ID-Based Self-encryption via Hyperledger Fabric Based Smart Contract

Grishkov, Ilya; Kromes, Roland; Giannetsos, Thanassis; Liang, Kaitai

DOI
10.1007/978-3-031-31420-9_1

Publication date
2023

Document Version
Final published version

Published in
Blockchain Technology and Emerging Technologies - 2nd EAI International Conference, BlockTEA 2022, Proceedings

Citation (APA)
Grishkov, I., Kromes, R., Giannetsos, T., & Liang, K. (2023). ID-Based Self-encryption via Hyperledger Fabric Based Smart Contract. In W. Meng, & W. Li (Eds.), *Blockchain Technology and Emerging Technologies - 2nd EAI International Conference, BlockTEA 2022, Proceedings* (pp. 3-18). (Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST; Vol. 498 LNICST). Springer. https://doi.org/10.1007/978-3-031-31420-9_1

ID-Based Self-encryption via Hyperledger Fabric Based Smart Contract

Ilya Grishkov[1], Roland Kromes[1(✉)], Thanassis Giannetsos[2], and Kaitai Liang[1]

[1] Cyber Security Group, Delft University of Technology, Delft, The Netherlands
I.Grishkov-1@student.tudelft.nl, {R.G.Kromes,Kaitai.Liang}@tudelft.nl
[2] Ubitech Ltd., Digital Security and Trusted Computing Group, Athens, Greece
agiannetsos@ubitech.eu

Abstract. This paper offers a prototype of a Hyperledger Fabric-IPFS based network architecture including a smart contract based encryption scheme that meant to improve the security of user's data that is being uploaded to the distributed ledger. A new extension to the self-encryption scheme was deployed by integrating data owner's identity into the encryption process. Such integration allows to permanently preserve ownership of the original file and link it to the person/entity who originally uploaded it. Moreover, self-encryption provides strong security guarantees that decryption of a file is computationally not feasible under the condition that the encrypted file and the key are safely stored.

Keywords: Blockchain · IPFS · Self-Encryption · Security · Hyperledger Fabric

1 Introduction

The modern world is increasingly adopting blockchain technology. The first major market adoption of blockchain happened in 2009 when Bitcoin was introduced [12]. Interest in blockchain solutions grew over the years and lead to the invention of Ethereum - Bitcoin peer but with support for smart contracts which are digital codes enabling the description of complete business logic [1]. The introduction of smart contracts leads to further development in the field of blockchain and created demand for more industry-friendly solutions that allow to identify users of the system (Know-Your-Customer, Anti-Money-Laundering). Hyperledger Fabric was then introduced as a highly modular permissioned blockchain that allows great customization to suit particular industrial needs [3]. Given its customizability and modularity, Hyperledger Fabric (HLF) is a perfect platform for extending it with various trust and privacy preservation solutions.

According to Huang et al. [7] the main component of a blockchain that is being attack the most is a smart contract. High frequency of attack on a component designed to handle user private data suggests a need for an alternative approach to handling sensitive information other than just sending it raw to the

W. Meng and W. Li (Eds.): BlockTEA 2022, LNICST 498, pp. 3–18, 2023.
https://doi.org/10.1007/978-3-031-31420-9_1

ledger. An local data encryption prior to sending data to the smart contract could be a solution.

Self-encryption was introduced as a mean of encrypting files that "requires no user intervention or passwords" [9]. This algorithm can be used for local encryption of files, encrypted chunks of which will be later uploaded to a cloud-based storage or to a distributed file system (e.g., IPFS[1]) Pointers to the encrypted chunks are then sent to the ledger. It can be noted that storing only the hash values of the encrypted data chunks on blockchain ledger is vital when the data size is significant. The authors in [5] point out that sending data to a blockchain frequently, when the data size is large, can cause the entire blockchain network to crash. Sending only the hash values of the given data is more optimal as a hash value is usually 32 bytes long. While this solution allows to keep file content private, the file itself is not linked in any way to its owner. A variant of identity based encryption can tackle this problem. If a file is self-encrypted with owners identity used during the encryption process, this file remains linked to the person who initially uploaded it to the blockchain. This way original ownership can be preserved.

This paper aims at exploring trust and privacy preserving solutions in Hyperledger Fabric blockchain. More specifically the goal is to further investigate the utility of a combination of identity based encryption and self-encryption as means of improving security of the data in the HLF; extend the previously done research by [13] and implement ID-based self-encryption via Hyperledger Fabric smart contract. Hence, this work is aimed at finding a possible solution to combining ownership information with blockchain architecture. Another goal of the paper is creating a proof of concept solution for integration of ID-based self-encryption into a blockchain context in a generic way.

Within this paper an approach of integrating ID-based self-encryption is presented. Moreover a detailed description of prototype implementation is given. In addition to this implementation of ID-based self-encryption, a practical fully decentralized network architecture for storing encrypted data has also been deployed. In this proposed network, the data owner can use ID-based self-encryption to store encrypted data in a decentralized and secure manner. The encrypted data chunks are stored in an InterPlanetary File System (IPFS) which is a decentralized systems for file storage. To store the references (hash values) of the encrypted data chunks, the Hyperledger Fabric blockchain was used.

This work is structured as follows. Section 2 describes related works used to achieve the goal. Section 3 gives a background about the implementation of ID-based self-encryption. Section 4 discusses the inner workings and the proposed implantation of ID-based self-encryption. The performance analysis and the overview of benefits of the proposed implementation is presented in Sect. 5. Finally, the work is discussed in Sect. 6, and concluded in Sect. 7.

[1] https://ipfs.io/.

2 State of the Art

Blockchain is a distributed ledger technology that provides immutability and transparency of data to members of the blockchain network [19]. The blockchain is also a peer-to-peer network in which participants are known to each other. Authentication of participants is ensured using elliptic curve cryptography. Today's blockchains enable the deployment of complete business logic in a seamless manner. These digital business logic are also known as smart contracts.

Blockchain technology is used in several use cases such as smart city, smart agriculture, vehicle networks [10] and also healthcare [15]. In the latter two cases, the privacy and ownership of data transmitted from a driver and medical patient is particularly important, as this data may contain privacy sensitive information. It can be noted that in these latter use case using a private blockchain such as Hyperledger Fabric can be a more optimal choice as they can provide higher security and privacy level.

The topic of security and privacy of the Hyperledger Fabric has been thoroughly studied [4,16,18]. Moreover a research has been conducted this year by a student of Delft University of Technology [13] addressing similar issue of improving HLF security using self-encryption.

The concept of self-encryption was introduced by Yu Chen [2]. The approach of the original paper involves converting a file into a bit stream, extract the key by randomly selecting bits from the stream and then doing the encryption using that key. After the encryption the key and the encrypted file should be stored separetely, e.g. the key can be stored locally, while the encrypted file can be sent to a server.

The original encryption scheme was also extended by Moch Rezky Debby Rahardjo [14]. According to the paper, "The modification is located in dividing the plaintext and ciphertext into 1024-bit chunks at XOR process and using the date when encryption process starts as a seed. The modification also adds the database for the key management function". Storing the key and the encrypted chunks in separate places makes in computationally not feasible to get the original data.

The later industrial adaptation of the self-encryption scheme happened when a team lead by David Irvine made self-encryption the core of his company's (MaidSafe) product - SAFE Network [9]. Irvine's implementation of the self-encryption scheme will be the basis of this work, hence a more detailed explanation of the implementation of the algorithm will be given.

Figure 1 shows the encryption process. First, the original file is getting split into minimum of 3 file chunks. After the file is split into chunks the algorithm creates a data map, where the key needed for decryption will be stored. Each chunk is then hashed and those hashes are written to the data map. Parts of those hashes are used as a key and initialization vector for AES 128 algorithm that encrypts each file chunk. When encryption is done, each encrypted chunk is obfuscated with the previously computed hash values by applying a XoR function. At the end of the process the encryption scheme returns a data map that is going to be later used for decryption, and the encrypted file chunks.

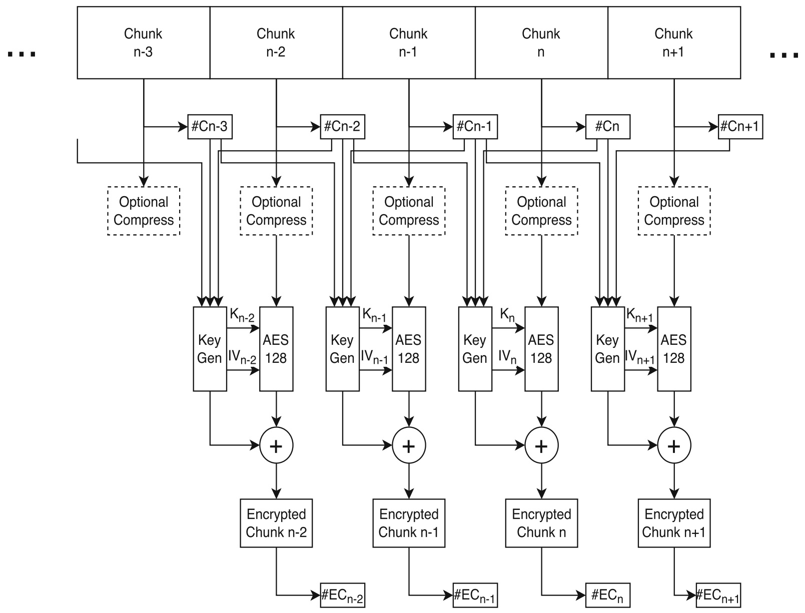

Fig. 1. Self-encryption process, adopted from [11]

3 Background

Original implementation of the self-encryption schema by David Irvine [11] was modified and used for this research. The use of rayon library (which adds parallelization to the code) was removed from the algorithm, due to the fact that the compilation target (WebAssembly) only supports single-threaded code. Additionally the code base was modified to include an interface for communication with the external code. Changes were also done to the Cargo.toml to make the code compatible with the target. Modified self-encryption algorithm was compiled to WebAssembly and run in a virtual machine (VM) and invoked from the code of the developed local application (which allows the interaction with the Hyperledger Fabric Smart Contract). A more detailed description of the process will be given in the Sect. 4. The benchmarks of this implementation will be provided in Sect. 5.

Hyperledger Fabric test network v2.4.3 was used. Test network was deployed to Docker based on the tutorials provided by Hyperledger Fabric[2].

Smart contract was then deployed (detailed in Sect. 4).

Encrypted file storage is handled by the IPFS, which is a distributed Torrent database, which uses hashes of files to address its content. IPFS node was also deployed to Docker. For IPFS deployment two directories (staging and data) were

[2] Usage of the command requires navigating to the root directory of the test-network, provided by the Hyperledger Fabric [8].

mounted on the host file system to persist the stored data, when the container is stopped. Hyperledger Fabric provides official software development kit (SDK) for 3 languages: Go, Java, Javascript. Go was chosen for implementation of the project, due to ease of integration with both Hyperledger Fabric and the IPFS. Encryption library is written in Rust and is compiled to WebAssembly, hence a way to call WebAssembly was needed. Go also provides support for Wasmer library that allows to call WebAssembly function directly from Go code.

4 ID-Based Self-encryption

4.1 Integrating Identity into the Encryption

This paper offers an extension to the algorithm proposed by Irvine [9]. Encryption step in the original algorithm is modified to include identity of a person who is running the algorithm into the encryption process. Instead of using part of the chunk hash as a key for AES 128, the result of XoR of the hashed identity and the chunk hash is used as a key. The identity can be any string of any length. If the length of this string is shorter than the length of the key, then the cycle function is applied to the string, which repeats the iterator of a string. The hashing function SipHash 1-3 is used to hash the identity of a user, before passing it to the XoR function. Figure 2 demonstrates the process of encrypting a file using the modified version of self-encryption with identity integrated into the encryption process.

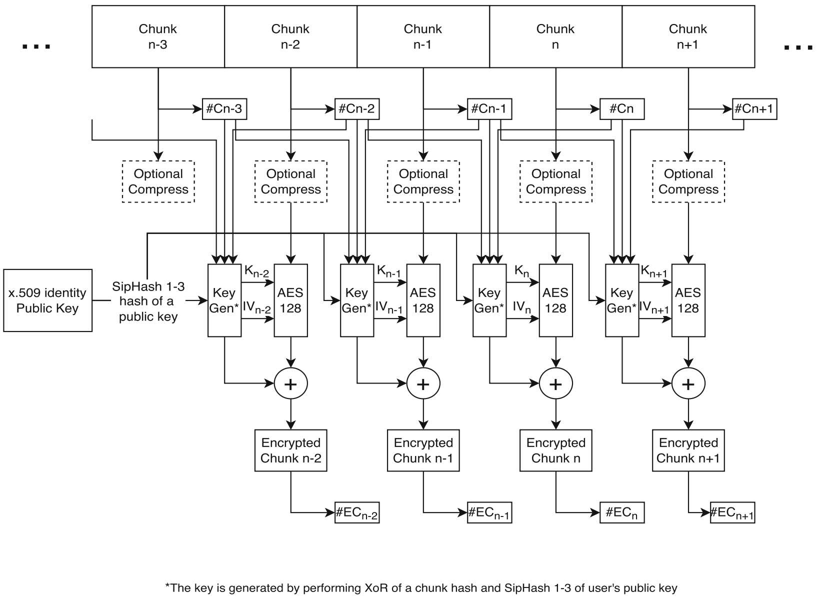

Fig. 2. ID-based self-encryption process

Decryption of the file, that was encrypted using ID-based self-encryption, is similar to that of a regular self-encryption, with the key for AES 128 being the only different part. The decryption calculates the key the same way the encryption does it by applying XoR function to the hash of identity and the chunk hash from the data map.

The implementation of the encryption scheme can be found on GitHub[3].

4.2 Connecting the Encryption Algorithm and the Local Application

The implementation of the identity-based self-encryption is written purely in Rust, while the client application is written in Go. This creates a demand for a way to integrate Rust library into Go code. Among the solutions to tackle the problem are:

1. Use Go tools to assemble the Go code and compile Rust code into a static library. Then link compiled code using additional assembly "glue-code" [17].
2. Compile Rust to a static library and call it from the Go code using Go build-in pseudo-library C for interacting with native interfaces.
3. Compile Rust to WebAssembly (WASM) code and call it from Go using Wasmer library[4].

All of the methods have been successfully tried. The first two methods do not allow cross compilation, because both of them require compiling Rust to a static library, which is platform-specific. Additionally, the first methods requires the use of assembly language, which is different on different processor architectures and operating systems. The second method also uses C pseudo-library, which does not allow cross-compilation of the Go code. Overall, both methods are very *platform-specific*, which makes them less preferable choice.

The third method was chosen for connecting Rust library to Go code. Compiling Rust to WASM to use as a standalone application or a library, can be done using the following command:

```
$ cargo build --target=[chosen_target]
```

where *chosen_target* is a WebAssembly target that can be either wasm32-unknown-unknown or wasm32-wasi. The latter was used, because it compile using WASI API[5], which is a system API that provides access to multiple operating system functionalities, such as access to the file system.

The resulting WASM file is then placed in a hidden folder in the home directory of a user, so it can later be loaded by the Go code. As the WASM code is used within a virtual machine (VM), it's independent from the operating system it will run on, so requires compiling only for one target.

[3] https://github.com/ilyagrishkov/ib-self-encryption-rust.
[4] https://wasmer.io/.
[5] https://wasi.dev/.

Calling WASM from Go Using Wasmer. In order to call WASM code a VM needs to be used. Wasmer library provides such VM, that can also be initialized from within Go code. The process of calling WASM code from the Go application is demonstrated in the Fig. 3. This process consists of the following steps:

1. Loading WASM code into a Wasmer VM
   (a) A directory on the host operating system, that will be accessible in the VM, need to be specified
   (b) Optionally, standard output of the WASM library can be inherited.
2. Invoking a function by specifying its name and the return type and passing arguments to it

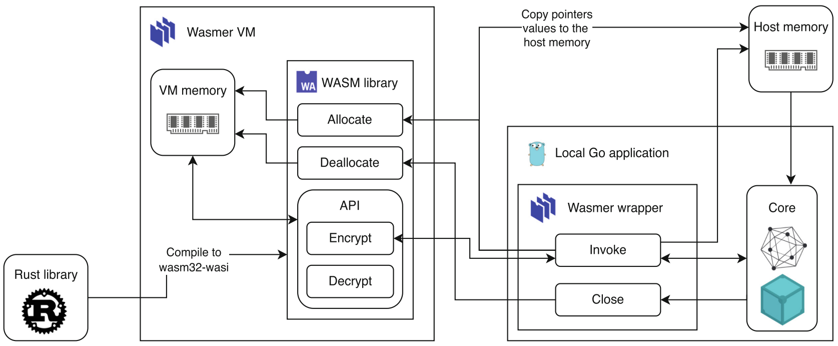

Fig. 3. Connection of the WASM encryption library to the client application via the Wasmer library and the developed wrapper

The communication between the Go code and WASM library and passing arguments for function invocation is happening using C types, which means that types like strings are not supported directly and need to be converted to corresponding C types. In case of a string being passed as an argument, it needs to be written to memory and end with a zero byte. The pointer to the first byte of this string is then passed to the invoked function as an argument.

As the host operating system memory is inaccessible for the VM, allocation and deallocation of memory need to happen within the VM itself. In order to facilitate the allocation and deallocation two dedicated Rust functions were developed as a part of id-based encryption library interface: allocate and deallocate.

In case the called function requires a string as an argument, the allocation needs to be performed before passing the pointer to that string. The allocate function has to be called to allocate memory inside the VM. The memory is then accessed from the Go code and each byte of the string argument is written to the newly allocated memory. The pointer to the memory and the length need to be preserved in order to deallocate the memory, before the program terminates. The pointer to the first memory cell containing the string argument is then passed as an argument to the function that is being called.

Wrapper Code for Wasmer Calls. A wrapper code has been written to simplify invocation of WASM functions. The major simplification that this code provides is the ability to pass Go native-type argument to the wrapper, which then performs all the necessary processing and allocation, if needed. The pointers to string or array types as well as their lengths are stored, so when the program terminates, the memory is getting deallocated.

Moreover, the developed wrapper code allows to pass simple numerical Go types (integers, floats, bytes, etc.) as pointers to the WASM library, so the changes that are happening to them when WASM functions run are also reflected in Go code, without the need to return anything.

Additionally, the wrapper requires return type parameter argument (which is represented as an enumerator), when calling the invocation function through the wrapper. It uses the return type to case the return of WASM function to corresponding Go type. In cases when a pointer to a string is returned, the wrapper reads bytes from the VM memory until the zero byte and creates a Go string from it. The return type of the wrapper's invocation function is a generic *interface*{}, which requires additional type casting. For example, in case the called function returned a pointer to a string, a Go string will be built from the pointer, but a user will still have to dynamically convert the returned value as it will be *interface*{}.

4.3 Smart Contract

The smart contract in Hyperledger Fabric allows to define assets that will be on the ledger. This paper defines an asset containing three fields: ID, Owner and CID. The code below shows the definition of an asset written in Go.

```
type Asset struct {
        ID      string    `json:"ID"`
        Owner   string    `json:"Owner"`
        CID     []string  `json:"CID"`
}
```

Listing 1.1. The struct representing an asset on the Hyperledger Fabric ledger

The ID is a universally unique identifier (UUID) that is generated, when the new asset is created. The Owner is a string of hexadecimal numbers representing a public key of a user, who created the asset. The CID is an array if unique identifier that reference encrypted file chunks saved in IPFS. The references to the encrypted data chunks remain immutable, and can also be used for verifying if encrypted data chunks were manipulated (the hash of an encrypted data chunk is a unique value).

Additionally, the smart contract defines a list of functions for creating, deleting and updating assets. The implementation can be found on GitHub[6].

4.4 Self-encryption Work Flow in a Blockchain-IPFS Based Network

The encryption and decryption process as well as interactions with the IPFS and the Hyperledger Fabric are orchestrated by a local application, which is a command line interface (CLI) tool written in Go. The prototype of the tool is accessible from GitHub[7]

Execution of any command starts with creating a new instance of a WASM wrapper and loading of the encryption library. When the command requires interaction with the Hyperledger Fabric, presence of the wallet, containing identity (which is necessary to enable interaction with the smart contract), is being checked. If the wallet is missing it's getting populated based on the certificates and keys of a user. When this preparation is done, the execution of the command starts.

At the end of the program execution the wrapper iterates over all allocated memory pointers and individually deallocates them.

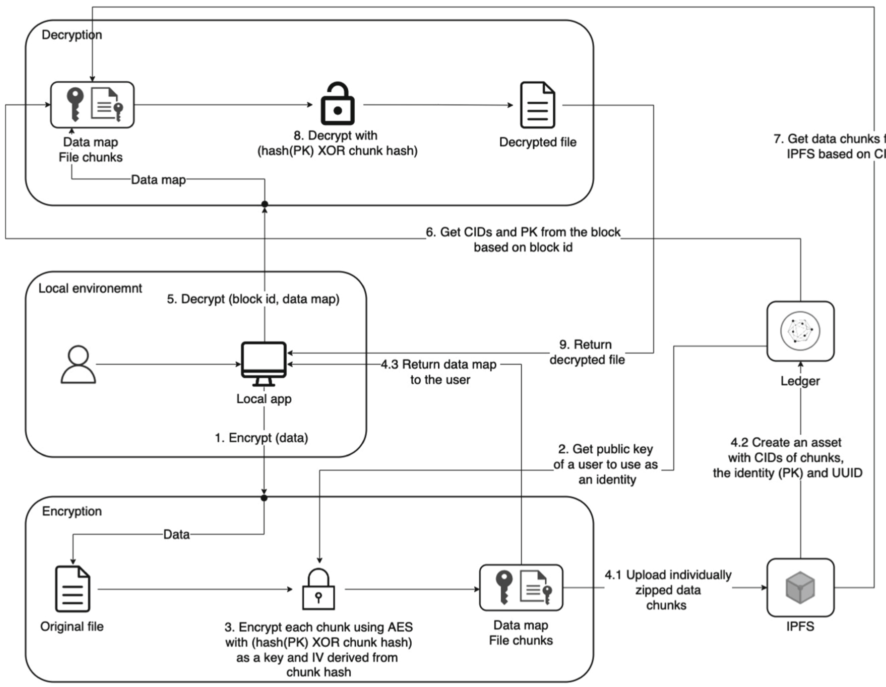
Decryption
Data map
File chunks
8. Decrypt with (hash(PK) XOR chunk hash)
Decrypted file
Data map
7. Get data chunks from IPFS based on CID
6. Get CIDs and PK from the block based on block id
Local environemnt
5. Decrypt (block id, data map)
Local app
9. Return decrypted file
4.3 Return data map to the user
Ledger
1. Encrypt (data)
2. Get public key of a user to use as an identity
4.2 Create an asset with CIDs of chunks, the identity (PK) and UUID
Encryption
Data
Original file
3. Encrypt each chunk using AES with (hash(PK) XOR chunk hash) as a key and IV derived from chunk hash
Data map
File chunks
4.1 Upload individually zipped data chunks
IPFS

Fig. 4. Work flow in the blockchain-IPFS based network when using self-encryption

[6] https://github.com/ilyagrishkov/ib-self-encryption-smart-contract.

[7] https://github.com/ilyagrishkov/ib-self-encryption.

There are two major parts of the system - encryption and decryption. Figure 3 demonstrates the workflow of both of them.

Encryption. The first part, encryption, that deals with encrypting a file and uploading data to the Hyperledger Fabric starts when the following command is called:

```
$ ibse add [file] [key_output_path]
```

where *ibse* is the name of the local app, *file* is the absolute path to the file that needs to be encrypted, and *key_output_path* is the absolute path to location where the key will be stored.

The original file is getting uploaded to the directory that was mapped during the VM initialization. From there it can be read by the WASM code. The encryption function is then called and the output is written to a new directory inside the mapped one. The output consists of multiple encrypted file chunks and a data map. The data map is moved to the location specified by the user and can later be shared via a secure channel. Each encrypted file chunk is being put into a zip archive to preserve their names, when uploading to the IPFS, and sent to the IPFS. The unique identifier, corresponding to each chunk (Content Identifiers or CIDs which are the hash values of the files) is returned. A smart contract function is then called that creates a new asset with all CIDs.

Decryption. The second part of the system, decryption, is invoked using the following command:

```
$ ibse get [block] [key] [destination]
```

where *block* is the UUID of an asset in HLF blockchain that contains CIDs of encrypted chunks, *key* is the absolute path to the data map, and *destination* is the absolute path to location where decrypted file should be written.

The UUID allows to identify an asset containing CIDs of encrypted file chunks. Each of the chunks is downloaded from the IPFS, unarchived, and written to the directory that is accessible from the VM. The data map is then copied to the same directory. After collecting all the necessary files for decryption, the decryption function is called and the restored file is written to a user-specified destination.

5 Results

5.1 Performance Analysis

Benchmarking of the system was done on the iMac 2019, 3,6 GHz 8-Core Intel Core i9 with 32 GB of memory running on MacOS 12.3.1.

Benchmarking of the implemented id-based self-encryption scheme was done. As the encryption itself is not implemented in the same language as the rest of the project (the encryption is implemented in Rust and the rest of the project is in Go), the execution time can differ when Rust functions are called from Go compared to pure Rust execution time.

Files of sizes 100-, 250-, 500-, 750 kilobytes, 1 megabyte, 10-, 25-, 50-, 75-, and 100 megabytes were created for benchmarking both the pure Rust implementation as well as the WASM + Go implementations. Moreover, for this benchmark both the Rust code and the WASM library were optimized using maximum level of optimization provided by the Rust compiler.

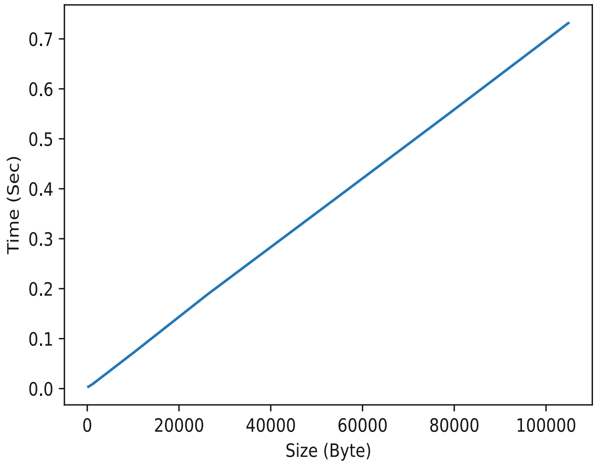

Fig. 5. Dependence of the execution time of id-based self-encryption algorithm in pure Rust from the file size

The initial benchmark was performed on the encryption function only and was measuring execution time of the pure Rust implementation. Figure 4 shows the results of the benchmarking.

The chart show near-linear dependence between the size of the file and the time it takes to encrypt it. This dependence can be explained by the fact that the most demanding computational is the AES 128 encryption process and with the increase of the file size, then number of chunks it is split to increases. Each chunk of the original file needs to be individually encrypted, hence the computation time grows linearly with the size of the file.

As the encryption function execution time grows linearly due to the computational demand of the AES 128 and the hashing algorithms, the decryption process will be identical, because it uses the same algorithms for decryption.

In order to achieve more objective benchmark results, file of each size has been encrypted 100 times and the average calculated. In order to visualize execution time a chart in Python using MatPlotLib[8] was created. The chart contains a 25-bin histogram, each representing density of a particular measurement. Following

[8] https://matplotlib.org/.

central limit theorem the distribution of the execution time measurements was assumed normal, so the mean and the standard deviation were calculated and the distribution plotted over the histogram. Figure 5 shows an example of combined charts for pure Rust and WASM + Go execution times, when encrypting 50 megabytes file. The blue histogram on the left-hand side shows results of the 100 measurements of the execution time of the Rust implementation; on the right-hand side - of the WASM + Go implementation.

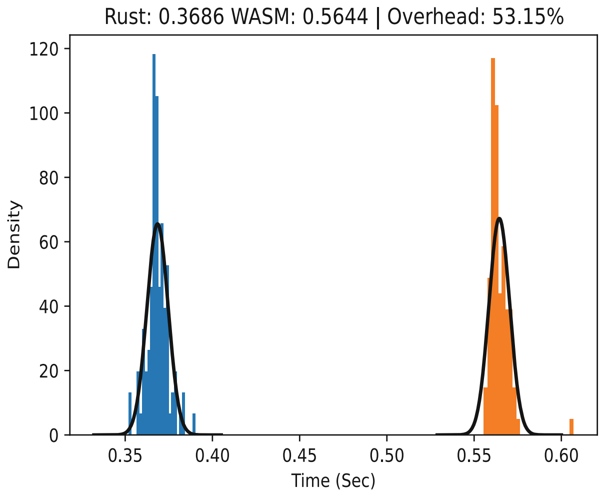

Fig. 6. Run time distribution for 50 MB file encryption using pure Rust and WebAssembly + Go implementations (Color figure online)

The results of execution time measurements have for various file sizes are summarized in Table 1. The execution time shown in the table is the average number of seconds it takes a corresponding implementation to encrypt a file of a corresponding size. In addition to the average execution time, the overhead of the WASM + Go implementation is calculated for every pair of measurements.

It is visible from the table that the overhead has a clear downwards trend (except the spike, when encrypting 250 KB file). When the execution time of a WASM + Go encryption implementation is less than 0.01 s, the overhead falls in the range between 70% and 85%. When the execution time is longer than 0.1 s, the overhead goes down to 50%–55% and stays in that range when the file size increases. Figure 6 demonstrates the overhead of WASM + Go encryption of file of different sizes.

Table 1. Average execution time and overhead when encrypting files of different sizes using id-based self-encryption

File size (Byte)	Average execution time (Sec)		Overhead (%)
	Rust	WASM + Go	
100 KB	0.0024	0.0042	75.89
250 KB	0.0038	0.007	84.51
500 KB	0.005	0.0088	75.48
750 KB	0.0069	0.0117	71.4
1 MB	0.0081	0.0139	71.29
10 MB	0.0747	0.117	56.55
25 MB	0.1885	0.2851	51.22
50 MB	0.3686	0.5644	53.15
75 MB	0.5492	0.8447	53.81
100 MB	0.7317	1.1201	53.08

Such decrease in the overhead, when the execution time becomes longer is explained by the presence of the Wasmer library invocation overhead, which occurs every time a call is made to the Wasmer VM. When execution time itself is less than 0.01 s the invocation overhead is significant compared to the execution time. At the same time, when the execution time becomes longer, the overhead from invocation becomes insignificant, and measurements start to approximate real WASM VM overhead, which is around 50%–55%.

5.2 Benefits Overview

The designed app has multiple surfaces of attack. The IPFS nodes, where the encrypted file chunks are stored can attacked. Also, an adversary can be gain unauthorized access to the ledger with references to file on the IPFS. Both of those possibilities are analyzed below.

The security of IPFS nodes (assuming the encrypted file chunks were stored individually on multiple nodes) can be compromised, in which case encrypted files will be leaked to the malicious user. As encrypted file chunks have been stored on different nodes, the probability that all of them being compromised is negligible and should not be considered. Additionally, individual files do not have any link to each other, so matching multiple encrypted chunks, that are needed for successful decryption, is not computationally feasible. The data map containing the keys for the decryption, was stored locally by the user, who encrypted the file. Without the original keys, the decryption of the self-encrypted data is computationally not feasible [9]. Moreover, as the proposed self-encryption is also related to the data owner or user identity, the decryption cannot be done until the identity is not provided. Thus the data ownership is also provided by the id-based implantation.

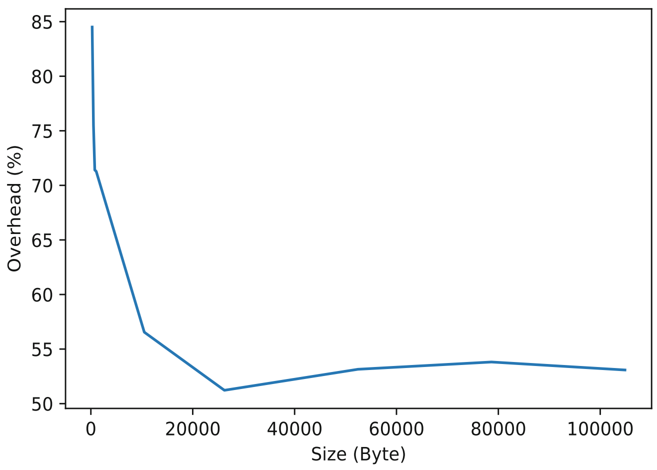

Fig. 7. WebAssembly + Go implementation overhead measurements over for files of different sizes compared to pure Rust implementation

6 Discussion

The results show high security guarantees of the id-based self-encryption scheme, when used for encrypting data, stored on the Hyperledger Fabric blockchain. This allows to use the implemented prototype as a secure medium for saving a retrieving information from the ledger.

In future works full Go implementation of self-encryption should be compared with the design proposed in this paper.

It was also beyond the scope of this study to create a standardized benchmarking for WASM and Rust libraries. It can be done by using multiple sample programs that test specific properties of the programming language (e.g. efficacy of memory allocation and deallocation) or very computationally intensive programs [6]. The objective could be running a containerized version of both libraries against a set of such programs and analyzing the run time.

Moreover, the study can be expanded by analyzing and comparing CPU and memory load of WASM and Rust libraries. Such benchmark could be also done using sample programs mentioned in the previous paragraph (Fig. 7).

7 Conclusion

In this study a new approach to storage of files on the Hyperledger Fabric blockchain was presented. The demonstrated approach allows for secure storage of data in a decentralized way, with ability to preserve the original file ownership and also information about the person, who encrypted it. This approach can be used where high security and trust in the integrity of data stored on the ledger is need. The prototype uses Rust implementation of id-based self-encryption that is compiled to WebAssembly and invoked from Go code.

Additionally, the study demonstrates the generic way of integrating ID-based self-encryption into the blockchain context with a relatively low overhead (the overhead is around 55%) and high performance level of WebAssembly library integration with the Go code base, compared to the pure Rust implementation. Relatively low overhead of WASM creates possibilities for developers to use WASM integration with Go and other languages that support Wasmer library as a cross-platform solution that allows to achieve high degrees of performance, while also being deterministic.

The wrapper proposed in this work can also be used in Golang-based back-end applications that aim to use the cryptographic libraries deployed in Rust providing a more memory-safe execution.

Acknowledgements. This research is supported by European Union's Horizon 2020 researchand innovation programme under grant agreement No. 952697 (ASSURED), No. 101021727 (IRIS), and No. 101070052 (TANGO).

References

1. Buterin, V.: Ethereum: a next-generation smart contract and decentralized application platform (2014). https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 22 Aug 2016
2. Chen, Y., Ku, W.S.: Self-encryption scheme for data security in mobile devices. In: 2009 6th IEEE Consumer Communications and Networking Conference, pp. 1–5. IEEE (2009)
3. Christian, C.: Architecture of the hyperledger blockchain fabric (2016). https://www.zurich.ibm.com/dccl/papers/cachin$_/$dccl.pdf. Accessed 10 Aug 2016
4. Dabholkar, A., Saraswat, V.: Ripping the fabric: attacks and mitigations on hyperledger fabric. In: Shankar Sriram, V.S., Subramaniyaswamy, V., Sasikaladevi, N., Zhang, L., Batten, L., Li, G. (eds.) ATIS 2019. CCIS, vol. 1116, pp. 300–311. Springer, Singapore (2019). https://doi.org/10.1007/978-981-15-0871-4_24
5. Gerrits, L., Kromes, R., Verdier, F.: A true decentralized implementation based on IoT and blockchain: a vehicle accident use case. In: 2020 International Conference on Omni-Layer Intelligent Systems (COINS), pp. 1–6 (2020). https://doi.org/10.1109/COINS49042.2020.9191405
6. Gouy, I.: Toy benchmark programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/why-measure-toy-benchmark-programs.html
7. Huang, Y., Bian, Y., Li, R., Zhao, J.L., Shi, P.: Smart contract security: a software lifecycle perspective. IEEE Access **7**, 150184–150202 (2019). https://doi.org/10.1109/ACCESS.2019.2946988
8. Hyperledger: Using the fabric test network (2020). https://hyperledger-fabric.readthedocs.io/en/release-2.2/test$_$network.html. Accessed 19 Mar 2022
9. Irvine, D.: Self encrypting data (2010). Unpublished Manuscript
10. Kromes, R., Gerrits, L., Verdier, F.: Adaptation of an embedded architecture to run hyperledger sawtooth application. In: 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 0409–0415 (2019). https://doi.org/10.1109/IEMCON.2019.8936264
11. Maidsafe: self_encryption (2022). https://github.com/maidsafe/self$_/$encryption

12. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf. Accessed 01 July 2015
13. Park, C.: Using self-encryption to safeguard data security in fabric's smart contract. Bachelor's thesis (2022). https://repository.tudelft.nl/islandora/object/uuid:15c5eee3-0be6-4d71-bf67-3ea5e576aa05?collection=education
14. Rahardjo, M.R.D., Shidik, G.F.: Design and implementation of self encryption method on file security. In: 2017 International Seminar on Application for Technology of Information and Communication (iSemantic), pp. 181–186. IEEE (2017)
15. Ray, P.P., Dash, D., Salah, K., Kumar, N.: Blockchain for IoT-based healthcare: background, consensus, platforms, and use cases. IEEE Syst. J. **15**(1), 85–94 (2021). https://doi.org/10.1109/JSYST.2020.2963840
16. Stamatellis, C., Papadopoulos, P., Pitropakis, N., Katsikas, S., Buchanan, W.J.: A privacy-preserving healthcare framework using hyperledger fabric. Sensors **20**(22), 6587 (2020)
17. Valsorda, F.: Rustgo: calling rust from go with near-zero overhead (2022). https://words.filippo.io/rustgo/
18. Yamashita, K., Nomura, Y., Zhou, E., Pi, B., Jun, S.: Potential risks of hyperledger fabric smart contracts. In: 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 1–10. IEEE (2019)
19. Zibin, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: a survey. Int. J. Web Grid Serv. **4**, 352–375 (2018). https://doi.org/10.1504/IJWGS.2018.095647