Using Self-Encryption to safeguard data security in Fabric's smart contract

Chaiwon Park[1]
Supervisor(s): Dr.Kaitai Liang1[1]
[1]EEMCS, Delft University of Technology, The Netherlands
c.park-3@student.tudelft.nl, kaitai.liang@tudelft.nl

Abstract

A rise in the use of blockchain systems implies an increase in smart contract usage. Blockchain systems can easily be found these days, most notably in cryptocurrencies and financial management. Furthermore, the Internet of Things (IoT) is being developed with blockchain, since blockchain ensures trust in IoT data and adds great flexibility. However, as with any other system, an increase in usage means the security of the blockchain system is becoming a more important aspect and an interesting area of research. Additionally, several pieces of research suggest that the smart contract is the most vulnerable component of the blockchain. In this paper, a simple prototype for a blockchain-based image encryption scheme for the Internet of Things will be implemented. This prototype will be implemented using Hyperledger Fabric Smart Contract and Self-Encryption scheme. Results will propose that the encryption method implemented in the smart contract is secure enough against any other malicious users.

1 Introduction

1.1 Motivation

Security has always been an important factor to be considered in any industry, but especially web security as the number of web users increased enormously over time. The blockchain is regarded by some as the most revolutionary technological invention since the dawn of the internet; the foundation of' Web3.0', then to proceed in the future of the internet [1]. However, with the increase of decentralised applications running on blockchains, security is becoming a more important aspect. Merging this idea with the IoT, the core idea of the blockchain system, its decentralised nature makes it ideal for IoT. In particular, usage of smart contracts for IoT device storage and provisioning. The blockchain system can also provide transparency, traceability and security of the data that is transmitted within IoT networks. The role of smart contracts in IoT systems is to achieve the following goals:

- Share data more securely across stakeholders
- Verify identification and authentication
- Reduce costs by eliminating the intermediaries

Hyperledger Fabric is one of the most popular permissioned blockchain architecture of any other blockchain framework. What makes the Hyperledger Fabric popular is that the peers can easily manage the transactions on the ledger, therefore it is faster than any other permissioned blockchains.

By Benhamouda et al. [2], putting private data on the ledger comes with an inherent dilemma. How can the data saved on the ledger be safe if everyone can see the same ledger? A common solution in many systems is to only encryption of the private data while keeping the data itself under the control of the party that owns it.
From the security perspective, analysis on four interconnected components of Hyperledger Fabric is done; the consensus, the chaincode, the network and its privacy preservation mechanisms, in possible attacks and leakage of data.

In this paper, specifically, the security of the smart contract will be discussed. According to Brotsis et al. [3], smart contracts are prone to code errors and inconspicuous vulnerabilities, while their accuracy and security can be violated by malicious programmers by means of exploits. The possible risks that derive from the platform's features or misunderstanding of the common practices can lead to inconsistencies to the peers' ledger:

- *Range query risks*
  Queries methods to access the Fabric's state databases and obtain private data are not executed again in the validation phase and can lead to phantom reads [1], in which the dirty data can not be detected.
- *Chaincode sandboxing*
  Hyperledger Fabric's chaincode is executed in an isolated Docker container and provides sufficient privileges. However, a malicious user could exploit vulnerable peers, install malicious software and execute attacks.
- *Log injection*
  Any corruption of the log messages possibly averts them from being executed automatically and allow the attacker to view the processed logs.

[1]A Phantom read occurs when one user is repeating a read operation on the same records but has new records in the results set.

Figure 1 shows the number of attacks to different components of the blockchain.

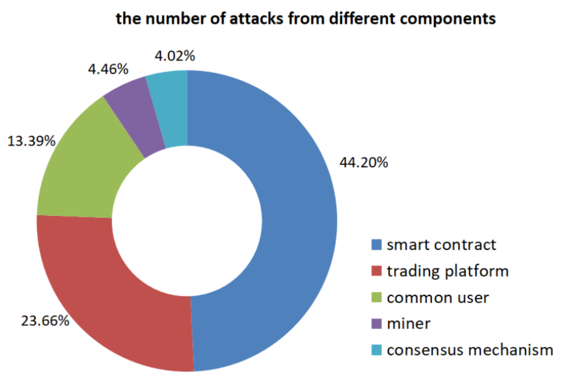


Figure 1: The number of attacks to different components [4]

1.2 Research Questions

A smart contract takes the largest proportion of the number of attacks, which shows that a smart contract is vulnerable. Therefore in this project, the vulnerability of the smart contract will be solved by combining Self-Encryption and Hyperledger Fabric Smart Contract. To answer the main question, "How can the Hyperledger Fabric Smart Contract be improved by Self-Encryption?". The following sub-questions have to be answered:

- What is Hyperledger Fabric Smart Contract?
- What is the definition of the Self-Encryption?
- How can Self-Encryption enhance the security of the Smart Contract?

Since a blockchain system and privacy of the data is of great interest for the researchers, there are existing research done.

1.3 Approach

W. Khan and Y. Byun [5] proposed a permissioned private blockchain-based result to secure the image encryption. In this scheme, the cryptographic pixel values of an image are stored on the blockchain, assuring the privacy and security of the image data. Encrypted results show that the proposed scheme is largely effective for data leakage averting and security.

Zheng et al. [6] proposed attribute-based encryption with outsourced decryption in the blockchain. The article proposed the combination of blockchain and IPFS, also the usage of the outsourced decryption, therefore decryption does not happen within the smart contract.

Based on the current research and works done, the prototype of a file upload system that could be used in an IoT system to securely transfer the data will be implemented via merging self-encryption and Hyperledger Fabric's smart contract, combined with IPFS.

In Section 2, the background of how blockchain works in general and more specifically permissioned blockchain technology, Hyperledger Fabric framework. Additionally, details about the encryption method used in this research, namely self-encryption and an explanation on how the chosen self-encryption library works. Section 3, contains a description of the research methodology, explaining what will be done to answer the research questions and why the following methods are chosen. Section 4 contains an exposition of the main ideas; the development of a theory, the analysis of the problem and some theoretical analysis. In Section 5, the results of the implemented tool will be discussed. Section 6 contains the reflection on the ethical aspects of this research and a discussion of the reproducibility of your methods. In Section 7, a summary of the main research question and the answers to the questions can be found. Furthermore, discussion of open issues, improvements and new questions that arise from this work.

2 Background

2.1 Blockchain and Hyperledger Fabric

A blockchain is a distributed database that is shared among the nodes of a computer network. As a database, a blockchain stores information electronically in digital format [7].

The most widely recognised application of blockchain is the Bitcoin cryptocurrency. Bitcoin is classified as public permission-less blockchain technology, which means the network is public and participants interact anonymously. However, for enterprise use, more requirements need to be considered, most importantly participants must be identified/identifiable and networks need permission. Such problems could be solved using Hyperledger Fabric.

Blockchain technology, especially Hyperledger Fabric consists of three components; a distributed ledger, consensus algorithm and smart contracts. Hyperledger Fabric is an open-source enterprise-grade permissioned distributed ledger technology platform [8]. The fabric has a highly modular and configurable architecture, also it supports pluggable consensus protocols which enable the platform to be more effectively customised to fit particular use cases and trust models.

The key elements of the blockchain network/Hyperledger Fabric framework are as follows:

- *Peer*
  Peer is a fundamental element of the network. A blockchain network is composed of peer nodes, each can hold copies of ledgers and copies of smart contracts. A peer can be created, started, stopped, reconfigured and even deleted. As it is a host for ledgers and chaincodes, applications and administrators must interact with a peer if they want to access ledgers and chaincodes. These communications between applications and peers are completed via channels, which is a mechanism by which a set of components within a blockchain network can communicate and transact privately [9].

- *Ledger*
  Specifically, in Hyperledger Fabric, a ledger is comprised of two distinct parts:

  – World State is a database that saves current values of a set of ledger states. It can be modified frequently as states can be created, updated and deleted.

  - Blockchain, all the changes in the current world state are recorded on a transaction log. However, it is different from the world state as it is immutable. Therefore, blockchain determines the world state [10].

- *Orderer*
  As Hyperledger Fabric is permissioned unlike many other blockchains, a node is called an orderer feature in the Hyperledger Fabric. An orderer does transaction ordering, along with other orderer nodes forming an ordering service so that any block validated by the peer is guaranteed to be final and correct. Additionally, it also maintains the list of organisations that are allowed to create channels [11].

- *Smart Contracts and Chaincode*
  The most important element of the Hyperledger Fabric in this project, Smart Contract/Chaincode. Usually, smart contracts and chaincode refer to the same thing, except that a chaincode is typically used by administrators to group related smart contracts for deployment. A smart contract defines the transaction logic that controls the lifecycle of a business object contained in the world state, which generates new facts that are added to the ledger [12].

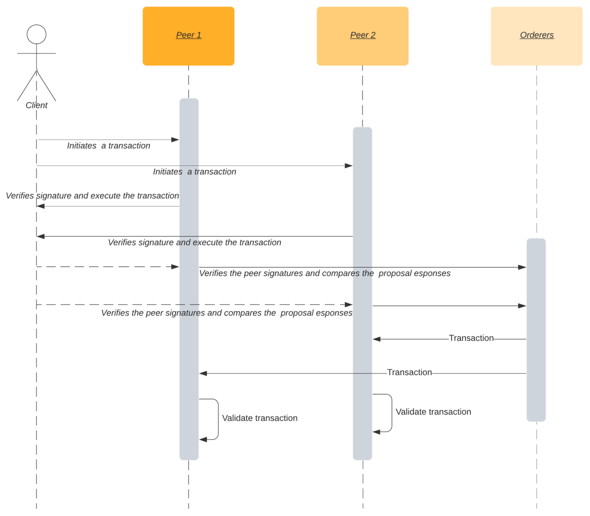

Peer 1
Peer 2
Orderers
Client
Initiates a transaction
Initiates a transaction
Verifies signature and execute the transaction
Verifies signature and execute the transaction
Verifies the peer signatures and compares the proposal esponses
Verifies the peer signatures and compares the proposal esponses
Transaction
Transaction
Validate transaction
Validate transaction

Figure 2: Simple Hyperledger Fabric transaction sequence diagram

Figure 2 shows the transaction flow of a standard asset change. As both peers must endorse any transaction, the client requests transaction for both peer 1 and peer 2. Then, each peer verifies the proposed transaction, then the chaincode is executed against the current state database and propose the response. After that, the response proposed from the previous step is inspected and verifies the peer signatures. Finally, the peers receive verified transactions, create blocks of transactions per channel. Once these transactions are validated and committed, the ledger will be updated.

2.2 Self-Encryption

Self-Encryption is a version of convergent encryption with an additional obfuscation step. It is a unique encryption method, as it has no separate keys unlike any other encryption method, where encryption methods usually have a separate public key and private key. It uses its file as the key to encrypt. The Self-Encryption library by the Maidsafe [13] is used in this research.

The following figure 3 briefly shows the encryption steps. First, the file is split into a minimum of three chunks, the number of chunks increases as the file size increases. Then each file chunk will be encrypted using AES 256. Lastly, file chunks are obfuscated via XOR operation between encrypted chunk and randomly chosen hash value of another chunk, i.e. excluding its hash value.

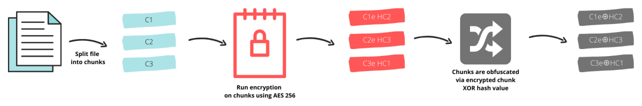
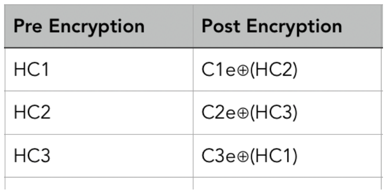

Split file into chunks
C1
C2
C3
Run encryption on chunks using AES 256
C1e HC2
C2e HC3
C3e HC1
Chunks are obfuscated via encrypted chunk XOR hash value
C1e⊕HC2
C2e⊕HC3
C3e⊕HC1

Figure 3: Encryption Steps

A more detailed overview of how the encryption method works is as follow [14]:

1. Split a file into several chunks, C_n.
2. Take the hash of each chunk, H_{c_n}.
3. Use [key size] CC_{n-1} as the key, use [next bytes iv size] C_{n-1} as the Initialisation Vector (IV).
4. Create obfuscation chunk, $OBFC_n$ by concatenating the hashes of other chunks, C_n, [unused part of] C_{n-1} and C_{n-2}.
5. Run encryption cypher on chunks C_n, to produce random chunks, C_{random}.
6. Data is considered to be randomised and of the same length as input data.
7. Obfuscation chunk, $OBFC_n$ is also random output, but of length less than the input data.
8. Now take $OBFC_n$(repeated) XOR C_{random} to produce output data.
9. Rename each with the hash of the new content and save these hashes.

Data map is the most important concept of this self-encryption scheme. The data map is used to decrypt the data. The data map contains the information about the file chunks, C_n. The following table is an example of the structure of the data map [14]:



Pre Encryption	Post Encryption
HC1	C1e⊕(HC2)
HC2	C2e⊕(HC3)
HC3	C3e⊕(HC1)

Figure 4: Data map

The names of all the encrypted chunks are in the right column and all keys and IVs are stored in the left column. The file hash, $fh = H(H(\mathrm{C}_1)$ + H(C_2) + ... H(C_{n-1}))^5 identifies the data element and acts as the unique key for this file. Thus, in the process of decryption, a data map is used as a lookup table to check which one of the encrypted chunks was encrypted with the hash value, which is the unique key. It is necessary to decrypt the chunk and merge them back to the normal file format.

The pseudocode below briefly describes how the self-encryption library works.

```
procedure ENCRYPTION($file\_in\_bytes$)
    $hashes \leftarrow HashFunction(file\_in\_bytes)$
    $pki \leftarrow get\_pad\_key\_and\_iv(hash\_of\_chunks)$
    $compressed\_file \leftarrow brotli\_compression(file\_in\_bytes)$
    $encrypted\_content \leftarrow AES(compressed\_file)$
```

In the encryption step, symmetric encryption, namely AES in this library is used. The encryption key (or password) and the Initialisation Vector (IV) are required to perform encryption. In hashing step, file chunks and hash function, namely SHA is used. In this implementation, it is assumed that a sufficiently secure algorithm is acting as a hash function which means that there are little or no collisions. In cryptography, secure hashing is an important aspect as the more secure hashing is the more infeasible to generate or modify a plaintext.

3 Methodology

At the starting point of the research project, thorough research on the tools that are going to be used to implement a merged system of self-encryption and Hyperledger Fabric smart contract was done. Three preeminent tools will be used for the implementation, Hyperledger Fabric and Self-Encryption Library [15] and InterPlanetary File System (IPFS) [2].

In this implementation, the test-network sample v2.2.2 from the fabric-samples [16] repository will be used. Within the test network, Docker [3] v19.03.8 is being used for the interaction between each node and user with a Fabric network. Additionally, smart contracts can be written in Go, JavaScript or TypeScript. JavaScript will be used to write the smart contract in this project, due to high compatibility with the Hyperledger fabric's smart contract and IPFS.

[2]https://ipfs.io

[3]https://www.docker.com

The Self-Encryption Library will be outsourced, in a way that the user has to encryption and decrypt outside the test network, due to a technical problem running the library on the Docker container.

Furthermore, InterPlanetary File System (IPFS) will be used to store the data map for security purposes. The function of uploading the file to the IPFS will be implemented in the smart contract. IPFS is chosen to store the data map since the data map needs to be stored off-chain and should not be accessible other than the validated user.
When a file is added to IPFS, the file is split into smaller chunks, cryptographically hashed and given a unique fingerprint called a content identifier (CID) [17].

4 Secure Enhancement for Smart Contract using Self-Encryption

This section contains a description of the methods/algorithms. The file the user wishes to upload on the ledger will be encrypted so that it is not accessible on the ledger and the encrypted data itself is meaningless. Figure 5 shows how users use the self-encryption library, interact with the smart contract and IPFS.

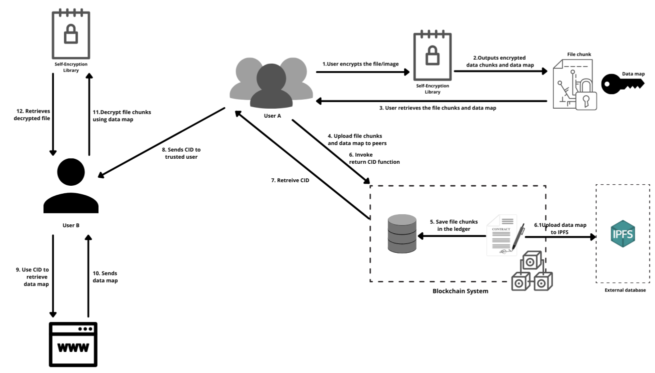

Self-Encryption Library
12. Retrieves decrypted file
11.Decrypt file chunks using data map
User A
1.User encrypts the file/image
Self-Encryption Library
2.Outputs encrypted data chunks and data map
File chunk
Data map
3. User retrieves the file chunks and data map
8. Sends CID to trusted user
4. Upload file chunks and data map to peers
6. Invoke return CID function
7. Retreive CID
User B
5. Save file chunks in the ledger
6.1Upload data map to IPFS
IPFS
9. Use CID to retrieve data map
10. Sends data map
Blockchain System
External database
www

Figure 5: Prototype diagram

In this implementation, there are two parties involved. One who uploads the file and another one who needs the file, thus need to download or retrieve the file from this application. First, the party who uploads the file, *user A* encrypts the file using the self-encryption library on the local machine. Upload both file chunks and data map to the system. Then, *user A* executes a query to initialise the ledger, which saves *only file chunks* to the ledger. Once more *user A* executes a query to upload the data map to the IPFS, which will return the CID. Lastly, *user A* provides the CID to the trusted party, *user B*. *User B* uses this CID to retrieve the data map, then uses the same application to retrieve file chunks. Finally, *user B* can decrypt the file.

4.1 Encryption Method

It was not feasible to run the Rust program on the Hyperledger Fabric test network, which is running on the Docker. Therefore, it is required for the user to encrypt the file within the local machine using the Self-Encryption library provided.

Then, the file chunks and data map should be uploaded to the Docker containers. Details on the possible improvement will be mentioned in section 6.

4.2 Implementation

The implementation extends the *asset-transfer-basic/chaincode-javascript* [16], provided from the Hyperledger Fabric and test network. The implemented prototype can be found from Github [4], under directory /self-encryption-chaincode/lib/assetTransfer.js.

The basic workflow of this whole system is:

1. Bring up the test network and create the channel
2. Package the smart contract
3. Install the chaincode package
4. Approve a chaincode definition
5. Commit the chaincode definition to the channel
6. Invoke the chaincode

Figure 6 presents the detailed workflow of the implemented prototype between the user, application, peers, orderer, smart contract, ledger and external database (IPFS).

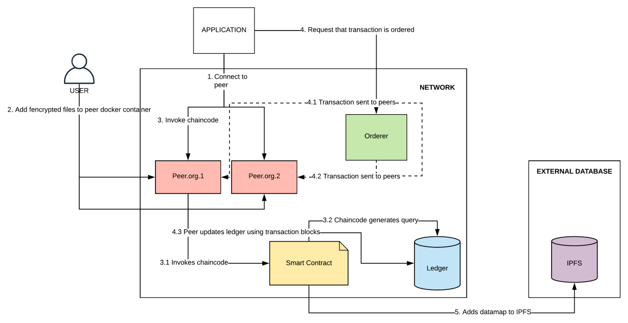

APPLICATION
4. Request that transaction is ordered
USER
1. Connect to peer
NETWORK
2. Add fencrypted files to peer docker container
4.1 Transaction sent to peers
3. Invoke chaincode
Orderer
Peer.org.1
Peer.org.2
4.2 Transaction sent to peers
EXTERNAL DATABASE
3.2 Chaincode generates query
4.3 Peer updates ledger using transaction blocks
3.1 Invokes chaincode
Smart Contract
Ledger
IPFS
5. Adds datamap to IPFS

Figure 6: Workflow diagram

The user has to upload the encrypted files and data map to the peer Docker container due to technical problems. More details will be discussed in section 6. Other processes are all discussed in section 2.

Details on functionalities implemented in the smart contract will be explained. There are two main functionalities within this smart contract. First, initialise the ledger and add the file/data into the ledger. Refer to code listing 1, function `getFileContents` in line 12-30. It retrieves the file chunks and read the contents as the buffer byte array. Then, it is saved as a `File` in one of the assets and saved in the ledger, excluding the data map. A user can query the ledger on the command line to get the list of all the files added to the channel ledger, using this command `peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'`. This is how the file chunks are saved in the ledger; `"File":"[{"filename":"3d20b45c49191b7fe0db46cbcdb3f00f12506e6ec7289de9c71955bcfa88c3d0","content":{"type":"Buffer","data":[35,251,91,206,80,74,192,208,246,6,150,27,163,115, ... ]}}]`. There are a few possible ways to handle the data map:

- Save the data map in the ledger
- Save the data map in the IPFS

Referring to code listing 2, `PutDataMapToIPFS` function puts the data map in the IPFS server. Line 4, the package `ipfs-http-client` is needed to connect to the running IPFS node. Then, line 5, actually connects to the daemon API server, which is saved as an IPFS object. Line 11-15 execute commands on the ipfs object. This function can be called with the command `peer chaincode query -C mychannel -n basic -c '{"Args":["PutDataMapToIPFS"]}'`. This query command returns `(base)chaiwonpark@Chaiwons-MacBook-Pro test-network %peer chaincode query -C mychannel -n basic -c '{"Args":["PutDataMapToIPFS"]}'QmeHdzTp8Ebef8MJuqKTj1LmpDwCuBq77NsjsLx3PdZhi5`. This function returns the hash value which allows the user to retrieve the data map via the link https://gateway.ipfs.io/ipfs/{CID}. Then, the data map will be automatically downloaded to a user's local machine.

5 Results and Analysis

5.1 Results and Findings

When the file chunks and data map are stored in the ledger, the plain text, which is saved in buffer byte array will be absurd as shown in Subsection 4.2. The testing of this prototype was conducted using different types of data set, JPEG, plain text, Docx file and JSON file. Additionally, data sets are real-world examples to produce more realistic test outputs for future works and compatibility. The tests were taken on MacBook Pro 2020, 1,4 GHz Quad-Core Intel Core i5 with 16 GB memory on macOS version 11.6.1. The results of the performance analysis may differ in different hardware.

Non-encrypted file

JPEG file

The normal file will be saved as buffer byte array just like encrypted file. The byte array itself may be absurd. However, using `fs.writeFileSync("filename.jpg",byte_array)` (JavaScript function) will return the image file. For example, byte array `<Buffer ff d8ff e00010 4a 4649 46 00 01 0 1 01 01 2c 012c 0000 ff e10f f545 78 69 66 00 00 49 49 2a 0008 00 00 00 02 00 32 01 02 00 14 00 00 00 26 0 0 ...76916more bytes>` will return the example images used.

JSON file

A not-encrypted JSON file is saved in ledger as buffer array. When command `peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'` is executed, the result `,"File":[{"content":{"data":[91,10,32,32,123,10,32,32,34,110,97,109,101,34,58,34,66,111,98,34,44,10,32,32,34,100,101,118,105,99,101,34,58,34,103,111,111,103,108,101,32,104,...],"type":"Buffer"},"filename":"example_data"}]` will be returned. It is saved as buffer array just like JPEG file. However, using

[4]https://github.com/chaiwon59/self-encryption-fabric

such function of JavaScript `{bufferArray}.toString()` will return `[{"name":"Bob","device":"google home","date":"2022-01-18T13:45:00.000Z","command":"Turn on spotify"},...]`. Therefore, the saved data might not be coherent, but there is way to produce data which user uploaded.

Docx file

A non-encrypted Docx file is saved in the ledger as a buffer array just like any other data type, thus not comprehensible just looking at the data saved in the ledger. However, using such JavaScript function, `fs.writeFileSync("path/to/destination",buffer_array_saved_in_ledger)` returns the exact Docx file uploaded.

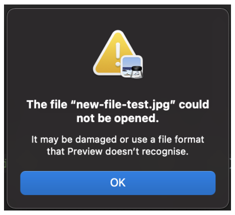
Plaintext

A non-encrypted plaintext is saved in the ledger as a buffer array too. However, equally, to the Docx file, an exact plaintext file could be returned using the existing JavaScript function.

As discussed, the non-encrypted file does not achieve data confidentiality or secure data sharing.

Encrypted file with the data map stored in the ledger

Both file chunks and data map are saved as buffer byte arrays. Converting the file chunks and data map using the JavaScript function returns 7.



The file "new-file-test.jpg" could not be opened.

It may be damaged or use a file format that Preview doesn't recognise.

OK

Figure 7: Error converting file

Thus, a malicious user can not retrieve any information from the data saved in the ledger. However, if the malicious user is aware of the self-encryption tool, decryption is not a problem since all the necessary data can be retrieved from the ledger. Hence, this method achieves security to a certain degree but is not ideal. This applies to all the other data sets/types used for testing in this paper.

Encrypted file with the data map stored in the IPFS

As previous results, files will be saved as a buffer byte array. However, the data map will be uploaded to the IPFS as mentioned in section 3. Once the file sender uploads the encrypted file and executes the `PutDataMapToIPFS` command, CID will be returned. Then this CID can be handed over to the user who wants to open the file, that user can access the data map using this CID which enables decryption of the file.

For analytical purposes, the performance of the smart contract with encrypted files and without encrypted files will be compared. The performance analysis will be completed under two categories; execution time and the size of the files.

Figure 8 shows the comparison of the execution time of ledger initialisation between encrypted files and non-encrypted files. The difference in execution in time is almost twice longer for the encrypted file, the main reason being the uploading of the data map to IPFS.

5.2 Analysis

Security Analysis

The practical side of the security analysis was done in the previous results section, when a malicious user has access to the ledger, how could the saved data be manipulated. Additionally, uploading the data map to the Docker container does not abate security. Since only trusted users should be allowed to control the Docker daemon, which allows the user to share a directory between the Docker host and a guest container only under consent or if required [18]. Thus, Docker containers are secure by default. In this section, more theoretical proof of the security of self-encryption will be discussed.

The self-encryption scheme is reduced to the discrete logarithm problem in chosen-ciphertext attacks. If there exists an adversary that can break the chosen-ciphertext attacks security of the self-encryption scheme, then the challenger can make use of the adversary to solve the discrete algorithm problem, which is assumed to be hard. Thus the existence of such an adversary is not possible. Mathematical and formal proof can be found in [19].

Performance Analysis

Conducive to conduct accurate and realistic results, a large JSON file is used at the same time as simple test data set. A JSON file is chosen as it is a common data type that needs to be handled in the IoT system.

Simple data set

Figure 8 shows the comparison of the execution time of the command to initialise ledger and to upload data map to IPFS (if necessary) between encrypted file and non-encrypted file. Uploading of the encrypted file takes longer than non-encrypted file due to the key management, which is uploading data map to the IPFS in this case. This result is inevitable as IPFS adds another layer of the process which leads to longer execution time even though uploading to IPFS takes a few hundred milliseconds.

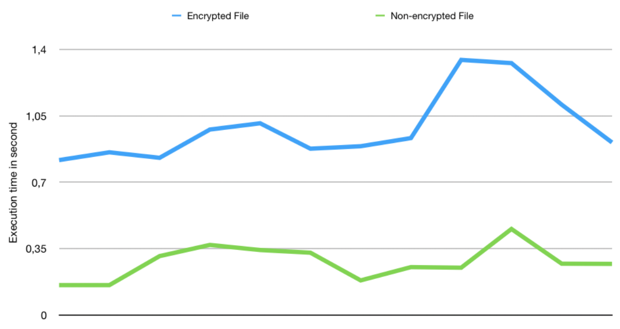


Figure 8: Execution time comparison (simple data set)

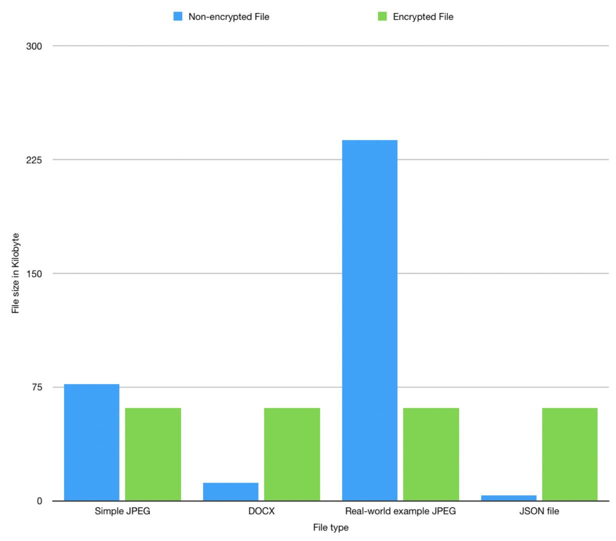


Figure 9: File size comparison (simple data set)

Figure 9 shows the file size comparison between non-encrypted files and encrypted files with the different file formats. The size of the encrypted files is all the same no matter what the size of the original (non-encrypted files) are, due to the Brotli [5] compression method implemented in the self-encryption (more information can be found in 2.2. However, extremely small files need to be split into chunks and data map, therefore the size may be larger.

Real-world example: JSON file [20]

In this scenario, the result was the complete opposite of the previous case. The execution time of the encrypted files was enormously smaller, almost 1/10 of the non-encrypted files. The reason is, self-encryption substantially decreases the file size as shown in figure 11 due to the compression. This considerable difference in file size makes the time of uploading data map to IPFS trivial and initialising ledger more significant factor. As the time to initialise the ledger and the file size are proportional to each other, non-encrypted file takes longer.

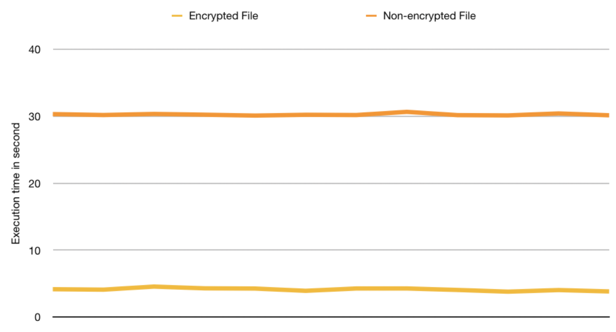


Figure 10: Execution time comparison (large JSON file)

[5]Brotli is a compression algorithm developed by Google and is primarily used by web servers and content delivery networks.

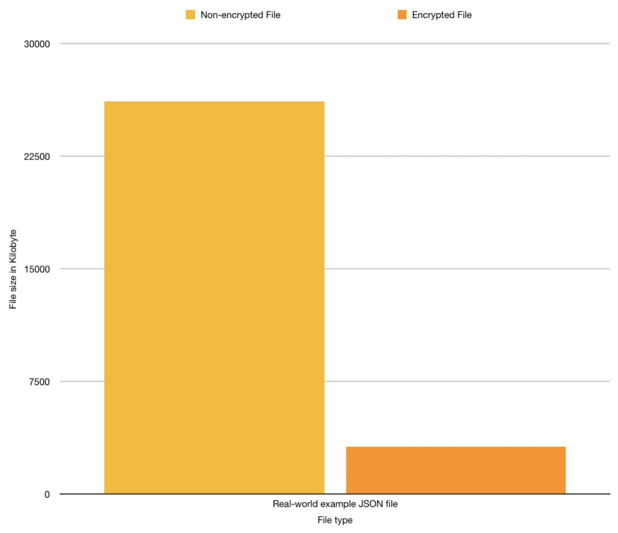


Figure 11: File size comparison (large JSON file)

The file size was an important aspect in this performance analysis since the execution time is proportional to the file size. This led to an interesting result that non-encrypted file needs a longer execution time to successfully conduct the function or query executed by the user. However, this is only the case when using an extremely large data set. The difference in execution time between simple example data sets and real-world examples was huge. In using encrypted scenarios, a real-world example was almost five times more than the simple data set. In using non-encrypted, the difference was even larger, where the real-world example was approximately a hundred times slower.

In this paper, simple performance analyses, such as execution time and file size measurements were taken place. However, different tests such as latency, error rates and throughput could have been conducted as well [21] [22].

6 Responsible Research

This section includes two subsections; Research Integrity and Reproducibility. The research Integrity part describes how the research was conducted in a way that this paper includes incorporates trusted methods and the findings in research. The reproducibility section describes how the implemented prototype is reproducible.

6.1 Research Integrity

A number of journals, websites and research papers were thoroughly studied and inspired this research. All resources were referenced correctly and cited on the necessary parts. This research paper contains the handling of the data set. All the data sets used for testing the implemented prototype is either self-made or the original author was cited. These test data sets do not include any personal or confidential information. Any quotation or rephrased paragraphs are referenced therefore readers can trace back to the sources. Additionally, since the implementation is based on source code, the source is properly cited on the Github README file.

6.2 Reproducibility

The implementation of this research is based on the sample network from Hyperledger Fabric [16], which applies to the Hypereldger Fabric v 2.2.2. The code is reproducible if the source code is properly cited and referenced. As mentioned above, the completed implementation can be found in the Github repository.

7 Conclusions and Future Work

7.1 Conclusions

In this research project, one of the encryption methods, specifically self-encryption was merged into the smart contract to implement the prototype of uploading a file to the ledger ensuring data confidentiality and security. Self-encryption enhances the security of the data in the Fabric's smart contract as the results discussed in section 5. This prototype could be extended to implement or create a blockchain application where external files are commonly used, where the security of the files is essential.

7.2 Future Work

Due to the limited time constraint, there are several possible improvements that could have been done better.

First of all, the encryption method itself could have been implemented in the smart contract. The following methods to import this library into the smart contract then encrypt the file within the test network were attempted:

- Convert the Self-Encryption Rust library into Node.js [6] library
- Running the smart contract as an external service as provided from the Hyperledger Fabric to make the Docker Rust runnable environment.
- Build separate Docker container for the Self-Encryption library, which will enable Docker containers to communicate through a network connection
- Build the Docker containers for encryption library and smart contract Docker

However, the attempted methods caused errors and could not be fixed within the given time constraint. It is not a big problem for the core functionality of the smart contract but it increases the usability for the application user.

Additionally, CID management could be improved via several different methods. Currently, the data map is handled via putting it in the IPFS. Although it ensures the security of the data map, the CID needs to be given to the verified user who wishes to retrieve the data map. The CID may be leaked in this process. Encrypting the data map within the smart contract using symmetric encryption methods such as AES, RSA, etc. However, this will add another aspect to manage as the way to transfer the decryption key needs to be decided.

Finally, more advanced encryption, namely attribute-based encryption [23] [24] could be used to securely escrow the CID to another user.

[6] https://nodejs.org/en/

References

[1] "History of blockchain," Jun. 2021. [Online]. Available: https://www.tradefinanceglobal.com/blockchain/history-of-blockchain/

[2] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on hyperledger fabric with secure multiparty computation," *IBM J. Res. Dev.*, vol. 63, no. 2/3, pp. 3:1–3:8, Mar. 2019.

[3] S. Brotsis, N. Kolokotronis, K. Limniotis, G. Bendiab, and S. Shiaeles, "On the security and privacy of hyperledger fabric: Challenges and open issues," in *2020 IEEE World Congress on Services (SERVICES)*. IEEE, Oct. 2020.

[4] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7, pp. 150 184–150 202, 2019.

[5] P. W. Khan and Y. Byun, "A blockchain-based secure image encryption scheme for the industrial internet of things," *Entropy (Basel)*, vol. 22, no. 2, p. 175, Feb. 2020.

[6] H. Zheng, J. Shao, and G. Wei, "Attribute-based encryption with outsourced decryption in blockchain," *Peer Peer Netw. Appl.*, vol. 13, no. 5, pp. 1643–1655, Sep. 2020.

[7] A. Hayes, "Blockchain explained," Investopedia. [Online]. Available: https://www.investopedia.com/terms/b/blockchain.asp

[8] "Introduction¶." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html

[9] "Peers¶." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/peers/peers.html

[10] "Ledger¶." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html

[11] "The ordering service¶." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html

[12] "Smart contracts and chaincode¶." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/smartcontract/smartcontract.html

[13] "Providing privacy, security and freedom." [Online]. Available: https://maidsafe.net/

[14] D. Irvine, "Self encrypting data - maidsafe," 2015. [Online]. Available: https://docs.maidsafe.net/Whitepapers/pdf/SelfEncryptingData.pdf

[15] Maidsafe, "Maidsafe/self_encryption: File self encryptor," GitHub. [Online]. Available: https://github.com/maidsafe/self_encryption

[16] Hyperledger, "Hyperledger/fabric-samples," GitHub. [Online]. Available: https://github.com/hyperledger/fabric-sample

[17] "Ipfs powers the distributed web," https://ipfs.io/#how,.

[18] "Docker security," Docker Documentation. [Online]. Available: https://docs.docker.com/engine/security/

[19] S. S. D. Selvi, A. Paul, S. Dirisala, S. Basu, and C. P. Rangan, "Sharing of encrypted files in blockchain made simpler," in *Mathematical Research for Blockchain Economy*. Cham: Springer International Publishing, 2020, pp. 45–60.

[20] "Json-iterator/test-data: Sample json file for testing," GitHub, Dec 2016. [Online]. Available: https://github.com/json-iterator/test-data

[21] Q. Nasir, I. A. Qasse, M. Abu Talib, and A. B. Nassif, "Performance analysis of hyperledger fabric platforms," *Secur. Commun. Netw.*, vol. 2018, pp. 1–14, Sep. 2018.

[22] J. Dreyer, M. Fischer, and R. Tönjes, "Performance analysis of hyperledger fabric 2.0 blockchain platform," in *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*. New York, NY, USA: ACM, Nov. 2020.

[23] S. Wang, K. Liang, J. K. Liu, J. Chen, J. Yu, and W. Xie, "Attribute-based data sharing scheme revisited in cloud computing," *IEEE trans. inf. forensics secur.*, vol. 11, no. 8, pp. 1661–1673, Aug. 2016.

[24] P. Zhang, Z. Chen, J. K. Liu, K. Liang, and H. Liu, "An efficient access control scheme with outsourcing capability and attribute update for fog computing," *Future Gener. Comput. Syst.*, vol. 78, pp. 753–762, Jan. 2018.

A Appendix

A.1 Code Listings

Listing 1 Initialising Ledger

```
const stringify  = require('json-stringify-deterministic');
const sortKeysRecursive  = require('sort-keys-recursive');
const { Contract } = require('fabric-contract-api');
const fs = require('fs');
const path = require('path');

let topDir = '/etc/hyperledger/fabric/chunk_store_test/';

let fileList = () =>
    fs.readdirSync(topDir);

function getFileContents(dir) {
    const array = [];
    fileList().forEach(filename => {

        if(!filename.includes('data_map')){
            // get current file name
            const name = path.parse(filename).name;

            const content = fs.readFileSync(dir + filename);

            let element = {};

            element.filename = name;
            element.content = content;
            array.push(element);
        }
    });
    return array;
}

class AssetTransfer extends Contract {

    async InitLedger(ctx) {

        const assets = [
            {
                ID: 'asset1',
                Color: 'blue',
                Size: 5,
                Owner: 'Tomoko',
                AppraisedValue: 300,
                File: JSON.stringify(getFileContents(topDir)),
            },
        ];

        for (const asset of assets) {
            asset.docType = 'asset';
            // example of how to write to world state deterministically
            // use convetion of alphabetic order
            // we insert data in alphabetic order using 'json-stringify-deterministic' and
                'sort-keys-recursive'
            // when retrieving data, in any lang, the order of data will be the same and
               consequently also the corresonding hash
            await ctx.stub.putState(asset.ID, Buffer.from(stringify(sortKeysRecursive(
               asset))));
        }
    }
}
```

Listing 2 Putting DataMap to IPFS

```
const { Contract } = require('fabric-contract-api');
const fs = require('fs');
const path = require('path');
const IPFS = require('ipfs-http-client');
const ipfs = new IPFS.create({ host: 'ipfs.infura.io', port: 5001, protocol: 'https'});

class AssetTransfer extends Contract {
    async PutDataMapToIPFS() {
        const getDataMap = () =>
            fs.readFileSync('/etc/hyperledger/fabric/chunk_store_test/data_map');
        const addFile = async () => {
            console.log('addFile called');
            const file = { path: 'testfile', content: Buffer.from(getDataMap())};
            const filesAdded = await ipfs.add(file);
            return filesAdded.cid.toLocaleString();
        };
        return await addFile();
    }
}
```