University of Bergen
Department of informatics

# Searchable Symmetric Encryption and its applications

*Author:* Kristoffer Borg Nilsen
*Supervisor:* Chunlei Li


UNIVERSITETET I BERGEN
*Det matematisk-naturvitenskapelige fakultet*

May 2022

**Abstract**

In the age of personalized advertisement and online identity profiles, people's personal information is worth more to corporations than ever. Storing data in the cloud is increasing in popularity due to bigger file sizes and people just storing more information digitally. The leading cloud storage providers require insight into what users store on their servers. This forces users to trust their cloud storage provider not to misuse their information. This opens the possibility that private information is sold to hackers or is made publicly available on the internet. However, the more realistic case is that the service provider sells or misuses your metadata for use in personalized advertisements or other, less apparent purposes. This thesis will explore Searchable Symmetric Encryption (SSE) algorithms and how we can utilize them to make a more secure cloud storage service. Source code for the project can be found here: `https://github.com/kni034/Symmetric-searchable-encryption`, and `https://github.com/kni034/secure-indexes`.

**Acknowledgements**

First of all, I would like to thank my supervisor Chunlei Li for his excellent guidance throughout the development of my project, and while writing the thesis. Thank you for always making time for me when I got stuck on a problem or simply wanted to discuss an idea.

I would also like to thank my fellow students at the Department of Informatics for creating an inclusive community, where I have felt at home. I am grateful for the many hours I have spent at the reading hall with interesting discussions and wonderful people. Special gratitude to the master students at the Selmer Center and Glassburet. Thank you for always having my back and reminding me to have fun even in stressful times. I truly could not have done this without you.

Last but not least, a big thanks to my family and friends, both at home and here in Bergen, for all the support and love throughout my studies.

Kristoffer Borg Nilsen
Bergen, 2022

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background on Searchable Encryption

When storing data remotely, secrecy is essential. Encrypting the information stored remotely has been standard from the start. The use of remote storage servers, also called Cloud Servers, is increasing rapidly for personal use and businesses. The standard way of storing information remotely has been to encrypt during transmission and then let the server decrypt and then re-encrypt the data before storing it safely. Allowing the server to use its own encryption makes sense and enables many valuable features for the users. However, this comes at the cost of letting the server view the user's data. In today's world, it is safer to assume that corporations are willing to misuse their user's information if they can get away with it. As users, this is practically impossible to stop. Therefore it is better to avoid the situation altogether. One solution to this problem is a type of encryption called Searchable Encryption (SE) which allows for searching on encrypted data. The two main branches of SE are searchable symmetric encryption and searchable asymmetric encryption [44].

### 1.1.1 Searchable symmetric encryption

Searchable Symmetric Encryption (SSE) is a type of encryption that makes it possible to make hidden searches on encrypted data. Suppose a user,

Alice wants to store a set of documents on a remote server. As Alice does not want the server to be able to view the contents of her documents, she encrypts her files before uploading them to the server. Whenever Alice wants to retrieve any documents containing a specific keyword, she can generate a special token(later referred to as a trapdoor) and ask the server to search for the token in her encrypted documents. The server never learns anything about the keyword or the documents but can determine if the documents contain the keyword or not and return the correct documents.

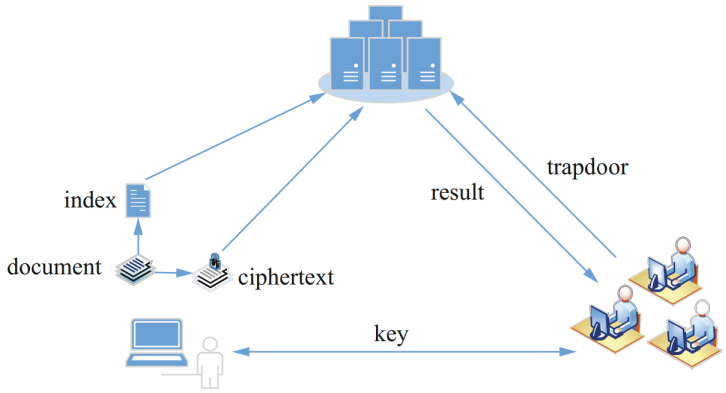


Figure 1.1: The general structure of SSE schemes [44]

The general structure of SSE schemes consists of four main functions: keygen, buildIndex, trapdoor, and search (The scheme by Song et al. in [41] is an exception to this structure).

**Keygen:** Generates a private key. Takes a security parameter as input-(usually the keysize of the system) and outputs the private key.

**BuildIndex:** Generates an index for a file or a set of files. Takes the private key and file(s) as input and outputs the index.

**Trapdoor:** Generates a trapdoor for a keyword. A trapdoor is the encoded (hidden) version of a keyword. Takes a keyword and the private key as input and returns the trapdoor for the given keyword.

**Search:** Finds all documents that contain the provided trapdoor. Takes an index or a set of indices and a trapdoor as input and returns a set of documents (or document references/identifiers).

The flow of most SSE schemes is as follows: During the initialization of a new user, the keygen algorithm is called by the data owner (can be the same person as the 'user'), and the private key is stored locally or distributed locally to the desired users. When the user wants to store a file, they generate its index(locally), encrypt the file with a normal symmetric encryption algorithm, and upload both the index and the file to the server. Depending on the scheme, there are either one index per document or one index for all document. When the user wants to search for a keyword, the user generates a trapdoor locally with the private key and sends the trapdoor to the server. The server calls the search function and returns any matching documents. The user can then decrypt the files with their private key.

### 1.1.2 Searchable asymmetric encryption

Searchable asymmetric encryption, also called Public-key encryption with keyword search (PEKS) works similarly to SSE but in a public-key setting. Suppose Alice uses several different devices to read her emails. Alice wants any emails containing the word 'urgent' to be sent directly to her phone, and the rest should be sent to her laptop. Like in the SSE example, Alice can create a token and send this to the server. In this example, Alice uses her private key to create a token with the keyword 'urgent' and sends this to the server with the instruction to send emails with a matching token to be sent to her phone. Now if another user wants to send an email to Alice, they encrypt the email with a relevant keyword(for example 'urgent') and with Alice's public key and send it to her. The server receives the email and can check if it has a matching token and route it accordingly. Alice then uses her private key to decrypt the email.

The general structure of PEKS schemes:

**Keygen:** Generates a private/public key pair. Takes a security parameter as input and outputs the private/public key pair.

**PEKS:** Encrypts a document while preserving searchability for a specific user. Takes the recipient's public key and a keyword as input, and returns an encrypted document with searchability for that keyword.

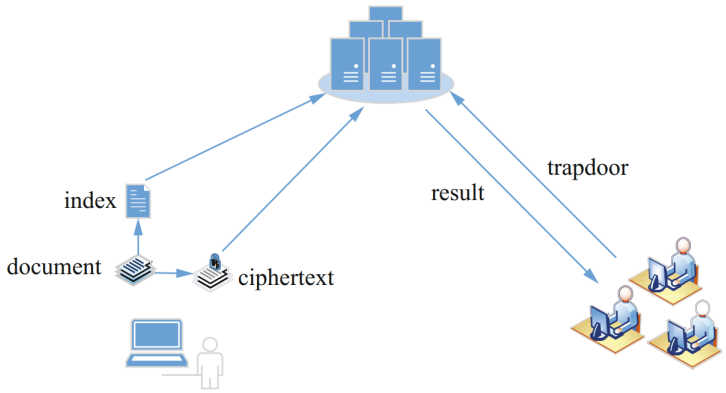


Figure 1.2: The general structure of PEKS schemes [44]

**Trapdoor:** Generates a trapdoor for a keyword. Takes a user's private key and a keyword and returns a token/trapdoor for the given keyword.

**Test:** Tests if an encrypted document was encrypted with a specific token or not. Takes a user's private key, the encrypted document, and a token as input, and returns 'true' if the document and the token were encrypted with the same keyword, otherwise 'false'.

Boneh et al. [14] proposed the first PEKS scheme in 2004, laying the foundation for a research line with several branches. The main branches of PEKS are largely the same as in SSE. Park et al. [38] were the first to propose a scheme using conjunctive search. Bringer et al. [15] proposed an error-tolerant searchable encryption scheme. Like in the first SSEs with fuzzy keyword search, the scheme uses Locality Sensitive Hashing to generate the same hash from similar keywords. Zheng et al. [47] proposed a scheme incorporating verifiable keyword search. Curious readers can find more information about PEKS in the survey paper by Wang et al. [44], and its citations. PEKSs are outside the scope of this thesis and will not be covered further.

## 1.2 Development of Searchable Symmetric Encryption

Initially, the only way to hide the contents of data being stored remotely was to encrypt the data before uploading. This approach worked but was highly ineffective as the user would have to download all the data, decrypt it and search locally every time they wanted to search for something specific in their documents. To solve this problem, Song et al. [41] proposed a scheme that made it possible to partially decrypt files on the server without revealing the file's contents. In this scheme, the user keeps a special private key which is used for both encrypting files and creating trapdoors(called search tokens in the paper). The trapdoors disclose no information about the search contents and can only be created by anyone with the correct private key. [39]

In the field of SSE, there are several different research branches focusing on different techniques of searching on encrypted data. The main branches include searching with a single keyword, fuzzy keyword search, conjunctive keyword search, ranked keyword search, and verifiable keyword search.

### 1.2.1 Single keyword search

In the branch of single keyword search, there have been proposed several ways of searching, mainly how to structure the index table used to perform the lookup of the trapdoor. As mentioned, the first scheme to use SSE was proposed by Song et al. [41].

This scheme does not use an index but performs the search directly on the encrypted text. The encryption is performed in a way that makes it possible for the server, when provided with a trapdoor to partially decrypt and check if the trapdoor is present in the encrypted text. Because of its clever design, only the user with the private key can generate a trapdoor from a search word. Neither the trapdoor nor the partially decrypted ciphertext reveal anything about its contents to the server. As the search is performed directly on the encrypted text and the server has to iterate over the entire document, the search time scales linearly with the length of the document.

As the scheme does not use an index, it does not require much more storage space than the encrypted document itself.

The next two techniques for searching with a single keyword use an index structured like a lookup table. The first uses what Poh et al. [39] call a direct index in their survey paper. With a direct index, the server stores a set of trapdoors for each encrypted file. The set of trapdoors is generated by the user before encryption and reveals no information about the words they were generated from. If a user wants to delete or modify a file on the server, its corresponding set of trapdoors should also be deleted/modified. When users want to search for a keyword, they generate the trapdoor with the search word and their private key and send it to the server. Goh [25] proposed a scheme that uses a Bloom filter as a per document index. A Bloom filter is a datatype that, in constant time can tell you if an element is part of a set or not. With this scheme, search time is linear with the number of documents on the server. This is because the server has to check each file's index during a search query. The storage space required however is increased as each file requires its own index to also be stored on the server.

Using an inverted index works similar to a direct index, but instead of the file pointing to a set of trapdoors, each trapdoor points to a set of files containing that word. Curtmola et al. [20] were the first to propose a scheme that uses an inverted index, they also propose an improved way of measuring the security of the system. The advantage of using an inverted index is that search times are sublinear and optimal in many cases. However, maintaining an inverted index is more complicated than keeping a direct index. When a user adds, removes, or updates a file in a scheme where an inverted index is used, the server has to linearly scan the inverted index table and update every relevant entry. The storage space needed is roughly the same as with a direct index.

The most recent addition to the development of single keyword search is dynamic SSE schemes. Van Liesdonk et al. [33] were the first to tackle the problem of making updates more efficient. In their paper, they propose two variants of their scheme with different qualities regarding search time and storage requirements. The first variant is interactive and the second is not interactive. Kamara and Papamanthou [28] proposed a new scheme that uses a three-structure as the index. The main difference between this method

and the previous methods is that instead of maintaining a map of [trapdoor, document] pairs, the user constructs a search tree where the nodes in the tree are used to perform the search.

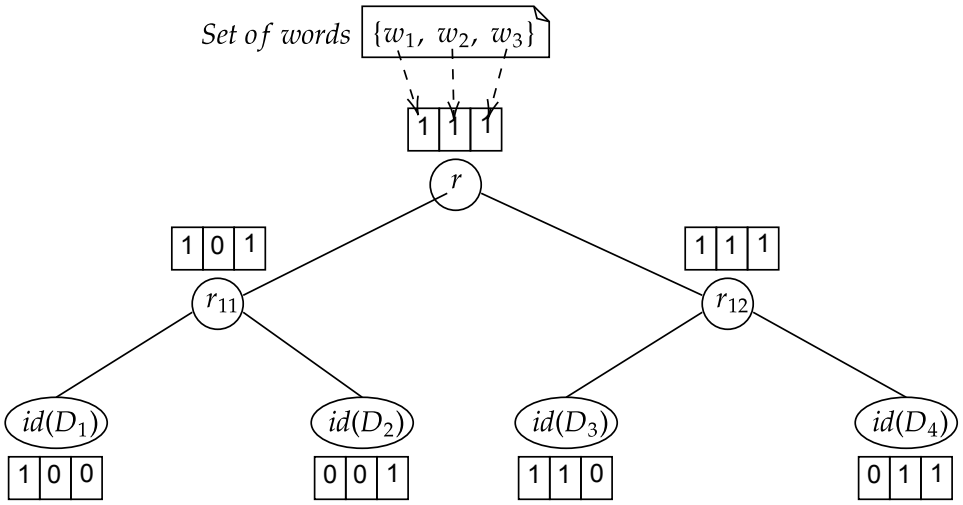


Figure 1.3: Tree-based construction [39]

In Kamara and Papamanthous's [28] scheme index, each node stores information about which words can be found in the documents represented by the children nodes. The tree index stores a boolean list where each bit represents a word being present or not in any of the documents in the children nodes. The example figure above shows that documents $D_1$ and $D_3$ contain word $w_1$ as they both have 1 in the first position of their boolean list. Stefanov, et al. [43] proposed a scheme that combines the positives of the earlier schemes i.e small leakage and high efficiency in search and updates. Their scheme uses sublinear time for searching and updating in the worst case.

### 1.2.2 Fuzzy keyword search

With the single keyword search technique, the search word has to match the stored keywords exactly. Any typo or small inconsistency in the search word will result in the search failing. Fuzzy keyword search tackles this problem by being able to handle minor differences in the search word and the stored keyword.

Li et al. [32] design and utilize a 'wildcard-based' technique related to the concept of edit distance to construct a fuzzy keyword set. They also propose a fuzzy search scheme utilizing the construction of the fuzzy keyword set. Adjedj et al. [13] use their fuzzy keyword scheme to perform fast and secure biometric identification. Like Kuzu et al. [30], they use locality-sensitive hashing (LSH), which will with a very high probability output the same value for two inputs with a small matching score(based on Hamming distance).

### 1.2.3 Conjunctive keyword search

In both single keyword search and fuzzy keyword search, the user provides a trapdoor to the server and receives the documents containing the keyword represented by that trapdoor. If a user wants to search for documents containing multiple keywords, they either have to search with all keywords separately and then locally separate the desired documents, or somehow embed several keywords into each trapdoor when generating the index. Both options are far from ideal. The first option leaks a substantial amount of information to the server, and the second option would make the size of each index scale exponentially.

To solve this problem, Golle et al. [26] proposed two schemes with conjunctive keyword search. The first scheme has communication costs linear to the number of documents but the work can be done offline before the request is sent to the server. The second scheme's search cost is on the order of the number of keywords and does not require any offline work. Cash et al. [18] solve the problem of searching for multiple keywords with boolean queries. Their scheme is not the fastest or most secure but provides a realistic and practical trade-off between security and efficiency. Faber et al. [24] propose an extension to the scheme proposed by Cash et al. [18], adding support for substring-, wildcard-, phase-, and range queries. The new query types do add some cost in performance and storage, but the authors claim that the extension is still practical today, even for large databases.

### 1.2.4 Ranked keyword search

With a ranked keyword search, only the most relevant documents are returned during a query. This can be used to make systems more effective and reduce unnecessary network traffic. Zerr et al. [46] present a ranking model used to create a relevance score transformation function. This lets a server return the most relevant results for a user query without revealing any information about the indexed data. Cao et al. [17] propose the first multi-keyword scheme with ranked search. Xia et al. [45] proposed a scheme with the same multi-keyword ranked search attribute, but also features efficient updates and deletions. The scheme uses a three-structure index and uses a "Greedy Depth-first Search" algorithm for searching.

### 1.2.5 Verifiable keyword search

With a verifiable keyword search, the recipient can check whether the result of a query is complete and correct. This attribute is widely used in general internet communication and helps avoid unnoticed hardware or software errors, as well as protecting the user from semi-honest servers trying to save computation resources. Chai and Gong [19] proposed the first verifiable SSE scheme. The scheme uses a trie-like (prefix tree) structure as the index. This index is used to search, and produce proof that the returned results are valid and complete. Li et al. [31] discuss an aspect of query authentication called query freshness that previously has not been explored. Query freshness means being able to verify that the search result comes from the latest version of the database. Kurosawa and Ohtaki [29] propose a verifiable SSE scheme with a focus on security against active adversaries, as opposed to the more common perspective of a passive (honest-but-curious server) adversary.

## 1.3 Project Summary

This thesis first aims to provide the reader with the knowledge necessary to understand the functions and mechanisms of the schemes implemented in the

main part of the project. The topics explained in the theoretical background include block ciphers, stream ciphers, and hash functions, as well as relevant functions and methods related to those topics.

Chapters 3 and 4 describe the schemes proposed by Song et al [41] and Goh [25] respectively. The two chapters have a similar structure but differ slightly in some areas. They first describe the scheme in detail. In Chapter 3 this is done by defining a very basic version of the scheme. The full scheme is described by introducing three extensions to the basic scheme, where each extension builds on the previous one. Inspiration for this structure of description is taken from the original paper and helps the reader break down the different parts of the rather complex scheme. In Chapter 4 this is done by separately describing the four main components of the scheme mentioned in the background section of the Introduction. Then the implementation of both schemes is presented. The aim of this section is to explain the major components of the implementation and their relation to each other. Various implementation decisions are also discussed. Lastly, a more detailed theoretical performance analysis is followed by a set of tests on the implementation with the same dataset. The goal of the tests is to show roughly how much performance varies between the two schemes, and how much performance varies when changing the internal parameters.

Chapter 5 presents my own proof of concept cloud storage application for images and videos using SSE. The application uses an altered version of the implementation of Goh's Secure Indexes scheme described in Chapter 4. The chapter starts with a brief introduction of the application and the motivation behind it. This is followed by describing the structure that is built on top of the main encryption scheme and how it works. The structure mainly consists of the authentication service and its communication protocols. The next section discusses keyword generation and the challenge of generating good and user-friendly keywords. The section presents several types of keywords and explains how they are generated. Then the performance of the system is discussed, followed by a section describing future work for the application.

Chapter 6 presents a final conclusion discussing the implementations and test results produced throughout the project.

# Chapter 2

# Theoretical Background

## 2.1 Cryptography

Cryptography is the study of techniques for achieving secure storage and secure communication between two or more parties. Secure communication happens when only the intended recipients can read and verify messages, and the contents are hidden from third parties. Historically, the techniques used to hide message contents were done by scrambling characters of the message to make it unreadable. This was done in a way that made it possible to reverse the process and obtain the original message if you knew the key to the hidden message. The most common example of this is the Caesar cipher used by the ancient Romans where the key consisted of a number and decided how much to shift the alphabet used in the message [34, p. 53]. In this cipher, the same key is used to encrypt(hide), and decrypt(reverse) the contents of the message. When the same key is used for encryption and decryption, we call it symmetric encryption. This was for many years the only way to encrypt messages. With the help of computers, the systems we use to perform encryptions have become more advanced. During the 1970s two of the most widely used cryptographic primitives were invented, namely the Diffie-Helman Key Exchange and the RSA cryptosystem [22][40]. What was special with these is that the key used for encryption and decryption is different. What is more fascinating is that the decryption key could be

generated by the recipient by combining their own secret component with a public component shared by the sender. This became known as asymmetric encryption.

In modern-day computer systems, both symmetric and asymmetric encryptions are used. Symmetric encryption is typically faster and more space-efficient, and asymmetric encryption is typically used to set up a secure connection between two parties over an insecure channel. A normal way to set up communication between two parties starts with the sender using asymmetric encryption to encrypt a symmetric key. The recipient then uses the private key and decrypt the symmetric key. Then they both can use the faster, and more efficient symmetric encryption scheme to encrypt their messages. In this thesis, I will not focus on the transmission of data, but rather on the way to encrypt data with symmetric encryption.

## 2.2 Block Cipher

Block ciphers are a type of encryption/decryption algorithm that turns a fixed size input into a pseudorandom fixed-size output with the intent of hiding the contents. Block ciphers are deterministic algorithms, meaning that if initialized with the same parameters, two encryptions of the same input will give the same output. The fixed-size inputs and outputs are called blocks. During the algorithm's initialization, keys to encrypt and decrypt are generated. Block ciphers are considered the standard method to encrypt data on the Internet due to their fast computational speed and that they produce relatively small file sizes.

### 2.2.1 AES

The Advanced Encryption Standard (AES), namely, the cipher Rijndael, was developed by Joan Daemen and Vincent Rijmen in 1998 [12]. Rijndael was chosen as the new encryption standard by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES consists of several versions

of the original Rijndael with different key lengths: 128 bit, 192 bit, and 256 bit, all with the same block size of 128 bits.

The encryption process consists of performing a set number of "rounds" on the current block. Each round, except for the last, consists of 4 different transformations done in order, where the output of one transformation is used as input for the next. The 128-bit block is initially divided into 16, 8-bit chunks called bytes. These 16 bytes are then placed into an array, represented as a $4 \times 4$ matrix during transformations. The four transformations used in AES are the following:

**SubBytes:** Each byte in the input is substituted with a new byte found in the predefined lookup table 2.1.

**ShiftRows:** Each matrix row is shifted a variable amount to the left, looping back on itself. The number of places each row is shifted equals the current row's y position in the matrix. Row 0 is shifted 0 steps to the left(stays the same), row 1 is shifted one position to the left, and so on.

**MixColumns:** Each column in the matrix is transformed into a new column by performing vector-matrix multiplication with a predefined matrix provided in [12].

**AddRoundKey:** Each bit in the matrix is EXORed with the current subkey generated from the key expansion algorithm explained in [12].

The number of rounds performed depends on the chosen key size, ten rounds for key size 128 bits, 12 rounds for key size 192 bits, or 14 rounds for key size 256 bits. Before the first round starts, an AddRoundKey transformation is applied to the block. Then in each round, the four transformations are performed in this order: SubBytes, ShiftRows, MixColumns, then AddRoundKey. In the last round, MixColumns is skipped because it does not add any extra security. The output is converted back into its original form and is now encrypted.

Even complex block ciphers like AES are only designed to securely encrypt a single block with the same key. There have been designed several methods for using block ciphers called modes of operation to get around this.

| | | y | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| x | 0 | 63 | 7c | 77 | 7b | f2 | 6b | 6f | c5 | 30 | 01 | 67 | 2b | fe | d7 | ab | 76 |
| | 1 | ca | 82 | c9 | 7d | fa | 59 | 47 | f0 | ad | d4 | a2 | af | 9c | a4 | 72 | c0 |
| | 2 | b7 | fd | 93 | 26 | 36 | 3f | f7 | cc | 34 | a5 | e5 | f1 | 71 | d8 | 31 | 15 |
| | 3 | 04 | c7 | 23 | c3 | 18 | 96 | 05 | 9a | 07 | 12 | 80 | e2 | eb | 27 | b2 | 75 |
| | 4 | 09 | 83 | 2c | 1a | 1b | 6e | 5a | a0 | 52 | 3b | d6 | b3 | 29 | e3 | 2f | 84 |
| | 5 | 53 | d1 | 00 | ed | 20 | fc | b1 | 5b | 6a | cb | be | 39 | 4a | 4c | 58 | cf |
| | 6 | d0 | ef | aa | fb | 43 | 4d | 33 | 85 | 45 | f9 | 02 | 7f | 50 | 3c | 9f | a8 |
| | 7 | 51 | a3 | 40 | 8f | 92 | 9d | 38 | f5 | bc | b6 | da | 21 | 10 | ff | f3 | d2 |
| | 8 | cd | 0c | 13 | ec | 5f | 97 | 44 | 17 | c4 | a7 | 7e | 3d | 64 | 5d | 19 | 73 |
| | 9 | 60 | 81 | 4f | dc | 22 | 2a | 90 | 88 | 46 | ee | b8 | 14 | de | 5e | 0b | db |
| | a | e0 | 32 | 3a | 0a | 49 | 06 | 24 | 5c | c2 | d3 | ac | 62 | 91 | 95 | e4 | 79 |
| | b | e7 | c8 | 37 | 6d | 8d | d5 | 4e | a9 | 6c | 56 | f4 | ea | 65 | 7a | ae | 08 |
| | c | ba | 78 | 25 | 2e | 1c | a6 | b4 | c6 | e8 | dd | 74 | 1f | 4b | bd | 8b | 8a |
| | d | 70 | 3e | b5 | 66 | 48 | 03 | f6 | 0e | 61 | 35 | 57 | b9 | 86 | c1 | 1d | 9e |
| | e | e1 | f8 | 98 | 11 | 69 | d9 | 8e | 94 | 9b | 1e | 87 | e9 | ce | 55 | 28 | df |
| | f | 8c | a1 | 89 | 0d | bf | e6 | 42 | 68 | 41 | 99 | 2d | 0f | b0 | 54 | bb | 16 |

Figure 2.1: Lookup table used during the SubBytes transformation

### 2.2.2 Modes of Operation

A block cipher in its basic form is not very useful in the real world since it will only work securely for a single block of input. Several methods for using block ciphers have been designed to describe how to repeatedly apply a block ciphers single-block operation to an input of much greater length with good levels of security and/or adds other desirable features.

The National Institute of Standards and Technology proposed five different modes of operation to cover the vast majority of needs for block ciphers [23]. The five modes of operation are: Electronic Code Book(ECB), Cipher Block Chain(CBC), Cipher Feedback(CFB), Output Feedback(OFB) and Counter(CTR). Some of the modes require defining an Initialization Vector (IV). The IV is used as the starting internal value of the block cipher. The modes of operation relevant to this project are ECB and CBC.

### 2.2.3 ECB

Electronic Codebook (ECB) is the simplest of the standard modes of operation. In ECB mode, each input block is encrypted with the same key. This

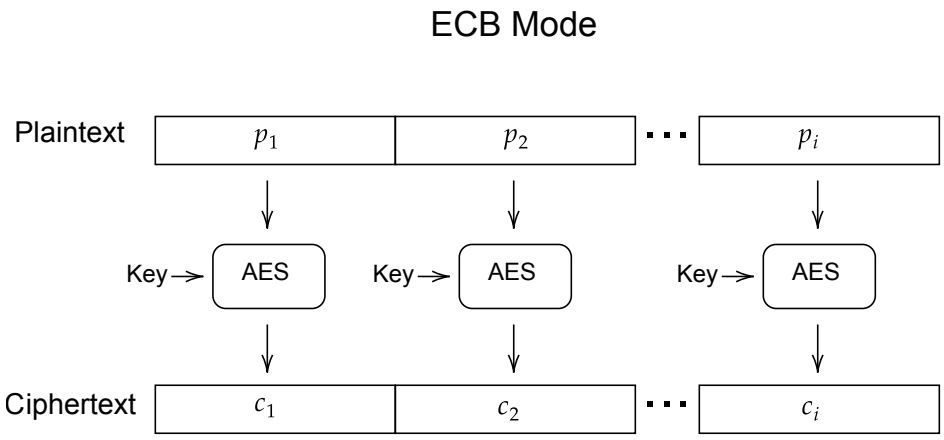


Figure 2.2: Encrypting with AES in ECB Mode

is only secure if it is used to encrypt a single block before changing the key. Because of this, it should never be used as the primary encryption method in a system. The main reason why ECB is not secure is that any two identical blocks will have the same output. This type of encryption is weak to frequency attacks and will reveal many encrypted words if given enough data. As a result of the independent encryptions of each block, an error during encryption will only corrupt the current block, and the previous and later blocks will still be intact.

In Song et als. Scheme IV [41], AES in ECB mode is used, but not as the primary encryption method. It is still secure to use in this context because another pseudorandom component is added to the output of the encryption before it is uploaded to the server. This will be explained in more detail in Chapter 3.

### 2.2.4 CBC

In Cipher Block Chaining (CBC) mode, the input from the current block is XORed with the plaintext of the next block before encrypting. Before encrypting the first block, the input is XORed with the Initialization Vector (IV), which should be generated randomly for each encryption. Using the previous encrypted block during encryption ensures that two identical blocks

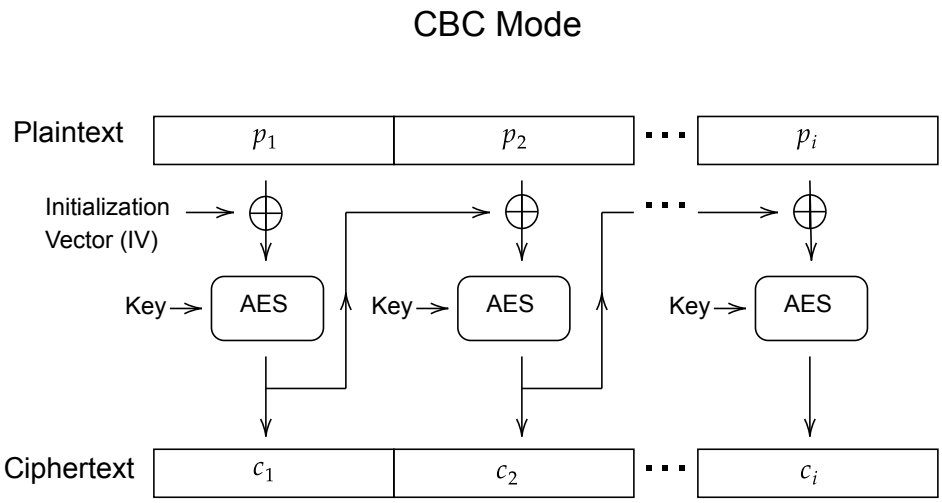


Figure 2.3: Encrypting with AES in CBC Mode

will look completely unrelated after encryption. This block chaining effect makes CBC resistant to statistical attacks like frequency analysis.

### 2.2.5 Padding

As block ciphers only accept input of a fixed size, a padding scheme is often used to ensure each block is the correct size. If padding was applied during encrypting, it must be removed after decryption. Because we need to know what should be removed, adding the length of the applied padding as part of the actual padding is common.

PKCS 5- and PKCS 7 Padding are predefined padding schemes and are used to pad byte arrays to a set size. Both schemes work the same but for different array lengths. They work by first finding the number of bytes that needs to be padded to give the array the correct size. The value of each added byte is the number of bytes added. If the current block needs to add 3 bytes to reach the desired size, the three bytes added will all have the value "03" [27, Ch.6.3].

## 2.3 Stream Cipher

As mentioned in the previous section, block ciphers process input in blocks at a time. Stream ciphers process much smaller units at a time, usually 1 bit or 1 byte(8 bits). A stream cipher does not do computations on the input(cleartext) as a block cipher does. Instead, they generate a pseudo-random bitstream called a keystream and combine it with the input. The combination is done with the XOR operation. If the XOR operator is applied twice with the same value, the output will be the same as the original input. This makes decryption very simple; Use the same key as during encryption, generate the same keystream, and combine it with the ciphertext to get the original cleartext.

### 2.3.1 Trivium

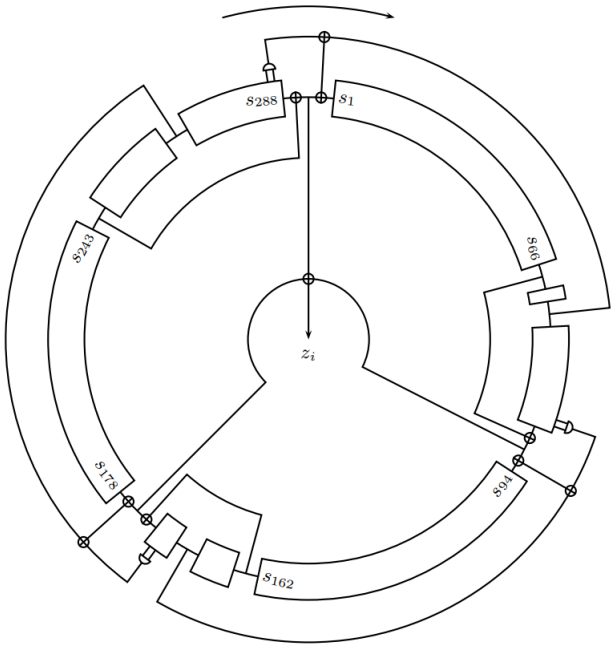


Figure 2.4: Visual representation of Trivium

Trivium is a fast and lightweight stream cipher that uses an 80-bit key and an 80-bit Initialization Vector (IV) to securely generate up to $2^{64}$ bits of pseudorandom output [16]. Trivium was designed as a challenge to how much a stream cipher could be simplified without sacrificing security, speed, and flexibility. Trivium still needs more testing before it is ready to be used in a real-world scenario where actual critical information is being handled but is very well suited for a project of this scale.

Trivium consists of 288 internal registers($s_1$,...,$s_{288}$), each containing one bit. Trivium also keeps track of three intermediate variables $t_1$, $t_2$ and $t_3$. These are mainly used to calculate the output bit $z_i$ in the main phase. Before any output is produced, an initialization phase is performed. In the initialization phase, the key and IV are loaded into the internal registers, and the rest of the registers are filled with predetermined bits 2.5.

$$
\begin{aligned}
(s_1, s_2, \ldots, s_{93}) &\leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0) \\
(s_{94}, s_{95}, \ldots, s_{177}) &\leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0) \\
(s_{178}, s_{279}, \ldots, s_{288}) &\leftarrow (0, \ldots, 0, 1, 1, 1)
\end{aligned}
$$

Figure 2.5: Initializing internal registers

In the last part of the initialization phase, the internal state is rotated 4 full cycles. One cycle consists of 288 rounds. After the initialization phase, the variable $z_i$ is calculated each round and is the output bit for that round. During the initialization phase, $z_i$ will not be outputted. $N$ rounds of Trivium are calculated by the figure below 2.6, and as visualized in Figure 2.4. As mentioned, 4 cycles ($4 \times 288$) are performed without outputting the variable $z_i$.

During the main phase of Trivium, one bit ($z_i$) is produced per round as shown in Figure 2.6.

**for** $i = 1$ to $N$ **do**
$t_1 \leftarrow s_{66} + s_{93}$
$t_2 \leftarrow s_{162} + s_{177}$
$t_3 \leftarrow s_{243} + s_{288}$
$z_i \leftarrow t_1 + t_2 + t_3$
$t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
$t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
$t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$
$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
$(s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
**end for**

Figure 2.6: $N$ rounds of Trivium

## 2.4 Hash functions

A hash function is a one-way function where the primary purpose is to map a variable-size input to a fixed size output without being able to reverse the process. Hash functions are used in several fields of computer science. They are, for example, the main component in the data structure hashtable and an essential tool in computer security. Hash functions do not require any additional key other than the input to be hashed. A cryptographically strong hash function should be easy to compute but hard to invert [35]. The primary and essential requirements for a hash function $H$ with input $x$:

1. $H$ can be applied to any argument of any size. $H$ applied to more than one argument is equivalent to using $H$ on the concatenation of the arguments.
2. $H$ always produces a fixed size output.
3. Given $H$ and $x$, it is easy to compute $H(x)$.
4. Given $H$ and $H(x)$, it is computationally infeasible to determine $x$.
5. Given $H$ and $x$, it is computationally infeasible to find an $x' \neq x$ such that $H(x) = H(x')$.

### 2.4.1 SHA-512

The Secure Hash Algorithms (SHA) is a set of cryptographic hash functions published by the Nasional Institute of Standards and Technology (NIST). The algorithms are divided into SHA-0, SHA-1, SHA-2, and SHA-3. SHA-512 is one of 6 similar hash functions included in SHA-2. The SHA-2 algorithms are made with the Merkle-Damgård construction. In the Merkle-Damgård construction, the input is divided into blocks and, one by one, fed into a compression algorithm. The output of each compression is combined with the next block and compressed again.

A message-digest with SHA512 is computed in 3 steps [42]:

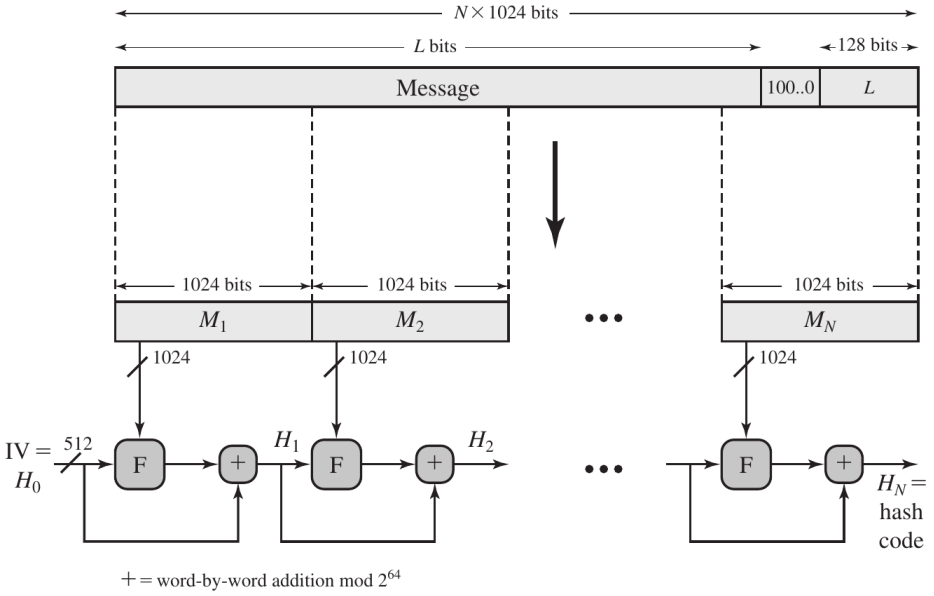


Figure 2.7: SHA512

**Step 1: Padding.** The compression algorithm $F$ of SHA512 takes an input of size 1024 bits and turns it into an output of size 512 bits. The message, therefore, needs to have a length that is a multiple of 1024. Like in a block cipher this is achieved by padding the message. A

critical component of the padding scheme used in SHA512 is adding the length of the original message (before any padding is applied) as the last part of the padding. The length $L$ is always formatted as an unsigned integer in binary with 128 bits. If the length $L$ of the original message happens to be longer than 896 mod 1024 bits[$L \equiv 896 \pmod{1024}$](the padding does not fit in the last block), a new block is added to make room for the padding. Any space in between the last bit of the message and the 128-bit message length is filled with one '1' bit and the rest '0' bits(from left to right).

**Step 2: Setup.** In this step, eight internal variables are defined and initalized. The eight internal variables $a$, $b$, $c$, $d$, $e$, $f$, $g$ and $h$ are 64-bit registers. The eight registers are initialyzed with the values (in hexadesimal format):

| | |
|---|---|
| $a$ = 6A09E667F3BCC908 | $e$ = 510E527FADE682D1 |
| $b$ = BB67AE8584CAA73B | $f$ = 9B05688C2B3E6C1F |
| $c$ = 3C6EF372FE94F82B | $g$ = 1F83D9ABFB41BD6B |
| $d$ = A54FF53A5F1D36F1 | $h$ = 5BE0CD19137E2179 |

**Step 3: Process Message.** In this step, we apply the compression function $F$ to each 1024-bit block of the message with padding. Between iterations, we keep track of the internal variables $H_i$ defined in the previous step. To produce $H_{i+1}$ we apply the compression function to the 1024bit message block $M_{i+1}$ and $H_i$. The 512-bit output is then XORed with $H_i$. The 512-bit value $H_N$ produced from message block $M_N$ and internal variable $H_{N-1}$ where $N$ is the number of blocks in the padded message, is the hash of the message.

The compression function $F$: The compression function used in SHA512 has 80 internal rounds of computations. Each round takes in a $W_i$ and a $K_i$, where i is the current round and the internal variables $a$, $b$, $c$, $d$, $e$, $f$, $g$ and $h$. For round 0, the internal variables are derived from splitting the previous internal variables $H_{i-1}$, or for rounds 1-79, the internal variables are the output of the previous round. Variable $W_i$ is calculated with the formula [21]:

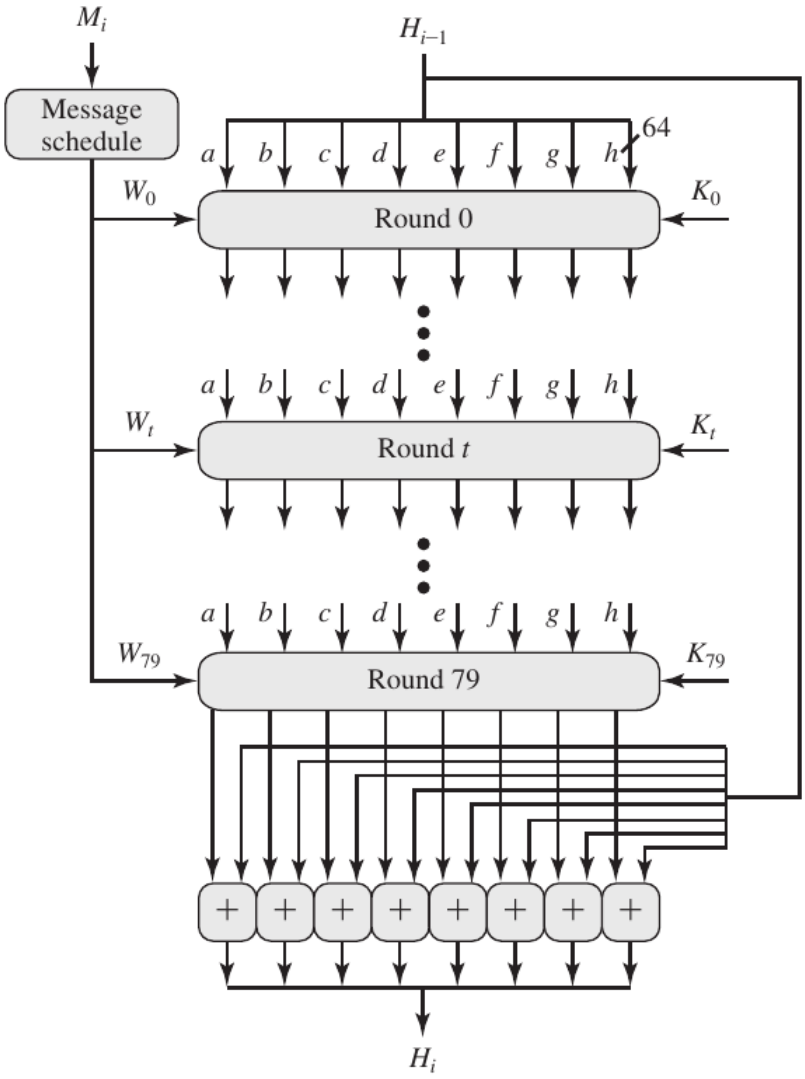


Figure 2.8: The compression function

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq 79 \end{cases} \tag{2.1}$$

Functions used in the compression function:

$$\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{2.2a}$$

$$\mathrm{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{2.2b}$$

$$\sum_{0}^{(512)}(x) = \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x) \tag{2.2c}$$

$$\sum_{1}^{(512)}(x) = \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x) \tag{2.2d}$$

$$\sigma_1^{(512)}(x) = \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^{6}(x) \tag{2.2e}$$

$$\sigma_0^{(512)}(x) = \mathrm{ROTR}^{1}(x) \oplus \mathrm{ROTR}^{8}(x) \oplus \mathrm{SHR}^{7}(x) \tag{2.2f}$$

where

$ROTR^n$(x) is a circular right shift operation where the bits in x are shifted n spaces to the right. Bits that overflow on the right side loop around to the left side.

and

$SHR^n$(x) is a noncircular right shift operation where the bits in x are shifted n spaces to the right. The n bits that overflow on the right side are lost, and n '0' bits are padded on the left side.

The $K_i$ values are predefined constants. These are calculated by taking the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. The constant $K_i$ is calculated from the i'th prime number. All $K$ values can be found in the Secure Hash Standard documentation [21].

In each round, the internal values are changed with the same formula. Figure 2.9 shows the computations of all 80 rounds of the compression function. One iteration corresponds to one round.

As shown in Figure 2.8, the output of the last round of computation(round 80) combined with the output of the previous compression block($H_{i-1}$) to create the output of this compression block($H_i$). The output is always size 512bits. The output of the last compression block is the output of the hash function.

For $t$=0 to 79:

{

$$T_1 = h + \sum_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t$$

$$T_2 = \sum_0^{\{512\}}(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

}

Figure 2.9: All 80 rounds of the compression function $F$, functions are explained in 2.2

### 2.4.2 HMAC

Hash-based Message Authentication Code (HMAC) is a method of securely adding a key to a standard hashing algorithm. The method requires the generation of two subkeys: $innerKey$ and $outerKey$. Both subkeys are derived from the original key. Generating the two subkeys is done by XORing the original key with two strings, *ipad*(inner pad) and *opad*(outer pad). The string *ipad* consists of the byte '0x36' repeated $B$ times where $B$ is the length of the original key. String *opad* is generated in the same way but with byte '0x5C' instead of '0x36'. The hash is then computed like this:

$$F(outerKey + F(innterKey + text))$$

where $F$ is the hash function and *text* is the input to be hashed.

### 2.4.3 Hash collisions

As one would expect from a function where the number of possible inputs is greater than the number of possible outputs, sometimes two different inputs result in the same output. With hash functions, this is called a hash collision. When mapping a variable-size input to a fixed-size output, this is an unavoidable problem. As hash functions are often used to generate a fingerprint/identification/checksum of an element, the possibility of a hash collision adds a small probability of error. A feature of a good hashing algorithm is that there should not be any way to predict or manufacture the output of the function. For a hashing algorithm that produces a hash of size 512 bits, producing a specific output has a probability of $1/2^{512}$. The average number of hashes before a collision is likely to occur is $2^{256}$.

### 2.4.4 Bloom filter

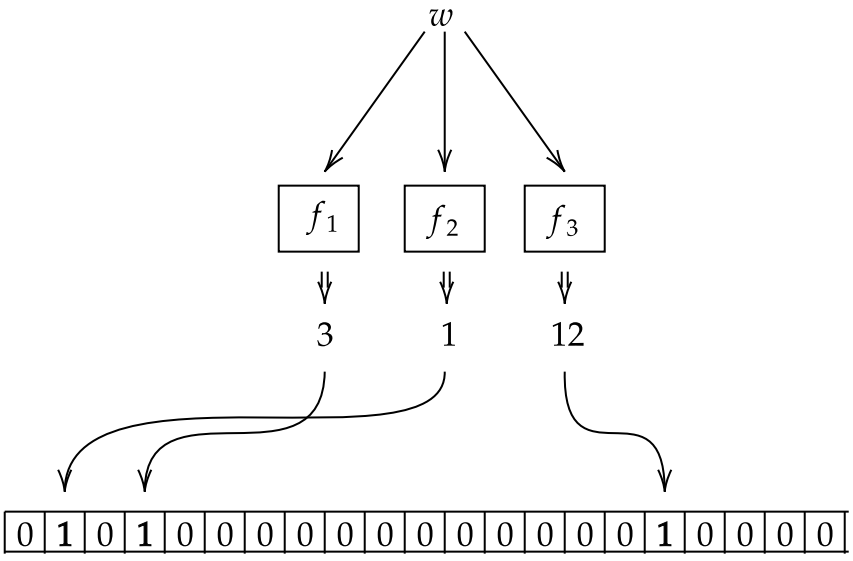


Figure 2.10: Bloom filter

A Bloom filter is a data structure used to check if an element is part of a set or not. A Bloom filter is always initialized without any elements. An empty Bloom filter consists of $m$ bits, all set to 0, where m is the number

of possible outputs of the pseudorandom function in use. When adding an element to the Bloom filter, the element is sent through $r$ independent pseudorandom functions. The number $r$ decides the number of functions in the system and is called the security parameter. The bits positioned at the index of the result of the functions are set to 1. To check if an element is in the Bloom filter, the element is sent through the same $r$ pseudorandom functions and checked if all the answers are in the Bloom filter. If all the results are in the Bloom filter, the element is present with a very high probability, if any of the results are not in the Bloom filter, the element is definitely not in the set. There is a very low probability of a false positive if all the pseudorandom functions have a collision at the same time. Suppose the probability that a pseudorandom function outputs the same value for two different elements is $p$. Then the probability of a false positive is $p^r$. The error probability drastically decreases when increasing $r$.

### 2.4.5 Password hashing

A requirement in a general-purpose hash function is speed. One should be able to perform message authentication or calculate checksums as efficiently as possible. When hashing passwords, however, the opposite is true. Two common ways for an attacker to obtain another user's password are with a brute force attack or a dictionary attack. In a brute force attack, the attacker tries all possible combinations to eventually guess the correct password. In a dictionary attack, the attacker uses a previously generated dictionary with (hash, password) pairs to be able to quickly find the password corresponding to the stored hash. This is made significantly harder with the addition of salt when hashing the password, but this is not good enough with computers getting faster.

A Password-Based Key Derivation Function (PBKDF2) is a function for deriving a random-like key based on a given password, which is provided by the user and is usually not strong and long enough. [36]. This can be used to make a function for hashing passwords that are computationally slow. The algorithm's speed depends on the number of iterations provided during initialization. OWASP recommends 120000 iterations when using PBKDF2 with SHA512 [7]. It works by applying a keyed hashing function

to a password and a salt. This process is repeated many times with the output of the last iteration as the new input.

# Chapter 3

# Practical Techniques for Searches on Encrypted Data

Song et al. [41] proposed a scheme to solve the issue of having to trust storage servers fully. In the paper, Song et al. use mail servers as the primary example but state that the uses are not limited to only text. The scheme works for any set of tokens called a document, where a user can search for the token. In the case of an email, the set of tokens would typically be each section of text separated by a space. Throughout the thesis, a token will often be referred to as a word.

## 3.1 Scheme Overview

In the paper, Song et al. describe four schemes, where each scheme adds better security or functionality to the previous scheme. The final scheme, scheme four, is provably secure and ensures that the untrusted server cannot learn anything about the plaintext given only the ciphertext. The final scheme features controlled searching, meaning that only the data owner or anyone the data owner has chosen to share their secret key with can perform searches on the encrypted data. Anyone that has rightfully been given access to the key (data owner or other) will throughout the thesis be referred to

as a user. The scheme features hidden queries, which means that the search word is encrypted and will not be revealed in the clear to the server. The last main feature is query isolation, meaning that the server learns nothing more than the search result when performing queries. The schemes require a stream cipher $G$, a pseudorandom function $F$, for example, a hash function, and a block cipher $E$. The block cipher is essential to be deterministic and does not use any randomness. The encrypted block should not rely on anything other than the key and the current block. In other words, it should be a block cipher in ECB mode.

### 3.1.1 Scheme I - The Basic Scheme

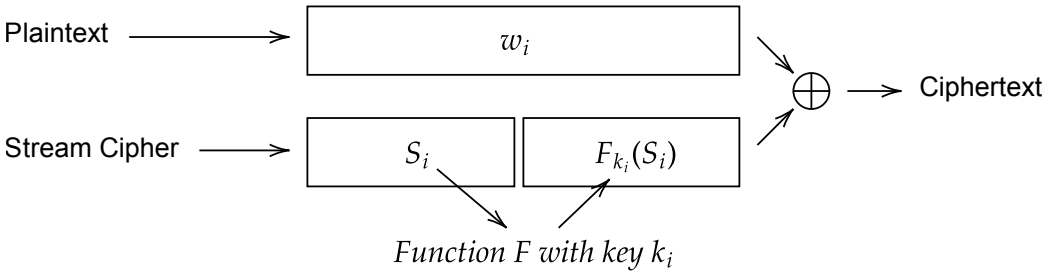


Figure 3.1: Scheme I

The first scheme introduces the basic concept of the encryption method. During initialization, the block size $n$ is chosen. Each word $w$ in document $D$ is padded to be length $n$, or if $w$ is longer than the block size, it is split into several blocks. For each $w$, during encryption, a sequence of bits $S$ of length $n - m$ is generated with the pseudorandom generator $G$, where $m$ is the length of the output of the pseudorandom function. When encrypting word $w_i$ in position $i$, $S_i$ is generated, and $T_i =< S_i, F_{k_i}(S_i) >$ with secret key $k_i$ is created. The ciphertext $C_i$ is computed by applying bitwise exclusive or (XOR) on $T_i$ and $w_i$. For this scheme version, the user can choose whether to use the same key $k$ for all words in the document or use a different one for each word.

When a user wants to search for a word in the documents, the user sends the search word $w$ and the key $k$ to the server. Then the server checks each

word in each document whether the current ciphertext word XORed with the search word is on the form $< S_i, F_k(S_i) >$ (this is the $T_i$ from the encryption). The encrypted word matches the search word if a valid $T$ is found. The server can then return the document to the user.

When the user wants to decrypt the returned document, they apply the encryption in reverse. Since the XOR operator returns the original input if the operation is used twice, the user can generate the keystream $S$ again(since they know the seed), calculate $T$, and XOR it with the ciphertext $C$ to get the original cleartext document.

There are two obvious problems with this version of the scheme. The first is that the search word is shown to the server in the clear. The second is that the user must provide the key $k$ to the server when searching. The server decrypts the whole document if the same $k$ has been used for all words. However, if the user has chosen a different $k$ for each word, then they would have to know the exact position the word might appear in and the $k$ for that exact word. This defeats the purpose of searching, and a fix to this will be presented in the next version.

### 3.1.2 Scheme II - Controlled Searching

As the name suggests, this scheme extension provides a way to choose $k_i$ to achieve controlled searching. This requires an additional pseudorandom function $f$, and a secret key $k'$. $k'$ should be selected randomly and be kept secret by the user. This scheme suggests using $f$ with $k'$ on each word and the output as the key $k_i$ during encryption. This would guarantee that the server could not learn anything about the other words when performing a query.

When the user wants to search for a word $w$ in their documents stored on the server, the user will compute $k = f_{k'}(w)$. The user would then send the server $< w, k >$. Then, like in the basic scheme, the server would XOR each ciphertext word $C_i$ with the provided search word $w$ and check if the output is in the form $< S_i, F_k(S_i) >$.

It is possible to use different $k'$ keys for some documents to limit which documents can be searched for in a single query. This can be done either for the convenience of being able to categorize documents or to add an extra layer of security. This feature will be explained in detail in the Scheme features section.

This scheme is much more secure than the basic scheme, but the user still has to provide the search word in plaintext to the server. Scheme III aims to solve this problem.

### 3.1.3 Scheme III - Support for Hidden Searches

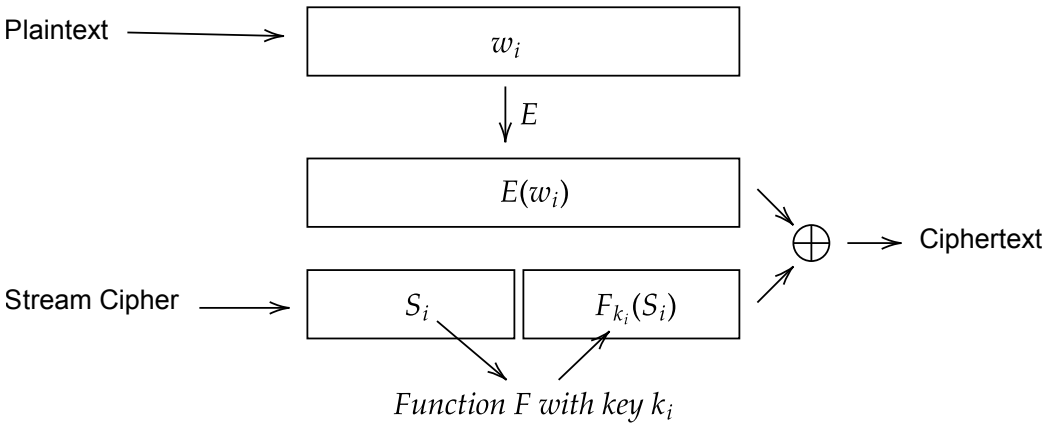


Figure 3.2: Scheme III

Scheme III provides a simple extension to the previous scheme to make the searches hidden from the server. To achieve this, pre-encrypt the word $w$ with a block cipher in ECB mode before applying the stream cipher. The key $k''$ used for the block cipher should also be kept secret by the user. $k$ is now computed from $f_{k'}(X)$ where $X = E_{k''}(w)$.

Careful readers may have noticed that when pre-encrypting each word with ECB before the main encryption, the user can no longer decrypt the documents. This is because if the user generates keys $k_i = f_{k'}(E_{k''}(w_i))$, they would need to know $E_{k''}(w_i) = X_i$. This does not make sense because

there would be no point in generating $T_i =< S_i, F_{k'}(S_i) >$ to find $X_i$ if the user already knew $X_i$. In the last scheme, Song et al. fix this issue without compromising the algorithm's security.

### 3.1.4 Scheme IV - The Final Scheme

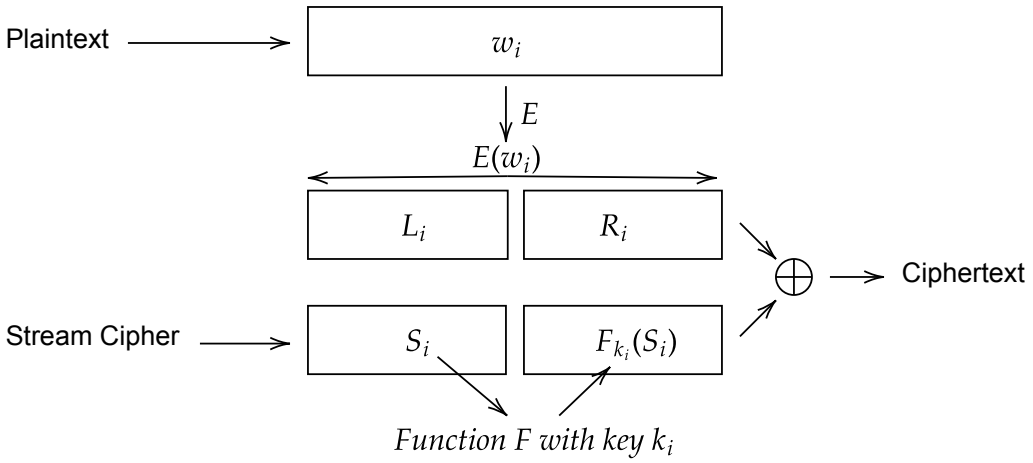


Figure 3.3: Scheme IV

The final scheme provides a simple fix to the problem described at the end of Scheme III. In the final scheme, the user should split the pre-encrypted word $X$ into two parts, $L$ and $R$. $L$ should be of length $n - m$, the same length as $S$, and $R$ should be length $m$, the same as $F_k(S)$. Now, instead of calculating $k = f_{k'}(X)$, the user should calculate $k = f_{k'}(L)$.

The full **encryption** now looks like this: the user first divides document $D$ into a set of $n$ size words $w$ where $n$ is the block size. In an English text, this would usually mean splitting the text on each space character and padding the words to length $n$. Then the user loops through the set of words, and for each word $w_i$ in position $i$, pre-encrypt it: $X_i = E_{k''}(w_i)$. The pre-encrypted word $X_i$ is split into $L_i$ and $R_i$. Then the user generates $S_i$ as $n - m$ lenght output of the pseudorandom generator $G$, with a randomly chosen secret seed, which should be stored locally for use during decryption. The user computes $k_i = f_{k'}(L_i)$ and sets $T_i =< S_i, F_{k_i}(S_i) >$. The current ciphertext block $C_i$ is computed by XORing $X_i$ and $T_i$.

When users want to **search** for a word, they pre-encrypt the search word $w$: $X = E_{k''}(w)$, then split it into a left and right part, $L$ and $R$ respectively, like during encryption. The user then computes $k = f_{k'}(L)$ and sends $< X, k >$ to the server. The server can now loop through each block in each document and partially decrypt it to check if it has a valid $T$. For ciphertext block $C_i$ in position $i$, the server splits the block into a left, and a right part $C_{i1}$ and $C_{i2}$. The same is done for the pre-encrypted search word $X$ -> $[L, R]$. Then the server computes $S_i = C_{i1}$ XOR $L$. Then the server computes $F_k(S_i)$ and checks if $F_k(S_i)$ XOR $C_{i2}$ equals $R$(this makes it a valid $T$). If so, a match has been found, and the document can be returned to the user.

When **decrypting**, the user generates $S_i$ with the psuedorandom generator $G$ and their secret seed, computes $L_i = S_i$ XOR $C_{i1}$, calculates $k_i = f_{k'}(L_i)$, computes $F_{k_i}(S_i)$ and then computes $R_i = C_{i2}$ XOR $F_{k_i}(S_i)$. The user can then combine $L_i$ and $R_i$ to get $X_i$. Then use the block cipher to decrypt $X_i$ into $w_i$.

## 3.2 Scheme Features

When talking about features, I am exclusively referring to features of the final scheme: Scheme IV. In the introduction of this chapter, the main features of controlled searching, hidden queries, and query isolation were mentioned briefly. The main features and some extra, nice-to-have/alternative features will be explained in the two subsections below.

### 3.2.1 Main Features

The main features: **controlled searching**, **hidden queries**, and **query isolation** are vital to the scheme's security. Controlled searching addressed in Scheme II ensures that the secret key generated by the user is needed to make search queries on that user's files stored on the server. The user can choose to share their secret key with others, giving them administrative rights to their account, but the key should never be shared with the server. Query isolation ensures that the server cannot learn anything about the

words that do not produce a match when performing a search query. Query isolation paired with Hidden queries, which ensures that the server cannot learn anything about the word being queried, makes the system only leak an acceptable amount of information.

### 3.2.2 Extra Features

Although there is no direct way of **querying with wildcards** (unknown character(s)), a user could simulate the feature by creating queries with all the different combinations of characters for the wildcard(s). If a user wanted to search for "Ca?" where "?" is an unknown character, the user could create 26 different queries (or more if special characters and/or numbers are included), one for each character in the alphabet instead of the wildcard. The number of queries and, therefore, also search time increases exponentially for each extra wildcard, so this feature should probably only be used in a particular type of system where this is used sparingly. When using this feature, it is also essential to remember that each query sent to the server leaks a small amount of information about the search. Making thousands of queries for a single search might not be a great idea.

An alternative to using a fixed block size is to use a feature with **variable-length words**. Using variable-length words removes the space inefficiency caused by padding in the regular scheme but adds some complexity and a lot of extra computation to the server. When encrypting with variable-length words, the user only generates enough random bits of the bitstream to perform the XOR. The user would have to store the length of each word to be able to separate them during decryption. The length of each word has to be shared with the server for it to know where to search in the document. Sharing the word lengths with the server is generally a bad idea but does remove the considerable space inefficiency of the main scheme. The alternative to sharing the word lengths is to make the server search all possible $r$ length sequences of the document where $r$ is the length of the search word. The complexity increase of this approach is very high, as the server needs to perform $r$ times more searches than the fixed size approach.

The last extra feature is an extension of the controlled search and provides functionality to **categorize documents**. This is mentioned briefly in the

"Scheme II - Controlled Searching" subsection. To explain this feature, I will use an example: Assume a user; Alice wants to store their emails on a server that uses this scheme. Alice wants to categorize her emails as "important" or "spam". When encrypting the emails, she can choose to generate $k_i = f_{k'}(L_i)$ with two different $k'$ keys, one for the "important" emails and one for "spam". Then later, when Alice wants to search for a word in only the "important" emails, she generates $k$ with the same $k'$ as when encrypting those emails. When performing the query, the server will look through all the files like normal, but only the "important" emails have a chance of finding a match. From the server's point of view, there is no way to tell that the emails are encrypted with a different key. If Alice wants to search through all her emails, she performs separate search queries with every $k'$ she has used to categorize emails.

To add to the extension of controlled searching explained above, using different $k'$ keys in different parts of a single document is possible. To continue the example from above, if Alice wants to be able to search for only who sent an email, she can use a different $k'$ for the part of the document where the information about the sender is stored. Then if she wants to search for only the sender, she creates the search token with the same $k'$. This feature can be extended as far as the user wants in both directions. Although an extensive amount of keys can be more challenging to manage, and if the user often wants to search through all documents, the number of queries will increase.

## 3.3 Implementation Specifications

In my implementation of the algorithm, I use a client-server solution. One server can have many clients and each client connects to a single server. The system is hosted locally and is not meant to be used outside of a controlled environment for demonstration purposes only. This implementation does not

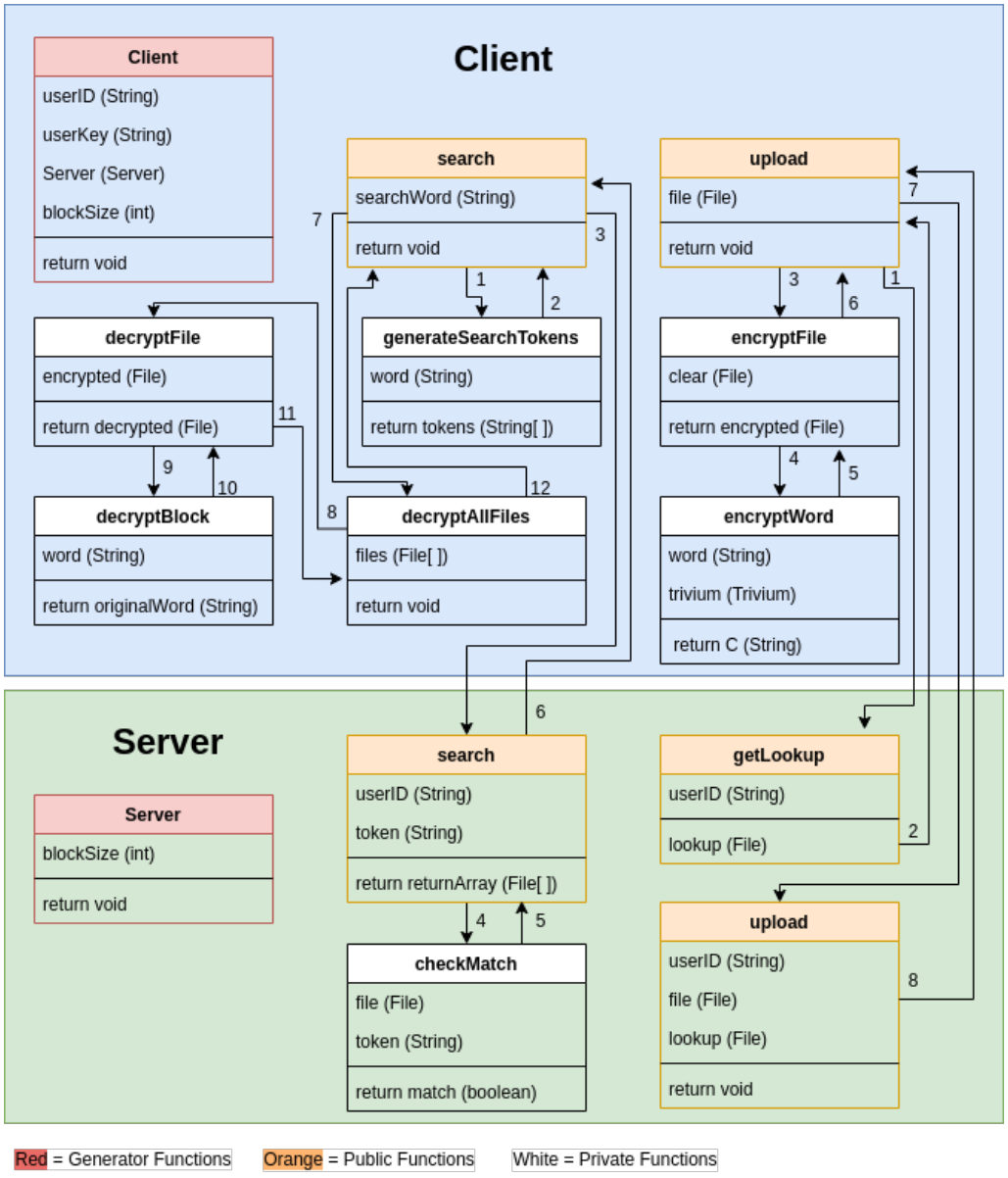


Figure 3.4: Components of my implementation and their relation

include any extra security features like encrypted file names, user authentication, or integrity checks. I have chosen not to focus on the mentioned extra

security features because they are not part of the proposed scheme.

I implemented the scheme in Java, specifically Java 18, and some of the design decisions are based on the standards and common practices of the language. For example, a common way to work with bits in Java is in the datatype byte. A byte is represented by a number in the range $[-127, 128]$ and effectively holds 8 bits($2^8 = 256$). For example, a 128-bit key will be represented as a list of 16 bytes. Text, on the other hand, can be a bit more tricky. The standard text charset is UTF-8. UTF-8 is a variable width character encoding, meaning that a single character can be represented by 1 - 4 bytes. The more common symbols are represented by only 1 byte and uncommon characters use more bytes. As mentioned in section 2, block ciphers convert a fixed-size input to a fixed-size output. Using UTF-8 causes the number of characters in each encryption block to vary depending on which characters are included in the current block.

As the user needs to generate the same bitstream when decrypting as when encrypting, there needs to be a way for the user to get the correct seed that corresponds to the current document. A very simple way to achieve this is that the user stores a dictionary of document identifiers and the corresponding seed locally and gets the seed whenever they download something from the server. If the user were to lose the dictionary or log on from a different computer, there would be no way to decrypt files even though the user knows the password for their account. I decided to store the dictionary encrypted on the server to solve this problem. The dictionary should not be encrypted with the SSE scheme like the rest of the documents because it should always be included whenever the user downloads other documents. From looking at the image of the components in Figure 3.4, if the server finds any matches when searching, the dictionary(called lookup in the code) is returned with the encrypted documents. The downside to storing the dictionary on the server is that the user would have to download the dictionary and update it before encrypting and uploading the actual document, this can be seen as the setLookup function in the image of the components 3.4. This adds an extra round-trip to the communication protocol between client and server. This dictionary is referred to as the lookup table in the code and later in the thesis.

The code runs in a loop that lets the user interact with the program through the terminal. First, the user is asked to choose a working directory.

This is where the program creates client directories and server directories used to store the user's files and the server's files. The working directory is also where the program creates and updates temporary files. Before the main loop begins, the user enters a username and password, which will be used to generate the secret key. In the main loop, the user has the option to upload files, search for files log out, or quit. The commands are written in the terminal but choosing files and the working directory is done in a primitive file-selection popup.

In the upload function, before **encryption** starts, the user asks the server for its lookup file. If there is none stored on the server, the user creates a new one. Then the encryption starts by randomly generating a seed for trivium with a SecureRandom Java object. A Scanner object is used to read the entire document that is to be encrypted and iterate over each word(any group of characters separated by a space). Each word is processed by a "separateWords" function used to split the word into the correct number of 32-byte chunks(for blocksize 32) where the correct padding is added to each chunk. The padding scheme in this implementation works as follows: the last two bytes of each block are always reserved for padding. If the length of the current word is not a multiple of 30, the missing bytes are filled with "*" characters. The last two padding bytes indicate how many "*" characters were added. If the current word is longer than 30 bytes, the word is split into the correct number of chunks, and only the last chunk is padded as mentioned above, the other chunks are padded with the bytes "-1" to indicate that the chunk should be combined with the next chunk when decrypting.

After padding, each 32-byte chunk is pre-encrypted with AES in ECB mode, the AES option "No-Padding" is used as I need to be able to control the padding of each word myself. I decide to make the left and right parts of the block the same size by setting $m = blocksize/2$. Generating $k$ is done with HMAC with SHA512 on $L$ as explained earlier. Whenever I work with strings during the encryption, the charset ISO_8859_1 is used. This is because, unlike UTF-8, ISO_8859_1 always translates 1 byte into 1 character. This is generally easier to work with when performing operations directly on the words. When the encryption of the entire document is complete, a hash of the document is computed and used as a key in the lookup dictionary to store the seed for trivium. Before uploading the updated lookup and the encrypted document, the lookup is encrypted with AES in CBC mode.

When a user wants to **search**, they type the search command in the terminal and provide the keywords they want to search for. The search token is generated with the same functions as where used in the encryption. The pre-encrypted word $X$ and the key $k$ are sent to the server with the current user's $userID$.

When the server receives the search request, it finds the files owned by the current user. Each file(except the lookup) is sent through the "checkMatch" function. The ckeckMatch function performs the search as described in the introduction by checking for a valid $T$ in each block. If the server finds any matches, those files are returned along with the lookup.

**Dectryption** is performed automatically for any files returned by a search. From the lookup table accompanied by the returned files, the user finds the correct seeds for trivium to generate the correct keystream $S$. With the $S$ and the private key, the user can successfully decrypt all their files.

## 3.4 Performance Analysis

As the scheme uses sequential search and is encrypted/decrypted by scanning through the document's words like in a block cipher, it is obvious that the main factor of performance is document length and number of documents. The four main functions **encryption**, **search**, **token generation** and **decryption** mostly consists of performing a combination of the three main operations **pre-encryption** with AES in ECB mode, **generating pseudorandom bitstream** with Trivium and **hashing** with SHA512. When scanning through a document during encryption, decryption, or search, each block of input is processed in the same way throughout the entire scan. It is, therefore, logical to analyze the performance of processing a single bock of input for the three mentioned functions. Token generation does not require scanning over several words and is therefore treated as a single block of input in this analysis.

The table below counts the number of executions the major algorithms (AES ECB, Trivium, and SHA512) has to do for each block processed during

the functions: encryption, search, token generation, and decryption. Careful readers may notice that the number of SHA512 executions is doubled from what it should be. This is because the implementation uses HMAC to perform keyed hashes. HMAC as explained in the theoretical background section uses two executions of SHA512 per hash calculated. Trivium generates a single bit per execution, so the number of executions required to generate $S_i$ is always $n - m$. The default block size of my implementation is 32. The number of AES ECB encryption blocks is therefore 2 per 32-byte block of input.

| Function | AES | Trivium | SHA512 |
|---|---|---|---|
| Encryption | 2 | $n - m$ | 4 |
| Search | 0 | 0 | 2 |
| Token Generation | 2 | 0 | 2 |
| Decryption | 2 | $n - m$ | 4 |

Table 3.1: Number of operations per block

As there is no great way of analyzing the speed of algorithms without going to extreme depths regarding internal operations, I decided to perform some tests on the main algorithms of the scheme. To make test results comparable, all performance tests are run on the same computer, specifically a Thinkpad T580 with a Intel(R) Core(TM) i5-8350U CPU [2]. When testing the speed of SHA512 and AES, OpenSSL [6] was used. OpenSSL is a cryptography toolkit that among other things lets anyone test the speed of several cryptography algorithms locally. As there is no official standard for trivium, it is not supported by OpenSSL. To test the performance of my implementation of Trivium, I created a simple testing program [10].

| Function | AES | Trivium | SHA512 |
|---|---|---|---|
| Bytes | 877142.15k | 501.05k | 57772.28k |

Table 3.2: Number of bytes processed per second displayed in 1000s

To try to simulate the conditions of how the algorithms were used in the implementation, the tests were performed with as similar as possible parameters as in the scheme implementation. The AES test displays how many bytes are processed with AES in ECB mode with block size 128. The test on SHA512 uses input size of 128 bits. The test on trivium is the average

of 10 runs of the test program mentioned earlier. The result from the trivium test counts the number of bits produced in one second, this is divided by 8 to make comparison with the other algorithms easier.

The encrypted documents do not require any extra overhead other than the encrypted contents of the document. Therefore, the storage space needed is determined by the number of words in the document and the block size. My implementation supports any block size that is a multiple of 128 bits(16 bytes), but the default block size is 256 bits(32 bytes). If a word is longer than 30 bytes(the last 2 bytes are reserved for padding), the word is encrypted in two(or more) blocks. As a general rule, a word of any size(smaller than 30 bytes) will take up 32 bytes of storage space.

Padding is excellent for masking the length of words in the document to the server but adds redundancy. How much redundancy depends on the length of the original input. If all words in the original document were of size 30, very little extra redundancy would be added. The encrypted document would almost be the same size as the unencrypted document. To analyze the redundancy, one could look at the average length of words in the language used in the documents.

The above tests analyze the cryptographic algorithms in a closed environment. To provide a more practical result, I performed tests on the implementation itself. The tests analyze the major functions of the scheme, specifically:

**Upload:** Test includes encryption of 100 txt files and sending them to the server.

**Search, 0 matches:** Test includes creating a trapdoor with a bogus word, sending it to the server, and executing a search query.

**Search, 85 matches:** Test includes creating a trapdoor with the word 'to', sending it to the server, executing a search query, returning the 85 matches, and decrypting them.

| Block Size | Upload | Search, 0 matches | Search, 85 matches |
|---|---|---|---|
| 16 | 4894 | 434 | 709 |
| 32 | 6852 | 779 | 1178 |

Table 3.3: Execution time for encrypting and uploading 100 txt files, Searching with 0 matches and 85 matches over the 100 encrypted txt files with block sizes 16 and 32. Results are the average of 10 separate executions, results are displayed in milliseconds.

| Block Size | Filesize | Percent increase |
|---|---|---|
| Cleartext | 66.4 | 0% |
| 16 | 183.6 | 176% |
| 32 | 363.3 | 446% |

Table 3.4: Storage space required for storing all 100 files encrypted with different block sizes. Filesizes are displayed in kilobytes (KB).

The dataset used for testing consists of the first 100 poems in the collection 'The Sonnets' written by William Shakespeare [8] as separate txt files. The smallest of the files is 595 bytes and the biggest is 706 bytes. The combined size of all 100 files is 66.4KB.

It is clear from looking at the storage space required in Table 3.4 that using block size 32 on this dataset adds a large amount of redundancy. If this is worth it or not is unclear and needs further testing. When most of the words in the dataset are short, choosing block size 16 can save a lot of storage space. However, if the dataset consists of longer words, choosing blocksize 32 can save some storage. This is because the last two bytes in each block are reserved for padding. This means that with block size 32, there are 30 bytes reserved for the actual word, compared to 14 bytes for block size 16. A 30-byte word when encrypted with block size 32 requires 1 32-byte block whereas the same word encrypted with block size 16 would require 3 16-byte blocks($30 \text{ bytes} > 2 \times 14\text{bytes}$).

# Chapter 4

# Secure Indexes

In the Discussion section of the paper [41] Song et al. describe searching with an encrypted index as an option to the method of sequentially scanning each document. The process of searching with an encrypted index described by Song et al. is later categorized by Poh [39] as searching with an inverted index. When searching with a reversed index, the server keeps an updated map of pointers from encrypted words to the documents they appear in. Searching with a direct index table is simpler because a pre-computed lookup table can show a user with valid credentials if a word appears in a given file or not, without having to search through the file during each query. However, the downside to using an inverse index table is that it adds extra overhead, and keeping the index updated can be cumbersome.

Goh [25] proposed a scheme that uses a direct index instead of a reversed index like Song et al. described. The overhead is still there, but updating is much simpler. Goh's scheme "Secure Indexes" uses a per document index [25]. Each document has a corresponding index table to check if a provided word appears in a given document. The scheme lets a user generate a trapdoor, which is used to make queries to the server. Trapdoor generation requires the user's private key and leaks very little information about the contents of the query to the server.

The big difference in searching with an index table instead of on the main document like in the previous scheme is that the document itself does not

need to be encrypted in a special way. In the secure indexes scheme, the main document is encrypted with a standard up-to-date block cipher in the most secure manner. Therefore, security relies on creating and maintaining the index table in an intelligent way.

## 4.1 Scheme Overview

The secure indexes scheme is constructed with four main algorithms: **Key Generation**, **Trapdoor Construction**, **Index Construction**(also called BuildIndex) and **Search Index**.

### 4.1.1 Key Generation

The key generation algorithm is an algorithm to generate a secret key. The user will keep this key, never to be revealed to the server. The algorithm takes an integer $s$, which is the security parameter of the scheme. The security parameters decide the key size of the system. The private key $Kpriv$ takes the form of a set of $r$ subkeys, where each subkey consists of $s$ bits. The variable $r$ should be chosen when initializing the system. The number of subkeys in the key has to be the same for all users of the system, but the user can choose the size of each subkey $s$. In a secure system, there should be a minimum requirement for the length of $s$, for example, 128 or 256. The bits in each subkey in $Kpriv$ should be chosen at random.

### 4.1.2 Trapdoor Construction

The trapdoor algorithm is used to generate a trapdoor (which serves a similar purpose as the search token from the previous scheme). The trapdoor algorithm takes the private key $Kpriv$ and the search-word $w$ in plaintext as input. The algorithm uses a keyed pseudorandom function $f$, which takes $Kpriv$ and $w$ as inputs and produces a trapdoor for $w$ in the same format

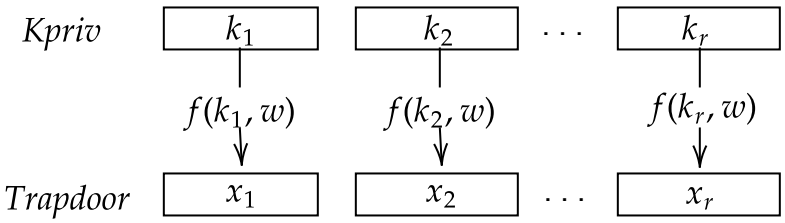


Figure 4.1: Generating a trapdoor

as the private key $Kpriv$(set of bit arrays). As the construction of the trapdoor uses the private key, the trapdoor for two identical words generated by two different users will be completely different. One could not look at the trapdoor for any given word and determine the original word or the private key used to generate it.

### 4.1.3 Index Construction

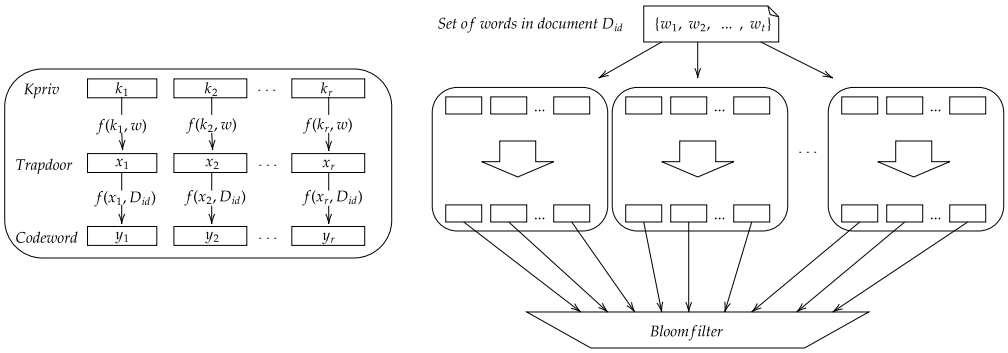


Figure 4.2: Build Index

The index table used in this scheme is a Bloom filter with hash functions as the pseudorandom functions. When generating the index, the private key $Kpriv$ and document $D$ are needed. The index generation algorithm loops through $D$ and for each word generates its trapdoor. For each trapdoor, all bitsets are hashed with the identifier ID($D_{id}$) of $D$ to create the current trapdoors codeword. The last step is to make sure two identical words in different documents do not produce the same codeword. This is to avoid the possibility of statistical attacks.

### 4.1.4 Search

The main feature of this scheme is being able to search for keywords on an encrypted document. Searching is done by the user generating a trapdoor for a word and sending it to the server. Generating valid trapdoors requires the user's private key and can therefore only be done by the account's rightful owner. The server generates a codeword with the trapdoor and each document identifier and checks if the current index contains the codeword or not.

## 4.2 Features of Secure Index

### 4.2.1 Efficient Update

Updating a document in the secure index scheme does not require the user to rebuild the index. Instead, the user computes the codewords that are removed from a document when changing the contents of a document and remove those codewords from the documents index. Similarly, when adding words to a document with an already existing index, the user computes the codewords of the newly added words and adds them to the index. When updating, the user needs to decrypt the actual document and encrypt it again after updating it.

If the number of words updated is more than half of the original document, rebuilding the entire index from scratch is more efficient. This is because the user would have to compute the codewords of the removed words and the updated/added words. In a smaller application, where documents are not frequently changed, I would argue that rebuilding the index during updating is fine. However, this feature will save a lot of computation and time when handling larger documents, where smaller updates are regular.

## 4.3 Implementation Specifications

The code structure of the implementation of this scheme is relatively similar to the code structure of the previous scheme. Like in the implementation of the previous scheme, I use a client-server structure where one client can connect to a single server, but one server can have many clients. When the server is initialized, security parameters $s$ and $r$ are chosen, these decide the size of each subkey and the number of subkeys respectively. A user object is created with a username and password to initialize a user. As there is no authentication service other than each user having their own private key to encrypt with, the user does not need to connect/login to the server. The only form of authentication is that the user attaches their user identification ($userID$) to the messages they send to the server. Unfortunately, the implementation cannot be run in this state as the implementation is used as the base to create the application described in Section 5.

Whenever a user wants to upload a document, the user first **builds an index**(Bloom filter), then encrypts the main document with AES in CBC mode. The user then uploads the encrypted document and its index to the server. Building the index starts by extracting the words from the document. As this implementation is made for encrypting text documents, this is done by separating each group of characters separated by a space. Then, for each word its corresponding trapdoor is created. To make searching easier for the user, each word is converted to lowercase before calculating the trapdoor. This avoids the hassle of remembering if a word starts with a capital letter or not when searching. Trapdoor generation in my implementation is done by hashing the word with all $r$ subkeys of the private key separately. The hashing is done with HMAC with SHA512. This creates a trapdoor for the word consisting of $r$ different hashes. To generate the codeword that is inserted into the Bloom filter, each part of the trapdoor is hashed again, with the same keyed hashing algorithm (HMAC with SHA512) but with the document identifier ($D_{id}$) as the key. As the Bloom filter stores elements as indexes of an array, the hash needs to be converted into a BigInteger. As the size of the output of SHA512 is 512 bits, the number of possible outputs is $2^{512}$. Therefore, the size of the Bloom filter array needs to be $2^{512}$. As the vast majority of bits in the Bloom filter would be 0, I decided only to store the indexes as BigIntegers instead of using the BigInteger as an index

in the boolean array. This method of storing only the index itself is more space-efficient when the document contains a small number of words. I chose to use the datatype HashSet for the Bloom filter as it performs addition and lookup operations in constant time. After every word has been turned into codewords and inserted into the Bloom filter, the Bloom filter is written to a file.

When the user wants to **search** for a document, they create a trapdoor with their private key and search word. This is done in the same way as during the construction of the index but without converting the trapdoor to a codeword. The trapdoor and the user identification ($userID$) are sent to the server as a search request. The server finds all documents belonging to the given $userID$ and generates the codeword from the provided trapdoor and each $D_{id}$. The server then performs a lookup in the current documents Bloom filter. Any document where the Bloom filter contains all subsets of the generated codeword is returned to the user.

As well as the encrypted documents and their indexes, the server stores a hashmap for each user. The hashmap maps a document to its index. The hashmap is used only for easier handling of the files and does not give the server any information it otherwise wouldn't have. This hashmap is updated whenever the user uploads or removes a document index pair. The hashmap is stored as a file with name lookup.

## 4.4 Performance Analysis

The speed of encrypting and decrypting the main documents is equal to the speed of AES in CBC mode. Testing AES in CBC mode with OpenSSL [6] like in the previous chapter, shows that it can process 249382.53k bytes per second on the same machine.

What makes the secure indexes scheme interesting is the index tied to each document and how to create it. To analyze the performance of the BuildIndex algorithm. We generate the codeword for each plaintext word in the original document to build an index. A codeword consists of the hash of each element in that words trapdoor. The trapdoor consists of $r$ hashes, where $r$ is the

number of subkeys in the private key $Kpriv$. The number of hashes that needs to be performed per word is, therefore, $2 \times r$. In my implementation, I use HMAC with SHA512 to calculate trapdoors and codewords. Like in the implementation of the previous scheme, the use of HMAC doubles the number of SHA512 operations per hash. The actual number of hashes per word is $2 \times 2 \times r = 4 \times r$. After generating the codewords, randomly generated codewords are inserted into the Bloom filter until it reaches the size of the upper bound $u$.

Generating the private key can be compared to the computation time required for generating $s \times r$ random bits. To generate the random bits, I use the SecureRandom Java class. The underlying algorithm which is used in the SecureRandom class is platform-specific. The preferred algorithm when running the code on a laptop with Manjaro (Linux) operating system is NativePRNG. NativePRNG gets random numbers based on SHA1-based PRNG from the underlying operating system [3] [5].

Searching for a word in the encrypted documents on the server requires the user to generate a trapdoor. As mentioned in the first part of the analysis, generating a trapdoor requires $2 \times r$ hashes. Then for each document, the server generates the codeword and performs a constant time lookup in the index. Generating the codeword also requires $2 \times r$ hashes.

The storage space required for the index is determined by the upper bound $u$. Choosing a suitable value for $u$ can be tricky as it limits how many words a document in the system can contain while still masking the number of keywords to the server.

When initializing the system, the three parameters $s$, $r$, and $u$ are chosen. These all play important roles in the performance of the system. When choosing the parameters, we essentially have to find a balance of security and performance. Requiring a very long private key will make the system more secure, but increases the number of computations per encryption. To quickly repeat the function of each parameter, $s$ and $r$ decide the size of each subkey and the number of subkeys in the private key $Kpriv$ respectively. Because the private key is used to create trapdoors and codewords, $s$ and $r$ also determine the size and number of elements (BigIntegers) in the index of each file. The upper bound $u$ determines the minimum number of masked

keywords(real or fake) in each index. If we set $u = 50$, the index of a txt document with 40 unique words(keywords) will contain 10 fake, randomly generated keywords. This is done to mask the number of keywords in the file.

As I think running time and storage space are two of the most interesting factors when analyzing the performance of the implementation of this kind of system, I decided to perform three separate tests with different values for $s$, $r$, and $u$. Each result for the two first tests on running time is the average of 10 separate executions with the given parameters. To make it possible to compare results with the previous scheme, the tests are performed on a Thinkpad T580 with an Intel(R) Core(TM) i5-8350U CPU [2] with the same dataset [8]. The first test measures the time it takes to encrypt, build the index and upload all the files in the dataset. The second test measures the time it takes to perform a search with no results. And the last test measures the storage space required to store all the encrypted files with their corresponding index on the server. I decided not to include the test that measures search time with a high number of matches like in the previous scheme. This is because I don't consider it to add much value over the search time without matches test as it only adds the time it takes to decrypt with AES in CBC mode.

It is clear from looking at the results of all the tests that increasing $u$ severely decreases the performance of the system. Choosing a good value for $u$ can be very tricky, as it very much depends on the use case of the system. I think the topic of choosing a suitable value for $u$, and even if it is needed is cause for an interesting discussion. To add to the discussion, I propose three solutions with various upsides and downsides to the problem. The first is to not use an upper bound $u$, this is equivalent to choosing $u = 0$(produces the exact same results as $u = 1$ with our dataset). The obvious upside of this solution is that there is no added computation and storage cost used for redundancy(fake keywords). The downside is that the number of keywords associated with each file is not hidden. This may or may not be a big security risk largely depending on the use case of the system. The second solution is to use a large enough $u$ to cover all files in the system. This can be a good solution if we know that all files have roughly the same number of keywords. The last solution is to use a 'good enough' value for $u$. This solution might be a decent tradeoff between security and performance. The

big problem with the first two solutions is that the downsides are very bad in a worst-case scenario. It is easy to make the claim that if an adversary can see the size of the encrypted file, masking the number of keywords does not add much extra security. This only applies if we know the file types of the encrypted files. An encrypted txt file with 100 keywords can typically use 500bytes of storage, as opposed to an encrypted image with 15 keywords which easily can use 10MB(10000000bytes) of storage space.

Encrypt, BuildIndex, and Upload

| $s$ | $u$ | $r = 2$ | $r = 3$ | $r = 5$ |
|---|---|---|---|---|
| 128 | 1 | 688 | 735 | 1100 |
| | 100 | 651 | 839 | 1247 |
| | 2000 | 7835 | 11798 | 19861 |
| 256 | 1 | 543 | 730 | 1137 |
| | 100 | 590 | 918 | 1291 |
| | 2000 | 8247 | 12363 | 20582 |
| 512 | 1 | 565 | 759 | 1148 |
| | 100 | 613 | 871 | 1290 |
| | 2000 | 8825 | 12995 | 22158 |

Table 4.1: Average of 10 iterations to encrypt and upload 100 txt files. Time in milliseconds.

| Search, 0 matches | | | | |
|---|---|---|---|---|
| $s$ | $u$ | $r = 2$ | $r = 3$ | $r = 5$ |
| 128 | 1 | 160 | 198 | 326 |
| | 100 | 157 | 229 | 378 |
| | 2000 | 2974 | 4485 | 7846 |
| 256 | 1 | 134 | 201 | 330 |
| | 100 | 156 | 231 | 381 |
| | 2000 | 3013 | 4519 | 7498 |
| 512 | 1 | 139 | 201 | 331 |
| | 100 | 157 | 232 | 385 |
| | 2000 | 3043 | 4546 | 7572 |

Table 4.2: Average of 10 iterations to Search with 0 matches over 100 encrypted txt files. Time in milliseconds.

| Storage space | | | | |
|---|---|---|---|---|
| $s$ | $u$ | $r = 2$ | $r = 3$ | $r = 5$ |
| 128 | 1 | 974.4 | 1419.6 | 2309.9 |
| | 100 | 1119.1 | 1636.6 | 2671.7 |
| | 2000 | 20784.2 | 31134.2 | 51834.3 |
| 256 | 1 | 1243.2 | 1822.8 | 2981.9 |
| | 100 | 1431.6 | 2105.4 | 3452.9 |
| | 2000 | 27034.2 | 40509.1 | 67459.2 |
| 512 | 1 | 1780.9 | 2629.3 | 4326.0 |
| | 100 | 2056.6 | 3042.9 | 5015.4 |
| | 2000 | 39534.2 | 59259.3 | 98709.2 |

Table 4.3: Storage space required to store 100 encrypted txt files and their index. Original plaintext size without index = 68,4. Results in kilobytes (KB).

# Chapter 5

# Cloud storage application with SSE for images and videos

During the implementation of the Secure Indexes scheme, I realized that the potential use-cases of SSE are not limited to only text documents. There still needs to exist a set of words that can be searched for, but this does not need to be the words of the original text document. In my application, I have altered the Secure Indexes scheme to work with images and videos. The set of words that can be searched for is generated from the image metadata and an object recognition API. The main scheme is surrounded by an authentication service that uses a database to store and manage users. I have created a very simple graphical user interface 5.1 to make the application more user-friendly.

Figure 5 shows an overview of the components of the application and their relation to each other. The figure is somewhat simplified to make it more manageable to understand. A good way to read the figure is by following the arrows connecting each component. The number displayed at the start of each arrow indicates in which order that component is executed. The program always starts by running the generator function of the GUI(function displayed in red). When a user is successfully logged in(mainScreen function in GUI is called), the numbers indicating the order of execution are reset to make it easier to follow. Some of the functionality left out of the figure includes reading and updating the server's internal lookup table(for

keeping track of which index corresponds to which file), the extra round trip required when logging in or registering a new user (this will be explained later, visualized in 5.3 and 5.4), generating the keywords for each file (part of the buildIndex algorithm) and buildIndex generating codewords for each trapdoor.

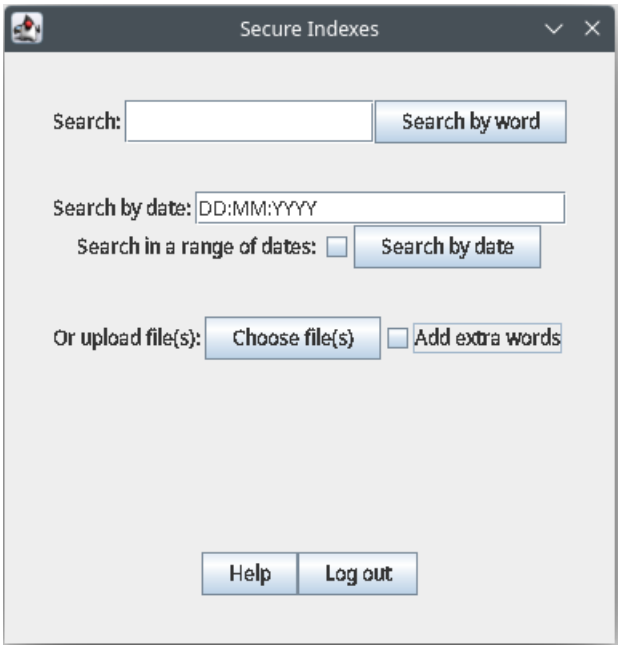


Figure 5.1: Simple Graphical User Interface

## 5.1 Motivation

My application is meant to show the various upsides and downsides that might come with using an SSE scheme, specifically Secure Indexes in a cloud storage application. I am aware that more secure and well-polished databases exist that utilize searchable encryption, CryptDB [1], for example. Still, I think it is helpful to show a working model of an application running SSE.

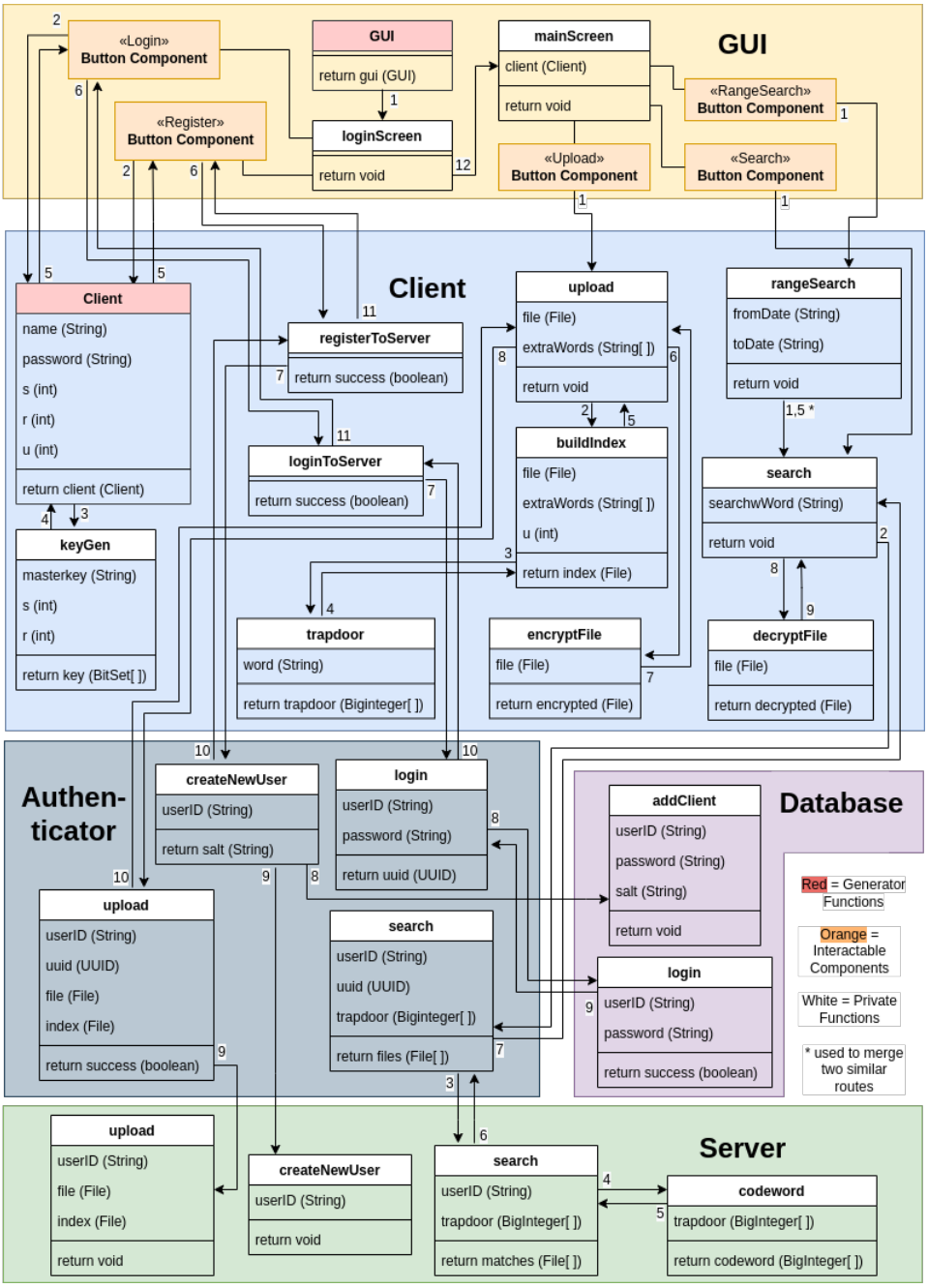


Figure 5.2: Components of the application and their relation

## 5.2 Authentication

It is essential to realize that even though the Secure Indexes scheme is cryptographically secure, it needs a service to handle user authentication. A user should not be allowed to try to interact with another user's documents, even though they have no way of generating valid queries. Another good reason to protect the application with an authentication service is that we could not let users remove their stored documents without being authenticated. There would be no way of preventing a user from deleting other users' stored documents.

A reasonable thought is that an authentication service is needed to store passwords securely, hashed with a good hashing algorithm with salt. However, this is not the case because, without an authentication service, there would be no reason to store the user's passwords. The only reason a password is used in the main scheme is to generate trapdoors and the password is never stored on the server in any way. The only reason to store user credentials on the server is to authenticate the users before giving them access to making queries.

In my application, I have created an authentication layer referred to as the authenticator on top of the Secure Indexes scheme on the server. All communication between a user and the server passes through the authenticator. The authenticator keeps an SQLite [9] database with user credentials used for authenticating users. The authenticator also acts as a salt distributor, used by the user when hashing their password. Before the user can interact with the server(through the authenticator), they have to be given a Universal Unique Identifier UUID. This happens after they are logged in or registered if they are new.

**Registration** is done in two steps. Step 1 is that the user requests salt from the authenticator. The authenticator will generate a 16-byte salt value with a SecureRandom object in java. The generated salt is stored in the database with the current users $userID$. The $userID$ of any user is the hash of the username. As the name suggests, the username will never be shown to the server, only their identification('ID'). The salt is then sent back to the user. Step 2 starts with the user computing their hashed password. The

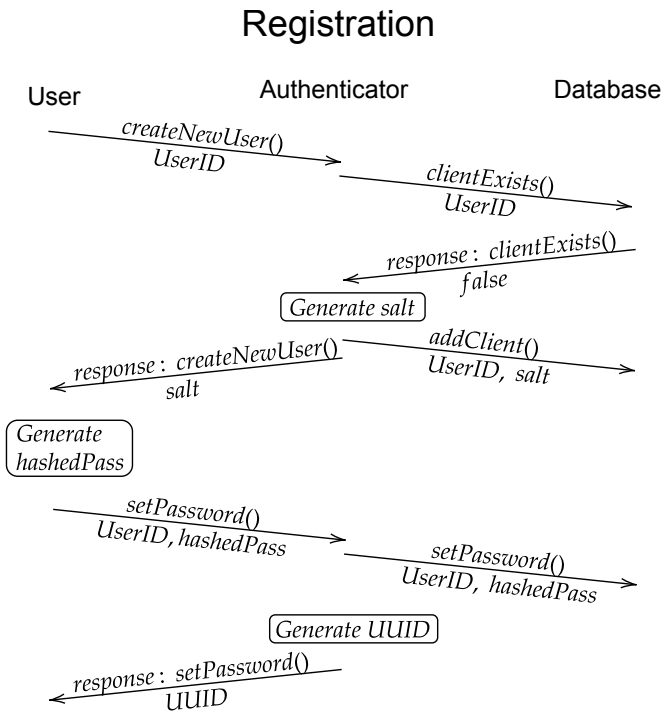


Figure 5.3: Registration

password-salt combination is hashed with PBKDF2, designed explicitly for hashing passwords. The hashed password is then sent to the authenticator with the $userID$. The authenticator updates the database with the provided password. The authenticator then generates the UUID for this session and sends it back to the user. A visual representation of the registration protocol is displayed in Figure 5.3.

**Login** looks similar to the registration and is also done in two steps. Step one starts with the user requesting their salt from the authenticator. The salt value is not secret, so the user does not need verification to get it. The authenticator then uses the $userID$ to get the salt value and return it to the user. Like in step two, during registration, the user computes their hashed password with the salt and sends it to the server. The authenticator then checks if the $userID$ and the provided password match an entry in the database, if so generates a UUID, and sends it to the user. A visual representation of the login protocol is displayed in Figure 5.4.

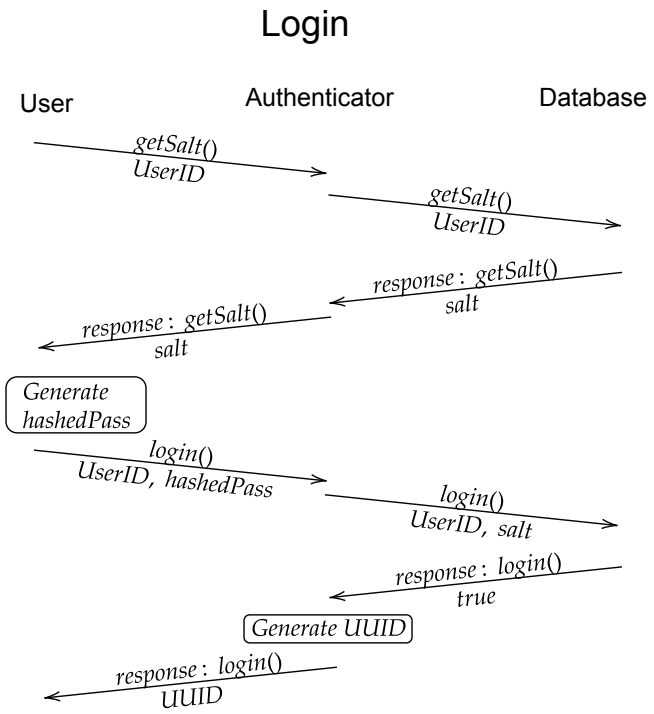


Figure 5.4: Login

After login or registration, all communication to the server is sent through the authenticator. The session UUID is attached for each message the user sends to the authenticator. The server will not process the message if the UUID does not match the one the user was issued during login/registration.

## 5.3 Generating keywords

The main factor in how well this application works in practice is the quality of the keywords for each file. The three main elements that can be searched for in my application are location, objects in the image, and date/time. As the application serves as online storage for pictures and videos, I have used various data points from the files to generate the keywords. The user can also add custom words to each file for easier searching.

The way to obtain information about an image/video is to use a service to extract metadata from the file. In my application, I use a java library called metadata-extractor [37].

### 5.3.1 Location

In many cases, the most natural keyword to search for to find specific images or videos is location. From the metadata, it is possible to extract the longitude and latitude of where the picture/video was taken. I then used an API called LocationIQ [4] to retrieve place names for the provided longitude and latitude. This feature is called reverse geocoding. The "zoom" level (how specific the place names should be) can be adjusted in the API call. The different "zoom levels" can provide country, state, county, city, suburb, street, or building. Since this information is private to the user, I have chosen to include as specific location keywords as possible.

### 5.3.2 Object recognition

As I think an excellent object recognition algorithm can make image search much more user-friendly, I decided to try to add this to my application. It did not work quite as well as I hoped; I still think it shows the possibilities of the feature. This feature unfortunately only works for images.

To get keywords of the images, I use Google Vision AI [11]. This will analyze the image and return any labels it found. The API gives all labels it finds a score between 0 - 1, where 0 means that the algorithm is unsure, and 1 means that the algorithm is confident that the label is correct. I chose to use all keywords that score more than 0.7 as the ones below I found too inaccurate. Most of the time, the API produces good keywords, but sometimes the keywords produced by the API would not be very natural search words. An example of this can be seen on the next page, by comparing the two images 5.6 and 5.5 and their labels from the API in the table 5.1

Figure 5.5: Example picture 1

Figure 5.6: Example picture 2

| Keywords 5.5 | Scores 5.5 | Keywords 5.6 | Scores 5.6 |
|---|---|---|---|
| Plant | 0.9644 | Cloud | 0.9747 |
| Tree | 0.8863 | Sky | 0.9721 |
| Deer | 0.8844 | Wheel | 0.9665 |
| Natural landscape | 0.8803 | Vehicle | 0.9126 |
| Grass | 0.8160 | Motor vehicle | 0.9107 |
| Fawn | 0.8156 | Road surface | 0.8894 |
| Grassland | 0.7971 | Asphalt | 0.8800 |
| Groundcover | 0.7910 | Tree | 0.8460 |
| Terrestrial animal | 0.7842 | Electricity | 0.8236 |
| Landscape | 0.7615 | Rolling | 0.8215 |

Table 5.1: Keywords for picture 5.5 works much better as searchwords than for picture 5.6

### 5.3.3 Date

The date of the image/video can be found in the metadata. To make searching more convenient for the user, I used the date to create other related search words. Keywords like "summer" or "Sunday" are not listed in the metadata but can be generated by analyzing the date. I also included year, month, and day combinations as search words. This makes it possible to search for images/videos in "march 2020" or "17. of may" (Norway's national day). This feature can be expanded as much as needed. It would be easy to add "morning", "afternoon" and "night" depending on the time of day for example, or the name of holidays for specific dates.

I added a range search option as an extension to the Date keyword. This makes it possible to search for all dates between two dates. This can also be done without specifying the day or month and day. Internally in the application, this means to send requests for all dates between the two provided dates. In the case of searching between two months, you will only need one request per month between the two provided months.

### 5.3.4 Filename

Searching for the filename of the image/video is probably the most common way of searching for files. I added support for searching for the full filename with extension and the filename without the extension. This is a very simple feature but can be very helpful as searching for the filename is a very common way of sorting and searching for files.

### 5.3.5 Custom Keywords

A natural feature for an application like this, where the keywords are generated automatically per file, is the option to add custom keywords. During upload, the user has the option to add custom words.

One issue with the feature of custom keywords is that checking for errors with hash collisions is no longer possible. When the keywords were generated purely based on the current file, the set of keywords could be recreated after a search to make sure that the search word was valid for the given file. With custom keywords, however, there is no way to accurately recreate the set of keywords, without remembering the exact custom words that were used during encryption. I don't think this is a big problem because of the very low chance of error. The only way to produce a false positive is to encounter $r$ independent hash collisions at the same time.

## 5.4 Performance

### 5.4.1 Precomputation

As I'm using the free version of both APIs described earlier, the number of allowed queries per second is low. In the google vision AI API, I'm limited to 1000 requests per month but the speed is good. The free version of LocationIQ claims to handle 2 requests per second but I encountered problems unless I waited for one full second between requests. The API calls add one

full second of extra computation time per file uploaded. Since the speed of the API calls depends on how much money you are willing to spend on the API services, I added the possibility of storing the result of the API calls locally to avoid having to wait for the APIs each time. This, however, is not a very good solution if the application were to be used in a real-world scenario. This is because requiring the user to store information locally defeats the purpose of uploading it to the cloud. However, it could be a useful extra feature in the right circumstances, like in this project for example.

### 5.4.2 Search

Since each file has its own index that has to be checked during a search operation, the search time scales linearly with the number of files in the system. The number of operations needed for each search is the same as the $r$ number(number of subkeys) of the system. Checking a single codeword in the index of a single file consists of calculating a hash and performing a hashtable lookup(hashtable lookups are a constant time operation). Checking for a single codeword is therefore a constant time operation. The full search time is then $O(n \times r)$.

## 5.5 Future Work

### 5.5.1 Better keywords

As mentioned earlier in the thesis, a lot of the application's usefulness relies on the quality of the keywords. If the user cannot guess/remember the keywords for the file they want, the service does not serve its intended purpose. The process of generating better keywords starts by understanding what the application should be used for. If the application is for teachers to store their recorded lectures, find a way to extract the subject and topics of the video. The date and location would also be great keywords in this setting.

### 5.5.2 Support for more metadata formats

As mentioned in the "Generating keywords" section, there are several formats for structuring metadata. This is because different file types can store different types and amounts of metadata. With the method of handpicking out the different parts of information, I need from the metadata, adding support for more formats is very cumbersome. I have not looked into more efficient ways of extracting metadata, but I am sure there are better methods.

### 5.5.3 Data transmission protocols

As the application is not meant to be deployed on an actual server, the communication between a user and the server happens internally, like sending objects between classes. Although protocols would look very similar, the information would have to be sent through streams, and there would need to be implemented ways to make sure the correct data is received without errors. This also means implementing a system to resend information that is not received correctly.

### 5.5.4 Forgot password feature

In its current state, the only way to connect to your account is with your username and password. If you forget or lose your password, there is no way of retrieving or changing it. This is especially devastating because you lose access to the files stored on the account. One way to solve this is to add support for alternative authentications methods. This would allow the user to authenticate themselves with other credentials than the username and password, making it possible to retrieve or reset one's password. If we add the possibility of chaining a user's password, we would either have to change the current direct connection between the user's password and their private key or download and re-encrypt every file with the new private key after changing passwords.

# Chapter 6

# Conclusion

The first goal of this thesis was to provide an implementation for the two SSE schemes by Song et al. [41] and Goh [25], and test their performance. In Chapters 3 and 4, I discuss various implementation decisions made throughout the project and the reasoning behind them. This could be a valuable resource for anyone wanting to implement similar schemes in Java.

By comparing the tests performed on the implementations, we can draw the simple conclusion that the Secure Index scheme is significantly faster than Song et als. scheme. Storage space requirements however are substantially higher in the secure indexes scheme. We can tell from the tests that an important factor in the storage space requirements, is the size of the variable $u$. The amount of security provided by increasing $u$ is not analyzed in this thesis and might be an interesting topic of research in the future.

The second goal of the thesis was to propose a working example of an application running SSE as the main encryption scheme. This was to show some of the positive features of Secure Indexes, and to demonstrate some of the limitations of the scheme.

# Bibliography


[1] CryptDB. https://css.csail.mit.edu/cryptdb/.

[2] Intel Core i58350U Processor 6M Cache up to 3.60 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/124969/intel-core-i58350u-processor-6m-cache-up-to-3-60-ghz.html.

[3] Java securerandom algorithms. https://docs.oracle.com/javase/10/docs/specs/security/standard-names.html#:~:text=Example%3A%0APBKDF2WithHmacSHA256.-,SecureRandom%20Number%20Generation%20Algorithms,-The%20algorithm%20names. (Accessed on 05/21/2022).

[4] LocationIQ - Free & Fast Geocoding, Reverse Geocoding and Maps service. https://locationiq.com/geocoding.

[5] Nativeprng - github. https://github.com/frohoff/jdk8u-jdk/blob/master/src/solaris/classes/sun/security/provider/NativePRNG.java.

[6] Openssl - cryptography and ssl/tls toolkit. https://www.openssl.org/. (Accessed on 05/21/2022).

[7] Password storage - owasp cheat sheet series. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. (Accessed on 04/13/2022).

[8] The sonnets. https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt.

[9] SQLite Home Page. https://www.sqlite.org/index.html.

[10] Trivium performance test source code. https://github.com/kni034/Trivium-performance-test. (Accessed on 05/21/2022).

[11] Vision AI | Derive Image Insights via ML | Cloud Vision API. https://cloud.google.com/vision.

[12] *Announcing the Advanced Encryption Standard (AES)*. 2001.

[13] Michael Adjedj, Julien Bringer, Hervé Chabanne, and Bruno Kindarji. Biometric identification over encrypted data made feasible. In *Information Systems Security*, Lecture Notes in Computer Science, pages 86–100. Springer Berlin Heidelberg, Berlin, Heidelberg.

[14] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004*, Lecture Notes in Computer Science, pages 506–522, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[15] J Bringer, H Chabanne, and B Kindarji. Error-tolerant searchable encryption. In *2009 IEEE International Conference on Communications*, pages 1–6. IEEE, 2009.

[16] Christophe De Canniere and Bart Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2005.

[17] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE transactions on parallel and distributed systems*, 25(1):222–233, 2014.

[18] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer Berlin Heidelberg, Berlin, Heidelberg.

[19] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *2012 IEEE International Conference on Communications (ICC)*, pages 917–922. IEEE, 2012.

[20] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Conference on Computer and Communications Security: Proceedings of the 13th ACM conference on Computer and communications security; 30 Oct.-03 Nov. 2006*, CCS '06, pages 79–88. ACM, 2006.

[21] Quynh Dang. Secure hash standard (shs), 2012-03-06 2012.

[22] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[23] Morris Dworkin. Recommendation for block cipher modes of operation methods and techniques, 2001-12-01 2001.

[24] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Report 2015/927, 2015. `https://ia.cr/2015/927`.

[25] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.

[26] Philippe Golle, Jessica Staddon, and Brent Waters. Secure conjunctive keyword search over encrypted data. In *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 31–45, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[27] Russ Housley. Cryptographic Message Syntax (CMS). RFC 5652, September 2009.

[28] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 258–274. Springer Berlin Heidelberg, Berlin, Heidelberg.

[29] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 285–298. Springer Berlin Heidelberg, Berlin, Heidelberg.

[30] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. Efficient similarity search over encrypted data. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1156–1167, 2012.

[31] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *International Conference on Management of Data: Proceedings of the 2006 ACM SIGMOD international conference on Management of data; 27-29 June 2006*, SIGMOD '06, pages 121–132. ACM, 2006.

[32] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *2010 Proceedings IEEE INFOCOM*, pages 1–5. IEEE, 2010.

[33] P.P Liesdonk, van, S Sedghi, J.M Doumen, P.H Hartel, W Jonker, and M Petkovic. Computationally efficient searchable symmetric encryption. In *Lecture notes in computer science*, Lecture Notes in Computer Science, pages 87–100. Springer, Berlin, Heidelberg, 2010.

[34] Keith Martin. *Everyday Cryptography: Fundamental Principles & Applications*. Oxford University Press, 2nd edition, June 2017.

[35] RALPH CHARLES MERKLE. Secrecy, authentication, and public key systems, 1979.

[36] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017.

[37] Drew Noakes. drewnoakes/metadata-extractor. `https://github.com/drewnoakes/metadata-extractor`, May 2022. original-date: 2014-11-19T00:13:55Z.

[38] Dong Jin Park, Kihyun Kim, and Pil Joong Lee. Public key encryption with conjunctive field keyword search. In *Information Security Applications*, Lecture Notes in Computer Science, pages 73–86. Springer Berlin Heidelberg, Berlin, Heidelberg.

[39] Geong Sen Poh, Ji jian Chin, Wei chuen Yau, Kim kwang Raymond Choo, and Moesfa Soeheila Mohamad. Searchable symmetric

encryption: Designs and challenges. *ACM Computing Surveys*, 50(3), 2017.

[40] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[41] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. 2000.

[42] William Stallings. Computer security : principles and practice, 2018.

[43] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. *IACR Cryptology ePrint Archive*, 2013:832, 2013.

[44] Yunling Wang, Jianfeng Wang, and Xiaofeng Chen. Secure searchable encryption: a survey. *Journal of communications and information networks*, 1(4):52–65, 2017.

[45] Zhihua Xia, Xinhui Wang, Xingming Sun, and Qian Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE transactions on parallel and distributed systems*, 27(2):340–352, 2016.

[46] Sergej Zerr, Daniel Olmedilla, Wolfgang Nejdl, and Wolf Siberski. Zerber +r: top-k retrieval from a confidential index. In *Proceedings of the 12th International Conference on extending database technology*, EDBT '09, pages 439–449. ACM, 2009.

[47] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. Vabks: Verifiable attribute-based keyword search over outsourced encrypted data. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 522–530. IEEE, 2014.