A report on Trapdoor-based Searchable Symmetric Encryption scheme

EN.601.745 | Advanced Topics in Applied Cryptograph

Sunitha Narayan (snaray18/snaray19)

Table of Contents

Introduction

Project is available on Github.
See code here

Searchable encryption

Symmetric Searchable encryption (SSE) is an encryption technique that allows users to search on encrypted data in a way that privacy of both, the files and search queries are preserved. This way an untrusted server cannot learn anything about the plaintext other than the search result. This encryption scheme provides query isolation for searches. This project demonstrates one such trivial implementation of SSE.

In this construction, we use trapdoor-based Searchable Symmetric Encryption scheme to query data on the server characterized by attributes using keywords. We build an index to track items in the input file. In our scheme, a word is represented in an index by a codeword derived by applying AES encryption twice –once with the word as input and once with a unique document identifier as input building a secure index. Also, there is a trapdoor generated by the client for every keyword as an input under the masterkey of the client. After receiving a query with the keyword information from the client, the server runs the SE algorithm and returns a list of identifiers from the document that contains the keyword. This way, the server learns nothing about the index or the keyword of the client.

Cryptographic schemes used

We first review AES encryption, trapdoor functions, and secure index.

AES Encryption

Block cipher

A block cipher is an algorithm that allows us to encrypt blocks of a fixed length. It provides an encryption function E that turns plaintext blocks P into ciphertext blocks C, using a secret key k. In this implementation, we use block cipher Advanced Encryption Standard (AES) of a symmetric-key encryption scheme.

Block cipher modes of operation

A common way of producing a stream cipher is to use a block cipher in a particular configuration. Here we choose CBC mode over EBC mode.

AES-EBC Mode

Electronic Codebook (ECB) mode of operation has some fundamental and well-known problems. Firstly, identical blocks of plaintext yield identical corresponding blocks of ciphertext will not achieve any desirable privacy. SInce it operates in deterministic mode with no initialization vector, randomness, or state it has no Semantic CPA security.

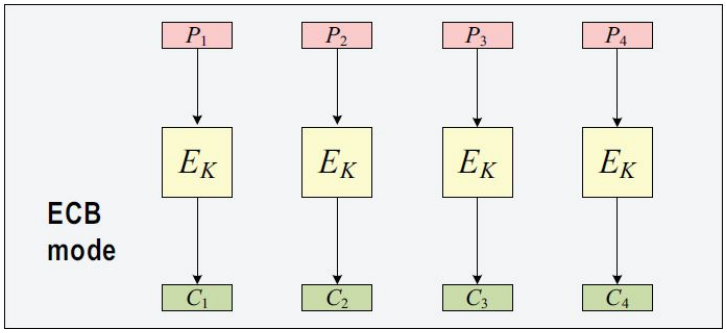

Figure 1: AES-EBC mode encryption [Ref.4]

AES-CBC Mode

We pick one of the classical IV-based modes of operation - Cipher block chaining (CBC). Here the plaintext blocks are XORed with the previous ciphertext block before being encrypted by the block cipher along with an Initialization vector(IV) which is unpredictable and ideally, cryptographically random. This is Semantic CPA secure because of the random IV is used in the implementation. It is important to ensure that the Initializing Vector (IV) used for the encryption is truly random as there are cryptographic attacks that are successful on encryptions where the same IV is used or just hardcoded into the program.

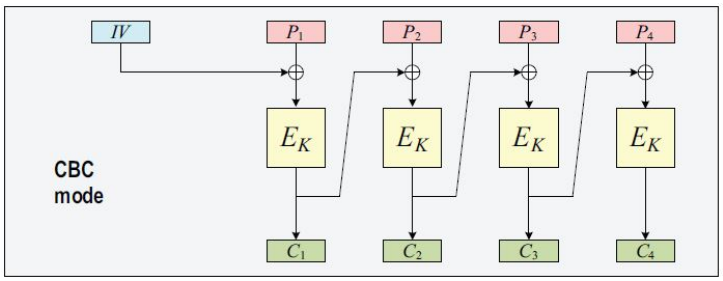

Figure 2: AES-CBC mode encryption [Ref.4]

Why we choose AES mode CBC over EBC

Firstly EBC is not Semantic CPA secure, it means that repetitions of plaintexts, in a deterministic scheme, reveals the corresponding ciphertexts. It is suggested that the key should be used only once in this mode. In our implementation we use AES encryption twice with the same master key per query. Hence, this scheme will not give us a strong encryption. We use EBC only in the sse query program that searches on already encrypted files and keyword.

There arises a need to use AES in CBC mode which provides security if the Initialisation Vector (IV) is truly random. The value of IV can be a nonce, a random value, or a tweak. We use urandom() to generate out values for every encryption and a masterkey.

Trapdoor

We use trapdoor as the cryptographic primitive here to search the encrypted file for a particular keyword. Since we don't want to disclose the search query directly, a trapdoor is generated for every keyword which is in turn used in the search algorithm. This will be discussed in detail in further sections.

Proposed Architecture of search over encrypted file

In this construction, we use trapdoor-based SSE scheme to query data on the server characterized by attributes using keywords. A trapdoor is generated by the client for every keyword as an input under the master key of the client. This implementation is the interaction between two parties. The client owns the data which is encrypted along with its associated search index. After receiving a query with the keyword information from the client, the server runs the Searchable Encryption algorithm and returns a list of identifiers from the document that contains the keyword. This way, the server learns nothing about the index or the keyword of the client.

1. Keygen: Key generation algorithm is run by the user to set up the scheme. It takes as input a security parameter k, and outputs a master key MK.
2. Trapdoor: Generated by the client using master key MK and keyword, w as the input, and outputs the trapdoor, t of keyword, w.
3. BuildIndex: The input is the document D as the input file that is to be encrypted.

A secure index is a data structure that allows user with a trapdoor for a keyword, k to test if the index contains x; The index reveals no information about its contents without valid trapdoors, and trapdoors can only be generated with a secret key. This makes it efficient to search on encrypted files.

1. For each unique word, compute
   (a) A trapdoor
   (b) A codeword for w
   (c) Insert the codeword into the index.
   (d) Output the encrypted index file
2. SearchIndex: Now we take input as the trapdoor T for word w and the index I.
   (a) Compute the codeword for w in the document
   (b) If the keyword exists, output the identifier.

Implementation notes

Setup and demo

1. Generate a masterkey using: $ python3 masterkey_gen.py > masterkey

   The key generated will be used as the masterkey for encrypting the input file, input.csv and generate a trapdoor for the keyword.

Build Index

2. Build an Index using build_index.py: $ python3 build_index.py

   (a) Phase I: input_index: The input file is encrypted using the masterkey and index is generated.

   (b) Phase II: Generate trapdoor: Uses masterkey and keyword as the input, and output the trapdoor of keyword.

```
sn@sn:~/Desktop/Advanced_Crypto/sse$ python3 build_index.py

        Phase I: Building Index

Please input the file/directory where the master key is stored:  masterkey

Successfully added master key.

An index file will be generated for the input file.
        Please input the file to be encrypted:  input.csv
        please input the file where the keywords are listed:  keyword

The time taken to generate the input_index.csv file is
3.9344851970672607
None

        Phase II: Trapdoor Generation for a keyword

Please input the keyword you want to search for in the encrypted input file:  Month

A trapdoor for the keyword is successfully generated.
        The trapdoor is saved as keyword_trapdoor.csv file

b'\x87\xc3)8\xfe\xd7\xfe\xc1\x81\x06\xc2\xe5\xe0YB\xf4\x13\x8b.x\xec"\xcb\x0b\x0fjF\t\xe3/\x1f\xf3'
Finished
```

Search Query

1. sse_query.py: Used to search the encrypted input file for specific keywords that are saved as trapdoors. The output prints the list of all identifiers for the keyword that is searched for.

```
sn@sn:~/Desktop/Advanced_Crypto/sse$ python3 sse_query.py

Please input the index file you want to search:  input_index.csv

Please input the file stored the trapdoor you want to search:  keyword_trapdoor.csv
The identifiers that contain the keyword are:
[2049, 2050, 4, 15, 23, 27, 2084, 41, 46, 50, 56, 2108, 70, 2118, 2119, 86, 94, 2148, 2149, 2151, 2168, 130, 2183, 2185, 2202, 2203, 2207, 2214, 171, 2
224, 188, 191, 209, 2261, 2264, 220, 2271, 231, 2282, 2295, 248, 2301, 253, 255, 256, 264, 270, 273, 283, 2337, 302, 305, 2357, 2359, 2363, 2378, 2383,
 343, 348, 2400, 354, 2404, 2405, 363, 364, 366, 2419, 2425, 382, 2432, 2439, 393, 396, 2449, 405, 2461, 416, 421, 2477, 2485, 443, 2498, 452, 466, 251
4, 470, 2521, 2531, 485, 2535, 2537, 495, 2550, 512, 519, 2569, 2584, 2591, 2596, 551, 2603, 555, 2625, 2626, 2631, 585, 586, 603, 2654, 610, 2658, 267
0, 641, 644, 2697, 650, 2699, 2701, 662, 678, 679, 680, 2728, 693, 2750, 710, 2765, 2767, 726, 2784, 739, 740, 743, 746, 2800, 771, 2820, 774, 777, 784
, 785, 2836, 2837, 791, 795, 2855, 811, 813, 820, 2869, 2886, 2894, 847, 849, 854, 857, 2906, 2908, 863, 878, 2927, 2935, 2940, 2941, 909, 916, 2966, 9
22, 943, 3003, 3004, 959, 3014, 968, 981, 985, 3038, 3040, 1005, 3056, 1033, 3084, 3089, 3091, 3100, 3101, 3104, 3106, 3110, 3119, 1076, 3125, 1084, 31
35, 3136, 1092, 1097, 1100, 3152, 1109, 3168, 1126, 3174, 3176, 1133, 3193, 3195, 3196, 3198, 3200, 3206, 3209, 1162, 3214, 1178, 3226, 3228, 3231, 323
2, 1186, 1194, 3242, 3251, 1204, 1209, 1211, 3264, 3269, 3270, 1225, 1232, 3280, 1235, 1246, 1249, 1256, 1258, 3321, 3337, 3339, 1308, 1312, 3372, 3373
, 1325, 1330, 1337, 1352, 3401, 3407, 1361, 1364, 1373, 1374, 3421, 3427, 3428, 1395, 1396, 3449, 1406, 1407, 1408, 1409, 3466, 1422, 1425, 1426, 3476,
 3481, 3482, 1438, 1439, 3490, 3491, 1444, 1445, 1446, 1448, 3496, 3501, 3507, 3514, 3517, 3521, 1475, 3523, 1481, 1483, 1500, 1503, 3554, 3564, 3566,
3568, 1523, 3575, 3578, 1532, 3593, 1545, 3595, 1553, 3604, 1568, 1571, 1572, 3625, 3632, 3637, 3641, 3646, 1599, 3654, 3663, 1623, 1628, 3684, 1641, 3
698, 1653, 1658, 3707, 1661, 1665, 1678, 1687, 3737, 1692, 3741, 3745, 3748, 3753, 1708, 1717, 1718, 3771, 1730, 1732, 1733, 3788, 1752, 3801, 1759, 17
62, 1768, 1783, 3832, 1785, 1787, 1791, 3847, 3848, 1803, 3851, 3854, 3870, 1824, 3876, 3880, 1834, 3888, 1849, 1854, 3902, 3908, 3921, 1881, 1882, 393
1, 1890, 3940, 1896, 3945, 1898, 1900, 1903, 1905, 3954, 1911, 1912, 3959, 3963, 1921, 3978, 1933, 1947, 1950, 4000, 1953, 1972, 1981, 1983, 1984, 2004
, 2007, 2013, 2018, 2022, 2034]
```

Cryptographic Attacks on implementation

We mostly focus on how CBC mode performs and what could be other choices of modes using IV. The table [Ref4] below summarises the characteristics of modes CBC, CFB, and OFB which are all semantic CPA secure if the IV is random.

Characteristic	CBC	CFB	OFB
SemCPA secure if the IV is **random**.	**Yes**. Proof in [14]. The ind$ security notion is easy to establish.	**Yes**. Proof in [2]. The ind$ security notion can also be shown.	**Yes**. Follows directly from ind$-security of CBC.
SemCPA secure if the IV is a **nonce**.	**No**. Enciphering nonce under E_K also incorrect. Incorrect use is common.	**No**. Enciphering nonce under E_K also incorrect.	**No**. But secure with a fixed sequences of IVs, like a counter.
SemCCA secure.	**No**.	**No**.	**No**.
Nonmalleable (cannot manipulate ciphertexts to create related plaintexts)	**No**. Often wrongly assumed to provide some sort of nonmalleability.	**No**. Trivially malleable.	**No**. Trivially malleable.
Padding oracle (says if the ciphertext is correctly padded) is well tolerated.	**No**. Pad oracle destroys SemCPA for most pad methods [42, 165, 194].	Usually **Yes** (or **NA**) as no padding is needed for small enough s.	Usually **Yes** (or **NA**) as no padding is ever needed with OFB.
Error propagation: will recover if a ciphertext bit is flipped (a "bit error").	**Yes**. Changes to C_i affect two blocks, P_i and P_{i+1}	**Yes**. Changes to C_i affect about n/s data segments	**Yes**. Changes to C_i affect only P_i
Self-synchronous: will recover if insert / delete ciphertext bits (a "slip").	**No**, unless the slip is a multiple of n-bits.	**Yes** for slips of s-bits (so always if $s = 1$). **No** for other slips.	**No**.
Maximal rate scheme: processes n-bits for every blockcipher call.	**Yes** (n bits processed per blockcipher call).	**No** (when $s < n$). The mode processes s bits per blockcipher call.	**Yes** (n bits processed per blockcipher call).
Parallelizable with respect to **encryption**.	**No**. Can always use interleaving to enhance parallelizability.	**Yes**, but awkward. See ISO 10116 for less awkward generalization.	**No**.
Parallelizable with respect to **decryption**.	**Yes**. Block P_i depends only on C_i, C_{i-1}.	**Yes**, but awkward.	**No**.
Inverse-free (blockcipher inverse is unnecessary for decryption).	**No**.	**Yes**.	**Yes**.
Preprocessing improves encrypt/decrypt speed.	**No**.	**No**.	**Yes**.

1. Malleability of CBC: An encryption scheme is malleable if an adversary can modify a ciphertext C for a previously unknown plaintext P to create a ciphertext C'whose plaintext P close enough to the plaintext P.

2. Chosen-ciphertext attacks on CBC: Let us look at an attack for CBC with random IV, which is our implementation of AES. The adversary can sends a query to encrypt a message, M and gets back (IV, C). Next, the adversary queries to decrypt (IV' , C') which are random getting back M". If M = M"then the adversary guesses that it is the real encryption oracle. If not otherwise the adversary knows that it is not.

3. Padding attacks on CBC: Padding Oracle lets one decrypt arbitrary ciphertexts that were encrypted under CBC mode with a pad (P) where it is passed as a parameter in the CBC mode. Now, let's look at an attack. Given, E = IV || C, computes P', the CBC-decryption of C with respect to the underlying key K. If P = Pad(P) for some plaintext P. Oracle returns 1, indicating a valid ciphertext–the CBC-encryption of a properly padded plaintext–was provided. Otherwise, the oracle returns 0, indicating an invalid ciphertext.

Bloom filter

Bloom filters are particularly useful in searching on encrypted text. The user maintains the bloom Filter of the document, encrypts the document using an encryption algorithm and then sends both the encrypted document as well as its corresponding bloom filter to the server. For searching an item the user sends its keywords to the server and the server checks the document bloom filter for the presence of the keyword. If the presence of the keyword is established, the encrypted document is returned to the client which is decrypted with the key that was previously used to encrypt the document.

Advantages of using Bloom filter

1. Space efficiency: Bloom filter does not store the actual items but only an array of integers. This makes it space efficient. Saves expensive data scanning across several servers depending on the use case.

A few concerns with bloom filter are:

1. Deleting and Item: If an item has to be deleted or specifically clear set bits corresponding to the bloom filter, it might erase some other data as well because of the many set bits that could be shared among multiple items.
2. Inserted items are not retrievable: Bloom filters do not track items, it just sets different positions in the array.
3. False Positive result as Bloom filters are probabilistic: The size of the bloom filter is usually fixed before populating it. Due to the small array size it will saturate quickly and yield in false positive results. It is advised to have a strategy to reset the bloom filter or use a scalable bloom filter. Addition of elements will increase the probability of generating false positive results.
4. Insertion Search Cost: Efficiency primarily depends on the hash function as an item is passed through k hash function. The hash function used should be independent and uniformly distributed. Performance during insertion or search operation depends heavily on the efficiency of the chosen hash functions. Inserting element searching element takes O(k) times as you run k hash functions to set/check k number of indices in the array.

Extensions

1. Locating Words Within Documents. Instead of using an index for every document, we can divide documents into chunks and create indexes for each chunk. With this modification, words can be located with chunk size granularity within a document.
2. Implement sse using bloom filter choosing suitable Filter Parameters and small array size to make it efficient.
3. Extend this prototype to support search in multi-user settings.
4. Use other AES encryption modes that provide better security and privacy like CBC-MAC, CMAC, HMAC, and GMAC.

Conclusion

A major take away after implementing a single user searchable encryption scheme is that it tries to solve the problem with searching on untrusted servers. Though this is a trivial implementation of the SSE, it is helpful to understand the practical ways to achieve strong and secure encryption. Extending this work could be implementing AES for other modes of operation which provides both confidentiality and authenticity.

References

1. *Dawn Xiaodong Song, David Wagner, Adrian Perrig* "Practical Techniques for Searches on Encrypted Data"

2. *D. V. N. Siva Kumar, P. Santhi Thilagam* "Searchable encryption approaches: attacks and challenges" 2018

3. Block cipher mode of operation, Wikipedia

4. *Phillip Rogaway* "Evaluation of Some Blockcipher Modes of Operation"