2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

Blockchain-Based, Decentralized Access Control for IPFS

Mathis Steichen, Beltran Fiz, Robert Norvill, Wazen Shbair and Radu State
University of Luxembourg, SnT, SEDAN
29, avenue J.F. Kennedy, L-1855 Luxembourg
{mathis.steichen, beltran.fiz, robert.norvill, wazen.shbair, radu.state}@uni.lu

***Abstract*—Large files cannot be efficiently stored on blockchains. On one hand side, the blockchain becomes bloated with data that has to be propagated within the blockchain network. On the other hand, since the blockchain is replicated on many nodes, a lot of storage space is required without serving an immediate purpose, especially if the node operator does not need to view every file that is stored on the blockchain. It furthermore leads to an increase in the price of operating blockchain nodes because more data needs to be processed, transferred and stored. IPFS is a file sharing system that can be leveraged to more efficiently store and share large files. It relies on cryptographic hashes that can easily be stored on a blockchain. Nonetheless, IPFS does not permit users to share files with selected parties. This is necessary, if sensitive or personal data needs to be shared. Therefore, this paper presents a modified version of the InterPlanetary Filesystem (IPFS) that leverages Ethereum smart contracts to provide access controlled file sharing. The smart contract is used to maintain the access control list, while the modified IPFS software enforces it. For this, it interacts with the smart contract whenever a file is uploaded, downloaded or transferred. Using an experimental setup, the impact of the access controlled IPFS is analyzed and discussed.**

I. Introduction

Blockchain applications interact either directly with blockchains or with smart contracts in order to achieve consensus on transactions, data or code execution. A network of heterogeneous nodes stores the blockchain, processes transactions and, if necessary, executes smart contracts. This leads to the following issue when working with large data files. Because the files are usually not required for the blockchain nodes to function, the blockchain becomes bloated, resulting in data being replicated on a large amount of nodes.

On one hand, storing large files on the blockchain is inefficient. Limitations on the block size require files to be split and reassembled off-blockchain. Additional data relevant to reassembling files would also have to be stored, requiring either even more space or a distinct system that provides the reassembly information. If smart contracts are leveraged to directly store file parts, the data can more easily be accessed and the reassembly information could be stored as well. However, sending and storing large files, even partially, using smart contracts is expensive (for example regarding gas costs) and needs to be executed at every mining or verifying node.

On the other hand, operating the mining nodes becomes more expensive. More data needs to be propagated through the network, processed and stored by the node. Mining nodes would thus require connections with higher bandwidths and more storage space to store the blockchain, even partially, thus leading to increased costs.

It concludes that blockchains are not the right platform to share and store large files. Fortunately, file sharing platforms can be leveraged to support applications while keeping the blockchain small in size. Users can efficiently share large files and still benefit from the blockchain. Cryptographic hashes that serve to securely identify a file's content, can be send to the latter, thus proving that the file was available to someone at a certain time. One particularly interesting file sharing platform for this purpose, combining file sharing and the mentioned hashes, is the InterPlanetary File System (IPFS) [1]. IPFS identifies, verifies and transfers files relying on the cryptographic hashes of their contents.

Similar to public blockchains, files stored on ipfs can be requested and viewed by anyone who can connect to or deploy an ipfs node. This is an issue for blockchain applications working with large files that contain sensitive or personal data. Therefore, this paper leverages the Ethereum blockchain [2] to provide an access controlled IPFS. An Ethereum smart contract stores and allows dynamic modification of the access control list. The modified IPFS software, hereinafter named acl-IPFS, whose design and implementation is discussed in the following, can then connect to the smart contract and enforce the permissions given by the access control list. Acl-IPFS allows users to register new files, and grant and revoke permissions by forming and sending transactions to the smart contract. With every request for a file, acl-IPFS nodes provide the public key and sign the message using a linked Ethereum account. This creates a relation between the nodes and the account, thus allowing the nodes to rely on the smart contract to request permissions and enforce them.

This paper is structured as follows. Section II discusses related work. Section III gives an overview of ipfs, describes the access control smart contract and its implementation. Section IV describes experiments conducted with acl-IPFS and illustrates the results. A use case for acl-IPFSis presented in Section V, and this work concludes with Section VI.

II. Related work

This section reviews related work which leverages blockchain-based data storage approaches and access control

978-1-5386-7975-3/18/$31.00 ©2018 IEEE
DOI 10.1109/Cybermatics_2018.2018.00253

methods. Blockchain systems that focus on privacy are also considered.

A. Blockchain-based data storage solutions

Blockchain technology provides an immutable data storage in that it allows transaction to only be appended and never to be modified or removed. However, the data storage on blockchain has a cost model that differs from conventional data storage in terms of size and cost. For instance in terms of size, the Bitcoin blockchain [3] provides the `OP_RETURN` *opcode* to store arbitrary data in transactions. It was limited to 80 bytes which has been reduced to 40 in February 2014. In term of cost, storing 80 bytes on the Bitcoin blockchain costs roughly US$0.03617 and US$0.007 when using Ethereum [4]. As described, blockchain transactions hold only very small amounts of data, wherefore it is crucial to choose what data should be placed *On-chain* and what should be kept *Off-chain*.

There are several Off-chain data storage solutions designed to be friendly to blockchain, such as Storj[1], FileCoin[2], Sia[3], and IPFS[4]. These solutions share the idea of peer-to-peer distributed file system, where the data is shredded, encrypted, and distributed to multiple nodes in the network to ensure their safety and availability. One main issue with these systems is the lack of access control. These features are extremely important if blokchain-based applications equipped with off-chain data storage solutions are deployed in operational and sensitive environments such as the financial and public sectors.

B. Blockchain-based access control approaches

Traditional access control mechanisms are based on centralized databases containing user identities and their access rights. However, this method has many issues with distributed systems across multiple sites [5]. The authors in [6] provide a survey of decentralized access control mechanisms in distributed file systems intended for large-scale environments.

In the context of blockchain technology, various studies [7], [8], [9] investigate using blockchain as an access control manager for distributed systems (e.g. cloud computing). Zyskind et al. [7] propose a decentralized privacy platform for personal data that is collected and controlled by a third-party. The solution enables this by combining a blockchain with an off-chain storage solution. When a user signs up to use a service, a new identity described by a tuple, user and service, is created. The identity contains signing key-pairs for both, and a symmetric key used to encrypt and decrypt the data. The blockchain verifies the signature for either the user or the service and checks whether the service is granted permission to access the data, then provides the hash to retrieve the data from the off-chain storage.

In [8] the authors propose an identity and access management system for cloud federations. The system guarantees the integrity of the policy evaluation process by using the Ethereum blockchain and Intel SGX trusted hardware. The main benefit of using blockchain is to to ensure that users identity attributes and access control policies cannot be modified by a malicious user, while Intel SGX protects the integrity and confidentiality of the policy enforcement process.

Maesa et al. [9] use blockchains to manage the rights to access resources and to transfer them from one user to another through blockchain transactions. The proposed approach has been implemented on the Bitcoin blockchain.

All the aforementioned studies use blockchains to provide access control features for cloud resources. The work of Zhang et al. [10] follows the same approach but in the context of Internet of Things (IoT). They propose a smart contract-based framework to implement distributed and trustworthy access control. Multiple access control contracts (ACCs) have been implemented on Ethereum blockchain to manage the access control between multiple subject/object pairs, one judge contract (JC) for judging the misbehavior of the subjects, and one register contract (RC) for managing the ACCs and JC.

The main difference between these works and this paper is that off-chain data storage resources are targeted herein. To achieve that access control features are integrated into IPFS, an open source file sharing platform, that can be used as off-chain storage. At the same time, a smart contract provides and manages the access control list.

C. Privacy

Certain blockchains focus heavily on being able to transact financially with assurances of anonymity and secrecy. Monero makes transactions unlinkable to a particular account by deriving a key pair per transaction from the user's mast key. In this way the user has control over the account, but that account is not visibly linked to the owner. It uses ring signatures in a process similar to mixing to hide a user's transaction amongst others (whether the other have been spent or not does not matter [11]). Doing so makes it more difficult to associate accounts with a particular user with accounts based on transactions.

Zcash also focuses heavily on anonymity and secrecy. It makes use of zero knowledge, succinct, non-interactive arguments of knowledge to be able to show, with an acceptably small chance of being lied to, that a particular address owns a particular amount. Thanks to the cryptographic methods used this can be done without revealing the sender, receiver or amount to the system at large [12] [13].

Both of these cryptocurrencies have dispensed with scripts. As a result they do not have smart contracts. While they currently lack the capability of smart contracts required to produce the system discussed in this paper, their provision of privacy may hold some useful applications, for example in the financial world where a blockchain is used by a number of parties that shall not all have access to all the information.

III. Architecture and Implementation

This section provides a description of IPFS, elaborates on the Ethereum smart contract that manages the access control

[1] https://storj.io/
[2] https://filecoin.io/
[3] https://sia.tech/
[4] http://ipfs.io/

list and presents the modifications done to IPFS that allow users to selectively share files.

A. The Inter-Planetary File System (IPFS)

The InterPlanetary File System, IPFS [1] is a decentralized file sharing platform that identifies files through their content. It relies on a distributed hash table (DHT) to retrieve file locations and node connectivity information.

The following example illustrates how IPFS functions. When a file is uploaded to IPFS, it is split into chunks[5], each containing at most 256 kilobytes of data and/or links to other chunks. Every chunk is identified by a cryptographic hash, also named content identifier, that is computed from its content. The mentioned links also contain content identifiers, thus forming a Merkle directed acyclic graph (Merkle DAG) that describes the file as a whole and can be used to reconstruct any file from its chunks. Because of the Merkle DAG, an entire file can be identified by just using the root hash. Once a node has divided the file into chunks, and the Merkle DAG has been formed, the node registers itself as a provider by means of the DHT.

The DHT is essentially a distributed key-value store. As described in [14], it uses node identifiers and keys - both need to have the same length - together with a distance metric to easily store and retrieve information. When looking for a value, a node attempts to find nodes that are close to the key and requests the value from them. It does that by using buckets to keep track of nodes within the network. The buckets are organized such that any node of the network has precise information of its near environment. However, a node's knowledge of other nodes decreases as the distance increases. Thus, in order to find the value associated with a key, a node contacts a node that is closer to the key than it is itself. The latter replies either by returning the value, or by referring to nodes that are even closer to the key. This continues until the key is found. To write a value, a node determines a predefined number of nodes that are closest to the key and informs them of the value of the key. This allows nodes to efficiently write values to and retrieve them from the network. Keys, however, are only valid for a certain amount of time, for example 24 hours, and need to be updated regularly to be kept in the DHT.

IPFS employs S/Kademlia [15] as its DHT. It stores two different types of information. First, whenever a file is uploaded through a node, the latter registers itself as a provider of the file's chunks. Second, the DHT contains information on how to connect to a node with a specific identifier, for example by providing an IP-address. Thus, an IPFS node can request providers through the DHT and then connect to them to retrieve a file.

Due to IPFS' use of content identifiers to identify, verify and transfer chunks and files, it is particularly suitable to be used with blockchains. In fact, the root hash of a file's Merkle DAG can be sent to the blockchain using a transaction, thus not bloating the latter. At the same time, no information other than the hash is required to retrieve the file from IPFS. This is different to file sharing systems that do not rely on cryptographic hashes as content identifiers, because they require additional name resolution systems to obtain a file whose hash is found in the blockchain. Furthermore, a second, distinct file with the same hash cannot easily be created, thus making it impossible to flood IPFS with files with a given target file identifier. To conclude, a file served through IPFS can easily be verified and it is difficult to thwart a user by providing different files with the same name/identifier.

[5]In IPFS, the chunks are referred to as blocks. However, in order to more easily distinguish between IPFS blocks and blockchain blocks, the former are called chunks in this paper.

B. Ethereum and the access control smart contract

Ethereum is one of the world's most prominent blockchain systems. It allows users to make financial transactions in its native cryptocurrency ether and custom currencies called tokens. It is set apart by its extensive integration and use of smart contracts. They allow code to be executed, and the results to be stored with the same assurances as financial transactions can be made on the blockchain. Ethereum's Virtual Machine reads a custom form of bytecode that can be compiled from various languages, most commonly Solidity. Ethereum smart contracts are Turing complete, as such they can be utilized for a large number of purposes. Ethereum is stateful, code executed on the blockchain can change the state of system, which for example depends on account balances and contract storage. This allows the implementation of a smart contract based access control system, with addition, removal and changes in permissions being recorded on the blockchain, and forming part of Ethereum's state.

The benefits of using a smart contract for access control are the same as the benefits for financial transactions. Firstly, decentralization ensures that no entity has control over the execution. As such the code execution can be trusted, with any resulting changes being agreed upon by consensus. Secondly, the code cannot be tampered with or changed given that Ethereum is (largely) append only due to consensus and proof of work. This work utilizes the Byzantium version of Ethereum, at the time of writing it is the latest hard fork. It provides the `revert` opcode, and a status field to receipts which indicates the success or failure of a transaction. Acl-IPFS makes use of these features.

The contract storing and managing the access control list is written in Solidity and compiled using compiler version 0.4.20. Solidity is statically typed and makes use of functions, variables and structures [16]. The contract consists of one structure, one mapping and several functions. It makes use of the implicitly available `msg.sender` which contains the address of the user sending the transaction.

The structure named `FileData` and defined in the contract contains one variable of type address, called `owner`, a mapping called `access` and a dynamic array of type address that is called `allowedAddresses`.

The contract contains a mapping from a variable of type bytes32 to an instance of the structure `FileData`, called `fileMapping`. It it used heavily in the contract, its purpose

is explained in the function description below. Many of the functions have a corresponding version with 'Multiple' appended to the function name. 'Multiple' functions make use of arrays and call the corresponding single version multiple times.

All of the functions can be called externally by the user. Each function can be passed input data from the user as an argument. It is validated by the function using if-statements. In the event that the checking condition resolves to false the revert function is called. Revert rolls the contract back to its previously held state. Functions that do not change Ethereum's state can be declared as constant. Constant functions do not require a paid transaction to execute as they are analogues to simply reading data, and make no changes to contract storage. Such functions reduce the amount of work required of nodes as the functions can be called locally instead of broadcasting a transaction to be included in a block. The access control contract contains 4 such functions: amIOwner, amIOwnerMultiple, checkAccess, checkAccessMultiple.

addBlock & AddBlockMultiple: This function is used to register an IPFS file chunk in the access control list. Chunks are identified by their cryptographic hashes. The function accepts a hash as a value of type bytes32 as its only argument. It checks that it is not empty and that the hash does not already have a record denoting its owner. If either of these checks fail, revert is called. Otherwise, the chunk's hash is used as a key in the `fileMapping` mapping, with the value being a new instance of the `FileData` structure, which has its value for `owner` set to `msg.sender`. This new data is committed to the contract's storage. The 'Multiple' version of the function accepts an array of byte32 values and calls `addBlock` for each one.

grantAccess & grantAccessMultiple: Accepts an address and a bytes32 variable representing a chunk hash as argument. It reverts if the address is empty (has the value 0x0) or the hash is empty. Otherwise, it recalls the relevant data from storage by accessing `fileMapping`, using the hash argument as the key. It checks that `msg.sender` matches the owner recorded in the instance of `FileData` accessed via `fileMapping`. If it does not match, revert is called. If it matches, the `access` mapping has the address argument added as a key and the corresponding boolean is set to true. Lastly, the address argument is added to the `allowedAddresses` array and kept in the contract's storage. The 'Multiple' version accepts one address and an array of bytes32 type values. It calls `grantAccess` for each element in the array.

removeAccess & removeAccessMultiple: This function and its 'Multiple' version operate in the same way as the `grantAccess` function pair, the only difference being that it sets the boolean in `fileMapping` to false.

amIOwner & amIOwnerMultiple: Accepts a bytes32 value representing a chunk hash as an argument. If this is an empty value, it returns false. If not, it returns true if `msg.sender` equals the owner listed in `fileMapping` with the chunk hash as the key, and false otherwise. The 'Multiple' version accepts an array of bytes32 values, calls `amIOwner` for each one and returns an array containing the results.

checkAccess & checkAccessMultiple: Accepts one address and one bytes32 value representing a chunk hash as arguments. It returns false, if either value is empty and returns true, if the address argument matches `owner`, or if the address has previously been granted access. If the address has not been granted access, false is returnd. The 'Multiple' version accepts one address and an array of chunk hashes, it calls `checkAccess` for each value in the array.

deleteBlock & deleteBlockMultiple: Accepts one bytes32 value as an argument. It reverts if the hash is empty or if `msg.sender` is not the owner of the chunk identified by the hash. If the checks pass, it iterates over the `allowedAddresses` array and sets all booleans to false. In a second step, it assigns a new, empty instance of `Filedata` to the data in `fileMapping` for the given hash. In effect, all accesses and the owner are removed. The file can no longer be requested by any user. Deletion of the actual data is left to individual nodes, as is required by IPFS.

C. Requirements for sharing critical data using IPFS

In order to extend IPFS with an access control mechanism, the following requirements need to be considered.

Adding permissions. After a file is added to acl-IPFS, no users except the owner have access to it. Thus, the owner needs to be able to easily grant access permissions to other users for the file to effectively be shared. Ideally, for ease of use, an owner can grant permissions to a particular user for multiple files or file chunks at once.

Revoking permissions. Even though blocks in IPFS are immutable, and this is still the case in acl-IPFS, granted permissions also need to be revoked. This is necessary if regulations for example limit the duration for which a non-owning user can view a file. Similar to granting permissions, it is convenient to allow an owner to revoke a user's access to multiple files or file chunks at once.

Collaboration in the network. Even though an acl-IPFS node might not leak any files, it is still necessary for the operators of nodes to collaborate on some level. It is crucial that participants that obtain files and therefore the sensitive information are incentivized not to circumvent the system by sharing the information through other means. This, however, is an issue with every file sharing system used for sensitive information and needs to be controlled through collaborative agreements that are outside the scope of this work. It is henceforth assumed that such agreements exist and govern the behavior of participants in the acl-IPFS network.

Transferring files. Files are not allowed to be sent to any node that is not permitted to view the file. Therefore, every node needs to have an identity and it needs to be capable of proving its identity if necessary. This is the case whenever a node requests a chunk of a file.

Compatibility with IPFS. In addition to the requirements regarding sensitive information, it is also necessary to allow the IPFS functionality to operate largely without restriction.

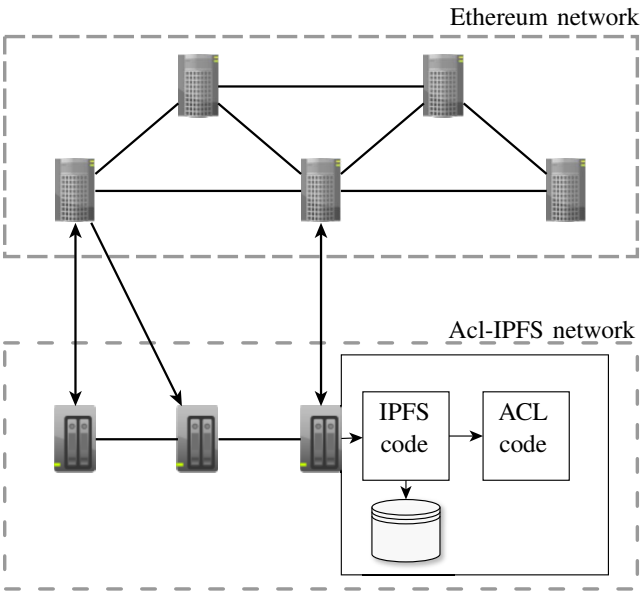

Fig. 1. The architecture of the acl-IPFS network. It is composed of Ethereum nodes executing the smart contract that handles the access control list and acl-IPFS nodes that enforce the permissions. One of these nodes is shown with the modified IPFS software, the local storage and the permissions package, referred to as acl code, that interacts with the blockchain.

Thus, an additional goal is to use as much of the code of IPFS as possible, while still achieving the requirements regarding access control. This is achieved by adding access control in specific, target locations. It results that the DHT and the data stored in it are completely unaffected. Furthermore, unless the access control list does not allow sharing a particular file, the exchange system is also largely unmodified.

D. Architecture

Figure 1 depicts the architecture of the acl-IPFS network. The modified IPFS software interacts with the *permissions* package, which then communicates with the smart contract through a blockchain node. It is therefore able to send transactions to the smart contract's methods that are described in Section III-B. The IPFS code manages its local storage, interacts with the network and with the DHT.

1) Adding a file: Whenever a node uploads a file, as per IPFS the node creates chunks and the associated Merkle DAG. During this process all content identifiers corresponding to the file are collected. These are then passed on to the permissions package, which registers the files in the smart contract, using the `addBlock` functions. This is done by forming the transactions necessary for this purpose. The permissions system furthermore allows the verification of transactions based on the corresponding content identifiers. If the transactions succeed, the execution returns to the IPFS code which stores the blocks in the local storage and then registers itself as a provider of the content identifiers. If the transaction fails, ownership of at least one of the content identifiers and therefore the whole file could not be claimed. Thus, an error is returned that interrupts the execution and therefore the storing of the file within the acl-IPFS node's storage and the registration as a provider.

While attempting to claim a content identifier, an adversarial node that sees the corresponding transaction, can try to claim the same content identifier, thus obtaining the ability to set its permissions through the smart contract. Although this can happen, it is not certain that the adversarial node is successful, because the two transactions compete and the first one to be included into the network renders the second one obsolete, due to verifications performed by the smart contract. Even so, the acl-IPFS determines whether or not a transaction has been executed successfully and can decide to interrupt the process. In that case, the legitimately owning node does not become a provider, does not store the blocks in its local storage and will therefore not be able to serve any chunks of the file. Thus, the actual owner, being the only one having the file, can choose to modify it, for example by prepending random bytes, and attempt the adding process again, eventually being able to successfully register as its owner. The successfully hijacked content identifiers serve no purpose, as nodes other than the owner do not have access to the original file and cannot serve chunks of it.

It is worth mentioning that transactions are limited in size, wherefore it is not always possible to fit all corresponding content identifiers in one transaction. If that is the case, the content identifiers can be registered in several transactions. Acl-IPFS then keeps track of all of these transactions to be able to inform on the success of the process as a whole.

2) Granting permissions: Acl-IPFS allows users to easily grant access permissions to specified users through the command line. For this purpose, a transaction is also formed and sent to a blockchain node. Now, however, since the content identifier has already been registered, only the owner, identified by its Ethereum account, can grant or revoke a permission. As previously described, a condition in the smart contract ascertains that this is the case.

3) Retrieving a file: If the required permissions have been provided, a node is able to request and obtain the chunks corresponding to the files. For this purpose, it determines providers of these chunks in the network and connects to them to obtain the chunks needed to reassemble the file.

If the permission has not been granted, or if it has already been revoked, the request will not be answered. Anyone connecting to an Ethereum node can verify a user's permissions regarding any content identifier and thus make the appropriate decision.

4) Linking the smart contract data to the acl-IPFS nodes: Although IPFS nodes also have an identity, it is not used in this paper. The drawbacks of using the IPFS nodes' identities are discussed in the following. On one hand, although it is possible to use the smart contract to linking IPFS and Ethereum identities, it is expected to require additional storage and processing to be performed by the Ethereum mining nodes. On the other hand, multiple nodes operated by one entity would require distinct permissions to be able to access a file. It would thus be more complex and more expensive to set

up all the necessary permissions. Nonetheless, the user is still capable of selecting this option by using multiple Ethereum accounts, if he or she wishes to keep the data only on a selected subset of its nodes, for example.

When sending a request, a node retrieves the public and private keys of its linked Ethereum account. With this information it signs the request and appends the public key. Other nodes can use this additional data to verify the sent public key, from which the address can be derived, thus verifiably identifying a participant in the network.

E. Implementation

The implementation of the aforementioned permissions package is discussed next.

As Golang implementations of Ethereum and IPFS exist, this is the programming language that best suits the needs of this work. This allows the use of go-ethereum's *abigen* that is capable of compiling Solidity code and creating a go package that allows easy interaction with the smart contract.

The permissions package is implemented with flexibility in mind. It provides an interface that can be implemented using different blockchains or access control systems, without having to adapt the already modified IPFS code.

As described in Section III-D, the go permissions package keeps track of transactions. The functionality in the package can verify the successful execution of a transaction using the corresponding content identifier(s), thus not requiring depending packages to handle implementation details, such as transaction hashes. Additionally, the package can wait until transactions associated to one or several specified content identifiers have been mined and reports on the state of the content identifiers. However, the permissions package needs to clear data regarding old transactions to make sure that the used memory space does not become too large. For this purpose, whenever instantiating a new instance of the permissions class, a go routine is started. It is executed concurrently to other go code, and is triggered regularly to remove data related to old transactions that are assumed either to have passed or failed.

The described go package interacts with the functions described in III-B and selects the appropriate kind, singular (e.g.: grantAccess) or multiple (e.g.: grantAccessMultiple).

IV. Experimental Results

This chapter focuses on the evaluation of the proposed decentralized access control version of IPFS (acl-IPFS) in regards to its efficiency as a file sharing system when compared to the original IPFS.

All experiments are performed by deploying IPFS version 0.4.13, acl-IPFS and an Ethereum node in docker [6] containers. The Ethereum container is limited to use only one CPU core, which leaves enough processing power to handle the IPFS-based containers. Before using acl-IPFS, the access control smart contract is deployed and its address informed to the acl-IPFS nodes.

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

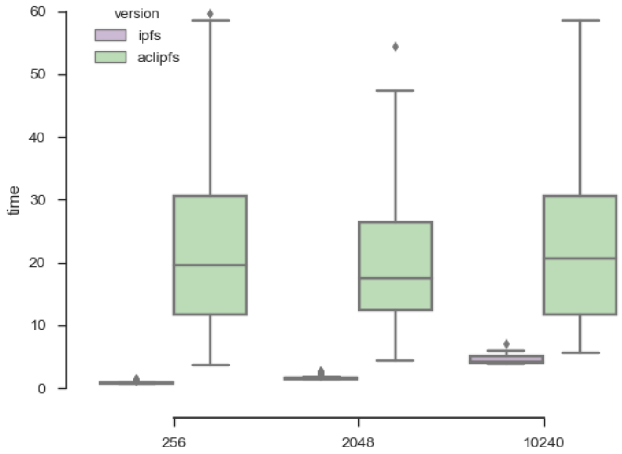

Fig. 2. Boxplot for comparing adding files of 100 files three different sizes (256 kB, 2MB and 10MB) to IPFS and acl-IPFS.

In the following, two tests are performed. First, a latency test shows the additional delay caused by the interaction with the smart contract. Second a stress test is performed to determine the maximum number of file operations that the system is capable of handling in a given slice of time.

A. Latency Test

The goal of the latency test is to compare the delay taken in acl-IPFS and IPFS when adding a file into the IPFS network.

The tests are performed using three different file sizes: small (256 kB), medium (2 MB) and large (10 MB). These sizes are based on the fact that an IPFS file chunk by default is set to 256 kB.

For each of the proposed file sizes, 100 random files are generated. This yields an array of execution times for each file size. The result is shown in Figure 2 as a boxplot.

As expected, the delay displayed by the acl-IPFS add file operation is seconds slower on average than the standard IPFS operation. This is due to the fact that in acl-IPFS a file is only considered once the transaction registering the file's ownership has been received and mined successfully by the Ethereum network.

For the use-case presented in Section V, the additional delay added by the decentralized access control list does not present any major drawbacks.

In addition, while these values might seem relatively large in comparison with standard IPFS, it is worth noting that these tests are performed on a file by file comparison. In reality, users would send multiple file operations at once and have the ability to have most of them mined within the same block. The limits of this are shown in the stress test below.

B. Stress Test

The stress test is designed to determine how many operations can be performed on ACLIPFS given the constraint that

an operation such as adding a file requires a transaction to be mined into a block.

In the genesis file of the private Ethereum network, a gas limit is defined in order to fix a maximum amount of gas all the transactions in the block may use. Miners have the option to increase this value slightly between blocks, allowing the gas limit to slowly grow or shrink as needed. This effect can be reduced by using the `targetgaslimit` flag when setting the ethereum nodes in order to force the system to tend towards a determined value.

In this experiment the access control smart contract is deployed in a private Ethereum network and then targeted with 100,000 *add file* operations. This acl-IPFS operation generates a transaction in the private Ethereum network with the hash address of the IPFS file as an input parameter.

The gas cost for every sent transactions was 50171 or 50235. This means that, given the initial gas limit setting of 134217728, between 2671 and 2675 add operations can be added in each block. Given that a block takes on average 14 seconds, this yields around 190 operations per second.

V. Use case: Know your customer

A. KYC

Financial institutions in the EU must comply with legislation for Know Your Customer (KYC). The phrase refers to the requirements for banks and financial institutions to know their customers' identities through a process of verification. This takes the form of costly document collection and analysis. For financial institutions operating within the EU, recent legislation has mandated stricter rules around KYC. The latest General Data Protection Regulation (GDPR) asserts the right of the user to have control over, and access to their data upon request, as well as the right to have their data shared with another company upon request [17]. The 4th Anti-Money Laundering Directive (AMLD) came into effect in 2017 [18]. It places stricter requirements on the checks that financial institutions must carry out, with a focus on checks for known terrorists and Politically Exposed Persons (PEP). It mandates the creation of national level registers containing the personal details of beneficial owners of companies and trusts.

Both pieces of legislation increase the cost of compliance for financial institutions operating within the EU. A large portion of this cost comes from the requirements for availability and sharing of data; sharing at the customer's request in the GDPR legislation, beneficial owners register in the 4th AMLD. As such, a system that could automate and control the flow of data and enable sharing, in-line with current legislation would save financial institutions a significant amount of time and money while ensuring compliance is achieved.

Blockchain and IPFS for KYC: Blockchain technology has a number of inherent properties that make it an attractive potential solution for modern KYC data sharing.

Distribution & consensus: The distributed nature of blockchain requires and ensures that no single entity is responsible for storing the ledger that makes up the chain. As such, records cannot be tampered with. Additions to the blockchain are made with the knowledge and agreement of the majority of the network. For KYC this means that if a document is stored on the chain at a given point in time, all participants in the system agree upon the contents of the document, when it was uploaded and by whom.

Append only: As adding a block to the chain is done by a majority consensus of nodes in the network, and each block is cryptographically linked to the previous one, once something has been added to the chain it cannot be modified. In terms of KYC this ensures documents cannot be changed or tampered with once they have been uploaded as no single entity has the power to change the agreed upon history of the chain.

Smart Contracts: For KYC, any process that is common across financial institutions can be automated by writing a program to carry out the process. The assurances provided by blockchain consensus mean the result of each execution can be trusted.

Blockchain's weaknesses: Blockchain's two main weaknesses in relation to data sharing and privacy are it's append only nature and the growth in size associated with this. The EU's GDPR legislation includes the 'right to be forgotten', article 17 states that "The data subject shall have the right to obtain from the controller the erasure of personal data concerning him or her without undue delay and the controller shall have the obligation to erase personal data without undue delay..." [17]. As blockchain relies on being append only in order to provide a true record of events, data cannot be removed from the chain. If personal data is stored directly on the chain this means that financial institutions would be unable to fully comply with the GDPR legislation. It is theoretically possible to change historical data in the chain by achieving a majority consensus on what changes should be made. However this would require an extremely large amount of computation, especially in large chains and those using proof-of-work. Each block is linked to the preceding block by a cryptographic hash, and any change in a block changes the hash and breaks the link with the next block. Even with consensus on the changes to be made all verifying nodes in the network would have to recompute the hashes for all blocks from the point of the change up to the present. If the chain is secured by proof-of-work this becomes even more computationally expensive. In a network with a lot of customers, and therefore a lot of removals, this would require constantly recomputing large portions of the chain, with the number of blocks that require recalculation increasing with the age of the data to be removed. Moreover, this lessens blockchain's usefulness for auditing purposes. Ideally, the chain should show that certain entities had access to a certain customer's data at a certain point in time, rather than simply erasing historical events from the chain. Storing all personal data on the chain would cause the chain to grow very rapidly, making it harder to store and search over time.

B. IPFS and moving data off-chain

The weaknesses of blockchain can be overcome by moving the files containing personal data off-chain. This ensures files

can be deleted as required and are not causing the chain to grow excessively fast. However, for data sharing between organizations the assurance that files have not been edited or changed must be retained. Moreover the blockchain tracks which entities have access to which data.

IPFS stores files in a distributed way by splitting them into chunks which can be requested and transferred between nodes. Each file is identified by it's cryptographic hash. Doing so makes it easy to ensure one has received the correct data by generating the hash of the concatenated file chunks and checking it matches the hash that was requested.

IPFS hashes identifying files that contain personal data can be stored on the blockchain instead of the data itself. Doing so enables compliance with GDPR legislation. The blockchain containing the hash ensures that the file has not been tampered with. The file itself being stored in IPFS means it can be deleted as required, by nodes removing it from their local storage. The hashes of files can be used to associate files with owners and access permissions. Chain growth is reduced as hashes are generally smaller than the data they represent, if SHA256 is used the on-chain storage required for a file of any size becomes 32 bytes. As such chain growth is vastly reduced.

Enforcing Privacy: IPFS is designed to share information as widely as possible and does not attempt to restrict connections or data flow between nodes. The scenario considered herein deals with private data. IPFS cannot be allowed to share files with any nodes that request them, it must only share files with those that have permission according to the permissions recorded on the blockchain. As such, the provided solution is a modified version of IPFS that uses a smart contract which is capable of adding, removing and updating file ownership and accesses. Entities are identified by their Ethereum public key and files are identified by their unique cryptographic hash.

The combination of blockchain assurance and a private-network capable IPFS allows for compliance with current data protection and sharing legislation and reduces the cost of doing so.

VI. Conclusion

This paper addressed the requirement of blockchain applications to share larger files containing sensitive information. As discussed, the files can neither be efficiently stored on the blockchain, or be uploaded through unmodified ipfs nodes.

For this purpose, the design and implementation of acl-IPFS, a blockchain-based extension to ipfs that provides access control, have been discussed. Acl-IPFS leverages Ethereum smart contracts to handle the access control list. Through the smart contract, users can register files, and grant or revoke access to them. The to this end modified IPFS software provides access to the smart contract and enforces the permissions stored in the access control list.

Furthermore, a use-case for acl-IPFS has been presented and experiments conducted in order to show the impact of the system with reference to the unmodified ipfs. The observed additional delay is mainly due to the interaction with the blockchain and has been shown to be noticeable, but insignificant to operations.

References

[1] J. Benet, "IPFS - Content addressed, versioned, p2p file system (draft 3)," https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf, 2014.

[2] G. Wood. (2018) Ethereum: A secure, decentralised generalised transaction ledger. [Online]. Available: http://yellowpaper.io/

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[4] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *2017 IEEE International Conference on Software Architecture (ICSA)*, April 2017, pp. 243–252.

[5] C. K. and T. M. V., *Distributed Access Control in Cloud Computing Systems*. Wiley-Blackwell, 2016, ch. 35, pp. 417–432. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118821930.ch35

[6] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis, "Decentralized access control in distributed file systems," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 10:1–10:30, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1380584.1380588

[7] G. Zyskind, O. Nathan, and A. Pentland, "Decentralizing privacy: Using blockchain to protect personal data," in *2015 IEEE Security and Privacy Workshops*, May 2015, pp. 180–184.

[8] S. Alansari, F. Paci, and V. Sassone, "A distributed access control system for cloud federations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 2131–2136.

[9] D. Di Francesco Maesa, P. Mori, and L. Ricci, "Blockchain Based Access Control," in *17th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, vol. LNCS-10320. Neuchâtel, Switzerland: Springer International Publishing, Jun. 2017, pp. 206–220, part 5: Making Things Safe (Security).

[10] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the internet of things," *CoRR*, vol. abs/1802.04410, 2018. [Online]. Available: http://arxiv.org/abs/1802.04410

[11] N. v. Saberhagen, "Crypto note v 2.0," *CryptoNote. org.[Online]*, vol. 17, no. 10, 2013.

[12] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 459–474.

[13] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 397–411. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.34

[14] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 53–65. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687801

[15] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 02*, ser. ICPADS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2007.4447808

[16] , "Solidity 0.4.21 documentation," http://solidity.readthedocs.io/en/v0.4.21/, 2018, [Online; accessed 12th April 2018].

[17] Council of European Union, "Council regulation (EU) no 2016/679," 2016, https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679.

[18] ——, "Council regulation (EU) no 2015/849," 2015, https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex\%3A32015L0849.