Secure Data Search via
Searchable Encryption Using Hyperledger Fabric Smart Contracts

Mălina Bulină
Supervisors: Dr. Kaitai Liang and Dr. Roland Kromes
EEMCS, Delft University of Technology, The Netherlands

A Dissertation Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Data outsourcing has become one of the primary means for preserving information as it passes the responsibility of storage management to the service provider. However, storing sensitive data remotely poses privacy threats for the data owners. Searchable encryption (SE) is a technique that allows performing search queries over encrypted data. The majority of SE solutions model the server as an honest-but-curious entity. If this is not the case, the results of the queries might not be reliable. The issue can be mitigated by implementing SE within blockchain technology. This paper proposes a searchable encryption scheme that uses smart contracts in Hyperledger Fabric. For storing a set of documents securely, the data owner chooses an identifying keyword for each document. The identifying keywords and documents ids are stored in a matrix that facilitates keyword search; consequently, the matrix is appended to the ledger. For retrieving a document, the data owner builds an encrypted query (trapdoor) using the identifying keyword; the trapdoor is passed to the smart contract. Thus, the data owner delegates the smart contract to perform the query on their behalf. The data owner receives the document id, which can then be used to retrieve the respective content. The proposed protocol achieves faster data pre-processing, i.e., matrix computation, when the number of documents is smaller. The file size does not affect the time efficiency of the scheme. Nonetheless, the execution time for pre-processing increases with regard to the number of documents. As a result, the system is Input/Output (I/O) Bound.

1 Introduction

Data confidentiality has gained substantial interest due to the proliferation of information stored in open infrastructures. Outsourcing data to the cloud is a preferred alternative compared to preserving it in local storage. In this manner, the concerns regarding memory management and data retrieval are circumvented by the user.
As possibly confidential data is stored on remote servers, there exist risks regarding potential attackers such as administrators, or hackers having root rights, which can lead to them having full access to the servers and consequently to the plaintext (PT) data. Thus, it is important to have the data stored encrypted on an untrusted server. By encrypting the PTs into cyphertexts (CTs), it is guaranteed that no attacker can recover the PT without the necessary decryption keys. However, it becomes more difficult for trusted parties/data owner(s) to search the CT data compared to searching the PT data [1].

Searchable encryption (SE) is a technique that allows users to search encrypted data in a secure manner. In the majority of SE schemes, the remote server is modelled as an honest-but-curious entity that always executes correctly search queries on behalf of its users. It turns out that this is not always the case. If the server is malicious, the results of the request might not be reliable.
Existing protocols, e.g., [2; 3], implement verifiable searchable encryption schemes in order to check the validity of search results. Nonetheless, malicious behaviour can be detected without penalty in most schemes. Thus, a verifiable searchable encryption scheme that can restrain the demeanour of all participating parties is required [4].

1.1 State of the Art

One candidate solution for mitigating this problem is depicted by implementing searchable encryption within distributed ledger technology (DLT). There have already been proposed solutions implementing SE within DLT [5; 6; 7]. DLs are distributed databases shared across a network of peers. Distributed ledgers provide trust, immutability, transparency and provenance.
An example of DLT is Hyperledger Fabric (HLF) [8]. HLF is a modular and versatile distributed ledger framework that supports the collective development of blockchain-based [9] distributed ledgers. Ledgers hold records of all the state transitions in the fabric. Records are stored as a chain of blocks in order to preserve sequencing and immutability. State transitions are the result of the transactions submitted by the participating peers. The current state is maintained in the state database contained in the ledger. The ledgers and the smart contracts are hosted by the peer nodes.
Transactions (TXs) can create, update or delete the information on the ledger. TXs are committed to the ledger as a collection of assets represented by key-value pairs. Assets define the structure of the resources shared between the participating nodes.

Chaincode is used for asset manipulation, i.e., asset definition, modification and other actionable logic. In HLF the terms smart contract and chaincode are used interchangeably. Smart contracts (SCs) [10] are transaction protocols that are executed in an automated manner in accordance with some predetermined conditions. These encapsulate the shared information and procedures in a network. SCs define the TX logic that alters the lifecycle process of a business object contained in the world state. The SC is packaged into chaincode which is then deployed to the network.
Peers interact among each other through channels. Chaincode invocation can be done through peer connections. Peers can belong to different organisations which share common rules for assets manipulation. Smart contracts can be created in any organisation in order to share the business logic among the consortium members.
The design of HLF is intended for enterprise use, satisfying a broad range of industry use cases. It was started in December 2015 by the Linux Foundation [11].

1.2 Searchable Encryption

A traditional searchable encryption scheme allows the client to delegate a server for conducting the data search

on its behalf. The data owner can identify the documents $D_1, D_2, D_3, ..., D_N$ by selecting a variable number of identifying keywords $w_i, w_{i+1}, ...$ for each document D_j. For encrypting the PTs, the data owner has to first create an index table I that maps the keywords to the respective documents. Then, the content is encrypted and outsourced to the cloud. For encrypting the documents, the majority of the implementations use symmetric key encryption [2; 12; 13; 14; 15]. This allows only the key holder to create searchable CTs and trapdoors. Zhang et al. [16] proposed a searchable asymmetric encryption method that supports multiple data owners. Any user can create searchable CTs under the public key, whereas trapdoors are generated using the corresponding private key.
For searching the data, the key holder has to build a trapdoor $T = \text{Trapdoor}_K(w)$, with K the encryption key and w the identifying keyword. Trapdoors are encrypted queries used for search delegation. These allow seeing whether the encrypted keyword w_{ENC_K} is contained in the encrypted file D_{i,ENC_K} using the index table I. If it is, the identifier of the document D_i is returned and the client can decrypt the index and search for the document.

This paper studies **searchable encryption using smart contracts in Hyperledger Fabric**. Firstly, the necessary concepts and open issues are discussed more in-depth, followed by modelling a candidate solution for the problem. Then, an implementation is provided using a simulation environment. Lastly, the setup and results of the model are discussed.
The structure of the paper is as follows. In section 2 the methodology is presented, followed by the protocol overview and system model in section 3; section 4 describes the simulation scenario, along with the results obtained. The ethical aspects of this study are discussed in section 5. section 6 reviews the results obtained and discusses any possible edge cases. section 7 concludes this paper and lays the foundations for future work.

2 Methodology

For developing greater understanding about searchable encryption and Hyperledger Fabric, the literature available has been reviewed. After gathering enough information, an initial design of the protocol was proposed, followed by implementing the actual model and correcting any details. Subsequently, the performance of the algorithm was assessed. The work was concluded by logging all the findings and drawing the conclusions that followed.
The implementation was done using as programming languages Java [17] and Go [18]. For the integrated development environment (IDE), IntelliJ IDEA and Visual Studio Code (VS Code) were used. The network can be run either by using the command line interface, by installing the IBM Blockchain Platform VS Code plugin [19] or by cloning the application gateway in Java [20]. The simulation environment can be reproduced using the test network provided by Hyperledger Fabric [21].

2.1 Problem Description

Storing data remotely has become one of the most prominent ways of information preservation. In the most classical scenario, the user directly uploads their data to the remote service, leaving the encryption process to the service provider. This method, however, does not guarantee privacy and security.
A searchable encryption scheme is considered to be secure if there is no possibility for an unauthorised entity to learn the plaintexts based only on the cyphertexts [12]. Privacy is achieved by encrypting the data locally prior to outsourcing it. Nonetheless, the difficulty of performing data search increases significantly.

Symmetric Searchable Encryption (SSE) was first proposed by Song et al. [22]. In this scheme, the user data is stored remotely in a secure manner. Information retrieval is done by selecting search segments of the data, while document decryption is done locally. The data owner is able to allow access to other users who can perform search queries using the provided trapdoor.
Su, Zhang and Mu introduce in [23] a search framework for health systems. The BA-RMKABSE scheme is based on two smart contracts, one for determining whether the query trapdoor matches the secure indexes (search smart contract) and one for checking the correctness of the results returned and calculating the score of the corresponding files. It returns the top-k files matching the search criteria. However, BA-RMKABSE is based on the Ethereum blockchain [24] and focuses on multi-keyword ranking search.
Tahir and Rajarajan provide in [25] a framework, Permissioned Blockchain-based Searchable Encryption (PBSE), that facilitates keyword search over encrypted data stored on the blockchain network. The scheme is based on probabilistic trapdoors, thus it is privacy-preserving. For each keyword search, probabilistic trapdoors generate different search tokens. This way, the search pattern remains hidden and protected against distinguishability or passive attacks. PBSE comprises of eight polynomial time algorithms for key generation, signature generation, index construction, index embedding, trapdoor creation, data appending, search outcome and data decryption. The scheme proposed in this study is based on the theoretical concepts presented in the PBSE framework.

3 Protocol Overview

This section provides an overview of the steps involved in implementing searchable encryption in Hyperledger Fabric Smart Contracts. The terms notations and abbreviations used for describing the protocol are shown in Table 1.

3.1 System Model

The first step of the scheme is **key generation**. The result of the function invocation is a symmetric key K generated using the Advanced Encryption Standard (AES). AES is a variant of the Rijndael block cipher [26].

K	*main key*
$D = \{D_1, D_2, .., D_n\}$	*set of documents*
$\tilde{D} = \{\tilde{D}_1, \tilde{D}_2, ..., \tilde{D}_n\}$	*set of encrypted documents*
$w = \{w_1, w_2, ..., w_n\}$	*set of selected keywords*
$ENC_K(PT)$	*encryption of PT using key K*
$DEC_K(CT)$	*decryption of CT using key K*
$HMAC_{H,K}(M)$	*HMAC of message M using hash function H and key K*
I	*index table*
λ	*security parameter*
$T_K(w)$	*trapdoor function on keyword w using key K*

Table 1: Terms Notations and Abbreviations

Data Appendment

Before appending the data to the blockchain, the data owner (DO) selects an identifying keyword for each document, followed by constructing the **index table** I. The index table shows the presence or absence of a keyword from a document. For building I, the DO has to provide the symmetric key K generated in the first step, along with the documents and their respective keywords. The function returns a two-dimensional index array, I. The first row of the matrix contains the modular inverses (with respect to some big prime number P) of the hashed keywords, whereas the first column holds the encrypted document identifiers $\{\tilde{id}_{D_1}, \tilde{id}_{D_2}, ..., \tilde{id}_{D_n}\}$. The rest of the matrix is filled with values corresponding to the frequency of the keywords for each file.

Hashing the data is done using HMAC-SHA-X. SHA-X represents one of the hash functions that have digests (hash values) equal to 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 or SHA-512/256. Hash functions are mathematical algorithms that map data of variable length to a fixed-size bit array, known as the hash value or the message digest. These are one-way functions (preimage resistant) which implies that is practically infeasible to invert or reverse the computation [27]. HMAC [28] is a keyed-hash message authentication code that uses a cryptographic hash function, in this instance SHA-X, and a secret cryptographic key, i.e., K. HMAC can be used to determine whether a message sent over a channel has been tampered with, provided that the sender and receiver share a secret key.

The **encryption** is done using the AES algorithm in Cipher Block Chaining (CBC) mode with PKCS#5 Padding. Compared to the Electronic Codebook (ECB) mode, CBC excels in hiding away patterns in the PT by making the encryption of each block dependent not just on the key, but also on the CT of the previous block. This is done by doing XOR operations between the current block and the ciphertext from the previous block. For the first block in the text, an initialisation vector is required for XORing. PKCS#5 Padding is a padding scheme that pads the PT to be multiples of 8-byte blocks [29].

After generating the index table I, the data owner encrypts the content of the documents using the algorithm previously described (AES/CBC/PKCS#5Padding) and stores the set of encrypted documents $\tilde{D} = \{\tilde{D}_1, \tilde{D}_2, ..., \tilde{D}_n\}$ on the ledger. The index table is stored by invoking the smart contract.

Smart Contract Model

The smart contract defines two assets, one for the document and one for the index matrix. For each document, the document identifier (id), name, corresponding keyword and document content are stored. For the matrix, the variables tracked are the identifier of the matrix (id) and the actual two-dimensional array I. The private fields corresponding to the two classes are presented in Listing 1.

```
public class Matrix {
$$@Property()
private String id;
$$@Property()
private String[][] I;
}
public class Document {
$$@Property()
private String id;
$$@Property()
private String name;
$$@Property()
private String keyword;
$$@Property()
private String content;
}
```

Listing 1 Structure of the Matrix and Document classes

The smart contract provides functions for asset operations such data creation and data retrieval. Apart from these methods, the SC implements methods for storing the index matrix I on the ledger and for searching the data.

Data Search

Once the encrypted documents and the index table I are recorded in the blockchain network, the data owner can perform search queries for retrieving the files corresponding to the keyword searched for. In order to do this, DO has to first build the **trapdoor**. This is performed locally. The trapdoor $T_K(w)$ is created by providing the keyword w to search for, the key K previously used for encrypting the document D_i, along with the initialisation vector and the hash function H. The resulting $T_K(w) = (d, c)$ is composed of two numbers, one that contains the HMAC of the encrypted keyword modulo the big prime number P: $d = HMAC_{H,K}(ENC_K(w) \bmod P)$ and one that holds the product between the HMAC of the keyword $a = HMAC_{H,K}(w) \bmod P$ and the encrypted keyword $b = ENC_K(w) \bmod P$ reduced modulo the prime number P: $c = a \cdot b \bmod P$.

Once $T_K(w)$ is generated, the trapdoor holder can delegate the smart contract to perform **search** operations. Thus, the algorithm is run by the smart contract and not locally. The

search method requires as input the trapdoor $T_K(w)$ and returns the encrypted document identifier $\tilde{id}_{D_i}$ that satisfies the search condition, i.e., D_i corresponds to the keyword w DO was looking for.
After retrieving the respective encrypted document identifier $\tilde{id}_{D_i}$, the data owner decrypts it locally, $DEC_K(\tilde{id}_{D_i})$. DO retrieves the encrypted document contents $\tilde{D}_i$ by calling the method contained in the smart contract for document retrieval. Lastly, the encrypted document $\tilde{D}_i$ is decrypted locally $D_i = DEC_K(\tilde{D}_i)$ and the content of the document D_i can be accessed.

3.2 System Implementation

This subsection provides the implementation details of the protocol described in subsection 3.1.
The construction of the index table I is shown in algorithm 1. It takes as input the symmetric key K, a dictionary that maps the set of documents D to the set of corresponding keywords w and the index table I. Initially, I is a matrix containing only zeroes. The function updates the index table and returns the encrypted documents $\tilde{D}$.

Algorithm 1 buildIndex $(K, Dict\langle D, w\rangle, I)$

```
for each kw ∈ w do                          ▷ compute keyword frequency
    for each d ∈ D do
        for each t ∈ d do                   ▷ current word in d
            if t = kw then
                update frequency of kw for d in I
            D̃ ← D̃ + ENC_K(t)
j ← 1
for each (kw, d) ∈ Map(w, D) do             ▷ kwd-doc pair
    I[0][j] = HMAC_{H,K}(kw) modInverse P
    I[j][0] = ENC_K(id_d)
    j ← j + 1
for each i ∈ I do                           ▷ mask frequency values
    // i = elements of I excl. first row and column
    i ← (i · R) mod P                       ▷ choose R ∈ Z_P
return D̃
```

Trapdoor creation is illustrated in algorithm 2. It takes as input the symmetric key K along with the keyword to search for and returns the corresponding trapdoor.

Algorithm 2 buildTrapdoor (K, w)

```
b ← ENC_K(w) mod P
a ← HMAC_{H,K}(w) mod P
c ← (a · b) mod P
d ← HMAC_{H,K}(ENC_K(w) mod P)
return T_K(w) = (d, c)
```

In algorithm 3 the search procedure is presented. The trapdoor $T_K(w)$ is passed as input. The algorithm verifies if the modular inverse of the hashed keyword contained in the first row of I, i.e. variable i, reduces to one when multiplied by the variable c of the trapdoor. The HMAC of the product is stored in h. If the product yields 1, then h is equal to the HMAC of the of the encrypted keyword, $HMAC_{H,K}(ENC_K(w) \bmod P)$, thus equal to the trapdoor variable d.

Algorithm 3 search $(T_K(w) = (d, c))$

```
j ← 1
while j ≤ I.length do
    i ← I[0][j]
    m ← (c · i) mod P
    h ← HMAC_{H,K}(m)
    if d = h then
        return ĩd_{D_j}
    j ← j + 1
return
```

4 Experimental Setup and Results

The first dataset used for evaluating the performance consisted of 50 text files (10 files for each data size of 250 KB, 500 KB, 1 MB, 5 MB, and 25 MB). It was used for Figure 1 and Figure 2. The second dataset contained 10, 20, 30, 40, and 50 grouped documents, each of size 500 KB. The 5 groups of files were compared in terms of time performance; these are illustrated in Figure 3. The figures were plotted in Python [30] using Matplotlib [31]. The algorithms were run either remotely using the smart contract or locally using IntelliJ IDEA and Visual Studio Code on a macOS with 2.4GHz quad-core Intel Core i5 processor with 128 MB of eDRAM and 8GB memory. For evaluating the protocol, the test network needs to be started using the command line interface (CLI). The smart contract is developed using VS Code and deployed from CLI. After the deployment of the smart contract, the Fabric Gateway client API is used to connect to the Fabric Gateway. The gateway connection has three requirements: a gRPC connection, a client identity and a signing implementation. Once the parameters are established, the ledger is populated using chaincode invocations. The transactions are executed using the client API.

The proposed protocol works as follows. Firstly, the data owner has to set up the security parameters. The key is generated using AES, as specified in subsection 3.1. Subsequently, DO constructs the dictionary that maps each document to its corresponding keyword. The dictionary is passed to the buildIndex (algorithm 1), along with the generated key and the index matrix that contains zeroes.
The buildIndex method consists of two main subroutines: the computation of the middle elements, i.e., all elements except the first row and column, and the computation of the remaining (border) elements, i.e., the first row and column. The execution time of the latter subroutine is shorter than the time to compute the middle elements by more than three orders of magnitude.
In Figure 1 is illustrated the average time to perform the

buildIndex method. The duration of the subroutines for border and middle matrix computation are shown as well. The averages depicted are computed over multiple runs of the algorithm. The y-axis shows the time necessary to finish the procedure. The difference between the time to compute the middle matrix and the time to compute the border is so significant that the latter was not visible in the graph unless zoomed in. In order to solve this, the logarithmic scale was used for the time axis (y-axis). The unit of measurement is minutes. The x-axis illustrates the average size of the files used. The time required to complete algorithm 1 grows as the size of the files increases.

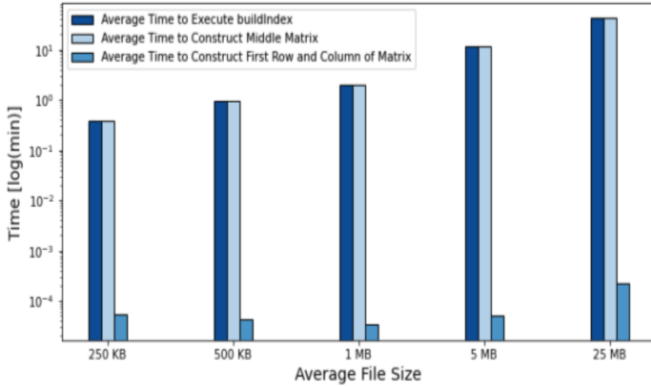

Figure 1: Plot of the average time to execute the buildIndex algorithm and the two subroutines contained in it: build middle matrix and build border matrix. The average size of the files is shown on the x-axis, whereas the average time to complete the corresponding method (in units of minutes, logarithmically scaled) is on the y-axis.

It can be seen that the total time to execute the buildIndex method differs from the time to construct the middle matrix only by a small value. As middle matrix creation is contained in algorithm 1, the remaining difference in the execution time of buildIndex is left for the border matrix computation and for other $\mathcal{O}(1)$ operations. This time complexity is negligible compared to the complexity of middle matrix computation. This result was expected, as the middle matrix subroutine includes the encryption of the documents. It reads the PTs and ensures that the CTs are saved after encryption. Each document is encrypted word by word, then the subroutine updates the frequency of the keywords found for each document. In terms of complexities, the middle matrix subroutine has $\mathcal{O}(|w| \cdot |D| \cdot \mathcal{C})$ time complexity, where $|A|$ represents the cardinality of set A and $\mathcal{C}$ is the maximum length of any document from the set D of documents. The border matrix computation does not affect the time complexity of the buildIndex method, thus algorithm 1 is $\mathcal{O}(|w| \cdot |D| \cdot \mathcal{C})$.

Upon completion of the buildIndex method, the resulting matrix and the encrypted documents are stored by invoking the smart contract and executing the required transactions. Consequently, the data owner can perform search queries. The search procedure is presented as pseudocode in algorithm 3. It uses the output of algorithm 2: $T_k(w)$, the trapdoor for keyword w using key K. By invoking the search procedure, the smart contract performs the query on behalf of the DO. The time complexity of algorithm 3 depends on the size of of the index matrix I and is $\mathcal{O}(|w| \cdot |D|)$. The procedure starts by iterating through the first row of I and once it finds the respective keyword, it looks for the documents that are mapped to it. The average execution time of the buildIndex and search methods are depicted in Figure 2. One important thing to note is that algorithm 1 is run locally by DO, as specified in section 3. The search routine consists of iterating through the index matrix I. This is done by the smart contract, thus the procedure is executed remotely.

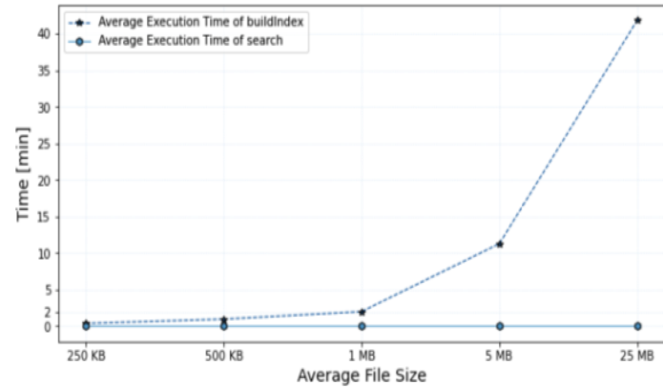

Figure 2: Plot of the average execution time of algorithm 1 and algorithm 3. The x-axis shows the average size of the files. The average time to perform the methods is depicted on the y-axis in units of minutes.

As observed in Figure 2, the search method is significantly faster than buildIndex. The time required for trapdoor creation, which is done locally, is in the order of nanoseconds, therefore it was not included in the performance analysis. The time t of algorithm 1 grows proportionally to the inputted file size: from 500 KB to 1 MB, $\frac{t_{1MB}}{t_{500KB}} \approx 2$ and from 1 MB to 5 MB, $\frac{t_{5MB}}{t_{1MB}} \approx 5$.
According to [32], the normal distribution for file sizes of work-related documents has mean 1473 KB (1.47 MB) and median 8 KB, whereas for personal-use documents, the mean is 2284 KB (2.28 MB) and the median 7 KB. However, IT-related files are smaller, their distribution having mean 1165 KB (1.17 MB) and median 2 KB. 346 participants contributed to the study. The size distribution for all the category of documents, i.e., operating system files, personal, work and study related, has file size median 5 KB and mean 1528 KB (1.53 MB). Thus, as observed in Figure 2, the average time required to compute the index matrix for 10 files of 1.5 MB is around 5 minutes. However, the duration of the method execution is dependent on the machine that runs the implementation.

Thus, it can be seen from both, Figure 1 and Figure 2 that the time increases proportionally to the file size. Having compared the time efficiency when operating with files hav-

ing different sizes, the next analysis focuses on comparing varying number of files having similar size. In Figure 3 the execution time is evaluated for a dataset having files of similar size ($\approx$ 500KB), divided into groups of 10 to 50 files.

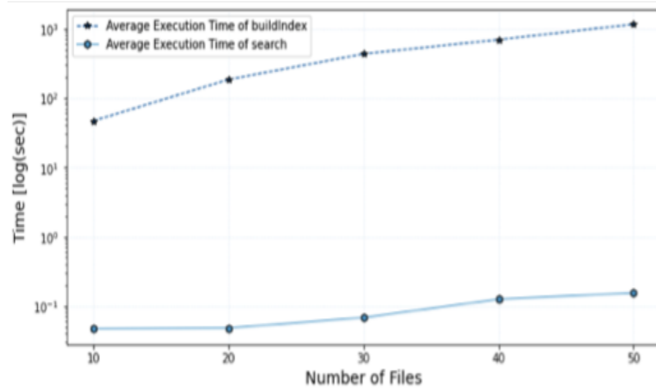

Figure 3: Plot of the average execution time of algorithm 1 and algorithm 3. On the x-axis the number of files is depicted. The time to perform the methods is shown on the y-axis in units of seconds. For the y-axis, the logarithmic scale was used.

As the values in the time distribution for executing algorithm 1 are significantly larger compared to those for algorithm 3, the logarithmic scale was applied. The time necessary to compute the index matrix for 10 documents, as illustrated in Figure 3, is 47 seconds (without the logarithmic scale). It increases then to 185, 433, 694 and 1160 seconds for 20, 30, 40 and 50 documents, respectively. For the latter case, when 50 documents are used, it takes 1160 sec $\approx$ 20 min to compute the matrix *(scenario I)*. The total data size is approximately 50 docs $\cdot$ 500 KB $\approx$ 25 MB.
In the scenario depicted in Figure 2, for 10 files of 5 MB each, it takes 11 minutes to build the matrix. The total data size for the 5 MB files is 10 docs $\cdot$ 5 MB = 50 MB. As 11 minutes are required to build the index using 50 MB of data (in the scenario with 10 files of 5 MB each), it should take approximately 6 minutes to finish the execution for a set of documents having 25 MB in total *(scenario II)*.
When using a 25 MB documents set, scenario I (SI) takes 20 minutes to complete, whereas scenario II (SII) takes approximately 6 minutes. In the first scenario, there are 50 documents (each of 500 KB); in the second scenario, there are 10 documents (each of 5 MB). As $\frac{t_{SI}}{t_{SII}} = \frac{20}{6} \approx 3$, it follows that the buildIndex algorithm takes longer when there are more files. Despite the fact that the files have larger sizes in scenario II, scenario I takes longer. Thus, the bottleneck of algorithm 1 is in the input/output processing (reading the PTs and writing the CTs). In the event that the second scenario took longer, one possible reason could have been the complexity of the middle matrix computation. However, this is not the case.

For the search procedure (algorithm 3), when querying over a set that consists of a varying number of files (each having 500 KB), the duration of execution is 0.047, 0.048, 0.068, 0.125, and 0.154 seconds or 10, 20, 30, 40 and 50 documents, respectively. This result was expected, as the size of the index matrix increases proportionally to the number of documents (and keywords), thus it takes more time to iterate over a matrix that has more rows (and columns).

5 Responsible Research

The purpose of this study is to develop secure data search via searchable encryption using Hyperledger Fabric smart contracts. The contributors of this research are not affiliated with any distributed ledger, smart contract or searchable encryption technology that could result in a conflict of interest. For conducting the study, no personal information was used. All the resources that were utilised are referenced and cited in the respective excerpts of this paper.

This research focuses on secure data search using searchable encryption. The searchable encryption scheme provided is built upon the Permissioned Blockchain-based Searchable Encryption protocol provided in [25], as previously specified in subsection 2.1. The main methods used for implementing SE are illustrated in subsection 3.2 as pseudocode. These can be modified and integrated into other searchable encryption frameworks or extended to provide supplementary functionalities within HLF. Additionally, the proposed protocol can be used to create secure indexes for searchable encryption outside the Hyperledger Fabric Environment, e.g., for storing data using a remote storage provider. Furthermore, the smart contract can be enhanced to store more complex data types. This matter is discussed more in-depth in section 7.
The experiments discussed in this paper can be reproduced using the smart contract implementation [33] described in section 3.1 and the Fabric Gateway Client Application [34]. The latest version (v2.4) of Hyperledger Fabric was used. When starting the test network, multiple Docker containers are created. The containers are used for running peer nodes, orderers, certificate authorities and other necessary Fabric images.

With regard to the security of the proposed searchable encryption protocol, only the authorised users can query the encrypted documents and retrieve the relevant document identifier. The trapdoor creation requires as input parameter the key used by the data owner for building the index matrix. Thus, if DO does not share the key, no other user can perform read operations. Additionally, the chaincode used for manipulating the index matrix can be executed only if the participating peer is a member of the channel on which the smart contract is implemented. As Hyperledger Fabric is a private blockchain network, it supports memberships based on permission. All the participating nodes must have known identities, leading to an additional level of security.
HLF uses the Raft consensus algorithm which reaches general agreement through leader election. In comparison, the blockchain for the Bitcoin network uses Proof of Work (PoW). The PoW mechanism requires a vast amount of com-

puting power, leading to significant amounts of energy being consumed by the miners for solving blocks. Each miner succeeds with a probability proportional to the computational effort expended, making the protocol energy inefficient. In the Raft algorithm, every node has an equal probability of becoming a leader, thus it is not dependent on the specific hardware. Additionally, it is much faster than PoW, but limited in terms of scalability due to its architecture. Thus, in terms of energy efficiency and environmental ethics, Hyperledger is the preferred alternative.

6 Discussion

As this research focuses on secure data search using distributed ledger technology, various factors need to be taken into account when developing the protocol.

First of all, the proposed scheme has to be secure and have no privacy breach. The algorithm is based on trusted atomic primitives such as encryption and hashing. Due to the architecture of the system, the scheme is suitable for data outsourcing, i.e., single-writer/single-reader. In order to extend the implementation to support multi-writer, the encryption should be done using a public key encryption (PKE) scheme. In this scenario, the decryption of a document is performed with the private key corresponding to the public key used for encrypting the respective document. Thus, multi-user writing is supported by using the public key. However, multi-user reading cannot be performed as only the key holder can execute the searches. For supporting multiple readers, public key encryption with keyword search (PEKS) should be used. Besides PKE and PEKS, there exist other techniques used for implementing SE, i.e., predicate encryption (PE), inner product encryption (IPE), anonymous identity-based encryption (AIBE) and hidden-vector encryption (HVE).
For speeding up the search procedure, a common tool used is indexing. By introducing indexing, the search complexity is significantly reduced, leading to more efficient scheme. However, the increased performance comes with the cost of introducing a pre-processing step: computing the indices (algorithm 1). The index can be either forward (one index per document) or inverted (one index per keyword). Sublinear time complexity can be achieved using inverted indexing. The current implementation uses a dictionary that maps documents to their respective keywords; the dictionary is stored as a Java map structure, having the index as key and the corresponding document as the value. This implies having one index per keyword, thus the scheme uses the inverted index model.

7 Conclusions and Future Work

This study provides a secure searchable encryption protocol using smart contracts within Hyperledger Fabric. The scheme consists of two phases: index matrix creation and keyword search. The first phase happens locally, thus the time required for completion is dependent on the machine that runs the algorithm, whereas the second phase takes place remotely. The protocol achieves $\mathcal{O}(|w| \cdot |D| \cdot \mathcal{C})$ time complexity for the index computation phase and $\mathcal{O}(|w| \cdot |D|)$ for searching.

As observed in section 4, the most expensive operation with regard to the time analysis is computing the middle elements of the matrix. More specifically, it was found that the bottleneck of the system lays in the processing of the plaintexts and ciphertexts. The algorithm is more efficient when the number of files is smaller. Thus, the completion time of algorithm 1 is affected more by the number of documents than it is affected by the file sizes. When performing the buildIndex procedure for two datasets, SI and SII, having the same total size (25 MB), the algorithm was 3 times faster for the dataset that contained a smaller number of documents (SII, $|D_{SII}| = 10$). The search algorithm was slower for the dataset that had a greater number of document (SI, $|D_{SI}| = 50$), despite the fact that documents in SI had a smaller file size ($size_{SI} = 500KB, size_{SII} = 5MB$).
For decreasing the duration of the buildIndex phase, one possible solution would be altering the middle matrix computation method. A different encryption algorithm can be used such that it avoids processing the document word by word. However, this might lead to another impediment: not being able to keep track of the keyword frequency in each document.
The search phase has polynomial time complexity in the worst case scenario due to the pre-processing step, i.e., algorithm 1. However, the current implementation uses symmetric key, therefore it does not support multiple users, as specified in section 6. Another key encryption scheme can be used in order to achieve the multi-writer/multi-reader architecture.

The protocol proposed in this study can be easily extended to include additional features. As specified in section 6, the inverted index structure is used for storing the index-keyword dictionary. However, it currently supports one document for each keyword. In order to support multiple-keyword documents, the implementation can be modified to map each keyword to multiple documents.
The main use case of the the protocol is to provide searchable indexes over an encrypted set of documents. Nonetheless, this scenario can be modified to support encryption over other data types. This can be done by modifying the asset definition provided in the smart contract section 3.1. For instance, the content of the document can be sent to a file storage system, e.g., The InterPlanetary File System (IPFS). Then, the content field of the Document class can be replaced by the address at which the document is stored in IPFS. The matrix class can be modified as well in order to provide additional functionality. One suggestion would be to use a more efficient data structure instead of the two-dimensional index array I. For decreasing the search duration furthermore, the location of the hashed keywords in the matrix can be tracked in order to avoid looping over the matrix.

References

[1] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, aug 2014. [Online]. Available: https://doi.org/10.1145/2636328

[2] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," pp. 917–922, 2012. [Online]. Available: https://doi.org/10.1109/ICC.2012.6364125

[3] Q. Zheng, S. Xu, and G. Ateniese, "Vabks: Verifiable attribute-based keyword search over outsourced encrypted data," pp. 522–530, 2014. [Online]. Available: https://doi.org/10.1109/INFOCOM.2014.6847976

[4] H. Li, T. Wang, Z. Qiao, B. Yang, Y. Gong, J. Wang, and G. Qiu, "Blockchain-based searchable encryption with efficient result verification and fair payment," *Journal of Information Security and Applications*, vol. 58, p. 102791, 2021. [Online]. Available: https://doi.org/10.1016/j.jisa.2021.102791

[5] L. Chen, W.-K. Lee, C.-C. Chang, K.-K. R. Choo, and N. Zhang, "Blockchain based searchable encryption for electronic health record sharing," *Future Generation Computer Systems*, vol. 95, pp. 420–429, 2019. [Online]. Available: https://doi.org/10.1016/j.future.2019.01.018

[6] Y. Chen, L. Meng, H. Zhou, and G. Xue, "A blockchain-based medical data sharing mechanism with attribute-based access control and privacy protection," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–12, 07 2021. [Online]. Available: https://doi.org/10.1155/2021/6685762

[7] X. Qin, Y. Huang, Z. Yang, and X. Li, "An access control scheme with fine-grained time constrained attributes based on smart contract and trapdoor," pp. 249–253, 04 2019. [Online]. Available: https://doi.org/10.1109/ICT.2019.8798859

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[9] "Blockchain lab - whitepapers." [Online]. Available: https://blockchainlab.com/whitepaper

[10] N. Szabo, "Smart contracts: Building blocks for digital markets." [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[11] "Linux foundation unites industry leaders to advance blockchain technology," *The Linux Foundation*, 2015.

[12] H. Li, H. Tian, F. Zhang, and J. He, "Blockchain-based searchable symmetric encryption scheme," *Computers Electrical Engineering*, vol. 73, pp. 32–45, 2019. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2018.10.015

[13] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, pp. 895–934, 01 2011. [Online]. Available: https://doi.org/10.1145/1180405.1180417

[14] Y. Watanabe, T. Nakai, K. Ohara, T. Nojima, Y. Liu, M. Iwamoto, and K. Ohta, "How to make a secure index for searchable symmetric encryption, revisited," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 948, 2021.

[15] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," p. 965–976, 2012. [Online]. Available: https://doi.org/10.1145/2382196.2382298

[16] J. Zhang, C. Song, Z. Wang, T. Yang, and W. Ma, "Efficient and provable security searchable asymmetric encryption in the cloud," *IEEE Access*, vol. 6, pp. 68 384–68 393, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2872743

[17] K. Arnold, J. Gosling, and D. Holmes, "The java programming language," 2005.

[18] R. Griesemer, R. Pike, and K. Thompson, "Hyperledger fabric. open, proven, enterprise-grade dlt." [Online]. Available: https://talks.golang.org/2012/splash.miscf

[19] "Developing smart contracts with ibm blockchain platform developer tools." [Online]. Available: https://cloud.ibm.com/docs/blockchain?topic=blockchain-develop-vscode

[20] "Application gateway java." [Online]. Available: https://github.com/hyperledger/fabric-samples/tree/main/asset-transfer-basic/application-gateway-java

[21] "Using the fabric test network." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/test_network.html

[22] P. A. Song DX, Wagner D, "Practical techniques for searches on encrypted data," *IEEE symposium on security and privacy*, pp. 14–17, 2000.

[23] J. Su, L. Zhang, and Y. Mu, "BA-RMKABSE: Blockchain-aided ranked multi-keyword attribute-based searchable encryption with hiding policy for smart health system," *Future Generation Computer Systems*, vol. 132, pp. 299–309, 2022. [Online]. Available: https://doi.org/10.1016/j.future.2022.01.021

[24] V. Buterin, "A next generation smart contract and decentralized application platform," 2015.

[25] S. Tahir and M. Rajarajan, "Privacy-preserving searchable encryption framework for permissioned blockchain networks," pp. 1628–1633, 2018. [Online]. Available: https://doi.org/10.1109/Cybermatics_2018.2018.00272

[26] J. Daemen and V. Rijmen, “Aes the advanced encryption standard,” *The Design of Rijndael*, vol. 26, 01 2002. [Online]. Available: https://doi.org/10.1007/978-3-662-04722-4

[27] R. Sobti and G. Ganesan, “Cryptographic hash functions: A review,” *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, vol. Vol 9, pp. 461 – 479, 03 2012.

[28] N. Mineta, C. Shavers, U. Technology, and R. Kammer, “The keyed-hash message authentication code (HMAC),” 02 2001.

[29] K. Moriarty, B. Kaliski, and A. Rusch, “PKCS #5: Password-based cryptography specification version 2.1,” RFC 8018, Tech. Rep. 8018, jan 2017. [Online]. Available: https://doi.org/10.17487/RFC8018

[30] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[31] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55

[32] J. Dinneen and B. Nguyen, “How big are peoples’ computer files? file size distributions among user-managed collections,” 07 2021.

[33] “Searchable encryption using hyperledger fabric smart contracts.” [Online]. Available: https://github.com/b-malina/SE-HLF

[34] “Searchable encryption using hyperledger fabric smart contracts.” [Online]. Available: https://github.com/b-malina/application-gateway-java