数据安全实验报告

Lab6 对称可搜索加密方案
网络空间安全学院 信息安全专业
2112492 刘修铭 1036

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 实验原理

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：

- 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。

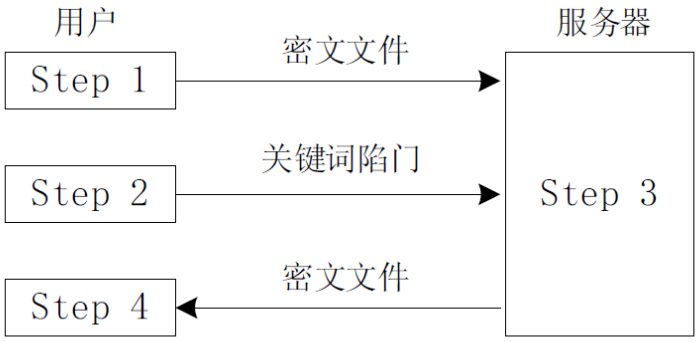

图 6.1.1 可搜索加密过程

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：

- 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。

- 非对称可搜索加密 (Asymmetric searchable encryption, ASE)：旨在加解密过程中采用公钥对明文信息加密和目标密文的检索，私钥用于解密密文信息和生成关键词陷门。非对称可搜索加密算法通常较为复杂，加解密速度较慢，其公私钥相互分离的特点，非常适用于私钥生成待检索关键词陷门，可用于多用户同时进行关键词检索的场景等。

本次实验中，主要关注对称可搜索加密，其流程为：在加密时，用户执行 KeyGen 算法生成对称密钥 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 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 K 解密所有返回文件，得到目标文件。

对称可搜索加密有如下两种实现思路。

1. 正向索引

基本构造思路是：将文件进行分词，提取所存储的关键词后，对每个关键词进行加密处理；在搜索的时候，提交密文关键词或者可以匹配密文关键词的中间项作为陷门，进而得到一个包含待查找的关键词的密文文件。

因为是按照“文档标识 ID：关键词 1, 关键词 2, 关键词 3, ..., 关键词 n”的方式组织文档与关键词的关系，称这种方式为正向索引（和后面倒排索引进行区分）。

2. 倒排索引

倒排索引，也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。如图所示，关键词 w_1 索引了一个列表，列表中存储了所有关联的文档的标识 ID。

Key	Value
w_1	$D_1, D_4, \ldots, D_{15}$
w_2	$D_1, D_3, \ldots, D_{22}$
$\vdots$	$\vdots$
w_m	$D_2, D_3, \ldots, D_n$

⇨

Key	Value
$E(w_1)$	$D_1, D_4, \ldots, D_{15}$
$E(w_2)$	$D_1, D_3, \ldots, D_{22}$
$\vdots$	$\vdots$
$E(w_m)$	$D_2, D_3, \ldots, D_n$

3 实验过程（含主要源代码）

3.1 正向索引

3.1.1 *数据初始化*

创建一个可搜索加密类，初始化 index/data 和 keys 为字典。在此例子中，使用一个简单的正向索引数据结构，其中每个索引是一个单词，每个值是包含该单词的文档ID的列表。index 存储了每个文档的加密单词列表，data 存储了每个文档的加密数据，keys 存储了每个单词的陷门（trapdoor）。

```
class SearchableEncryption:
    def __init__(self):
        self.index = {}  # The 'forward index'
        self.data = {}  # The 'encrypted data'
        self.keys = {}  # The 'trapdoors' (i.e., keys)
```

3.1.2 encrypt 函数

`encrypt` **函数**：接受一个 ID 和一段文本作为输入，并将文本加密后存储起来。加密过程是通过将文本拆分成单词，然后对每个单词进行哈希处理（使用 SHA-256 哈希函数），最后将所有单词的哈希值连接起来。

在这里，输入单词首先被编码为字节串（byte string），即由 ASCII 或 Unicode 字符组成的二进制序列，然后被传递给 SHA-256 哈希函数进行处理。最终，这个加密过的单词被存储在 self.keys 字典中，其中它是一个“trapdoor”，可以用于后续的搜索操作。由于 SHA-256 哈希函数具有单向性质，因此，只有知道原始输入的人才能计算出对应的哈希值。这种“trapdoor”机制可以实现可搜索加密的核心功能：在保持数据加密的同时，允许使用某些密钥搜索数据。

```
def encrypt(self, id, text):
    words = text.split()
    encrypted_words = []

    for word in words:
        # Simple 'encryption' by hashing the word
        encrypted_word = hashlib.sha256(word.encode()).hexdigest()
        encrypted_words.append(encrypted_word)

    # Store the encrypted data
    self.data[id] = ' '.join(encrypted_words)

    # Add the encrypted words to the index under the document ID
    self.index[id] = encrypted_words
```

3.1.3 generate_trapdoor 函数

`generate_trapdoor` **函数**：接受一个单词作为输入，并将其加密后的值存储为"trapdoor"（陷阱门）。这里的陷阱门实际上就是单词的加密哈希值。

```
def generate_trapdoor(self, word):
    # The 'trapdoor' is simply the encrypted version of the word
    self.keys[word] = hashlib.sha256(word.encode()).hexdigest()
```

3.1.4 search 函数

`search` **函数**：通过使用"trapdoor"来搜索索引，找到包含特定单词的文档 ID。如果输入的单词存在于索引中，则返回包含该单词的文档 ID 列表；否则返回一个空列表。

```
def search(self, word):
    # Use the 'trapdoor' to search the index and find document IDs
    if word in self.keys:
        encrypted_word = self.keys[word]
        return [id for id, words in self.index.items() if encrypted_word in words]
    else:
        return []
```

3.1.5 decrypt 函数

`decrypt` **函数**：这个函数简单地返回加密后的数据。在主函数中进行解密

```
def decrypt(self, id):
    # 'Decrypt' the data by simply returning it - in a real system, this would not be possible without the decryption key
    return self.data[id]
```

3.1.6 main 函数

在后面，对前面编写的函数进行实现并检验。首先创建一个 SE 类对象，接着插入两个数据用作测试。接着调用函数生成陷门，搜索数据，并进行解密。

```
se = SearchableEncryption()

# Encrypt some data
se.encrypt(1, "hello world")
se.encrypt(2, "goodbye world")

# Generate some trapdoors
se.generate_trapdoor("hello")
se.generate_trapdoor("world")

# Search for some data
print(se.search("hello"))  # [1]
print(se.search("world"))  # [1, 2]

# Decrypt some data
print(se.decrypt(1))
print(se.decrypt(2))
```

3.2 倒排索引

3.2.1 数据初始化

构造一个可搜索加密类，有三个属性：index、data 和 keys，分别存储索引、加密数据和陷门（trapdoors）。索引使用了 defaultdict 类型，这样在索引中不存在某个加密单词时，会自动创建一个空列表，而无需手动检查键是否存在。

```
class SearchableEncryption:
    def __init__(self):
        self.index = defaultdict(list)
        self.data = {}  # The 'encrypted data'
        self.keys = {}  # The 'trapdoors' (i.e., keys)
```

3.2.2 encrypt 函数

`encrypt` **函数**：接受文档的 ID 和文本作为输入，然后将文本加密并存储。加密过程是将文本拆分成单词，然后对每个单词应用哈希函数（SHA-256）进行加密，得到单词的哈希值。同时，将文档 ID 添加到与加密单词关联的索引列表中，以便后续的搜索。

```
def encrypt(self, id, text):
    words = text.split()
    encrypted_words = []

    for word in words:
        # Simple 'encryption' by hashing the word
        encrypted_word = hashlib.sha256(word.encode()).hexdigest()
```

```
        encrypted_words.append(encrypted_word)

        # Add the document ID to the index under the encrypted word
        self.index[encrypted_word].append(id)

    # Store the encrypted data
    self.data[id] = ' '.join(encrypted_words)
```

3.2.3 generate_trapdoor 函数

`generate_trapdoor` **函数**：接受单词作为输入，然后将单词的加密哈希值存储为该单词的“陷阱门”。这里的陷阱门实际上就是单词的加密哈希值，它用于后续的搜索操作。

```
def generate_trapdoor(self, word):
    # The 'trapdoor' is simply the encrypted version of the word
    self.keys[word] = hashlib.sha256(word.encode()).hexdigest()
```

3.2.4 search 函数

`search` **函数**：这个方法接受一个单词作为输入，然后使用该单词的陷阱门在索引中查找与之关联的文档 ID 列表。如果该单词存在于陷阱门中，就返回与之关联的文档 ID 列表；否则返回一个空列表。

```
def search(self, word):
    # Use the 'trapdoor' to search the index and find document IDs
    if word in self.keys:
        return self.index[self.keys[word]]
    else:
        return []
```

3.2.5 decrypt 函数

`decrypt` **函数**：这个函数简单地返回加密后的数据。在主函数中进行解密

```
def decrypt(self, id):
    # 'Decrypt' the data by simply returning it - in a real system, this would not be possible without the decryption key
    return self.data[id]
```

3.2.6 main 函数

在后面，对前面编写的函数进行实现并检验。首先创建一个 SE 类对象，接着插入两个数据用作测试。接着调用函数生成陷门，搜索数据，并进行解密。

```
se = SearchableEncryption()

# Encrypt some data
se.encrypt(1, "hello world")
se.encrypt(2, "goodbye world")

# Generate some trapdoors
se.generate_trapdoor("hello")
se.generate_trapdoor("world")

# Search for some data
```

```
print(se.search("hello"))  # [1]
print(se.search("world"))  # [1, 2]

# Decrypt some data
print(se.decrypt(1))
print(se.decrypt(2))
```

4 实验结果及分析

4.1 正向索引

运行程序，得到如下的结果，可以看到，按照如期要求检索出了对应的信息，说明程序编写正确。

```
lxm@lxmliu2002:~/datasecurity$ /usr/bin/python3 /home/lxm/datasecurity/forward_index.py
[1]
[1, 2]
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824 486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7
82e35a63ceba37e9646434c5dd412ea577147f1e4a41ccde1614253187e3dbf9 486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7
```

4.2 倒排索引

运行程序，得到如下的结果，可以看到，按照如期要求检索出了对应的信息，说明程序编写正确。

```
lxm@lxmliu2002:~/datasecurity$ /usr/bin/python3 /home/lxm/datasecurity/inverted_index.py
[1]
[1, 2]
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824 486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7
82e35a63ceba37e9646434c5dd412ea577147f1e4a41ccde1614253187e3dbf9 486ea46224d1bb4fb680f34f7c9ad96a8f24ec88be73ea8e5a6c65260e9cb8a7
```

5 遇到的问题及解决方案

本次实验中遇到的问题主要是对于可搜索加密方案的掌握不熟悉，编程实现时会比较生疏。

6 参考

本次实验主要参考教材内容完成。