Blockchain-based Multi-Party Authorization for Accessing IPFS Encrypted Data

Ammar Battah [1], Mohammad Madine [1], Hamad Alzaabi [1], Ibrar Yaqoob [2], Khaled Salah [1], and Raja Jayaraman [1]

[1]Affiliation not available
[2]Khalifa University of Science and Technology

October 30, 2023

Abstract

Multi-party authorization (MPA) typically involves multiple parties to control and grant access to shared data. MPA is used to solve the insider's attack problem by ensuring that a single authority or party is not acting alone. Currently, almost all existing implementations of MPA are centralized and fall short in providing logs and events related to provenance of granting permissions in a trusted, secure, immutable, auditable, and decentralized manner. Moreover, for sharing data, proxy re-encryption algorithms are often used to give secure access to encrypted shared data. These schemes and algorithms are also centralized and cannot be trusted. In this paper, we propose a fully decentralized blockchain-based solution in which MPA is implemented using Ethereum smart contracts, and proxy re-encryption algorithms (which are computationally expensive) are implemented using multiple oracles to give access to encrypted shared data stored on a public and decentralized storage platform, such as the Interplanetary File Systems (IPFS). The smart contracts help to validate results based on the majority of encrypted results determined by the oracles. For this, we incorporate reputation mechanisms in the proposed smart contracts to rate the oracles based on their malicious and non-malicious behaviors. We present algorithms along with their full implementation, testing, and validation details. We evaluate the proposed system in terms of security, cost, and generalization to show its reliability and practicality. We make the smart contract source code publicly available on Github.

Blockchain-based Multi-Party Authorization for Accessing IPFS Encrypted Data

Ammar Battah, Mohammad Madine, Hamad Alzaabi, Ibrar Yaqoob, Khaled Salah, Raja Jayaraman

***Abstract*—Multi-party authorization (MPA) typically involves multiple parties to control and grant access to shared data. MPA is used to solve the insider's attack problem by ensuring that a single authority or party is not acting alone. Currently, almost all existing implementations of MPA are centralized and fall short in providing logs and events related to provenance of granting permissions in a trusted, secure, immutable, auditable, and decentralized manner. Moreover, for sharing data, proxy re-encryption algorithms are often used to give secure access to encrypted shared data. These schemes and algorithms are also centralized and cannot be trusted. In this paper, we propose a fully decentralized blockchain-based solution in which MPA is implemented using Ethereum smart contracts, and proxy re-encryption algorithms (which are computationally expensive) are implemented using multiple oracles to give access to encrypted shared data stored on a public and decentralized storage platform, such as the Interplanetary File Systems (IPFS). The smart contracts help to validate results based on the majority of encrypted results determined by the oracles. For this, we incorporate reputation mechanisms in the proposed smart contracts to rate the oracles based on their malicious and non-malicious behaviors. We present algorithms along with their full implementation, testing, and validation details. We evaluate the proposed system in terms of security, cost, and generalization to show its reliability and practicality. We make the smart contract source code publicly available on Github.**

***Index Terms*—Blockchain, Access Control, Authentication, Ethereum, Encrypted files, Multi-Party Authority.**

I. Introduction

Recent years have witnessed unprecedented increase in identity theft, data loss, and security breaches. It has been reported that 3,813 data breaches occurred that led to expose 4.1 billion records in the first 6 months of 2019 [1]. Unauthorized access or intentional breach has become one of the major threats that is presented due to the improper implementation and maintenance of access control systems [2]. Access controls help to perform authentication and authorization of individuals. They serve as the first and most important line of defense against potential data access breaches. In the cybersecurity world, getting unauthorized access to the "root user" is often considered as game over. Gaining access to the most important user gives the attacker privileges to exploit and manipulate a system by executing further attacks without any constrictions. For example, unauthorized access made Edward Snowden, a former national security agent, has enabled to steal approximately 1.7 million documents containing secret data from the national security agency (NSA) and delivered them to news agencies [3]. This security breach has led to expose NSA's confidential data to the general public. Snowden used an NSA civilian's public key infrastructure (PKI) certificate to gain access to classified information on the NSANET [3].

Ensuring confidentiality of sensitive data is one of the primary requirements of today's systems. Access control policies play a vital role in terms of system security. Implementation and maintenance systems of access control policies must be secured enough because if they get compromised, then the internal and external defense systems are no longer be useful. Unauthorized access is mostly caused by users' negligence, which leads to fail the purpose of other implemented defense systems. In simpler terms, improper and insecure privileged access management can pose serious security threats. In addition, access controls can act as single point of compromise in the system. This problem can be addressed through multi-party authorization (MPA), which uses multiple authorities to perform authentication and authorization. Specifically, the MPA technology can be used to secure the most sensitive data against insider attacks that are mostly carried out by the insider acting alone [4]. It is somewhat similar to weapons systems, wherein two individuals are required to enable a system using two different keys [4]. Another example of the MPA is like accessing to a lockbox in a bank that usually requires multiple parties, such as the lockbox owner and a bank official to gain access [4]. In a simpler term, MPA acts as a second authority that helps to review and approve any activity before its commencement. Leveraging MPA for access controls can help to grant permissions in a reliable manner. However, MPA is a centralized solution and often falls short in providing provenance of log events related to access/permissions in a manner that is immutable, auditable, decentralized, and trustful. On the other hand, most of the existing proxy re-encryption schemes used to give secure access to shared encrypted data are centralized and cannot be trusted. In this paper. we propose a fully decentralized blockchain-based solution to address the aforementioned problems. Also, storing all the transactions related to MPA and access to files on the blockchain can lead to create and maintain a trusted, immutable, and secure audit trail that could be verified by anyone.

Ammar Battah, Mohammad Madine, Hamad Alzaabi, Ibrar Yaqoob, and Khaled Salah are with the Department of Electrical Engineering and Computer Science, Khalifa University, Abu Dhabi 127788, United Arab Emirates.

Raja Jayaraman is with the Department of Industrial and Systems Engineering, Khalifa University, Abu Dhabi 127788, United Arab Emirates.

II. Related Works and Contributions

Blockchain technology can be used to efficiently manage access controls. However, its potential has not been widely explored in this regard, and thus very limited literature is available on this topic. For example, the authors in [5] have proposed a solution called ”FairAccess” that utilizes smart contracts to enforce access control policies. This solution is designed for the Internet of things (IoT) devices, wherein a transaction processing is based on tokens that are generated through a ”GrantAccess” function that is called by a resource owner. Subsequently, a requester uses the GetAccess function to consume the token and obtain access to a certain resource or it can delegate the token to a new owner under certain conditions using a ”DelegateAccess” function. Access to the resource can also be revoked by the owner if some misbehavior is detected through a ”RevokeAccess” function. Each token is encrypted through the public key of the receiver, which is extracted from the user address. This system is designed for IoT devices to give their owners full access and control over their data. Another study conducted in [6] proposes an access control policy for an electronic health record (EHR) system. The data which needs to be shared is initially encrypted with symmetric keys. A proxy acts as a mediator between the sender and receiver that helps to fetch and send the data. The owner combines the private key with the receiver’s public key to create a re-encrypted key that is sent to the proxy. The proxy downloads the encrypted files and re-encrypts them with the new key and pushes the data to the receiver that is decrypted with the private keys. Through this approach, each new user always requires a new re-encryption key that makes it inefficient. Also, the re-encryption is performed using a centralized server that can not be trusted in most of the cases.

PriWatt is a token-based solution built on top of blockchain [7] for energy trading. The system uses the concept of multi-signature technology to verify the validity of transactions. The multi-signature requires a minimum of m of n keys to sign the transaction before a token is spent. This methodology requires that m is the minimum number of signatures that need to match a public key, and n represents the number of keys provided. A minimum of t keys, which are less than n, should be provided to proceed with the transaction. This helps to decentralize the transaction execution process. However, in the token-based systems, it is hard to verify whether or not the token is being spent by an authorized user. In [8], a blockchain and trusted oracles based decentralized access control solution for the IoT data has been proposed. The trusted oracles are used to fetch data from the IoT devices. The proposed solution consists of admins and smart contracts (IoT data access, reputation, and aggregator), wherein the former is responsible to manage, control, and delegate access to the IoT devices, and latter enables the communication between users and oracles. The IoT Data Access contract forwards requests to Aggregator contract that sends requests to a pool of oracles, where hashing is performed against the data and subsequently send them back. The Aggregator contract helps to compare the hashes. The Reputation contract updates the reputation scores and helps to choose the one with the highest reputation before generating an access token for the end-user to access the IoT device through that oracle. One of the major limitations of this solution is that it fetches all the data that is not fully used in one turn. Moreover, it is expensive as it requires multiple oracles. Besides, the whole process of fetching data and comparing the hashes before granting access is time-consuming.

In the past, the concept of multiparty access control has been employed in online social networks (OSNs), wherein it aims at protecting shared data associated with multiple users [9]. Another study conducted in [10] has formulated a collaborative multi-party access control model to allow all individuals in OSNs related to a resource to collectively participate in defining access control policies. Despite many advantages of multiparty access control in OSNs, it poses certain limitations in terms of centralization, log provenance information, immutability, audit trial, trust, and security that hinder its wide-level adoption. For coping with such limitations, a multi-authority attribute-based access control approach using smart contracts has been proposed in [11]. The proposed approach is tested on the Rinkeby Ethereum Testnet. The proposed approach employs smart contracts to define and enable interactions between the data owner (DO), data user, and multiple attribute authorities. The approach is based on the concept of attribute tokens. After collecting attribute tokens, a smart contract allows issuing secret keys to particular users to give them access to certain resources. Unlike this approach, in our proposed solution, tokens are not exclusive to MPA but they are used to perform authentication between different participants. Besides, our proposed solution is significantly different from this existing approach. Our solution is fully decentralized in which MPA is implemented using the Ethereum smart contracts, and proxy re-encryption algorithms are implemented using multiple oracles to give access to encrypted shared data stored on a public and decentralized storage platform, such as the Interplanetary File Systems (IPFS).

Our key contributions are summarized below:

- We propose a fully decentralized blockchain-based MPA solution to provide provenance of log events related to access/permissions in a manner that is immutable, auditable, trustful, and secure.
- We develop smart contracts to define and enable interactions between DO, data requester (DR), MPA, proxy re-encryption oracles, and IPFS. The implementation code is made publicly available on GitHub[1].
- We implement the solution of giving access to encrypted shared data stored on IPFS using multiple oracles that report their results to the proposed smart contracts.
- We introduce and incorporate reputation mechanisms in the proposed smart contracts to rate the oracles based on their malicious and non-malicious behaviors.
- We evaluate the proposed approach against various metrics, such as security, cost, generalization to find out the limitations, reliability, and practicality of the proposed solution.
- Our proposed solution can be customized and implemented on both public and private blockchain networks based on the needs and preferences of industries.

[1]`https://github.com/multipartyauthority/MPA-access-control`

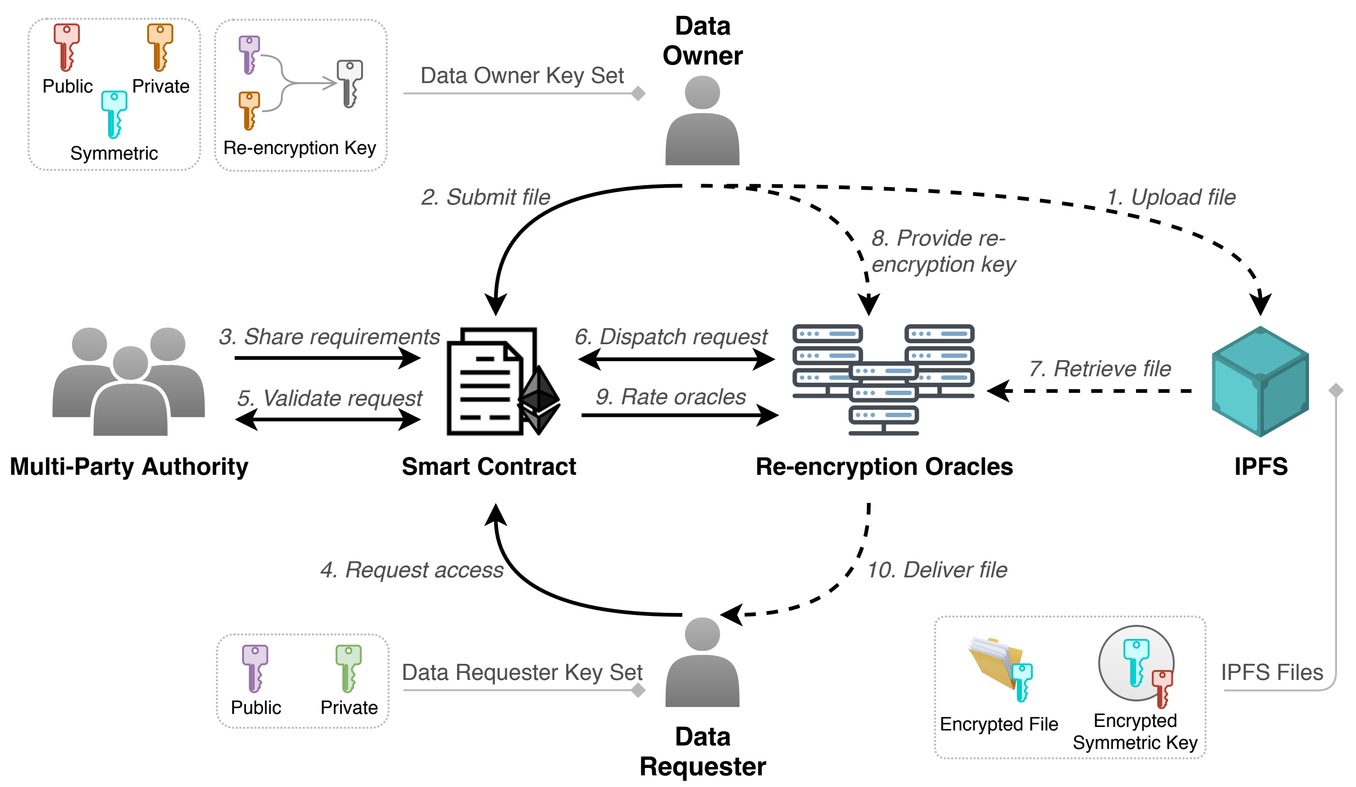


Figure 1: Architecture overview of the different participants.

The organization of the paper is as follows. In Section II, we review the existing literature. SectionIII presents the proposed solution along with its architectural components and shows sequential interactions between them. SectionIV discusses full implementation details. In SectionV, we provide security and cost analysis to show the reliability of the proposed solution along with important open challenges. Finally, a conclusion is given in SectionVI.

III. Proposed Approach

With the rapid technological advancements, risks have come as part of the bargain. While technology is advancing, security threats are also advancing. For any organization, a policy is what holds it together and governs the environment, and that is exactly what access control does. It defines digital policies to avoid threats and give role-based privilege to ensure authentication and authorization. In this section, we present the proposed Ethereum blockchain-based approach along with our system architecture components that include DO, DR, IPFS, proxy re-encryption oracles, and MPA.

A. Etheruem Blockchain

Blockchain has been a breakthrough in 2009 introduced by Sakatoshi Nakamoto to develop Bitcoin, a cryptocurrency that utilizes blockchain, as means of handling the decentralization of ownership and consensus over the cryptocurrency. Blockchain is a chain of blocks considered as a distributed ledger containing transaction records that are replicated across various computers assembled in a peer-to-peer (P2P) network. It contains information about the data of the transactions, time, value, hash, and encryption codes. It has hash functions that convert any input into fixed value output of each block so that each block will have the hash information of the previous one to provide integrity of the chain [12]. Moreover, it encrypts transactions using public and private keys to achieve confidentiality. In addition to that, it employs consensus algorithms, such as proof-of-work (PoW), proof-of-stake (PoS), proof-of-capacity (PoC) to name a few, to ensure that all entities involved in the system are agreed on a single source of truth.

Ethereum is a public and open-source blockchain platform that allows developers to create and deploy decentralized applications (Dapps) [13]. The Etheruem blockchain boasts attributes, such as modularity, simplicity, and universality. Ethereum virtual machine (EVM) is used to execute smart contracts to add application-specific logic to the Blockchain. The Ethereum-based smart contracts can help to form a closed network that consists of validators and participants that are responsible to perform actions and validation of the processes, and make important decisions. Moreover, through smart contracts, all the transactions can be monitored and controlled in a transparent, immutable, auditable, traceable, and secure manner. In summary, defining and implementing important properties (i.e., the access time of participants to confidential information, the extent of authorization, and the number of participants needed to satisfy the access policy) through the Ethereum blockchain can lead to offer a reliable, highly secure, trusted solution. The Ethereum-based permissioned blockchain is also gaining immense popularity in recent times. Such examples include Hyperledger Besu and Qourom [14] [15].

B. Architecture

The proposed system architecture is depicted in Figure 1. It consists of entities that communicate with the smart contracts to govern the access control of the encrypted data stored on the IPFS. Such entities include DO, DR, proxy servers/proxy re-encryption oracles, and IPFS. Each entity has a unique Ethereum address (EA) for communication purposes on the blockchain through Ethereum clients.

- **Data Owner (DO):** It is an initiation point of the system. The DO is responsible to upload data for sharing, perform communication, agree upon access requirements posed by the MPA, and register the address of the data (which is the hash of the data) on the blockchain. Also, it helps to encrypt the data using a symmetric key algorithm and send it to the P2P decentralized database along with the other key encrypted by the public key of a shared wallet between MPA and DO using multi-signature. Furthermore, the DO creates a smart contract that contains the hash of the mentioned components to act as the address of the data. Finally, the DO creates a re-encryption key from the public key of the DR and its own private key to send to the proxy servers.
- **Data Requester (DR):** The requester contacts the smart contract using its EA asking for access to the encrypted data provided by the DO. After the requester is validated, it waits to get an access token from the smart contract for the suitable proxy to receive the data. Once the data is downloaded from the proxy, the requester downloads encrypted data, encrypted symmetric key, and the hash of the file. Subsequently, it proceeds to decrypting the symmetric key along with the data using its private key and decrypting the data again with that symmetric key.
- **IPFS:** It is a P2P decentralized database that holds the data to be shared across multiple users [16]. The DO uploads the data encrypted with a symmetric key that is further encrypted with the DO's public key. Once the database is requested for the data (based on the hash of the file), it provides the proxy with the encrypted symmetric key and encrypted data.
- **Proxy re-encryption servers:** In our proposed solution, we use proxy re-encryption servers/oracles because performing encryption is a compute-intensive task. Note that implementing compute-intensive operations/functions using the Ethereum-based smart contract is a very expensive approach. Proxy re-encryption servers or oracles are used to fetch data and execute complex functions. They act as a medium to share data mainly between the DO and DR. The proxy servers have a reputation system managed by the proposed smart contracts. The reputation of a proxy fluctuates based on the response to the queries of the smart contract. If most proxies give the same hash results while others give different, then their reputation goes down. The proxy server has its own unique address that is sent within the access token, which is shared with the requester. On the other hand, the proxy server also receives a token with the requester address to perform validation. Once the proxy server receives the task of sharing the data to that requester, it ensures confidentiality, integrity, and privacy. The proxy server first gets the re-encryption key from the DO, then downloads the data from the decentralized database that includes the encrypted symmetric key and the encrypted data. Once the proxy server has the key and the data, it re-encrypts the symmetric key which is sent to the requester.
- **MPA:** It acts as a co-owner, which is included in each step of the access control mechanism. The MPA manages access to a shared wallet (with the DO) to avoid malicious acts. Using the multi-signature technology, the DO requires the keys of the MPA. It requires m of n (2 keys out of a total 3 keys) keys to use it. Note that m and n are adjustable based on a use case scenario. Although all the MPA entities do not need to be involved in this process, they must be qualified enough to verify the requirements needed to give access to the data requested by the DR. For example, lets assume that there is a highly confidential organization with an agent that wants to share a sensitive report. The agent would be the DO, but it does not make sense for him to have full sole control of the data, that is why it should be under the supervision of multiple people in the upper echelons. Some of them are given keys to access the account's wallet, wherein the supervisors would have the authority and knowledge of who has the right to access the file within the organization. In this way, the MPA technology can be used to secure the most sensitive data against insider attacks that are mostly carried out by the insider acting alone [4].

C. System Interactions

Figure 2 shows a sequential workflow between the entities involved in the proposed system. The dotted lines indicate an off-chain event; whereas, the solid lines are used to represent on-chain events. In the first stage, the DO needs to be registered to upload the needed data to the database with the symmetric key used for encryption. The symmetric key is encrypted with the public key of the owner. The owner creates a smart contract that contains the hash of the data, the address of the owner, and the access requirements that need to be shared with the authorities responsible to perform MPA.

- After the initial registration and setup, the smart contract shares the access requirements with the authorities that perform MPA.
- After that, the smart contract initiates a request made by the DR. The request gets validated by the MPA, which consists of authorized entities having enough knowledge to determine whether or not the user meets the specific requirements. Subsequently, the MPA sends the verification results to the smart contract.
- If the verification is succeeded, the smart contract asks the proxies/oracles to fetch the hash of the requested data. Based on the fetched results, the smart contract determines the suitable proxy. It proceeds to create a token with the address of the proxy to be sent to the DR and address of DR for the proxy. In this way, both of them are connected together to validate each other.

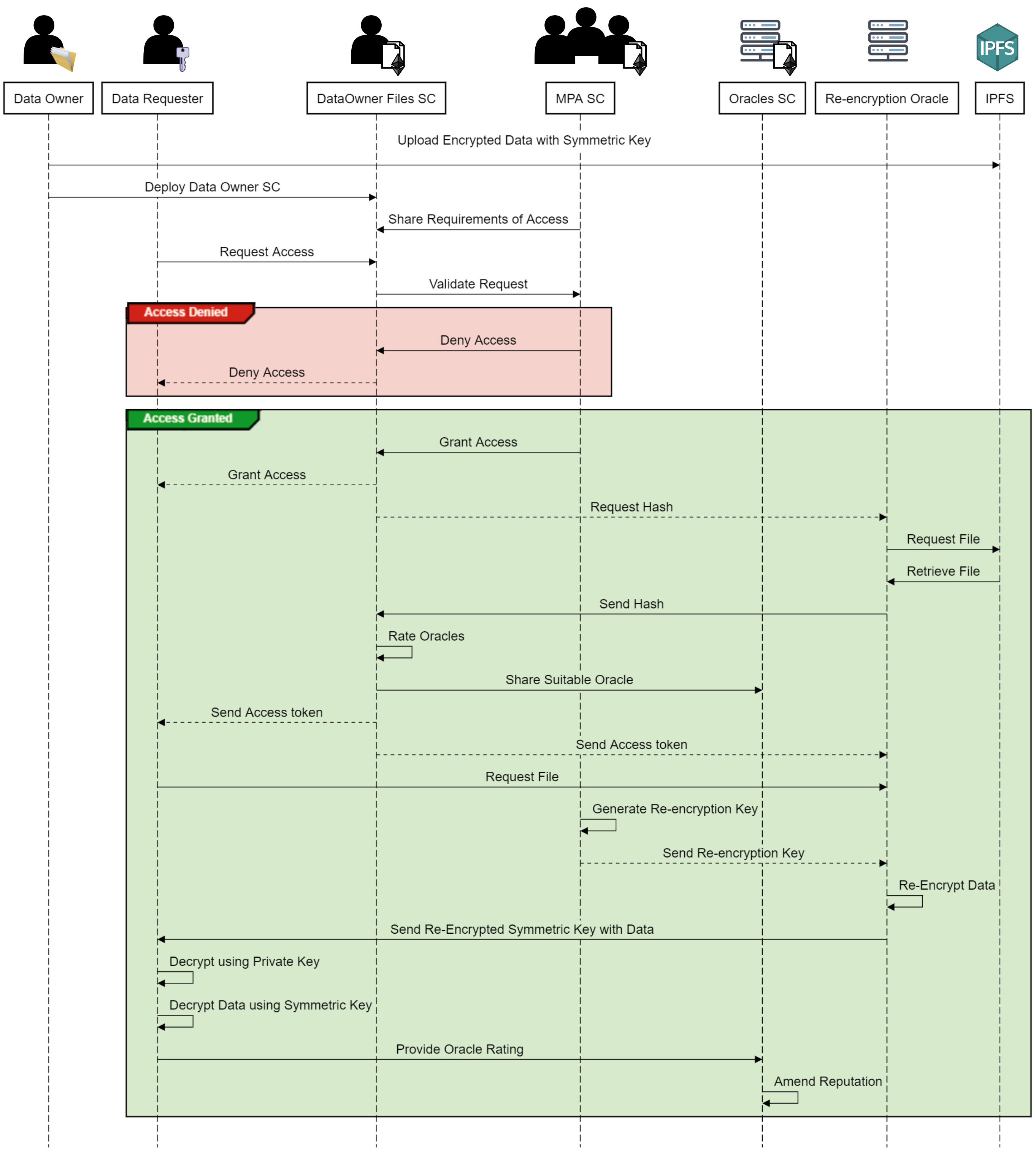


Figure 2: Sequence diagram of a file exchange between *DataOwner* and *DataRequester*.

- After that, the proxy server starts the process of fetching data while simultaneously getting the re-encryption key from the DO. The DO uses its private key and the public key of the DR to create the re-encryption key using the AFGH algorithm. The proxy server re-encrypts the symmetric key through atomic encryption, so it becomes invisible to the proxy server.
- Once the DR receives the needed data, it computes the hash of the data and compares it with that available on the blockchain to check its validity. After that, the symmetric key is decrypted using the DR's private key, and the data is decrypted using that symmetric key.

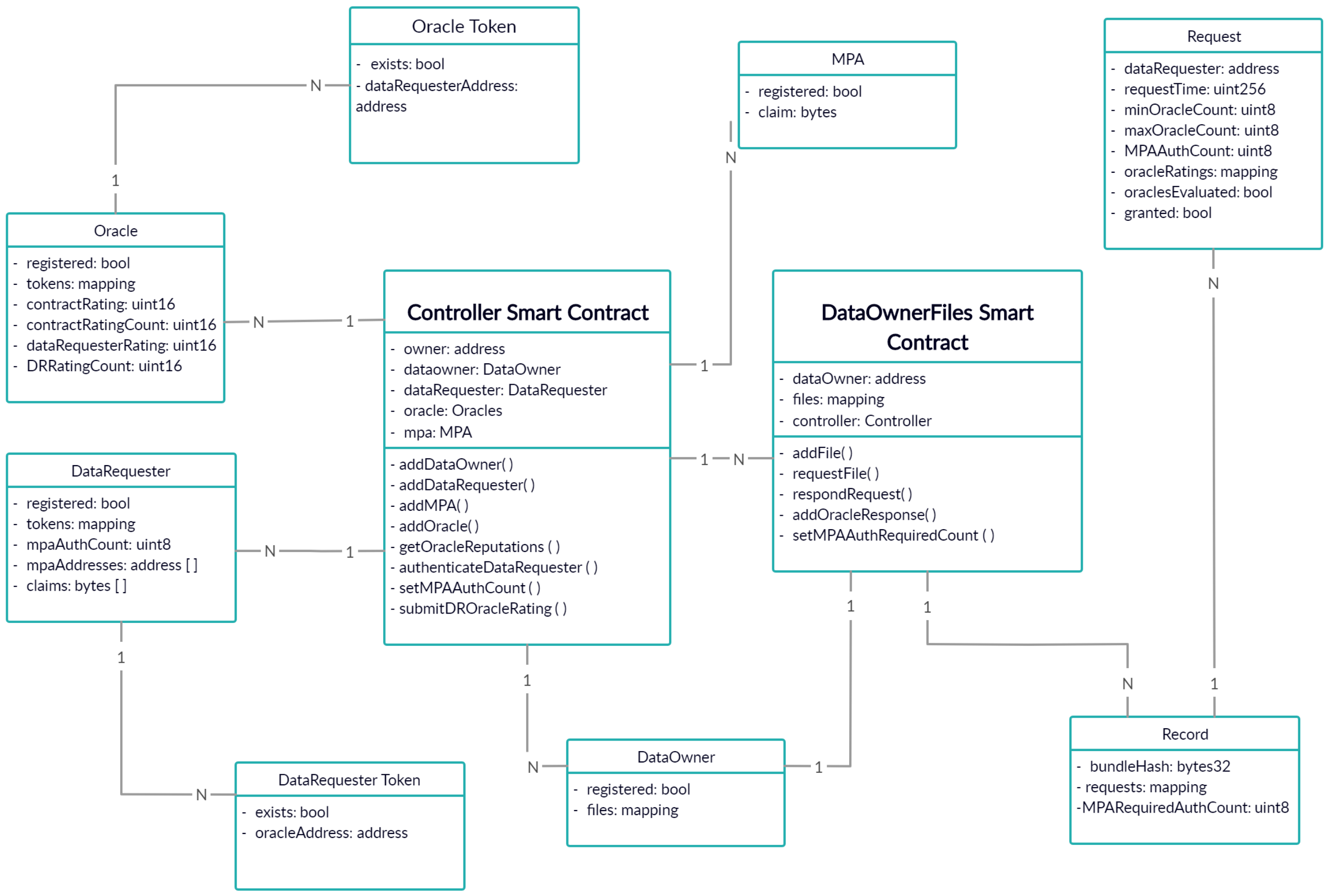


Figure 3: Entity relation diagram displaying the different entities along with their corresponding associations of interaction.

IV. Implementation and Validation

For the implementation, we use Remix Solidity IDE to write, run, and debug smart contracts [17]. The smart contracts allow to implement access controls for encrypted data stored on the IPFS. Also, the smart contract based reputation system is proposed to identify and mitigate malicious activities and threats by giving ratings to oracles or requesters. Figure 3 gives a simplified overview of the acting entities that are discussed in the following subsection.

A. Implementation

The first phase of the implementation is the registration and initialization of the network. The DO and its associated files, DR, oracles, and MPA are get registered with the variables set to the default states. The process gets started when the owner uploads file/s to the IPFS and subsequently submits its hash on the Ethereum-based blockchain. The owner only provides the hash of the data bundle containing the file encrypted by the symmetric key (K_s), which is further encrypted by a public key (K_p) of the MPA. Once the file is added, the MPA and the requesters are notified, as shown in the Algorithm 1. The events sent out are important and they enable the MPA to provide the access requirements to the smart contract.

Once the MPA gets a notification about the file, it sets the access requirements for that file. At this stage, the number

Algorithm 1 Add File

```
1: Input: $H$(uploaded bundle)
2: Require: (DataOwnerOnly)
3: Emit: file was uploaded to $MPA$ and requesters
4: add to bundle of hashes [] $\leftarrow$ $H$ (uploaded bundle)
5: create and add a new file with an empty request list
```

Algorithm 2 Request File

```
1: Input: FileIndex, DRPublicKey, OracleCountMin, OracleCountMax
2: Require: (DataRequesterOnly and ValidPublicKey and ValidOracleRange)
3: create new request(DRAddress, RequestTime, OracleCountMin, OracleCountMax)
4: add request to list of requests of the file
5: Emit: request submitted to $DR$ and DRPublicKey to $DO$
```

of authorities are also assembled. After that, the MPA shares this information with the smart contract of the DO. Such information helps to validate/verify whether the requester is eligible for this data access or not. The requester requires to meet all the requirements set by the MPA to get the data access. This can be seen in Algorithm 2.

Algorithm 3 Authenticate Request

```
1: Input: File, Request, GrantBool
2: Require: (MPAOnly and !File.Granted)
3: if (GrantBool) then
4:     Request.MPAAuthCount ← Request.MPAAuthCount + 1
5:     Emit: inform DR and DO of new MPAAuthCount
6: end if
7: if (Request.MPAAuthCount >= File.MPAAuthRequiredCount) then
8:     Emit: broadcase Request to oracles
9:     File.Granted ← true
10: end if
```

The MPA handles the authentication process by enabling each authority to verify each claim made by the requester. If the number and validity of the data are legitimate, then the request is entertained, as shown in Algorithm 3. The method for calculating the rating of oracles is to map the latency, whose value varies in a range between 1 second - 1 hour/3,600 seconds, to 65,535 - 1. Note that ”65,535” is the maximum value that can be stored in the uint16 data type.

Algorithm 2 shows the procedure for accessing the file, when the DR initiates a request. The DR needs to state its address, file information, and the number of oracles required to be queried. The higher number of oracles ensure high quality; however, they also impose extra costs. The lower you set the range of oracles, it leads to degrading the quality. It is important to note that in this scenario quality is referring to the throughput/latency of the oracle and its reputation. The said request is responded to DR through the DO smart contract, after ensuring that the requester and the file have the corresponding authentication according to Algorithm 3. Subsequently, an event is triggered and sent out for oracles so they can register to retrieve the file.

Algorithm 4 presents respond of oracles with their encrypted results. The hash retrieved is compared to the original, and the latency is computed based on the time it took to respond. Based on the certain number of oracles specified by the DR, a certain threshold is set for time delay, where if the maximum is not reached, it would stop accepting results at that time. If the timestamp is passed and the minimum number of oracles are not satisfied, the time period is further extended until the minimum is achieved. A rating is then given based on the latency and correctness of results. Later on, the oracles are evaluated based on the given rating that leads to find their reputation scores. Note that the oracles having less reputations are eliminated from the system after some time. In a simpler term, the oracle having the best score is chosen. Tokens are then created and sent to the chosen oracle and the DR.

Tokens contain a unique identifier and the address of the counterpart. Once the DR gets the token, it knows the address of the chosen oracle and thus the DR can contact it. Token ID is generated from hashing the requester address, oracle address, and the timestamp information. After the DR gets the data, it submits a rating based on the parameters defined in Algorithm 5. The old reputation ranking is updated based on the rating of the DR, rating of the smart contract, and the old reputation.

Algorithm 4 Add *Oracle* Response

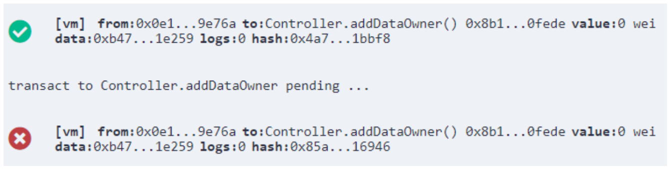
```
1: Input: File, Request, Hash
2: Require: (OracleOnly and File.Granted)
3: Latency ← Now - Request.Time
4: if (OracleCount < OracleMin) or (OracleCount >= OracleMin and OracleCount < OracleMax and notTimeoutBool) then
5:     HashResultBool ← File.Hash == Hash
6:     Rating = HashResultBool × $map$(Latency, From: $[1, 3600$ seconds], To: $[65535, 1]$ )
7:     add oracle to list of oracles of the request
8: end if
9: if (OracleCount >= OracleMin and TimeoutBool)
10:     or (OracleCount == OracleMax) then Reputations[] ← Reputations(Oracles)
11:     Ratings[] ← []
12:     BestOracleAddress ← 0x0
13:     BestOracleScore ← 0
14:     for $i$ ← 0 ... OraclesCount do
15:         OracleScore = Ratings[Oracle] × (Reputations[Oracles] + 1)$^2$
16:         if OracleScore >= BestOracleScore then
17:             BestOracleScore ← OracleScore
18:             BestOracleAddress ← OracleAddress
19:         end if
20:     end for
21:     SubmitContractOracleRatings(Oracles, Ratings)
22:     TokenID ← keccak256(Doctor || Oracle || Now)
23:     Emit: Token(TokenID, BestOracleAddress) to Doctor and Token(TokenID, Doctor) to Oracle
24:     Evaluated ← true
25: end if
```

Algorithm 5 Submit DR *Oracle* Rating

```
1: Input: oracleAddress, tokenID, rating
2: Require: (valid tokenID and valid oracleAddress)
3: averageDRRequesterRating ← contractRatingCount ∗ averageContractRating + rating / contractRatingCount + 1
4: DRRatingCount←DRRatingCount+1
```

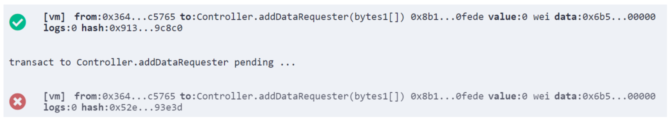
B. Validation

Let us take a sample example with different test cases to better understand the functioning of the system and validate



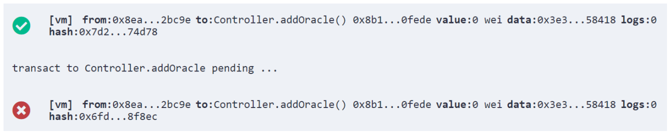
```
[vm] from:0x0e1...9e76a to:Controller.addDataOwner() 0x8b1...0fede value:0 wei
data:0xb47...1e259 logs:0 hash:0x4a7...1bbf8

transact to Controller.addDataOwner pending ...

[vm] from:0x0e1...9e76a to:Controller.addDataOwner() 0x8b1...0fede value:0 wei
data:0xb47...1e259 logs:0 hash:0x85a...16946
```

Figure 4: Registering a *DataOwner* to an existing *DataOwnerFile* smart contract, but failed.



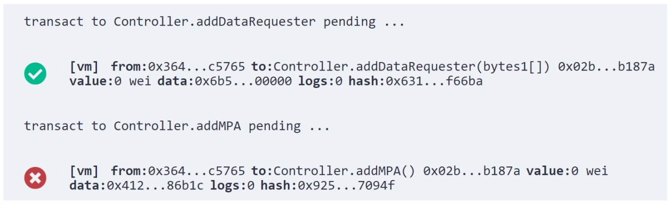
```
[vm] from:0x364...c5765 to:Controller.addDataRequester(bytes1[]) 0x8b1...0fede value:0 wei data:0x6b5...00000
logs:0 hash:0x913...9c8c0

transact to Controller.addDataRequester pending ...

[vm] from:0x364...c5765 to:Controller.addDataRequester(bytes1[]) 0x8b1...0fede value:0 wei data:0x6b5...00000
logs:0 hash:0x52e...93e3d
```

Figure 5: Registering a *DataRequester* twice using the same address, but failed.



```
[vm] from:0x8ea...2bc9e to:Controller.addOracle() 0x8b1...0fede value:0 wei data:0x3e3...58418 logs:0
hash:0x7d2...74d78

transact to Controller.addOracle pending ...

[vm] from:0x8ea...2bc9e to:Controller.addOracle() 0x8b1...0fede value:0 wei data:0x3e3...58418 logs:0
hash:0x6fd...8f8ec
```

Figure 6: *Oracle* tries to register twice with same address, but failed.



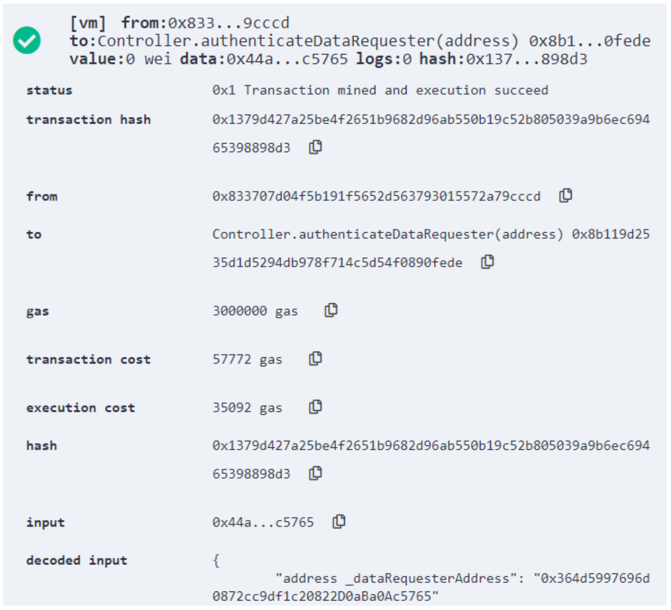
```
transact to Controller.addDataRequester pending ...

[vm] from:0x364...c5765 to:Controller.addDataRequester(bytes1[]) 0x02b...b187a
value:0 wei data:0x6b5...00000 logs:0 hash:0x631...f66ba

transact to Controller.addMPA pending ...

[vm] from:0x364...c5765 to:Controller.addMPA() 0x02b...b187a value:0 wei
data:0x412...86b1c logs:0 hash:0x925...7094f
```

Figure 7: *DataRequester* tries to pose as an *MPA* to exploit the system, but failed.

that the design goals are met. The testing environment to execute such an example is designed in the following way: The DO and the DR are the main entities providing the basis for all the interactions. The MPA process is managed by three authorities. Three re-encryption oracles are involved in our system architecture. The DO uses two files to explore the different test routes. Such two files require different privileges to be accessed. The addresses of the participants are shown in each of the provided figures across the validation section to follow through and fully understand the interaction process between the different entities.

Before starting this scenario, we first look at the enrollment of the actors into the system. The smart contract deployed by the DO is exclusive to him/her. The smart contract associates the ”owner” of the contract to the address of the deployer, so no other DO can have the same ”DataOwnerSC”, as shown in Figure 4. The DO can add several files to be accessed through the smart contract, when doing so the owner adds the corresponding privileges of each file given to the DR of the specified attributes.

It is important to note that the validation is required for the DR as well as for getting access to each file. This is important as each file could have different access privileges and requirements. Therefore, the first phase of the validation

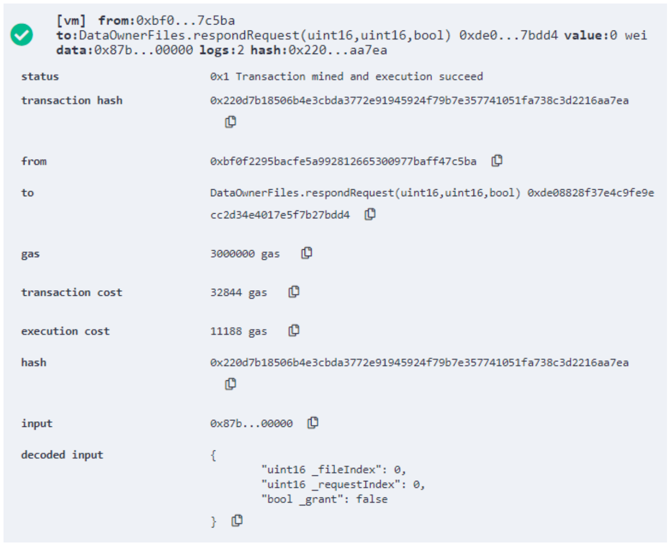


```
[vm] from:0x833...9cccd
to:Controller.authenticateDataRequester(address) 0x8b1...0fede
value:0 wei data:0x44a...c5765 logs:0 hash:0x137...898d3

status               0x1 Transaction mined and execution succeed
transaction hash     0x1379d427a25be4f2651b9682d96ab550b19c52b805039a9b6ec694
                     65398898d3
from                 0x833707d04f5b191f5652d563793015572a79cccd
to                   Controller.authenticateDataRequester(address) 0x8b119d25
                     35d1d5294db978f714c5d54f0890fede
gas                  3000000 gas
transaction cost     57772 gas
execution cost       35092 gas
hash                 0x1379d427a25be4f2651b9682d96ab550b19c52b805039a9b6ec694
                     65398898d3
input                0x44a...c5765
decoded input        {
                            "address _dataRequesterAddress": "0x364d5997696d
                     0872cc9df1c20822D0aBa0Ac5765"
```

Figure 8: Authenticating *DR* using an *MPA*.

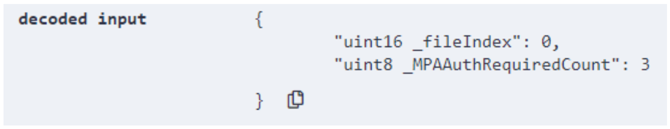
would be for the DR that has the minimum requirements to access the files of DO. It uses ”authenticateDataRequester” function to perform the validation, as shown in Figure 8. The second phase is responsible for checking whether or not the DR meets the requirements defined to access a certain file. This would be in the response to a request as shown in Figure 9. The DO can not be any of the other participants to avoid conflict of interest and to ensure separation of concerns. The same goes for other participants where they can not pose as different participants. They also can not take a role more than once; for example, an oracle can not register itself twice. Something similar can also be seen for other participants, an example can be seen in Figure 5 and Figure 7. Another example of an oracle trying to register twice with the same address is stopped preventing a whitewash of reputation and potential DoS attack from that channel, as can be seen in Figure 6.

As mentioned previously, the DR is authenticated by fulfilling the minimum requirements, so the concerned MPA entities submit an $authenticateDataRequester()$ function with the address of the DR, as shown in Figure 8. Before a request is initiated, it is important to ensure that the files have an associated number of each MPA member to verify attributes to access the file, as shown by the helper function in Figure 10. Now, when the DO initiates a request to access the file, the MPA sends a response through the smart contract (i.e., either grants or denies the access), as shown in Figure 9. The DR does not require to pass the verification procedure as this is something variable that is set by the MPA and the DO. Two claims out of three might be enough and some claims could be mandatory while others can be used in place of other claims. Figure 11 shows several responses from the MPA, which need to be directed towards the smart contract. Also, Figure 11 shows that file 1 gets only one permission; however, it requires permission from two entities. Thus, no



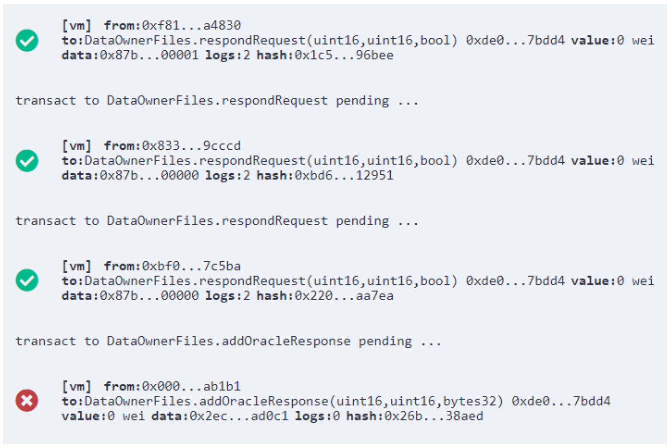
```
[vm] from:0xbf0...7c5ba
to:DataOwnerFiles.respondRequest(uint16,uint16,bool) 0xde0...7bdd4 value:0 wei
data:0x87b...00000 logs:2 hash:0x220...aa7ea

status              0x1 Transaction mined and execution succeed
transaction hash    0x220d7b18506b4e3cbda3772e91945924f79b7e357741051fa738c3d2216aa7ea

from                0xbf0f2295bacfe5a992812665300977baff47c5ba
to                  DataOwnerFiles.respondRequest(uint16,uint16,bool) 0xde08828f37e4c9fe9e
                    cc2d34e4017e5f7b27bdd4
gas                 3000000 gas
transaction cost    32844 gas
execution cost      11188 gas
hash                0x220d7b18506b4e3cbda3772e91945924f79b7e357741051fa738c3d2216aa7ea

input               0x87b...00000
decoded input       {
                            "uint16 _fileIndex": 0,
                            "uint16 _requestIndex": 0,
                            "bool _grant": false
                    }
```

Figure 9: Authenticating file access eligibility of *DR*.



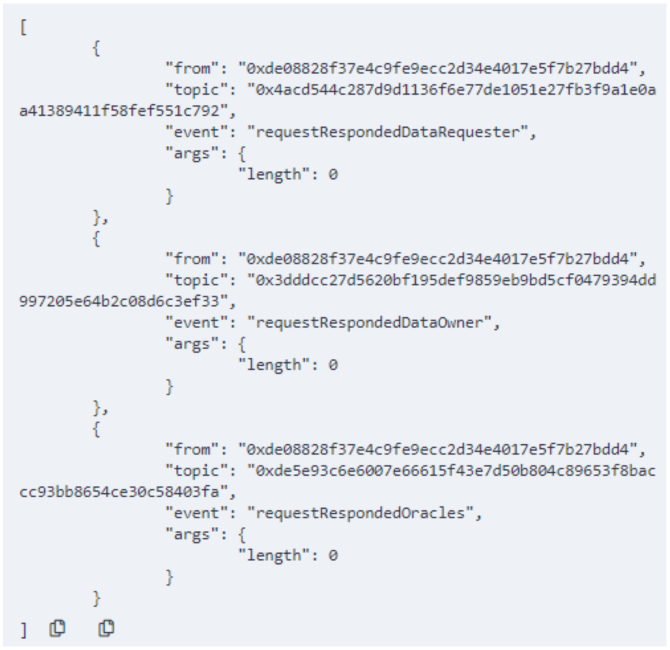
```
decoded input           {
                                "uint16 _fileIndex": 0,
                                "uint8 _MPAAuthRequiredCount": 3
                        }
```

Figure 10: Number of claim verification needed to access a particular file.



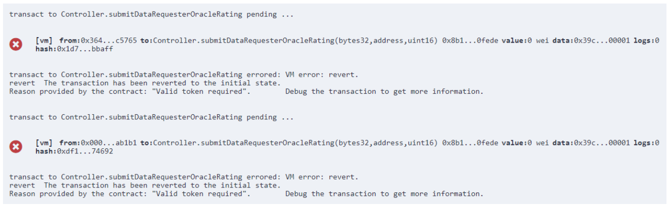
```
[vm] from:0xf81...a4830
to:DataOwnerFiles.respondRequest(uint16,uint16,bool) 0xde0...7bdd4 value:0 wei
data:0x87b...00001 logs:2 hash:0x1c5...96bee

transact to DataOwnerFiles.respondRequest pending ...

[vm] from:0x833...9cccd
to:DataOwnerFiles.respondRequest(uint16,uint16,bool) 0xde0...7bdd4 value:0 wei
data:0x87b...00000 logs:2 hash:0xbd6...12951

transact to DataOwnerFiles.respondRequest pending ...

[vm] from:0xbf0...7c5ba
to:DataOwnerFiles.respondRequest(uint16,uint16,bool) 0xde0...7bdd4 value:0 wei
data:0x87b...00000 logs:2 hash:0x220...aa7ea

transact to DataOwnerFiles.addOracleResponse pending ...

[vm] from:0x000...ab1b1
to:DataOwnerFiles.addOracleResponse(uint16,uint16,bytes32) 0xde0...7bdd4
value:0 wei data:0x2ec...ad0c1 logs:0 hash:0x26b...38aed
```

Figure 11: Denying *Oracle* participation for a rejected request.



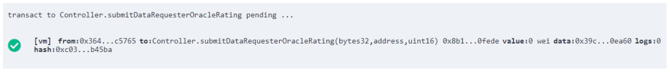
```
[
        {
                "from": "0xde08828f37e4c9fe9ecc2d34e4017e5f7b27bdd4",
                "topic": "0x4acd544c287d9d1136f6e77de1051e27fb3f9a1e0a
a41389411f58fef551c792",
                "event": "requestRespondedDataRequester",
                "args": {
                        "length": 0
                }
        },
        {
                "from": "0xde08828f37e4c9fe9ecc2d34e4017e5f7b27bdd4",
                "topic": "0x3dddcc27d5620bf195def9859eb9bd5cf0479394dd
997205e64b2c08d6c3ef33",
                "event": "requestRespondedDataOwner",
                "args": {
                        "length": 0
                }
        },
        {
                "from": "0xde08828f37e4c9fe9ecc2d34e4017e5f7b27bdd4",
                "topic": "0xde5e93c6e6007e66615f43e7d50b804c89653f8bac
cc93bb8654ce30c58403fa",
                "event": "requestRespondedOracles",
                "args": {
                        "length": 0
                }
        }
]
```

Figure 12: Sharing events once the request is granted access so *DR* and *DO* are notified.



```
transact to Controller.submitDataRequesterOracleRating pending ...

[vm] from:0x364...c5765 to:Controller.submitDataRequesterOracleRating(bytes32,address,uint16) 0x8b1...0fede value:0 wei data:0x39c...00001 logs:0
hash:0x1d7...bbaff

transact to Controller.submitDataRequesterOracleRating errored: VM error: revert.
revert  The transaction has been reverted to the initial state.
Reason provided by the contract: "Valid token required".       Debug the transaction to get more information.

transact to Controller.submitDataRequesterOracleRating pending ...

[vm] from:0x000...ab1b1 to:Controller.submitDataRequesterOracleRating(bytes32,address,uint16) 0x8b1...0fede value:0 wei data:0x39c...00001 logs:0
hash:0xdf1...74692

transact to Controller.submitDataRequesterOracleRating errored: VM error: revert.
revert  The transaction has been reverted to the initial state.
Reason provided by the contract: "Valid token required".       Debug the transaction to get more information.
```

Figure 13: Invalid doctor rating submissions.



```
transact to Controller.submitDataRequesterOracleRating pending ...

[vm] from:0x364...c5765 to:Controller.submitDataRequesterOracleRating(bytes32,address,uint16) 0x8b1...0fede value:0 wei data:0x39c...0ea60 logs:0
hash:0xc03...b45ba
```

Figure 14: Success submitting a valid rating token.

operation can be performed. In the second case, file 2 gets multiple permissions, and thus the process continues to be executed. The DR also employs proxy re-encryption oracles in this process to encrypt the files. In this way, the DR is given a choice to keep the balance between cost, quality, and time.

The concerned entities are notified through certain events. When the grant is done, an event is sent to DO, DR, and oracles, as shown in Figure 12. After that, the oracles start informing the smart contract that they want to participate in fulfilling a certain request. The oracles are mainly chosen based on the accuracy of the retrieved hash, latency, and reputation scores. After the best oracle is chosen and it fulfills its service, the DR requires to finish the interaction by submitting back a rating regarding the service received. To avoid illegitimate ratings, the DR is asked to provide the token for the oracle and is checked if it is a valid token or not. There is also a second case, where the token exists, but it is not linked to the right oracle in which it is being rejected by the smart contract, as shown in Figure 13. When the token does exist and matches the right oracle, it is being accepted, as can be seen in Figure 14. Subsequently, the reputation score is updated and the interaction gets finished.

V. Analysis and Discussion

In this section, we evaluate our proposed system using various performance measuring parameters including cost, security, and generalization. Besides, we present several limitations of our proposed solution acting as open research challenges.

Table I: Gas costs of smart contract functions.

Function	Executor	Transaction Cost	Execution Cost	USD
Deploy: Controller	Mediator	1,109,478	797,250	$ 3.55
Deploy: DataOwnerFiles	*DO*	1,317,140	956,324	$ 4.21
addDataOwner()	*DO*	44,192	22,920	$ 0.14
addDataRequester()	*DR*	49,628	28,036	$ 0.16
addMPA()	*MPA*	55,014	28,742	$ 0.18
addOracle()	DOF	87,955	66,683	$ 0.28
addFile()	*DO*	82,353	58,713	$ 0.26
setMPAAuthReqCount()	*MPA*	47,182	25,590	$ 0.15
authenticateDR()	*MPA*	66,463	51,343	$ 0.21
requestFile()	*DR*	143,612	121,316	$ 0.46
requestResponse()	DOF	42,557	20,858	$ 0.14
addOracleResponse()	*Oracle*	77,217	53,513	$ 0.25
addOracleResponse()	*Oracle*	103,549	94,845	$ 0.33
submitOracleToken()	DOF	92,095	65,895	$ 0.29
SubmitDROracleRating()	*DR*	49,640	24656	$ 0.16

A. Cost Analysis

In the Ethereum blockchain, smart contracts are executed on EVMs. The EVM is paid in return for its services. Ethereum gas is a unit used to measure the price for running or executing certain operations. Table I shows both the transactions and their execution costs. The transaction costs were those for establishing the transactions on the blockchain; whereas, the execution costs were compensation for the EVM execution of the operations. The major costs were imposed by the deployment functions as can be seen in Table I. The smart contracts are deployed once for each DO, so this is not a recurring cost, unlike the functions. The functions were mostly priced at below $ 0.3; however, the requestFile() function was priced at $ 0.46 that it is relatively higher in price when compared to other functions (excluding deployment functions). The presented gas price is set at 20 Gwei, which is safely above the current price that fluctuates between 10 and 15 Gwei.

B. Security Analysis

Herein, we present the security analysis of our proposed approach using important parameters, as discussed below.

- **Integrity and traceability**: The distributed ledger of the proposed blockchain-based solution acts as an immutable evidence for all the transactions recorded on it. It provides traceability features for access control related events. The proposed solution assures users that the data stored on it can not be modified or tampered with, thereby enforcing trust. Our blockchain-based solution maintains a chain of hashes, wherein each block uses the hash of the previous block. Therefore, altering one bit in a block leads to invalidating all the blocks that follow it.
- **Privacy**: The users in our proposed system can interact without compromising on their privacy. Although the EA is used to mask the identity in the Ethereum network, we further extend it in our implementation by only exposing needed data without breaching the privacy of users. There is no direct communication between the DO and DR. Early interactions are performed through the smart contract and logged on the chain. The later interactions are mediated through tokens that share the public addresses of the entities. The whole process of encryption and decryption only requires the public keys and the decryption is done at the DR's endpoint using the private key.
- **Authenticity**: Two-way authentication method is used to perform additional authentication. The EA acts as the identifier for each participant on the network. However, still the data remains visible to third parties most of the times for authentication purposes. Thus, a link is required to establish for off-chain interactions, and this can be achieved using tokens. Even if there are off-chain interactions that the blockchain do not mediate, tokens are sent to the entities so that they can authenticate each other. This helps to avoid potential impersonation attacks. On the other hand, reputation systems are also vulnerable to certain types of security attacks including ballot stung and bad-mouthing attacks [18]. Such attacks can be mitigated by the use of tokens, where each DR only submits one rating for one oracle. There is no real incentive for an oracle to carry out a Sybil attack as only one oracle is chosen and it mainly relies on the latency of that oracle. A denial of service (DoS) attack is one of the hardest to tackle. One of the possible ways to mitigate it is that there is a limit to the maximum number of oracles that would respond, but for a network with a large number of nodes it would be very challenging to completely prevent this type of attack. In the case of unreliable oracles that do not provide the required services, the DR can use the token to inform the network about such a case. After that, the reputation score of the oracle will go down due to its malicious action. In this way, it will be eliminated from the system after some time.

C. Generalization

Even though the starting point was examining organizations that prioritize the confidentiality of data; however, the implemented solution can be seen as a general template that encompasses all kinds of confidential data exchange. Any entity that values confidentiality and privacy can incorporate our solution regardless of the application domains.

The proposed framework can be seen as a foundation that dictates the exchange of data policy of a larger system. The proposed solution can be used for different intelligence agencies that need solid access control governance. To incorporate the different use cases and generalize them in a solution, different concerns should be taken into account. Performance, cost, privacy, integrity, and authenticity are all parameters that concern a user and should be adjustable to match their needs. To achieve a general solution, it might be a good initiative to make components of the system optional, where if there is an interaction between users that know each others' attributes, then they should be able to exclude the MPA to decrease cost and increase performance. This might prove to be useful in a system of a modest size where the participants are static, such as start-up businesses. Choosing who does the encryption, who acts as the MPA, how much power is given to the DO are all important aspects that must be studied if such a solution needs to be incorporated. One of the major things to be adjusted might be the hierarchical access policy, which dictates the privileges based on the already established hierarchy in the system.

D. Open Challenges

- The oracles were implemented to alleviate the overload on the blockchain in terms of cost and size. They help to increase the performance with their off-chain actions. They also incorporate a reputation system to mitigate the effects of centralization and make decentralization a viable option, which might be seen as making the system more complex.
- Some security concerns are not fully mitigated as they are very challenging. For example, the DoS attack for a large network would be challenging to prevent. Even though whitewashing is slightly mitigated by being given an average reputation; however, it is still something that might be exploited.
- To completely deploy a solution, it would be favorable to fully incorporate the re-encryption proxies and analyze the behavior of the system as a whole to be assessed in a better manner that would reflect the feasibility of shipping such a solution.
- A multi-signature wallet was not implemented but doing so would also decentralize the authority power at the endpoint. This would prove useful in cases where the DO does not have enough power to claim a file for himself. To ensure that nothing happens without the supervision of those entities they would be involved in a multi-signature wallet that would be needed to do any action with the file.
- The development and deployment of such a large scale solution would need a comprehensive study and analysis. It would be a challenge to fully assess how the solution can be generalized for the different use cases unless it is tested in the different domains, which would take extensive efforts.
- The blockchain network with the different entities interacting on it has been implemented as a software; however, the re-encryption oracles as physical entities have not been fully integrated with the solution. This could be something to be added in the future.

VI. Conclusion

In this paper, we have proposed a fully decentralized blockchain-based multi-party authorization (MPA) solution to provide provenance of log events related to access/permissions in a manner that is immutable, auditable, trustful, and secure. We proposed implementing proxy re-encryption using multiple oracles to give access to encrypted shared data stored on a public and decentralized storage platform, such as the IPFS. We incorporated reputation mechanisms in the proposed smart contracts to give ratings to the oracles based on their malicious and non-malicious behaviors. We developed Ethereum-based smart contracts to implement the functions, modifiers, and trigger events. Our smart contract code is made publicly available on GitHub, and it generic enough as it can be implemented on both types of permissioned and permissionless blockchain networks with minimal modifications based on the specific needs of industries. We presented the proposed algorithms and the system components along with their full implementation, testing, and validation details. We presented the cost analysis to verify the affordability and practicality of the proposed solution. We conducted a security analysis to verify the reliability of the proposed approach. As a future work, we aim to implement our solution using private blockchain platforms, such as Hyperledger Fabric and Hyperledger Besu. In addition, we plan to develop frontend decentralized applications (DApps) for the end-users.

VII. Acknowledgement

This publication is based upon work supported by the Khalifa University of Science and Technology under Award No. CIRA-2019-001.

References

[1] [Accessed on: April 08, 2020]. [Online]. Available: https://pages.riskbasedsecurity.com/hubfs/Reports/2019/2019MidYearDataBreachQuickViewReport.pdf

[2] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[3] J. Verble, "The nsa and edward snowden: surveillance in the 21st century," *ACM SIGCAS Computers and Society*, vol. 44, no. 3, pp. 14–20, 2014.

[4] WIKIPEDIA, "Multi-party authorization," 2019, [Accessed on: 05 August 2020]. [Online]. Available: https://en.wikipedia.org/wiki/Multi-party_authorization

[5] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, "Fairaccess: a new blockchain-based access control framework for the internet of things," *Security and Communication Networks*, vol. 9, no. 18, pp. 5943–5964, 2016.

[6] D. Tith, J.-S. Lee, H. Suzuki, W. Wijesundara, N. Taira, T. Obi, and N. Ohyama, "Application of blockchain to maintaining patient records in electronic health record for enhanced privacy, scalability, and availability," *Healthcare Informatics Research*, vol. 26, no. 1, pp. 3–12, 2020.

[7] N. Aitzhan and D. Svetinovic, "Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, pp. 1–1, 10 2016.

[8] H. Al Breiki, L. Al Qassem, K. Salah, M. H. U. Rehman, and D. Sevtinovic, "Decentralized access control for iot data using blockchain and trusted oracles," pp. 248–257, Orlando, FL, USA, 2019.

[9] H. Hu, G.-J. Ahn, and J. Jorgensen, "Multiparty access control for online social networks: model and mechanisms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 7, pp. 1614–1627, 2012.

[10] P. Ilia, B. Carminati, E. Ferrari, P. Fragopoulou, and S. Ioannidis, "Sampac: Socially-aware collaborative multi-party access control," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, Scottsdale Arizona, USA, 2017, pp. 71–82.

[11] H. Guo, E. Meamari, and C.-C. Shen, "Multi-authority attribute-based access control with smart contract," in *Proceedings of the International Conference on Blockchain Technology*, Honolulu HI USA, 2019, pp. 6–11.

[12] C. Chinchilla, "A Next-Generation Smart Contract and Decentralized Application Platform," 2019, [Accessed on: 23 April 2020]. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper/

[13] D. Bryson, D. Penny, D. Goldberg, and G. Serrao, "Blockchain technology for government," *Montgomery, AL: The MITRE Corporation*, 2017.

[14] "Besu Enterprise Ethereum Client," 2020, [Accessed on: 23 March 2020]. [Online]. Available: https://besu.hyperledger.org/en/stable/

[15] "Quorum Whitepaper," 2018, [Accessed on: 23 March 2020]. [Online]. Available: https://github.com/jpmorganchase/quorum/blob/master/docs/Quorum\%20Whitepaper\%20v0.2.pdf

[16] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," 2014, [Accessed on: 23 March 2020]. [Online]. Available: https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf

[17] "Remix Docs description," [Accessed on: 03 April 2020]. [Online]. Available: https://remix-ide.readthedocs.io/en/latest/#

[18] K. Hoffman, D. Zage, and C. Nita-Rotaru, "A survey of attack and defense techniques for reputation systems," *ACM Comput. Surv.*, vol. 42, no. 1, Dec. 2009. [Online]. Available: https://doi.org/10.1145/1592451.1592452