Peer-to-Peer Networking and Applications
https://doi.org/10.1007/s12083-020-00967-6

Themis: An accountable blockchain-based P2P cloud storage scheme

Yiming Hei[1] · Yizhong Liu[2] · Dawei Li[1] · Jianwei Liu[1] · Qianhong Wu[1]

Received: 1 March 2020 / Accepted: 14 July 2020

Abstract

Cloud storage is an effective way for data owners to outsource their data by remotely storing them in the cloud and enjoy on-demand high quality services. In traditional cloud storage systems, cloud data integrity verification relies on centralized entities and data is stored in a small number of storage servicers. However, these centralized entities and storage servicers may be untrustworthy, and malicious servicers may even refuse to perform a user's query or update request. Though a few blockchain-based themes have been proposed to address some of these problems, they do not achieve decentralization, accountability, flexibility and practicability simultaneously. In this paper, we present Themis, an accountable P2P cloud storage scheme with smart contracts on Ethereum. Our scheme has the following advantages: First, cloud data integrity verification is decentralized and implemented by miners on blockchain without any trusted third party. Second, by carefully setting up the reward and punishment mechanism within a smart storage contract, all rational nodes will participate in the storage service following an accountable rule. Third, based on reliable information published on the blockchain, users are free to choose appropriate storage servicers who want to share idle storage, making storage service decentralized and flexible. Fourth, compared with the existing related systems, by adopting a payment at maturity method, the malicious behavior of breaking the contract after the servicer obtains some revenue is prevented, and the availability of user data within the specified period is enhanced. Fourthermore, we implement a prototype of Themis on Rinkeby, an Ethereum test network. Extensive experimental results demonstrate that our scheme is able to support a PB-level data storage in a single P2P storage service at a low cost and is feasible for practical deployment. And the entire decentralized arbitration process takes only 40 to 110 seconds.

Keywords Cloud storage · P2P · Blockchain · Smart contract · Ethereum

✉ Dawei Li
lidawei@buaa.edu.cn

Yiming Hei
black@buaa.edu.cn

Yizhong Liu
liuyizhong@buaa.edu.cn

Jianwei Liu
liujianwei@buaa.edu.cn

Qianhong Wu
qianhong.wu@buaa.edu.cn

[1] School of Cyber Science and Technology, Beihang University, Beijing, China

[2] School of Electronic and Information Engineering, Beihang University, Beijing, China

1 Introduction

In the era of big data, users are more inclined to choose cloud storage as the main way of data storage [1, 2]. Cloud storage reduces the burden of local storage while facilitating data sharing. At present, enterprise users are the main target of cloud storage services. Services for individual users are not mature in terms of transmission efficiency and functionality. The emerging 5G [3] technology provides higher data transmission rate and lower transmission delay, allowing users to access cloud data at a higher speed. Therefore, it could be predicted that the development of cloud storage will be more rapid, especially for personal cloud storage.

However, there are still many problems restricting the popularity of cloud storage. In the following, we divide these problems into three categories and explain in detail the

Published online: 05 August 2020

definition of the problem, the existing solutions and their shortcomings.

1. **Centralization of Cloud Data Integrity Verification.** In order to prevent malicious cloud servicers from tampering with user uploaded data, data integrity verification is required [4]. Many cryptography-based solutions have been proposed to achieve cloud data integrity verification. Some of the solutions [5, 6] rely on users themselves to execute the verification, adding extra communication and computing costs to users. To reduce the burden on users, third-party-based solutions are proposed. Wang et al. [7] propose a general formal PoR model with public verifiability for cloud data, which supports fully dynamic data operations. A dynamic multi-replica provable data possession scheme [8] is proposed to achieve efficient data integrity verification by using techniques such as probabilistic homomorphic encryption and BLS signatures. Zhang et al. [9] design a scheme that supports batch verification and data dynamic operations. Storj [10] utilizes Proof of Retrievability to ensure data integrity and verifiers in this scheme are peers called Satellites. However, all the above solutions rely on a special center to perform verification. Once the verification center miscalculates or misbehaves, the legal rights of the participants will be impaired.

   Sia [11] adopts the M-of-N signature scheme to ensure the fairness of data services. It claims to support public data integrity verification, while implementation details are not given. As an incentive scheme for InterPlanetary File System (IPFS) [12], Filecoin [13] utilizes Proof-of-Spacetime as a mechanism to stimulate storage, and uses zk-SNARKs [14] technology to ensure that the integrity of data could be publicly verified. Currently, Filecoin is still in the process of implementation.
2. **Non-Accountability Problem of Denial of Service.** The cloud servicer may refuse to provide users with queries or data update services by pretending that they have not received the user's message. Even if a trusted third party is introduced, this problem will not be solved. Consider the following scenario: A user claims that he has sent a query (or an update) request to the cloud servicer. While the cloud servicer refuses to provide the service on the grounds that he does not receive the user's request. In this case, it is intractable to determine who should be to blame, even if there is a trusted third party.
3. **Centralization of Cloud Storage Servicers.** Today, only a few cloud servicers such as Google and Amazon are providing cloud storage services to most users, leading to a monopoly in the industry. The centralization of storage servicers may lead to the following problems: If the cloud servicer loses user data due to equipment failure or hacking activities, it is very difficult for the user to obtain satisfactory compensation. In order to solve the storage centralization problem, some P2P cloud storage projects based on blockchain [15] begin to sprout, among which the more famous ones include Storj, Sia and Filecoin. These projects motivate users on the network to rent out their own idle local storage to get service revenues. Smart contracts [16] are utilized to standardize storage behaviors. Not only do they achieve decentralized storage, but also conform to the idea of a shared economy. In general, the above schemes require cloud servicers to periodically provide proofs to ensure the security of the storage, increasing the amount of on-chain data and the computing burden of cloud servicers. In addition, they [10, 11, 13] utilize the micropayment channel [17] to solve the denial of service issue when querying, but they do not pay attention to this issue when updating, and servicers have chance to deliberately delete data after receiving some revenue and then reach a new deal with clients providing higher rewards.

1.1 Our contributions

In order to solve the above problems in cloud storage, we propose an accountable P2P storage scheme called Themis based on the current design idea of blockchain-based storage systems. We implement Themis on Ethereum test network.

An Accountable Blockchain-based P2P Storage Scheme. We propose Themis with novel smart storage contracts to enforce consensus among storage participants in P2P network in an accountable manner without any trusted third party. We elaborate storage smart contracts to regulate the behavior of both parties. A pub-sub model is used where storage servicers first publish storage contracts on the blockchain to declare their service, and then users choose an appropriate servicer according to the contract information. Under normal circumstances, data transmission is carried out off-chain. When a dispute arises, i.e., the request sent by a user has no reply or the response is incorrect, the participants request the storage contract to make a ruling. Moreover, users' update commands are recorded on the blockchain for service auditing. In addition, micropayment technology is used to support data sharing, while the access and payment records are also stored on the blockchain. Specifically, Themis has the following advantages:

- Decentralized cloud data integrity verification. We propose an efficient integrity verification algorithm that is publicly available, enabling automatic verification of data integrity. The method of integrity verification is decentralized and redundant on-chain proof operations is reduced compared with existing schemes based on blockchain.
- Accountable against malicious behaviors from servicers and users. Themis solves the non-accountability problem of denial of service existing in cloud storage system by a punishment mechanism. For servicers, they could not claim their storage capacity unconditionally. And assuming the servicers to be rational, they will provide the service honestly. For users, assuming the users to be rational, they always query data off-chain when servicers provide the service honestly. In addition, Themis realizes the reliable update of data.
- Decentralized servicers. Through a reward mechanism, rational nodes are motivated to provide storage service to other nodes. Users are free to choose storage servicers according to information on the blockchain.
- Payment at maturity. By converting the service reward method in exciting schemes [10, 11, 13] from off-chain micropayment to expiring settlement, the availability of user data within the specified period is enhanced, preventing malicious storage servicers from deleting data after receiving some revenue, and then reaching new service contract with customers providing higher potential revenue.

Implementation. We implement a full prototype of Themis on Rinkeby, an Ethereum test network and give a comprehensive evaluation of the system performance. Streamlined algorithms for data integrity verification and data update are adopted to lower the computational complexity of the algorithm. We test the costs of all functions within the smart contract and the time costs of data integrity verification. The experimental results show that our scheme is feasible and efficient, supporting a PB-level cloud data storage for a user in a peer-to-peer storage service process with low cost. And the entire decentralized arbitration process takes only 40 to 110 seconds. It should be noted that our scheme could be conveniently transplanted to any system that supports smart contracts.

1.2 Related work

Blockchain and Smart Contract. Introduced by Bitcoin [15], blockchain is a continuously growing public ledger where transactions or records are contained in blocks that are linked by cryptographically calculated hash values [18]. Blockchains could be classified into two categories: permissionless blockchains and permissioned blockchains [19]. In permissionless blockchains, nodes are free to join the network and contribute to the consensus as long as they complete some kind of proof, such as PoW (proof of work), PoS (proof of stake) or PoA (proof of activity). There is usually a consensus mechanism to guarantee that agreement is reached among all honest nodes. Formally speaking, a blockchain system has to satisfy the consistency and liveness property [20, 21] to guarantee safety. The consistency property ensures that honest nodes have the same view of the public ledger. As a result, decentralization is realized in permissionless blockchains while no trusted third party is in need. While the liveness property guarantees that valid transactions submitted by nodes will be processed ultimately.

Smart contract [16] is first introduced by Szabo in 1997. Szabo points out that computer codes could replace mechanical equipment to deal with complex transactions. This viewpoint does not receive attention at that time due to the lack of a credible operating environment. With the emergence and development of blockchains, smart contracts have been widely used in many fields: secure pub-sub (SPS) [22], new payment schemes [23], Data Rights Management [24], Internet of Things [25, 26] and P2P File System [27], etc. In addition, in order to enhance the practicability of smart contracts, many researchers begin to study the security of smart contracts and the data feed service for smart contracts.

Ethereum [28] is a platform supporting smart contracts where developers are encouraged to design next-generation distributed applications. In every participating node, there is one or several Ethereum virtual machines where smart contracts are running. Each operation executed on the virtual machines carries an inherent cost, denoted as gas.

Cloud Storage Based on Blockchains. The basic idea of blockchain-based cloud storage is to motivate all nodes to contribute their own idle storage space to obtain revenues in an untrusted environment. Representative schemes based on this idea include: Storj, Sia and Filecoin.

Based on the Ethereum platform, Storj is a cloud storage system that integrates platform, digital token and distributed application. The main roles in the system are users, storage nodes, Ethereum miners and Satellites. Users spend money in renting spaces from storage nodes. Ethereum miners are trying to become a block producer, and then package transactions into a block to gain the corresponding rewards. Satellites are specific auditors that send challenges to storage nodes with Heartbeat, validate the storage proofs,

and establish a reputation system according to the audit history of each storage node.

Sia is a decentralized cloud storage system similar to Bitcoin, aiming to compete with existing storage solutions in P2P and enterprise areas. There are three types of entities in a Sia system: miners, storage servicers and users. Miners receive rewards based on the PoW mechanism. Storage servicers earn revenues by renting their hard disks. Users obtain the data storage service through the system. The Sia system automatically generates a verification request for the data storage status at a fixed frequency, requiring storage servicers to provide proof of the correct storage of user data. Unlike Storj, there are no specific auditors in the Sia system. Any miner can independently verify the validity of proofs from servicers.

As the incentive layer of IPFS, Filecoin guarantees the security and stability of the storage process in IPFS. Filecoin consists of three types of entities: storage miners who are responsible for mining and storage; search miners who provide search capabilities; users that store and query data. Storage miners prove to users through Proof-of-Replication and Proof-of-Spacetime that they have correctly stored the users data for a continuous period of time. Unlike other schemes, Filecoin creates a storage and retrieval market to increase competition among storage servicers, offering users flexible options and concessional prices. However, Filecoin has not been launched officially. In the process of producing a proof of spacetime, the problem caused by the imbalance of the computational power of storage miners is not considered in Filecoin.

1.3 Organization

The rest of the paper is organized as follows. In Section 2, the system model and system components are given. Then, we present our scheme in Section 3 and provide security discussion in Section 4. In Section 5, we further analyze the experiment results and show the practicality of our schemes. Besides, we compare the performance of this scheme with existing related schemes. Finally, we give a conclusion in Section 6.

2 System model

We give a system overview and definitions of relative concepts in this section.

2.1 System overview

As Fig. 1 shows, our system involves three types of entities: miner, user and servicer. Miners try to solve the PoW puzzles to produce new blocks where valid transactions are recorded. Servicers could be companies or normal nodes that lease their storage space to earn revenues by publishing a smart storage contract on the blockchain. Users could conclude storage transactions with appropriate servicers according to their storage requirements and the parameters within servicers' smart storage contracts.

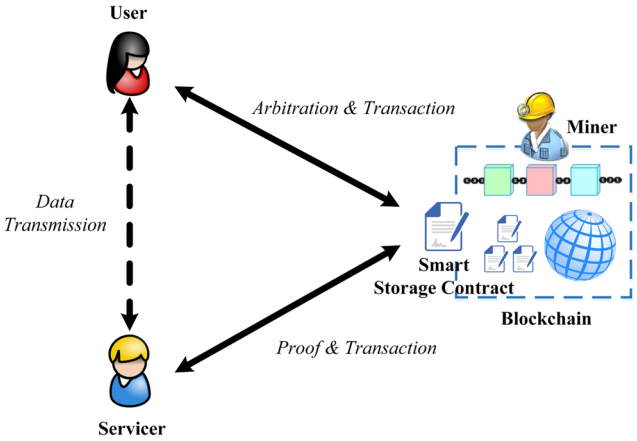


Fig. 1 The system model of Themis

After reaching an agreement through the smart contract on-chain, users divide their data to be stored into data blocks and use different symmetric keys to encrypt these data blocks for different servicers. Then, users send these encrypted data blocks with indices to the relevant servicers. Later, users are able to query their servicers for the stored data in an off-chain method. When a user does not receive a reply after a certain amount of time from his servicer, he could request arbitration to the corresponding storage contract. If the servicer fails to upload a correct proof to the contract within a specified time, the user will receive the pre-stored deposit of his servicer from the smart contract as a compensation. Besides, the data stored in the service provider could be updated according to the user's requirements.

2.2 Proof of storage

The servicer stores users data in the form of a Merkle tree [29] (Taking 8 data blocks as an example, the storage structure similar to Sia is shown in Fig. 2). It is important that the storage structure should be defined and exposed publicly by servicers on the smart contract before reaching an agreement.

The proof of storage is a list published by the servicer to prove the integrity of the data queried by users. The proof list includes a string-type data block D_c indexed by a challenge c , a hash list $\mathcal{H}$ containing the hash values on the path from D_c to the root node R_D. For example, in Fig. 2, the proof for challenge 5 is colored by red and $\mathcal{H} = [H_{0-3}, H_4, H_5, H_{67}]$.

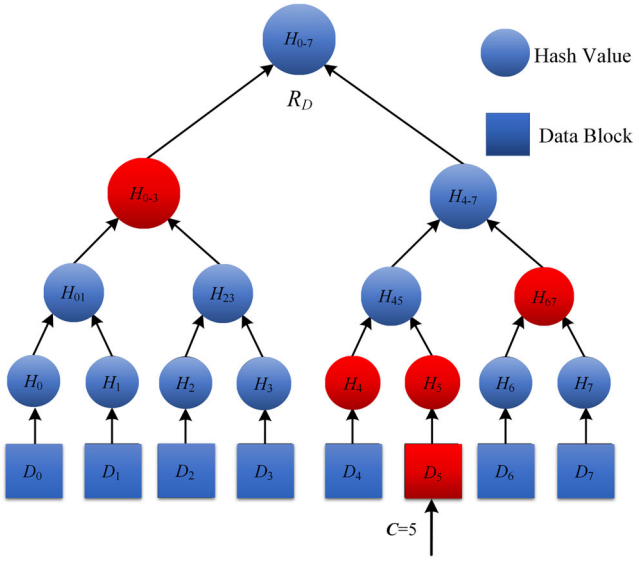


Fig. 2 The Merkle tree structure used for Proof of Storage

Miners run Merkle() (Algorithm 1) to calculate the result Res of the proof list, where $MaxSize$ denotes the total amount of data blocks, $newindex$ represents the index of the data block D_c's hash value in $\mathcal{H}$ and Hash() represents the keccak256 hash function. Specially, data blocks without data need to be filled with an empty string identifier such as "empty" to meet the requirement that $MaxSize$ has to be a power of 2.

Algorithm 1 Merkle.

Input:
a value c , a data block D_c , a hash list $\mathcal{H}$
Output:
a value Res

```
BEGIN
H_c := Hash(D_c)
if c ≥ 0 and c < MaxSize/2 then
    newindex := (c − (c + 1) mod 2 + 1)/2
end if
if c ≥ MaxSize/2 and c < MaxSize then
    c := c mod (MaxSize/2)
    newindex := 1 + (c − (c + 1) mod 2 + 1)/2
end if
if H_c == H[newindex] then
    Res := H[0]
    for i in 1 : H.length do
        Res := Hash(Res, H[i])
    end for
end if
END
```

2.3 Smart storage contract

In this section, we give a description of a smart storage contract to support the distributed file storage system without a trusted third party. The smart contract leverages Ether as money exchange between users and storage servicers. Our smart contract contains the transaction parameters and seven functions executed by miners publicly. The details of the smart storage contract are as follows.

2.3.1 Transaction parameters

The parameters used in smart contracts could be divided into two types.

Storage Service Parameters. These parameters are used to declare service capabilities and personal information of a storage servicer. Table 1 summarizes these parameters and their corresponding meanings.

User Parameters. The personalized rule parameters determined by each user. Table 2 shows these parameters and the corresponding meanings.

2.3.2 Functions

In this section, we only give the definition of the functional interfaces of the functions. Please see Section 3 for the details on algorithm implementation.

constructor(). Miners execute the function constructor() automatically to initialize a servicer's parameters when a storage contract is deployed on-chain. Given the volatility of the storage market price and to prevent a servicer from making claims about storage capacity (rent time and rent space) arbitrarily, we stipulate that a servicer cannot declare rent time and rent space in the contract at the same time. Therefore, a servicer only declares the storage time, while the storage space needs to be calculated according to the amount of deposit, storage time and storage unit price. A servicer who is the owner of the smart storage contract needs to transfer deposit through constructor() to claim its storage space implicitly.

concludeTrans(). The inputs of the function include a user's address on Ethereum, $T\_UserRent\_Begin$, $T\_UserRent\_End$, T_Prove_Limit, $Deposit\_User$, $DataRoot$, $Deposit\_User$ and $PubKey\_User$. This function could be invoked by any user or servicer to initialize the user's parameters in Table 2 and conclude a storage transaction. In detail, a servicer $\mathcal{S}$ triggers this function to upload parameters of a user $\mathcal{U}$ participating in a storage transaction. Then $\mathcal{U}$ also calls this function to upload his parameters. Finally, miners will check

Table 1 Storage service parameters

Parameter	Description
$T\_SerRent\_Begin$	The time when a servicer begin to rent out storage
$T\_SerRent\_End$	The time when a servicer finish renting out storage
T_Rent_Time	The total rent time of a servicer's storage space
$T\_Despoit\_Limit$	The time limit for deposit transfer
$Deposit\_Servicer$	The available deposit of a servicer
$Link\_Servicer$	The contact address of a servicer
$Pubkey\_Servicer$	The public key of a servicer
$Add\_Servicer$	A servicer's address on Ethereum
$Current\_User$	The address of a user within the latest transaction
$MapUsers$	The map recording the relation of a user's address and parameters
$MaxSize$	The number of data blocks

whether parameters from $\mathcal{U}$ and $\mathcal{S}$ are the same. If so, $\mathcal{U}$ and $\mathcal{S}$ could conclude the transaction with these agreed parameters.

update(). The inputs of this function include an index of data block to be changed, a new data block and a hash list that is the same as the hash list $\mathcal{H}$ mentioned in Section 2. A user calls this function to update his stored data.

finish(). The input of the function is a user's address. A servicer invokes this function to withdraw deposit when he finishes a storage transaction with a user and get the user's reward. If a transaction is invalid within stipulated time, the corresponding servicer calls this function to withdraw his deposit.

sendAsk(). The input of this function is an index of a data block. A user calls this function to send his query request and record it in the contract. Then he waits for a corresponding proof of storage from the servicer.

sendProof(). The inputs of this function include an address of a user who calls sendAsk(), and a proof of storage. A servicer invokes this function to send a proof of storage corresponding to the user's query. Then miners run Merkle() within the contract to verify the correctness of the proof. If the proof is correct, it returns the data queried by the user. If not, it returns error and deducts the servicer's deposit automatically.

proveTimeout(). A user calls this function to deduct a servicer's deposit if the servicer does not return a correct proof of storage within a specified time.

3 Concrete scheme

In this section, we present our concrete scheme, Themis. Themis consists of the following seven stages: Setup, Deal-OffChain, Deal-OnChain, Transmission, Update, Arbitration and Settlement. Figure 3 illustrates the detailed process of Themis, involving various entities, messages and sequences of events.

Table 2 User parameters

Parameter	Description
$T\_UserRent\_Begin$	The time when a user begin to rent storage
$T\_UserRent\_End$	The time when a user finish renting storage
$T\_Despoit\_Start$	The start time of deposit transfer
T_Ask_Begin	The time when a user begin to ask on-chain
$T\_Prove\_Receive$	The time to receive proofs from a servicer
T_Prove_Limit	The time limit for receiving proofs
$Flag\_Deposit$	The flag of deposit payment
$Flag\_Ask\_Receive$	The flag indicating that a user's ask has been received
$Flag\_Ask\_Permit$	The flag indicating that a user's ask has been permitted
$DataRoot$	The hash of a user's all data blocks
$PubKey\_User$	The public key of a user
$Deposit\_User$	The deposit of a user
Ask_index	The index of the data queried by a user

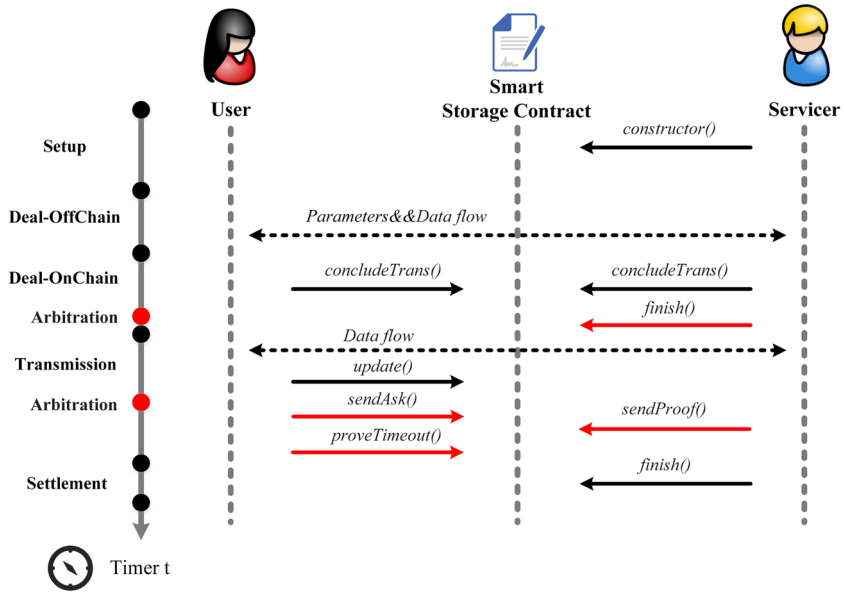


Fig. 3 Concrete process of Themis

3.1 Setup

Firstly, servicers and users register their accounts on Ethereum to obtain their key pairs and account addresses separately. Then, servicers publish storage contracts with initialized parameters ($Link\_Service$, $Pubkey\_Servicer$, $Add\_Servicer$, T_Rent_Time, $MaxIndex$, etc.). Finally, miners execute constructor() (see Algorithm 2 for details) automatically to initialize the other parameters in Table 1.

After this stage, servicers that publish storage contracts are ready to accept storage requests from users across the network.

Algorithm 2 constructor.

Input:
$\mathcal{S}.deposit$
1: BEGIN
2: $T\_SerRent\_Begin :=$ now
3: $T\_SerRent\_End := T\_SerRent\_Begin + T\_Rent\_Time$
4: $Add\_Servicer := \mathcal{S}.address$
5: $Deposit\_Servicer := \mathcal{S}.deposit$
6: $MapUsers := []$
7: END

3.2 Deal-OffChain

According to the information within storage contracts, users search for appropriate storage servicers. To improve the efficiency of matching, our scheme also supports third-party websites recommending servicers for users. In fact, in order to improve the robustness of storage, a user often chooses multiple servicers to store his data. For the sake of simplicity and clarity of description, we assume that a user $\mathcal{U}$ only finds a suitable servicer $\mathcal{S}$. Based on the storage service parameters of $\mathcal{S}$, i.e., $Link\_Servicer$ and $Pubkey\_Servicer$, $\mathcal{U}$ establishes a connection with $\mathcal{S}$ off-chain and negotiates user parameters of the new storage transaction. After reaching an agreement, $\mathcal{U}$ divides the data D into a block set $(D_1, D_2, \cdots, D_n)$ and obtains the set of encrypted data blocks $(E_1 = Encrypt(D_1, K_1), E_2 = Encrypt(D_2, K_2), \cdots, E_n = Encrypt(D_n, K_n))$, where K_i is the encryption key corresponding to the data block D_i. Then, $\mathcal{U}$ transmits the set of key-value pairs $[(1, E_1), (2, E_2), \cdots, (n, E_n)]$ to $\mathcal{S}$, where the key is a data index and the value is the corresponding encrypted data block.

3.3 Deal-OnChain

The Servicer $\mathcal{S}$ invokes the function concludeTrans() (see Algorithm 3 for details) with parameters negotiated with the user $\mathcal{U}$, which triggers the timing of the deposit transfer. Within the delay $T\_Despoit\_Limit$, $\mathcal{U}$ calls concludeTrans() with the same parameters and his deposit $Deposit\_User$. Then the miners compare the parameters of two parties. If their parameters are consistent, miners set $Flag\_Deposit$ to "true", indicating that the storage transaction begins to take effect.

Algorithm 3 concludeTrans.

Input:
$\mathcal{U}.address$, $\mathcal{U}.publickey$, a value dp, a value $DataRoot$, a timer $Tbegin$, a timer $Tend$, a timer $Tlimit$

1: BEGIN
2: **if** the caller is $\mathcal{S}$ **then**
3: $MapUsers[\mathcal{U}.address].(T\_Despoit\_Start, Deposit\_User, DataRoot, Pubkey\_Servicer, T\_UserRent\_Begin, T\_UserRent\_End, T\_Prove\_Limit, Flag\_Ask\_Receive, Flag\_Ask\_Permit) :=$ (now, dp, $DataRoot$, $\mathcal{U}.publickey$, $Tbegin$, $Tend$, $Tlimit$, $false$, $true$)
4: $Deposit\_Servicer \mathrel{-}= dp$
5: $Current\_User := \mathcal{U}$
6: **end if**
7: **if** the caller is $\mathcal{U}$ **and** $Current\_User == \mathcal{U}$ **then**
8: ***Assert***(now $< MapUsers[\mathcal{U}.address].T\_Despoit\_Start + T\_Despoit\_Limit$)
9: **if** the values in $MapUsers[\mathcal{U}.address]$ are the same with $\mathcal{U}$'s parameters **then**
10: $MapUsers[\mathcal{U}.address].Flag\_Deposit := true$
11: **end if**
12: **end if**
13: END

3.4 Transmission

Query. Since $\mathcal{S}$ and $\mathcal{U}$ have published their public keys on the blockchain, they establish a secure channel to transmit data off-chain through HTTPS or IPFS after their transaction takes effect. The query process is as follows: 1. $\mathcal{U}$ sends a index to $\mathcal{S}$; 2. Within a specified time, $\mathcal{S}$ needs to return the data block corresponding to the index.

Storage proof. This operation is optional. To confirm the integrity of the stored data, $\mathcal{U}$ could ask $\mathcal{S}$ to return its specified storage proof mentioned in Section 2 at any time off-chain. Then $\mathcal{U}$ triggers the Merkle() algorithm to verify the correctness of the proof locally. Obviously, since this operation is done entirely off-chain, it does not increase the storage burden on the blockchain.

Share. If a user $\mathcal{K}$ also wants to get $\mathcal{U}$'s data stored in $\mathcal{S}$, he should establish a micropayment channel with $\mathcal{S}$. Then $\mathcal{K}$ sends a index with $\mathcal{U}$'s tag (i.e., $\mathcal{U}.address||index$) to $\mathcal{S}$. And $\mathcal{S}$ returns the corresponding data block for rewarding. In general, $\mathcal{U}$'s data are encrypted before transmission, so only the users with $\mathcal{U}$'s decryption key could complete the decryption process and obtain the plaintext.

3.5 Update

If the user $\mathcal{U}$ wants to update his data D, he could call the algorithm update() (Algorithm 4) with a tuple ($index$, D_{new}, $\mathcal{H}_u$) as inputs, where $index$ is a index of data to be changed, D_{new} is the new data block and $\mathcal{H}_u$ is a hash list that is identical to the list $\mathcal{H}$ mentioned in Section 2. Then miners will run Merkle() to calculate a new $DataRoot$ for $\mathcal{U}$. And $\mathcal{S}$ should replace D_{index} with D_{new}.

In our example, the caller of update() is $\mathcal{U}$.

Algorithm 4 update.

Input:
a value $index$, a data block D_{new} , a hash list $\mathcal{H}_u$

1: BEGIN
2: $\mathcal{MS} := caller$
3: **if** $\mathcal{MS}$'s rent time hasn't expired **and** $index \geq 0$ **and** $index < MaxIndex$ **then**
4: $MapUsers[\mathcal{MS}.address].DataRoot :=$ Merkle($index$, D_{new}, $\mathcal{H}_u$)
5: **end if**
6: END

3.6 Settlement

Between time $T\_UserRent\_Begin$ and $T\_UserRent\_End$, if $\mathcal{S}$ responds correctly to all query requests from $\mathcal{U}$, he could take back his deposits worth $MapUsers[\mathcal{U}.address].Deposit\_User$ to $Deposit\_Servicer$ and get $\mathcal{U}$'s deposits as a reward by invoking finish() after $T\_UserRent\_End$ (see line 6-9 in Algorithm 5).

Algorithm 5 finish.

Input:
$\mathcal{U}.address$

1: BEGIN
2: **if** $\mathcal{U}$'s transfer of deposit exceeds $T\_despoit\_limit$ **and** $!MapUsers[\mathcal{U}.address].Flag\_Deposit$ **then**
3: $\mathcal{S}.Deposit\_Servicer \mathrel{+}= MapUsers[\mathcal{U}.address].Deposit\_User$
4: $Current\_User := 0x0$
5: **end if**
6: **if** $\mathcal{U}$'s rent time expires **and** $MapUsers[\mathcal{U}.address].Deposit\_User! = 0x0$ **then**
7: $\mathcal{S}.Deposit\_Servicer \mathrel{+}= MapUsers[\mathcal{U}.address].Deposit\_User$
8: $\mathcal{S}$ receives $MapUsers[\mathcal{U}.address].Deposit\_User$ as a reward
9: $MapUsers[\mathcal{U}.address].Deposit\_User := 0$
10: **end if**
11: END

3.7 Arbitration

Disputes might occur between users and their servicers in deal-onchain stage and transmission stage. Possible circumstances and corresponding arbitration methods are analyzed as follows:

Case 1: In the deal-onchain stage, we require the servicer to call concludeTrans() firstly, which will automatically deduct his deposits from $Deposit\_Servicer$. After $\mathcal{S}$ calls concludeTrans(), if $\mathcal{U}$ refuses to call concludeTrans() with his parameters and deposits within the time limit $T\_Despoit\_Limit$, then the transaction between $\mathcal{U}$ and $\mathcal{S}$ is invalid. In this case, $\mathcal{S}$ could call finish() (see line 2-4 in Algorithm 5) to withdraw his deposits.

Case 2: In the transmission stage, after $\mathcal{U}$ queries a data D_i or asks for a storage proof, if $\mathcal{S}$ returns a wrong result or refuses to response on time off-chain, then $\mathcal{U}$ calls sendAsk() (Algorithm 6) with a index i, and waits for the proof of storage corresponding to D_i. In our example, the caller of sendAsk() is $\mathcal{U}$.

Algorithm 6 sendAsk.

Input:
a value i
1: BEGIN
2: $\mathcal{MS} := caller$
3: **if** $MapUsers[\mathcal{MS}.address].Flag\_Deposit$ **and** $MapUsers[\mathcal{MS}.address].Flag\_Ask\_Permit$ **and** $index \geq 0$ **and** $index < MaxSize$ **then**
4: $MapUsers[\mathcal{MS}.address].(Flag\_Ask\_Permit, T\_Ask\_Begin, Ask\_index, Flag\_Ask\_Receive) := (false, \text{now}, i, true)$
5: **end if**
6: END

Within time T_Prove_Limit, $\mathcal{S}$ must call sendProof() (Algorithm 7) to upload his proof. Miners verify the proof of storage through the algorithm Merkle(). If the proof passes the validation, $\mathcal{U}$ could get the data he needs from this proof and their transaction continues. Else, $\mathcal{U}$ calls proveTimeout() (Algorithm 8) to withdraw his deposits worth $Deposit\_User$ and get compensatory payment worth $Deposit\_User$ from $\mathcal{S}$'s deposits. In this case, we set $Flag\_Ask\_Permit$ to "false" to ensure that only one penalty is imposed on the same disagreement.

Algorithm 7 sendProof.

Input:
$\mathcal{U}.address$, a string D_i , a hash list $\mathcal{H}_s$
Output:
a value res
1: BEGIN
2: **if** $caller == \mathcal{S}$ **then**
3: $MapUsers[\mathcal{U}.address].T\_Prove\_Receive :=$ now
4: **if** $MapUsers[\mathcal{U}.address].Flag\_Ask\_Receive$ **and** $MapUsers[\mathcal{U}.address].Flag\_Ask\_Permit$ **then**
5: verify($MapUsers[\mathcal{U}.address].DataRoot ==$ Merkle($MapUsers[\mathcal{U}.address].(Ask\_index, D_i, \mathcal{H}_s)$))
6: **if** the validation fails **then**
7: $\mathcal{U}$ receives twice the $MapUsers[\mathcal{U}.address].Deposit\_User$
8: $MapUsers[\mathcal{U}.address].Flag\_Ask\_Permit := false$
9: $MapUsers[\mathcal{U}.address].Deposit\_User := 0$
10: $res = false$
11: **else**
12: $MapUsers[\mathcal{U}.address].Flag\_Ask\_Permit := true$
13: $MapUsers[\mathcal{U}.address].Ask\_Receive := false$
14: $res := true$
15: **end if**
16: **end if**
17: **end if**
18: END

Algorithm 8 proveTimeout.

1: BEGIN
2: $\mathcal{MS} := caller$
3: **if** the response exceeds the time limit T_Prove_Limit **then**
4: $\mathcal{MS}$ receives twice the $MapUsers[\mathcal{MS}.address].Deposit\_User$
5: $MapUsers[\mathcal{MS}.address].Flag\_Ask\_Permit := false$
6: $MapUsers[\mathcal{MS}.address].Deposit\_User := 0$
7: **end if**
8: END

4 Security discussion

In this section, we discuss the security of our scheme. Possible attacks against the scheme are first given, and then security analysis for each attack is described and some countermeasures are proposed.

4.1 Security threats

Our proposed scheme is constructed on top of Ethereum and we assume that the Ethereum platform is reliable. Therefore, we mainly concern and analyze the impact of possible malicious behaviors from users and servicers on system security. We analyze potential security threats in the following.

4.1.1 Malicious servicers

Arbitrary Report on Storage Space In order to get more orders, a dishonest servicer may claim to have more storage capacity than he really has. This kind of attack is usually launched combined with the outsourcing attack. After obtaining an order that requiring larger storage space than his actual space, the servicer deposits data beyond its storage space to other servicers or the cloud.

Denial of Service In the transmission stage, a malicious servicer may refuse to return the corresponding data when a user initiates a query request off-chain. In addition, a malicious servicer may reject to execute the update instruction sent by a user. This will result in a decline in the availability of the entire system.

Outsourcing Attack Case 1: The servicer does not store data locally, but puts the data directly into some cloud. When a user queries, the server firstly queries the cloud and returns the result to the user. In this case, storage is still centralized. Case 2: When a user redundantly stores the same file to multiple storage servicers, the malicious servicer may not store the file, but only obtains and forwards data from other storage servicers for rewards. In this case, the redundant storage is meaningless.

4.1.2 Malicious users

Frequent On-Chain Queries. A malicious user may always query data by submitting challenges on-chain, not through an off-chain way. This will increase the size of data on the blockchain and bring higher costs to the servicer.

Illegal Challenge Values. A malicious user could submit an illegal challenge value to the contract, trying to make the servicer lose his deposits due to the failure of the corresponding proofs.

4.2 Security analysis

In the following, we give security analysis for each malicious behaviors mentioned in Section 4.1.

4.2.1 Malicious servicers

Servicers can not claim their storage capacity unconditionally Our system stipulates that a servicer could only declare the storage time, while the storage space needs to be calculated according to the amount of deposit. In other words, servicers need sufficient deposits to support their storage declaration.

Assuming that the servicers are rational, they will provide the service honestly We divide possible denial-of-service situations into the following two cases:

Case 1: The servicer fails to store data correctly, i.e., data damages or loss occurs. In this case, the servicer will lose his deposits since he can not provide the storage proof when queried by the user.

Case 2: The servicer stores data correctly. There are two sub-cases: Case 2a: The servicer refuses to answer to his users at any time. In this case, the servicer will lose his deposits without any benefit when his users ask for arbitration (as we mentioned in Arbitration stage). Case 2b: The servicer does not respond to his users off-chain until a user uploads his query on the blockchain by calling sendAsk(). In this case, the servicer have to call sendProof() to upload the corresponding proof on the blockchain. This will bring extra costs to the servicer since he have to pay for the fees to miners.

Since the record on blockchain is tamper-resistant, users' requests for data updates will be recorded on the blockchain to facilitate auditing. If the user's data update request appears on the blockchain for more than a certain time, and the servicer still does not update the data, then it could be determined that the servicer is to blame: In the decentralized scenario, the user can trigger the algorithm $sendAsk()$. If his servicer does not update the stored data, he can not use $sendProof()$ to return the correct proof of storage, causing the servicer to lose deposits and reputation.

Against Outsourcing Attack For the first case, note that T_Prove_Limit is a parameter negotiated by both parties. The user could limit this parameter to prevent its server from obtaining data from a third party. For the latter case, in addition to limiting T_Prove_Limit, users could divide their files into several data blocks and apply different symmetric key encryption for different servicers. In this way, malicious servicers can not obtain a reward by directly forwarding data stored by other servicers.

4.2.2 Malicious users

Assuming that the users are rational, they always query data off-chain when servicers provide the service honestly Apparently, users have to pay extra money when they call sendAsk() on-chain. On the other hand, the transactions associated with On-Chain Query will not take effect until they are packaged and recorded by miners, which makes the query inefficient.

Illegal challenge values submitted by users are invalid As we can see from Algorithm 6, only legitimate challenge values could be validated and recorded by miners.

5 Implementation and evaluation

In this section, we give the implementation details. Evaluation results like gas, fee and time costs are shown.

5.1 Implementation

We implement Themis with Solidity 0.4.18 and test it on Rinkeby, an Ethereum test network. We run our experiments on a computer with a 2.3 GHz Intel i5-4200 CPU and 8 GB RAM. The storage contract could be coupled with several addresses delegating a special servicer and several users. To facilitate testing, we use MetaMask, an Ethereum wallet, to generate the addresses for $\mathcal{U}$ and $\mathcal{S}$ (see Table 3).

5.2 Evaluation

In this section, we focus on the efficiency of data transmission phase. The efficiency of data transmission hinges on data transmission and arbitration. And our storage contract only plays a role of supervision in data storage,

Table 3 The addresses of participating entities

Entity	Address
$\mathcal{U}$	0xFE59795fC6BB864021eB916f30Cf889201a6B841
$\mathcal{S}$	0x8e9ABb5C3d285D393F4DD4c3E277ee1e861F8bc9

Table 4 The time cost of calling sendAsk() and proveTimeout()

Function	The Time Cost
sendAsk	14.3 s
proveTimeout	22.5 s

which means if both entities are honestly involved in this scheme, the efficiency of data transmission only depends on the transfer protocol adopted off-chain. Hence, we only test the time cost of arbitration to assess the efficiency of dealing with disputes. Moreover, to evaluate the economy cost of using Themis, we also test the gas and fee costs of the storage contract.

5.2.1 Time cost of arbitration

For more detailed and accurate analysis, we review the arbitration process which could be divided into the following steps.

1. A user initiates a query to the storage contract by calling sendAsk();
2. The servicer provides the proof to the contract with function sendProof();
3. If the proof is not submitted after a specified time, the user calls proveTimeout() to obtain compensation.

In order to get accurate data, we call the above three functions for 10 rounds and calculate the average execution time that they take. In particular, we test sendProof() with different data block sizes from 2KB to 30KB and four kinds of data block numbers: 2^5, 2^{20}, 2^{30} and 2^{40}. Table 4 and Fig. 4 illustrate the time costs of these functions.

Based on the test results, we get the following results.

1. The average time cost of sendAsk() and proveTimeout() is 14.3 seconds and 22.5 seconds, respectively;
2. The arbitration time grows as the size and number of data blocks increase. The results are consistent with

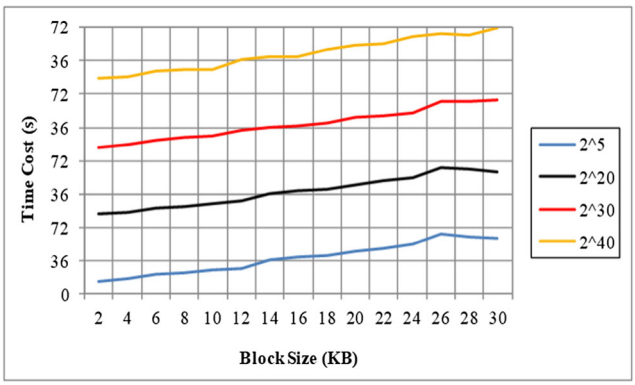


Fig. 4 The time cost of calling sendProof()

Table 5 The cost of executing the contract(a)

Caller	Function/Operation	Gas cost	Fee cost(1Gwei)	Fee cost(4Gwei)
Servicer	Deploy	1816520	0.37608$	1.50432$
	concludeTrans	220787	0.04571$	0.18284$
	finish(Transaction failed)	34625	0.00716$	0.02864$
	finish(Transaction success)	50137	0.01037$	0.04148$
User	concludeTrans	37007	0.00745$	0.0298$
	sendAsk	68083	0.01410$	0.0564$
	proveTimeout	41467	0.00859$	0.03436$

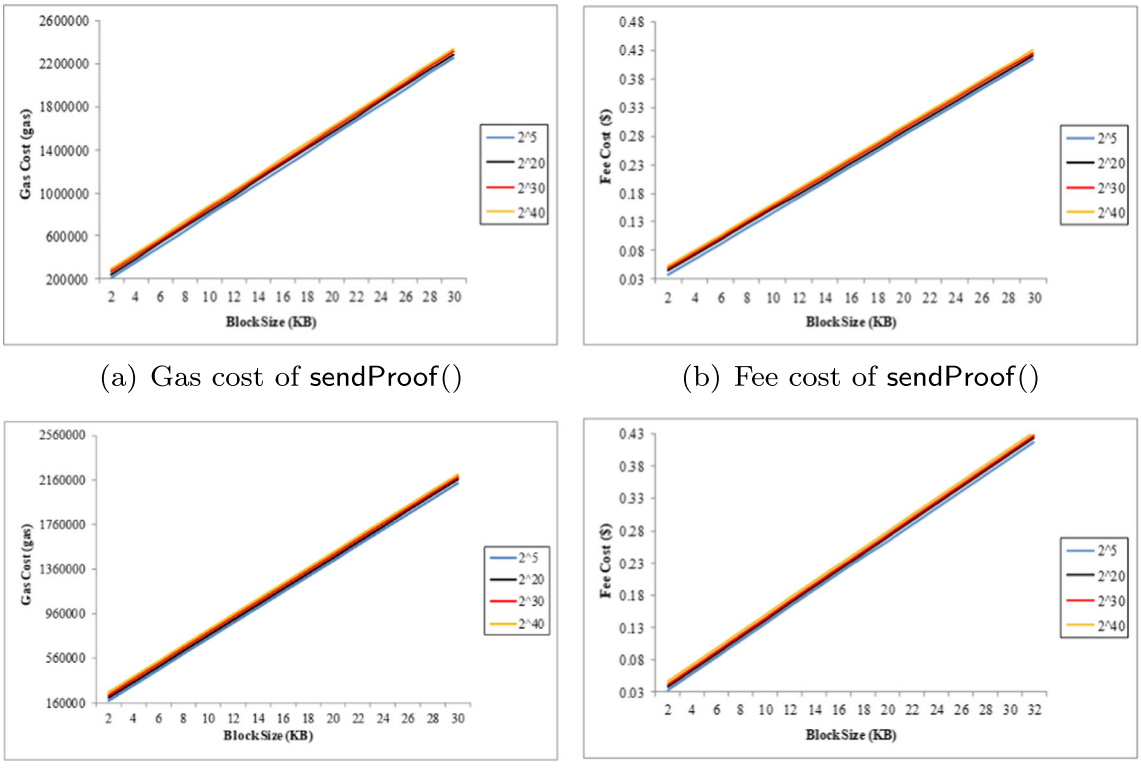
the design of the scheme. Since sendProof() needs to contain the contents of the data block, the time cost is positively related to the size of data block. Meanwhile, as the number of data blocks increase, the hash list $\mathcal{H}$ will contain more content, resulting in a longer time cost;

3. The whole arbitration process takes between 40 and 110 seconds.

5.2.2 Gas and fee cost

The gas price was denoted as 1×10^{-9} Ether (1 Gwei) and 4×10^{-9} Ether (4 Gwei), the exchange rate was 1 Ether = 209$ in September 2019. Table 5 and Fig. 5 show the gas and fee cost when deploying a contract and executing different functions of the storage contract. In particular, since the cost of sendProof() and update() are related to the block size and the number of blocks, we test them separately with different data block size from 2KB to 30KB and four kinds of data block numbers: 2^5, 2^{20}, 2^{30} and 2^{40}.

In Fig. 5, it could be seen that the gas cost of functions is linear to the block size. This phenomenon is reasonable due to the increase of the computation and transmission workload. Besides, the gas cost has a positive relationship with the number of blocks. We could deduce from the experimental results that Themis supports a PB-level cloud storage for a user in one peer-to-peer storage service process with a low cost. In details, according to the maximum size of the single data block (30KB) and the maximum amount of data block (2^{40}) supported by the system, we can



(a) Gas cost of sendProof()

(b) Fee cost of sendProof()

(c) Gas cost of update()

(d) Fee cost of update()

Fig. 5 The gas cost of executing the contract(b)

Table 6 Comparison with other blockchain-based cloud storage schemes

System	Platform	Adaptive scenarios	Way to conclude a tx	Proof submission	Transfer Protocol	Mining mechanism
Storj	Ethereum	Storage	Centralized platform	On-chain&&Periodic	HTTP	Proof-of-Work
Sia	Sia	Storage and Inquiry	Centralized platform	On-chain&&Periodic	IPFS	Proof-of-Work
Filecoin	Filecoin	Storage and Inquiry	Free	On-chain&&Periodic	IPFS	Proof-of-Spacetime
Themis	Ethereum	Storage and Inquiry	Free	Off-chain+On-chain&&Irregular	IPFS or HTTP	Proof-of-Work

get the maximum data size supported by Themis is 30PB ($2^{40} \cdot 30KB$). In addition, we find that calling the function update() requires a certain amount of cost, so users might selectively use the function for update operations (if the user trusts the servicer, the update operation could be done off-chain as much as possible).

Note that the cost of sendProof() is bigger than the cost of sendAsk(), which brings unfairness to servicers in P2P trading network. In details, if a malicious user intentionally calls sendAsk() for multiple inquiries, the cost of its servicer who calls sendProof() to reply will be much higher than the user. To ensure the fairness of the query process on-chain, we require users to transfer special money to the contract to make up for the difference while calling sendAsk().

Based on the above results, we could calculate the cost of participants in different situations. Suppose that there is a contract supporting a storage of 2^{20} data blocks, each of which is 2KB in size, and the gas price is set to 1 Gwei.

Under normal conditions, a user needs to spend $(0.00745 + L \times 0.3838)$\$, where L denotes the number of times that the user calls *update()*. If disputes arise, a user needs to spend about $(0.00745 + M \times 0.0141)$\$, where M denotes the number of times that the user calls sendAsk().

A servicer only need to perform the deploy operation whose cost is 0.37608\$ one time, then he could provide service for multiple users. Normally, a servicer merely spends about extra 0.05608\$ for each user. When arbitration is required, it costs a servicer about $(N \times 0.05287 + K' \times 0.04433)$\$, where N represents the number of users and K' denotes the number of times that the servicer calls sendProof().

5.3 Performance comparison

As Table 6 shows, we compare Themis with Storj, Sia and Filecoin from the aspect of implementation details.

Storj and Themis are based on Ethereum, and the others have their own platform. Storj is dedicated to storage scenarios, while the other schemes are suitable for both storage and inquiry scenarios. Users and servicers on Storj and Sia need a centralized platform to conclude storage transactions, while Filecoin and Themis advocate free matching between storage and users. To improve the matching efficiency, Themis also support matching through a trusted third party. To prove the correct storage of data, Storj, Sia and Filecoin require servicers to submit the storage proof to the blockchain regularly, resulting in extra gas and fee cost, a huge amount of proof data stored on-chain, and waste of servicers system resources. Therefore, Themis specifies that users make queries through an off-chain way when there is no dispute. Only when a user inquire through the smart contract, the servicer needs to return the proof on-chain. Storj uses HTTP as its data transfer protocol, while both Sia and Filecoin employ the IPFS protocol. For Themis, any transport protocol is feasible. Except that Filecoin uses Proof of Spacetime for mining, the other schemes adopt PoW.

6 Conclusion

We present Themis, a blockchain-based P2P cloud storage scheme based on smart storage contract, that provides an accountable and distributed storage environment for storage participants. Themis solves the problems of storage centralization, centralized verification of data integrity and denial of service in cloud storage scenarios. Our implementation and evaluation on the Ethereum test network show that when disputes arise, arbitration could be completed with low time and economic costs. In conclusion, Themis is a highly viable solution for practical deployment.

References

1. Chen K, Lang W, Zheng K, Ouyang W (2015) Research on the cloud storage security in big data era. In: 2015 international conference on applied science and engineering innovation, Atlantis Press, pp 659–664
2. Zhou R, Chen H, Li T (2015) Towards lightweight and swift storage resource management in big data cloud era. In: Proceedings of the 29th ACM on international conference on supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015, pp 133–142

3. Andrews JG, Buzzi S, Choi W, Hanly SV, Lozano A, Soong AC, Zhang JC (2014) What will 5g be? IEEE Journal on Selected Areas in Communications 32(6):1065–1082
4. Song D, Shi E, Fischer I, Shankar U (2012) Cloud data protection for the masses. IEEE Computer 45(1):39–45
5. Cash D, Küpçü A, Wichs D (2017) Dynamic proofs of retrievability via oblivious RAM. J Cryptology 30(1):22–57
6. Shi E, Stefanov E, Papamanthou C (2013) Practical dynamic proofs of retrievability. In: 2013 ACM SIGSAC conference on computer and communications security, CCS'13, Berlin, Germany, November 4-8, 2013, pp 325–336
7. Wang Q, Wang C, Li J, Ren K, Lou W (2009) Enabling public verifiability and data dynamics for storage security in cloud computing, in: Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings, pp 355–370
8. Mukundan R, Madria SK, Linderman M (2014) Efficient integrity verification of replicated data in cloud using homomorphic encryption. Distributed and Parallel Databases 32(4):507–534
9. Zhang Y, Xu C, Liang X, Li H, Mu Y, Zhang X (2017) Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation. IEEE Trans Information Forensics and Security 12(3):676–688
10. Wilkinson S, Boshevski T, Brandoff J, Buterin V (2014) Storj a peer-to-peer cloud storage network. https://storj.io/storj.pdf
11. Vorick D, Champine L (2014) Sia: Simple decentralized storage. https://sia.tech/sia.pdf
12. Benet J (2014) IPFS - content addressed, versioned, P2P file system. http://arxiv.org/abs/1407.3561
13. Benet J, Greco N (2018) Filecoin: a decentralized storage network. https://filecoin.io/
14. Ben-Sasson E, Chiesa A, Genkin D, Tromer E, Virza M (2013) Snarks for c: verifying program executions succinctly and in zero knowledge. In: Advances in cryptology - CRYPTO 2013 - 33rd annual cryptology conference, santa barbara, CA, USA, August 18-22, 2013, Proceedings, Part II, pp 90–108
15. Nakamoto S et al (2008) Bitcoin: a peer-to-peer electronic cash system https://bitcoin.org/bitcoin.pdf
16. Szabo N Formalizing and securing relationships on public networks, First Monday 2 (9)
17. Poon J, Dryja T (2016) The bitcoin lightning network: Scalable off-chain instant payments https://lightning.network/lightning-network-paper.pdf
18. Bonneau J, Miller A, Clark J, Narayanan A, Kroll JA, Felten EW (2015) Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: 2015 IEEE symposium on security and privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pp 104–121. https://doi.org/10.1109/SP.2015.14
19. Garay JA, Kiayias A (2018) Sok: A consensus taxonomy in the blockchain era. https://eprint.iacr.org/2018/754.pdf
20. Garay JA, Kiayias A, Leonardos N (2015) The bitcoin backbone protocol: Analysis and applications. In: Advances in cryptology - EUROCRYPT 2015 - 34th annual international conference on the theory and applications of cryptographic techniques, sofia, bulgaria, April 26-30, 2015, Proceedings, Part II, pp 281–310
21. Pass R, Seeman L, Shelat A (2017) Analysis of the blockchain protocol in asynchronous networks. In: Advances in cryptology - EUROCRYPT 2017 - 36th annual international conference on the theory and applications of cryptographic techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II, pp 643–673. https://doi.org/10.1007/978-3-319-56614-6_22
22. Zhao Y, Li Y, Mu Q, Yang B, Yu Y (2018) Secure pub-sub: Blockchain-based fair payment with reputation for reliable cyber physical systems. IEEE Access 6:12295–12303
23. Hu Y, Manzoor A, Ekparinya P, Liyanage M, Thilakarathna K, Jourjon G, Seneviratne A (2019) A delay-tolerant payment scheme based on the ethereum blockchain. IEEE Access 7:33159–33172
24. Pănescu A-T, Manta V (2018) Smart contracts for research data rights management over the ethereum blockchain network. Science & Technology Libraries 37(3):235–245
25. Hong S (2019) P2p networking based internet of things (iot) sensor node authentication by blockchain. Peer-to-Peer Networking and Applications, pp 1–11
26. Zhang Y, Wen J (2017) The iot electric business model: Using blockchain technology for the internet of things. Peer-to-Peer Networking and Applications 10(4):983–994
27. Chen Y, Li H, Li K, Zhang J (2017) An improved P2P file system scheme based on IPFS and blockchain. In: 2017 IEEE international conference on big data, BigData 2017, Boston, MA, USA, December 11-14, 2017, pp 2652–2657
28. Buterin V (2014) A next-generation smart contract and decentralized application platform. https://whitepaperdatabase.com/wp-content/uploads/2017/09/Ethereum-ETH-whitepaper.pdf
29. Merkle RC (1980) Protocols for public key cryptosystems. In: Proceedings of the 1980 IEEE symposium on security and privacy, Oakland, California, USA, April 14-16, 1980, pp 122–134

Yiming Hei received the B.S. degree from Xidian University, Xian, China, in 2017 and received M.S. degree from Beihang University, Beijing, China, in 2020. Now, he is pursuing the Ph.D degree in Beihang University. His research interests include applied cryptography and smart contract

Yizhong Liu received his B.S. Degree in Department of Electronic and Information Engineering from Beihang University, China, 2014. He is now pursuing a Ph.D. degree in Department of Electronic and Information Engineering, Beihang University, China. His research interests include information security, cryptography, blockchain and smart contracts.

Dawei Li received the BS degree from Beihang University, Beijing,China, in 2015. He received the Ph.D degree from Beihang University in 2019. Now, he is a lecturer of cyber science and technology at Beihang University. His research interests include cryptography, blockchain and 5G security.

Jianwei Liu received the BS and MS degrees from Shandong University, China, in 1985 and 1988, respectively. He received the PhD degree in communication and electronic system from Xidian University, China, in 1998. He is now a professor of electronic and information engineering at Beihang University, Beijing, China. His current research interests include wireless communication network, cryptography, and information security.

Qianhong Wu received his Ph.D. in Cryptography from Xidian University in 2004. Since then, he had been with Wollongong University (Australia) as an associate research fellow, with Wuhan University (China) as an associate professor, and with Universitat Rovira i Virgili (Spain) as a research director. He is currently a professor with Beihang University in China. His research interests include cryptography, information security and privacy, VANET security and cloud computing security. He has been a holder/co-holder of 15 China/Australia/ Spain funded projects. He has authored 30 patents and over 150 publications. He has served as associate/guest editor in several international ISI journals and in the program committee of dozens of international conferences.