param_id = 'SS512'
    [sk, pk] = Setup(param_id)
    group = PairingGroup(param_id)
    td = TdGen(sk, "yes")
    c = Enc(pk, salt_and_hash_keyword("yes",salt))
    assert(Test(td, c))
    td = TdGen(sk, "no")
    c = Enc(pk, salt_and_hash_keyword("yes",salt))
    assert(not Test(td, c))
    c = Enc(pk, salt_and_hash_keyword("Su*re",salt))
    assert(not Test(td, c))
    c = Enc(pk, salt_and_hash_keyword("no",salt))
    assert(Test(td, c))
    td = TdGen(sk, 9 ** 100)
    c = Enc(pk, salt_and_hash_keyword(9 ** 100,salt))
    assert(Test(td, c))
    td = TdGen(sk, 9 ** 100 + 1)
    c = Enc(pk, salt_and_hash_keyword(9 ** 100,salt))
    assert(not Test(td, c))
```

可以看到主函数这里调用 TdGen 函数生成陷门后由服务器进行加盐哈希运算并将结果返回调用 TdGen 的用户备份，以便后续该用户使用该结果搜索。而内部敌手若无法得到对应关键词的盐值（每一个关键词对应一个 salt）就无法计算出关键词的加盐哈希之后的结果，若直接输入关键词服务器将不会识别，故使用这种方案可以抵抗内部敌手对关键词的字典遍历攻击。

5、实验结果及截图



6、实验结果分析及感想

同样的输出结果的个数符合测试 assert 测试次数，并且每次测试输出的 Hash1(salt_and_hash_keyword(w,salt))的值相等说明实验结果正确。

通过加盐哈希的办法来实现模糊关键词的操作可以有效的防止内部敌手对关键词的字典遍历攻击，但是这样就对数据所有者即加入陷门的用户提出了一

定的要求，该用户需要存储大量的关键词及对应的加盐哈希之后的结果，对用户的计算机存储容量提出了要求。并且一旦该用户存储的映射表丢失或被外部敌手窃取，该方案将失去安全性。