南開大學
Nankai University

计算机学院

数据安全实验报告

基于正向索引的 SWP 对称可搜索加密

姓名：李欣

学号：2011165

专业：计算机科学与技术

2023 年 5 月 8 日

目录

1 实验要求

根据正向索引或者倒排索引机制，提供一种可搜索加密方案的模拟实现，应能分别完成加密、陷门生成、检索和解密四个过程。

2 实验环境

语言：Python
ide：pycharm 2022.1.3

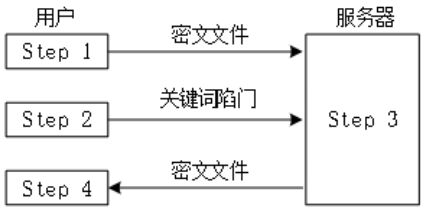
3 实验原理

3.1 可搜索加密

关键词检索是一种常见的操作，比如数据库全文检索、邮件按关键词检索、在 Windows 系统里查找一个文件等。可搜索加密（Searchable Encryption，简称 SE）则是一种密码原语，它允许数据加密后仍能对密文数据进行关键词检索，允许不可信服务器无需解密就可以完成是否包含某关键词的判断。

可搜索加密可分为 4 个子过程（如图5.7所示）：

- **加密过程**：用户使用密钥在本地对明文文件进行加密并将其上传至服务器；
- **陷门生成过程**：具备检索能力的用户使用密钥生成待查询关键词的陷门（也可以称为令牌），要求陷门不能泄露关键词的任何信息；
- **检索过程**：服务器以关键词陷门为输入，执行检索算法，返回所有包含该陷门对应关键词的密文文件，要求服务器除了能知道密文文件是否包含某个特定关键词外，无法获得更多信息；
- **解密过程**：用户使用密钥解密服务器返回的密文文件，获得查询结果。



3.2 对称可搜索加密

可搜索加密可以分为**对称可搜索加密**和非对称可搜索加密，分别基于**对称密码**和非对称密码来构建。

本次实验是基于对称可搜索加密实现的,对称可搜索加密 (Symmetric searchable encryption, SSE)：旨在加解密过程中采用相同的密钥之外，陷门生成也需要密钥的参与，通常适用于单用户模型，具有计算开销小、算法简单、速度快的特点。

3.3 对称可搜索加密算法描述

（1）算法描述定义在字典上的 $\triangle = \{W_1, W_2, ..., W_d\}$ 对称可搜索加密算法可描述为五元组：

$$SSE = (KeyGen, Encrypt, Trapdoor, Search, Decrypt)$$

其中，

- $K = KeyGen(\lambda)$：输入安全参数 λ，输出随机产生的密钥 K;
- $(I, C) = Encrypt(K, D)$：输入对称密钥 K 和明文文件集 $D = (D_1, D_2, ..., D_n)$，输出索引 I 和密文文件集 $C = (C_1, C_2, ..., C_n)$。对于无需构造索引的 SSE 方案，$I = \emptyset$;
- $T_w = Trapdoor(K, W)$：输入对称密钥 K 和关键词 W，输出关键词陷门 T_w;
- $D(W) = Search(I, T_W)$：输入索引 I 和陷门 T_w，输出包含 W 文件的标识符构成的集合 $D(W)$;
- $D_i = Decrypt(K, C_i)$：输入对称密钥 K 和密文文件 C_i，输出相应明文文件 D_i。

如果对称可搜索加密方案 SSE 是正确的，那么对于 $KeyGen(\lambda)$ 和 $Encrypt(K, D)$ 输出的 K 和 (I, C)，都有 $Search(I, Trapdoor(K, W)) = D(W)$ 和 $Decrypt(K, C_i) = D_i$ 成立，这里 $C_i \in C, i = 1, 2, ..., n$.

基于上述定义，对称可搜索加密流程如下：加密过程中，用户执行算法 $KeyGen(\lambda)$ 生成对称密钥 K，使用 K 加密明文文件集 D，并将加密结果上传至服务器。检索过程中，用户执行 $Trapdoor$ 算法，生成待查询关键词 W 的陷门 T_w；服务器使用 T_w 检索到文件标识符集合 $D(W)$，并根据 $D(W)$ 中文件标识符提取密文文件以返回用户；用户最终使用解密 K 所有返回文件，得到目标文件。

3.4 基于正向索引的 SWP 方案

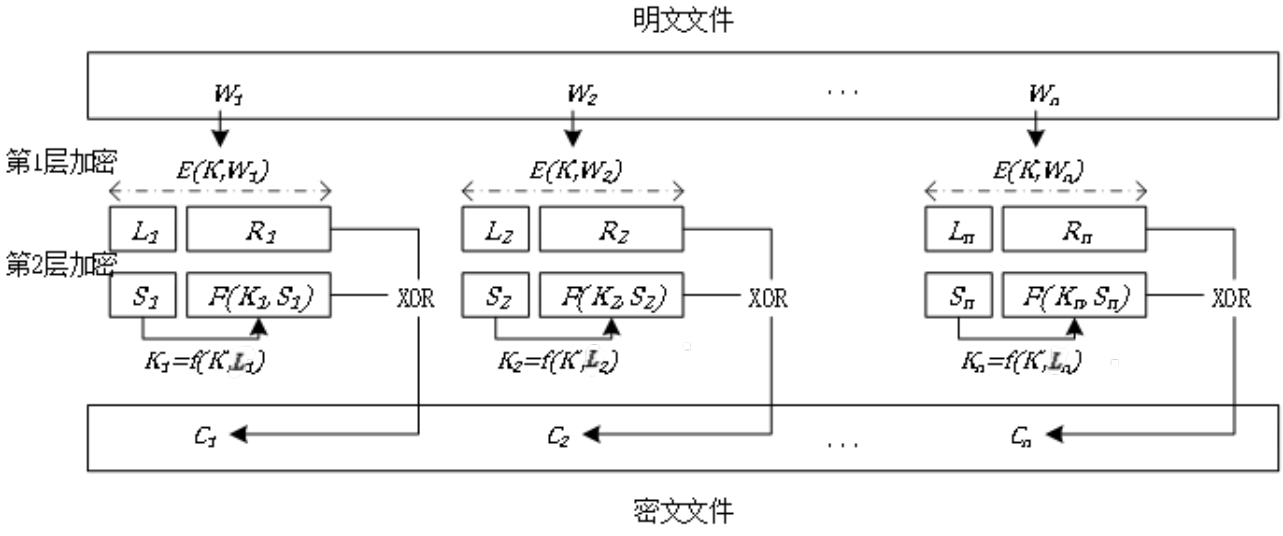


图 3.1: SWP 方案

3.4.1 加密过程

SWP 方案在预处理过程中根据文件长度产生伪随机流 $S_1, S_2, ..., S_n$（n 为待加密文件中“单词”个数），然后采用两个层次加密：在第 1 层，使用分组密码逐个加密明文文件单词；在第 2 层，对分组

密码输出 $E(K^{'}, W_i)$ 进行处理：1）将密文等分为 L_i 和 R_i 两部分；2）基于 L_i 生成二进制字符串 $S_i||F(K_i, S_i)$，这里，$K_i = f(K^{''}, L_i)$，$||$ 为符号串连接，F 和 f 为伪随机函数；3）异或 $E(K^{'}, W_i)$ 和 $S_i||F(K_i, S_i)$ 以形成 W_i 的密文单词。

3.4.2 检索过程

查询文件中是否包含关键词，只需发送陷门至服务器（为的左部），服务器顺序遍历密文文件的所有单词，计算，判断是否等于：如果相等，即为在中的密文；否则，继续计算下一个密文单词。

SWP 方案通过植入“单词”位置信息，能够支持受控检索（检索关键词的同时，识别其在文件中出现的位置）。例如，将所有“单词”以形式表示，为在文件中出现的位置，仍按图所示加密，但查询时可增加对关键词出现位置的约束。

3.4.3 解密过程

接收服务端发回来的 C_i，将其分为 C_iL 和 C_iR, 将左半部分与 S_i 异或，得到密文 X_i 的左半部分 L_i，再使用 H1 哈希函数，以 hash_key 为密钥对 L_i 加密得到 K_i；再使用 H2 哈希函数，以 K_i 为密钥对 S_i 加密得到 $F(K_i, S_i)$; 将 $F(K_i, S_i)$ 与 C_iR 异或得到 R_i；将 L_i 和 R_i 拼接得到 X_i，接着使用 DES 算法解密得到原文单词 W_i

3.4.4 小结

SWP 方案存在一些缺陷：1）效率较低，单个单词的查询需要扫描整个文件，占用大量服务器计算资源；2）在安全性方面存在统计攻击的威胁。例如，攻击者可通过统计关键词在文件中出现的次数来猜测该关键词是否为某些常用词汇。

4 代码实现

首先，定义加密信息类，主要存储第一层加密信息 X_i, 第二层加密所需要的 K_i 和 S_i

```
1  class Encrypt_W:
2      def __init__(self, Ci, Xi, Ki, Si):
3          self.Ci = Ci
4          self.Xi = Xi
5          self.Ki = Ki
6          self.Si = Si
7
8      def getCi(self):
9          if (self.Ci[0:2] == "0x"):
10             return self.Ci[2:]
11         return self.Ci
12
13     def getXi(self):
14         if (self.Xi[0:2] == '0x'):
15             return self.Xi[2:]
```

```
16          return self.Xi
17
18      def getKi(self):
19          if (self.Ki[0:2] == '0x'):
20              return self.Ki[2:]
21          return self.Ki
22
23      def getSi(self):
24          if (self.Si[0:2] == '0x'):
25              return self.Si[2:]
26          return self.Si
27
28      def ew_print(self):
29          print()
30          print("[ENCRYPT_W] CI = ", self.Ci)
31          print("[ENCRYPT_W] XI = ", self.Xi)
32          print("[ENCRYPT_W] KI = ", self.Ki)
33          print("[ENCRYPT_W] SI = ", self.Si)
```

然后定义服务端信息类，主要存储加密后的信息 C_i

```
1   class Server_W:
2       def __init__(self, Ci):
3           self.Ci = Ci
4
5       def getCi(self):
6           if (self.Ci[0:2] == "0x"):
7               return self.Ci[2:]
8           return self.Ci
9
10      def sw_print(self):
11          print("[SERVER_W] CI = ", self.Ci)
12          print()
```

4.1 加密过程

首先分割字符串，将原始字符串分割为一个一个的单词，这里我们使用正则表达式进行分割，生成原始单词 W_i

```
1   def split_str(str):
2       word_list = re.split(r"\b[\.,\s\n\r\n]+?\b", str)
3       return word_list
```

然后，我们使用 DES 算法对分割的每个单词进行**第一层加密**，生成 X_i

```
# 使用 DES 加密
def encrypt(s, DES_KEY):
    fin_str = ""
    s = s.encode()  # 这里中文要转成字节
    iv = b'abcdefgh'  # 定义初始化向量，长度为 8 个字符（64 位）
    # 初始化一个 des 对象，参数是秘钥，加密方式，偏移，填充方式
    des_obj = des(DES_KEY, pyDes.CBC, iv, pad=None, padmode=pyDes.PAD_PKCS5)
    while len(fin_str) < 64:
        secret_bytes = des_obj.encrypt(s)  # 用对象的 encrypt 方法加密
        secret_hex = secret_bytes.hex()
        secret_str = str(secret_hex)
        s = secret_str
        fin_str = fin_str + secret_str
    return fin_str
```

接着进行**第二层加密**，我们将第一层加密的结果 X_i 分割成 L_i 和 R_i，这里我们使用的方法是将 X_i 等分了，其实有点不太懂这里如何分会比较合适，也许会影响算法效率和算法安全性（?）

```
# 拆分成左右两边
def split_str_2(str):
    left = str[:len(str) // 2]
    right = str[len(str) // 2:]
    return left, right
```

使用带密钥的 hash 函数 H1（这里使用的是 MD5 算法）对 L_i 进行加密，以 $hash\_key$ 为固定密钥，结果记作 K_i

```
# 使用 hmac_MD5 加密
def hmac_md5(msg, key):
    return hmac.new(key.encode(), msg.encode(), hashlib.md5).hexdigest()
```

使用伪随机函数 Random，输入随机种子 seed，来得到伪随机序列 S_i。

接着，再使用带密钥的 hash 函数 H2（这里还是用 md5）对 S_i 进行加密，以 K_i 为密钥，得到 FK_i

将 S_i 与 FK_i 拼接就得到了 T_i，即第二层加密的结果。

最终将第一层加密结果 X_i 和第二层加密结果 T_i 异或得到最后的密文 C_i，上传至不可信服务器。

```python
# 做异或操作
def strXOR(str1, str2):
    if len(str1) != len(str2):
        print('ERROR: the length of two strings must be equal')
        print(len(str1), len(str2))
        return ""
    standard = len(str1) + 2
    num1 = int(str1, 16)
    num2 = int(str2, 16)
    bin1 = bin(num1)
    bin2 = bin(num2)
    res = int(bin1, 2) ^ int(bin2, 2)
    res = hex(res)
    # 这里是一种特殊情况，如果异或完之后出现了 0 开头的结果，
    # 就会导致返回的值不足原来的比特位的情况。
    if len(res) < standard:
        res = str(res)
        res = res[2:]
        while len(res) < standard - 2:
            res = '0' + res
        res = '0x' + res
    return res
```

上面的函数都是加密过程种需要用到的**函数**，我们将完整的加密流程封装到一个函数中：

```python
def Encrypt_pipeline(word, seed, DES_KEY, HASH_KEY):

    if word == "":
        print("THIS IS AN EXAMPLE")
        word_list = split_str(test_strs[1])
        print('[ENCRYPT_LOG] w:', word_list[2])
        word = encrypt(word_list[2], DES_KEY)  # 64 位
        print('[ENCRYPT_LOG] w:', word)
    else:
        print('[ENCRYPT_LOG] w:', word)
        word = encrypt(word, DES_KEY)  # 64 位
        print('[ENCRYPT_LOG] w:', word)
    left, right = split_str_2(word)
    Ki = hmac_md5(left, HASH_KEY)  # 32 位
    ran = gen_random(seed)  # 产生 32 位随机数
    Fki = hmac_md5(ran, Ki)
```

```
        res = str(ran) + str(Fki)
        Ci = strXOR(res, word)
        return Ci, word, Ki, ran
```

4.2 检索过程

客户端需要把要查询关键词对应的 X_i 和 K_i 告知服务器来进行检索，生成方式与加密过程相同。

```
def Search_msg(ss):
    # 直接检查所有的邮件列表里有没有这个词，有的话把对应位置的 Encrypt_word 对象取出来
    '''
    其实在这一步应该直接让用户输入对应的 CI、XI、KI、SI
    但是就一个简单的 demo 而言还要用户交互几百个 hex 不太现实
    不如我直接在所有的已有的 email 里搜索
    '''
    index_i = -1
    index_j = -1

    for i in range(0, len(test_strs)):
        words = Encrypt.split_str(test_strs[i])
        for j in range(0, len(words)):
            if words[j] == ss:
                index_i = i
                index_j = j
                break
    if index_i == -1 & index_j == -1:
        print("[ERROR] : DO NOT HAVE THIS WORD")
        return None, None

    ee = Encrypt_mg[index_i][index_j]
    correct_emails = []
    for email in Server_mg:
        for sw in email:
            if Encrypt.Search_pipeline(sw.getCi(), ee.getXi(), ee.getKi(), HASH_KEY):
                correct_emails.append(email)

    return correct_emails, ee
```

服务器得到 X_i 和 K_i 后，先计算 C_i 和 X_i 异或得到 T_i

$$T_i = X_i \oplus C_i$$

将 T_i 划分为 T_iL 和 T_iR

$$T_i =< T_iL, T_iR >$$

使用带密钥的 hash 函数 H2 对 T_iL 进行加密，以 K_i 为密钥，将结果与 T_iR 比较，相同则检索成功。

$$H_{2K_i} == T_iR$$

同样的，我们将服务器进行检索与返回 msg 的过程封装在一个函数中

```
# 检索算法的 pipline
def Search_pipeline(Ci, Xi, ki, HASH_KEY):
    print()
    print('SEARCH PROCESS:')
    Ti = strXOR(Ci, Xi)[2:]
    left, right = split_str_2(Ti)
    cal_Right = hmac_md5(left, ki)
    print('[SEARCH_LOG] right:', right)
    print('[SEARCH_LOG] cal_Right:', cal_Right)

    if right == cal_Right:
        print('[SEARCH_LOG] Search Success!')
        return True
    else:
        return False
```

4.3 解密过程

我们对服务器发回的检索结果进行解密，流程如下: 逐个解密然后打印最后的结果，解密过程在下面详述。

```
def Decrypt_emails(ew):
    mails = []
    for mail in Decrypt_mg:
        temp_mail = []
        for sw in mail:
            temp_mail.append(Encrypt.Decrypt_pipeline(ew.getSi(), sw.getCi(), ew.getKi(), DES_
        mails.append(temp_mail)

    print()
    print("RESULT:")
    for email in mails:
```

```
        print("[msg]", end=' ')
        for word in email:
            print(word, end=' ')
        print()
```

先使用伪随机函数 Random，输入随机种子 seed，来得到伪随机序列 S_i。
将 C_i 划分为 C_iL 和 C_iR。

$$C_i =< C_iL, C_iR >$$

S_i 和 C_iL 异或得到 L_i

$$L_i = S_i \oplus C_iL$$

使用带密钥的 hash 函数 H1 对进 L_i 行加密，以 hash_key 为密钥，得到新的密钥 k_i。
使用带密钥的 hash 函数 H2 对 S_i 进行加密，以 k_i 为密钥，得到 FK_i。
将 FK_i 与 C_iR 异或得到 R_i。

$$R_i = C_iR \oplus FK_i$$

拼接 L_i 与 R_i 得到 X_i，使用 DES 解密即可。

$$X_i =< L_i, R_i >$$

```
def Decrypt_pipeline(Si, Ci, Ki, DES_KEY, HASH_KEY):
    print('DECRYPT PROCESS:')
    Cil, Cir = split_str_2(Ci)
    Li = strXOR(Si, Cil)
    Fki = hmac_md5(Si, Ki)
    Ri = strXOR(Cir, Fki)
    X = Li + Ri
    word = decrypt(str(X)[2:18], DES_KEY)
    print('[DECRYPT_LOG] word:', word)
    return word
```

5 实验结果

- 加密过程

```
[ENCRYPT_LOG] w: My
[ENCRYPT_LOG] w: a35d95a95875e9743f60700c0961f8929e86aba244ca638ea8cbca152ea66bc9
[ENCRYPT_LOG] w: name
[ENCRYPT_LOG] w: e283cc655bc45c3eab79d1ae4b6d9b43efda17b59048dc3382a11dd498b7d03a
[ENCRYPT_LOG] w: is
[ENCRYPT_LOG] w: c3082096d63be756f63feaf57f3418b97cff33c4527ae13ea940e76b3ea0c0aa
[ENCRYPT_LOG] w: Gloria
[ENCRYPT_LOG] w: f7241783cd2a0ed860c0d8d80ab81021282ca05fd4cf8203d1aab89b0c95b5e4
[ENCRYPT_LOG] w: Your
[ENCRYPT_LOG] w: 719d6743ad677549694199472515e53242aba48c857771f0a5aab807d7ada6df
[ENCRYPT_LOG] w: name
[ENCRYPT_LOG] w: e283cc655bc45c3eab79d1ae4b6d9b43efda17b59048dc3382a11dd498b7d03a
[ENCRYPT_LOG] w: is
[ENCRYPT_LOG] w: c3082096d63be756f63feaf57f3418b97cff33c4527ae13ea940e76b3ea0c0aa
[ENCRYPT_LOG] w: data
[ENCRYPT_LOG] w: e520fc702195fae96d82f78b5e07bf36d48fe69217092ee4db15bac2bd41b01c
[ENCRYPT_LOG] w: I
[ENCRYPT_LOG] w: 00df2695661327f5876ec32db9fd33d9b6bd2a6f0269d4750b2538bbaa765204
[ENCRYPT_LOG] w: learn
[ENCRYPT_LOG] w: 92f07de12a594962c934443b211181ad308ff8f272d71c9972db549663438623
[ENCRYPT_LOG] w: 1
[ENCRYPT_LOG] w: beb054c0db6d7b11b4a18f1098a6c94e21783730e3475f61a2cdf1207e0f8fc2
[ENCRYPT_LOG] w: subject
[ENCRYPT_LOG] w: 72fb20a9b6e671328d4329d9c1cbd4047e5cea4c595cf0c29a3dbb0162da9ecb
```

图 5.2: 第一层加密结果（DES 加密）

```
No. 1  Client_msg

[ENCRYPT_W] CI =  0x8a8154e824024fb7286ea665c56a8c2e74a86c20293517f5aa3ed0b171019b12
[ENCRYPT_W] XI =  a35d95a95875e9743f60700c0961f8929e86aba244ca638ea8cbca152ea66bc9
[ENCRYPT_W] KI =  b17d1e9fa8d8b4d69704aced04e6eb60
[ENCRYPT_W] SI =  29dcc1417c77a6c3170ed669cc0b74bc

[ENCRYPT_W] CI =  0xcb5f0d2427b3fafdbc7707c78766efff146ec9a684e7435129930cacd3d9550a
[ENCRYPT_W] XI =  e283cc655bc45c3eab79d1ae4b6d9b43efda17b59048dc3382a11dd498b7d03a
[ENCRYPT_W] KI =  426ed9afa048d4ee86ed78c387488e77
[ENCRYPT_W] SI =  29dcc1417c77a6c3170ed669cc0b74bc

[ENCRYPT_W] CI =  0xead4e1d7aa4c4195e1313c9cb33f6c057672fcd316eea10a6a9204f914144e79
[ENCRYPT_W] XI =  c3082096d63be756f63feaf57f3418b97cff33c4527ae13ea940e76b3ea0c0aa
[ENCRYPT_W] KI =  95d0306ff700a3a64492686ee43224a5
[ENCRYPT_W] SI =  29dcc1417c77a6c3170ed669cc0b74bc

[ENCRYPT_W] CI =  0xdef8d6c2b15da81b77ce0eb1c6b3649d89c767365f601b9cf5ef419d203a09d5
[ENCRYPT_W] XI =  f7241783cd2a0ed860c0d8d80ab81021282ca05fd4cf8203d1aab89b0c95b5e4
[ENCRYPT_W] KI =  f3e2d037ee662f46bb5115db9c40ef05
[ENCRYPT_W] SI =  29dcc1417c77a6c3170ed669cc0b74bc
```

图 5.3: 第二层加密结果

```
No. 1  Server_msg
[SERVER_W] CI =  0x8a8154e824024fb7286ea665c56a8c2e74a86c20293517f5aa3ed0b171019b12

[SERVER_W] CI =  0xcb5f0d2427b3fafdbc7707c78766efff146ec9a684e7435129930cacd3d9550a

[SERVER_W] CI =  0xead4e1d7aa4c4195e1313c9cb33f6c057672fcd316eea10a6a9204f914144e79

[SERVER_W] CI =  0xdef8d6c2b15da81b77ce0eb1c6b3649d89c767365f601b9cf5ef419d203a09d5
```

图 5.4: 服务端存储 C_i

- 生成陷门检索过程

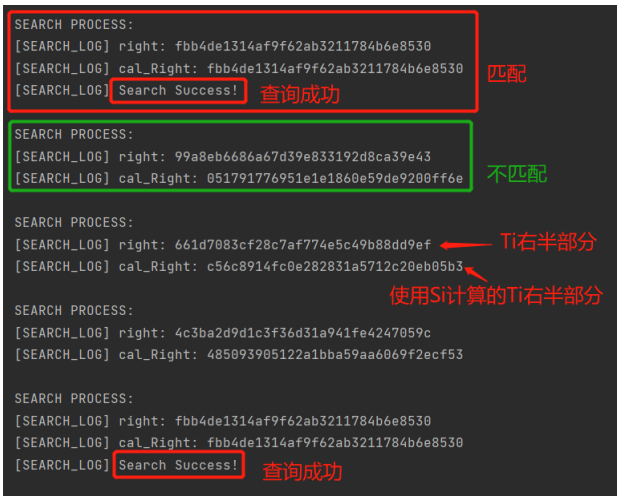


图 5.5: 检索过程

- 解密过程

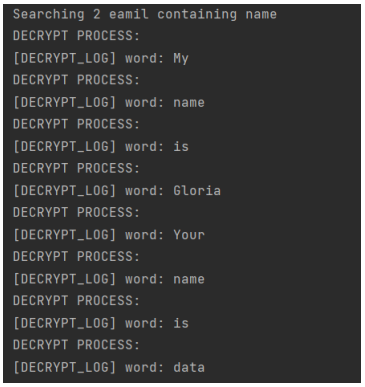


图 5.6: 解密结果

- 输出结果

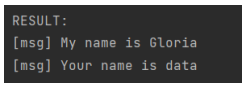


图 5.7: 输出结果

6 心得体会

通过学习可搜索加密，我了解了对称可搜索加密搜索和非对称可搜索加密搜索两种加密搜索技术。通过实现基于正向索引的 SWP 对称可搜索加密，我更深入地了解这项技术，并且掌握实现原理及其实际应用。在实验过程中，我意识到了以下几点：

- 对称可搜索加密技术能够保护数据的隐私，同时也允许用户对数据进行搜索。这种技术非常适合在云计算和数据共享领域中使用。
- 实现对称可搜索加密需要结合多种密码学算法，包括哈希函数、对称加密、伪随机数生成和搜索算法等。这些算法的结合可以实现对数据的安全加密、高效搜索、匹配和查询。

- 通过数据安全方面知识的学习，我明白了在实验中需要不断更新知识储备，了解最新的安全技术和算法，同时也需要结合具体应用进行优化。这些都有助于我们更好地理解和应用对称可搜索加密技术。