2018 IEEE 32nd International Conference on Advanced Information Networking and Applications

BlockStore: A Secure Decentralized Storage Framework On Blockchain

Sushmita Ruj†, Mohammad Shahriar Rahman‡, Anirban Basu* and Shinsaku Kiyomoto*,
† Indian Statistical Institute, India, Email: sush@isical.ac.in
‡ University of Liberal Arts, Bangladesh, Dhaka. Email: shahriar.rahman@ulab.edu.bd
*KDDI Research Inc., Japan, Email: basu@kddi-research.jp, kiyomoto@kddi-research.jp

Abstract—**In order to ensure faster audits, higher transparency and security, many applications are being designed using blockchains. We propose BlockStore, a decentralized storage framework using blockchain technology. The primary motivation is efficient utilization of storage resources of users. Users often have un-utilized or underutilized storage in their devices. They can choose to host their storage resources when they are not in use. Users rent storage from the host for a fee for a fixed period of time and release back after the time expires. BlockStore keeps track of un-utilized storage of hosts in *Space Wallet*, a structure that helps in assigning storage to renters on request. The ownership of storage can be proved by logging all storage transactions in a public ledger (the blockchain), which can be verified by any user. A host cannot host the same storage to two users at the same time, nor can it tamper with the data of the renter. Renters cannot frame a host of cheating. BlockStore uses proofs of storage and data possession to verify that the hosts do not tamper with data and penalizes parties for misbehavior. Users can encrypt data for privacy. Payment and penalty are handled using smart contracts. BlockStore differs from existing solutions, by providing stronger audit that detects and penalizes misbehaving parties earlier than existing schemes.**

Keywords: Blockchain, Decentralized Storage, Proofs-of-storage, Smart Contracts

I. Introduction

Everyday we generate a lot of digital information, through messages that we exchange, digital photographs and videos that we share; as well as through ubiquitous devices that we use that form the Internet of Things. This requires significant storage space. Cloud storage, although an option, can be expensive and access to the data may have high latency. Since data rests with, and accesses to it are often managed by a third party (the cloud service providers), data security and privacy are major concerns. Though several security features are now integrated with the cloud infrastructure such as homomorphic encryption, searchable encryption, zero-knowledge protocols, these are quite expensive for practical purposes. In many cases large storage space is required for archival purposes.

Users often have large un-utilized or underutilized storage space on their devices, which could be shared to form a collaborative storage network. This is an attractive option but with many challenges. Proper resource utilization, trust between users providing and availing service, verifying the integrity of data are crucial. Incentive mechanisms for hosts to share their idle resources and penalizing misbehaving parties are important for promoting peer-to-peer (P2P) collaborative storage. Many P2P file systems such as Gnutella [2] and Bit Torrent [1] exist but the main purpose of these P2P systems is file sharing. Our motivation is different in that a client (renter) wants to outsource data to a peer (instead of a cloud server) and retrieve it later. In this process, the client does not maintain a local copy.

Often a host can be tempted to overwrite the contents of a renter with that of another if the latter provides more incentive. It is important to ensure that two or more users do not use the same storage. There is a need for a transparent system, which logs all transactions for storage, so that neither can a host cheat a renter, nor can a renter frame a host. All these together will increase resource utilization, ensure integrity of data and maintain trust among peers.

Our proposed storage framework **BlockStore** does exactly that: it uses blockchains to ensure that transactions between peers (*Hosts* and *Renters*) are logged, and are publicly auditable. The *Space Wallet*, akin to a Bitcoin wallet, helps in coordinating the process without interfering with the transactions. Any deviation from the smart contract for space renting will be detected and the services aborted with a penalty on the deviating party. Thus, a host cannot cheat the renter by tampering with data or providing space less than that in the contract. Likewise, the renter cannot defame a host by accusing it of violating the conditions of the contract.

The renter can upload, modify, delete data anytime during the service contract. It may choose to encrypt data. It may want to run periodic audits to verify the integrity of the data. Audits can be done by third parties. All storage and payment transactions are maintained on the blockchain and audits are made off-chain. The reason for using blockchain instead of log servers is that often log servers are centralized and are prone to attack and there is no mechanism to verify if logged transactions are invalid. Blockchains provide transparency by consensus amongst the peers and is thus more robust against failure. BlockStore uses smart contracts on blockchain to ensure trustworthy P2P storage.

We show that BlockStore can be built using known cryptographic primitives like proofs of retrievability, proofs of space, signature schemes, and resource allocation algorithms. However, there are many challenges while integrating all these pieces together as we will see throughout the paper. We present a smart contract and discuss the various transactions in a step-by-step manner. This is an initial framework for smart contract for renting storage and can be extended to incorporate subletting. The ideas not only apply to storage but any type of sharing like digital content.

To the best of our knowledge, Storj and Sia also provide P2P

1550-445X/18/$31.00 ©2018 IEEE
DOI 10.1109/AINA.2018.00157

storage on blockchain. In Sections II, we discuss more about Storj and Sia. Both Storj and Sia require special cryptocurrencies, Storcoin and Siacoin (respectively), unlike BlockStore which can used with any blockchain platform like Bitcoin, Ethereum etc. If the host corrupts the files, then it cannot be known until the file is downloaded. In contrast, BlockStore helps in quickly detecting corruption.

The key features of BlockStore are as follows: (1) Separation of storage transactions (control) and data. (2) Transactions for hosting (or letting) and renting and payment are on-chain, while auditing for storage can be both on-chain and off-chain. (3) Audits can be done by either the renter or any other peer. Peers receive incentive for running audits. (4) The renter can either specify the host or depend on the space wallet assignment algorithm to pair it up with a host. Space wallets are maintained on every peer node and assignment algorithm is called from the BlockStore smart contract. Existing works like [15] do not discuss how hosts and renters are matched up. This is however crucial, for performance, security and trust. (5) Responsibilities of encryption and storage encoding rests with the renter, who can decide it depending on its requirement. (6) Penalty is imposed if the host or renter deviates from the contract.

The article is organized as follows. In Section II we present related work. We present our BlockStore frameworks and relevant details of the smart contract in Section III. In Sections V we present off-chain audits. In Sections VI and VII we present the security and performance of our framework. We present many possible extensions in Section VIII and conclude in Section IX.

II. Related Work

Information about Bitcoin appear in [17]. Throughout the paper, we use renter and client interchangeably. We first discuss about Storj [4], [24] and Sia [3], which offer P2P storage using blockchain. Our smart contract structure is very different from Storj, which prevents overwriting on existing files and prevents both client and host to cheat one another. For Storj, any misbehavior of a host cannot be detected, until the file is finally constructed by the client. In our case, any misbehavior can be detected very quickly during the audit process. The client does not have to wait till it finally downloads the file back. This helps the client to abort the procedure and initiate the penalty procedure.

In Sia [3], a host and user can share a secure channel and upload and download data. To prove that the data is stored securely, the host submits audits and these are logged in the blockchain. The host receives Siacoins for each audit it performs. Sia does not protect if a renter frames a host. BlockStore only stores the contract between the owner and renter. Only incorrect audits are logged on the blockchain, as they trigger the release of space and enforce penalty, in a timely fashion. Our scheme can be built on existing permissionless blockchain platforms with the underlying payment structure and consensus algorithm.

Very recently we came across [15], which also proposes a decentralized P2P storage framework. However it differs significantly in that the underlying blockchain is Koppercoin [14]. The payment is anonymous. The audit protocols are similar to Storj and can be done at the time of space release. The authors provide a high level view of their protocol and gives no details how renters are matched with the hosts. Our proposal helps in pairing up hosts and renters using space wallet. In "Accountable storage" [6] users can store files in a server and can run audits for the data. If some blocks become corrupted, then a penalty can be made in Bitcoins. Their motivation is very different from ours. A complete line of work is that of alternate cryptocurrencies like Permacoin [16], Retricoin [20], SpaceMint [18], which use proofs of storage instead of proof of work (PoW) or proofs of stake for mining. This saves huge computational overheads during mining and can be used for a useful purpose. The concepts of proofs of space

A. Proofs of storage

Various types of proofs of storage are available in literature. These include proofs of retrievability, proofs of data possession, proofs of erasure, proofs of space, etc.

Proofs of retrievability (PoR) and proofs of data possession (PDP) were proposed by Juels and Kaliski [13] and by Ateniese *et al.*[5] respectively. A file is outsourced to a user (Prover) who has to convince the verifier (either the file owner or third party auditor) that the file has been stored correctly. All channels between the data owner, server and challenger (in case of third party auditing) are secure. During such audits, each file is broken down into smaller blocks. Authentication tags are computed on each block and uploaded to the server. During audit, some random blocks are selected and the combined tags and blocks are sent back to the challenger. The challenger can verify the integrity of the blocks using its auxiliary information. In PoR, the files are erasure coded and can be retrieved (with high probability) if the audits are successful. There has been several works on PoR and PDP under different settings like static vs dynamic data [22], [10], private vs public auditing [21], with/without privacy of data [23]. We will use such auditing schemes for off-chain auditing.

Proofs of Space (PoSp) proposed by Dziembowski *et al.* [9] verifies that a prover $\mathcal{P}$ possesses a storage of size M. The space is generated by the prover $\mathcal{P}$ by using a special type of directed acyclic graph called hard-to-pebble graph consisting of n nodes. A node i contains the following label:

$$l_i = H(\mu, i, l_{p_1}, l_{p_2}, \ldots, l_{p_t}), \tag{1}$$

where $p_1, p_2, \ldots, p_t$ are parents of i, μ is random value chosen by $\mathcal{P}$ and $H : \{0,1\}^* \rightarrow \{0,1\}^L$ is a hash function. $\mathcal{P}$ creates a Merkle tree T with all the n nodes and sends the Merkle-tree commitment γ to the verifier $\mathcal{V}$. The total size of storage is $M = 2.n.L$. Prover $\mathcal{P}$ proves that it has stored this graph correctly.

PoS proceeds in three steps: *Initialization* (*Pos.SpaceCommit*), *commitment verification* (*Pos.CommitVerify*) and *prove space* (*Pos.ProveSpace*). In the initialization step, a space is created by the prover $\mathcal{P}$ using a hard-to-pebble graph as discussed above.

In the commitment verification step, $\mathcal{V}$ sends a set of c challenge nodes. $\mathcal{P}$ not only has to send the commitments of each of the nodes of the Merkle tree T, but also the parents of these nodes. Thus, $\mathcal{V}$ has to verify the values of the nodes and also check if the Merkle root γ verifies correctly. Since this step is expensive, this is performed once during initialization by $\mathcal{P}$. Any verifier can verify if the host has the space by executing the *prove space* step. It has to be remembered, that the initialization and commitment steps are computation intensive, however, these are executed only once (at the registration phase) in our protocol.

III. Our proposed scheme

A. Overview of our scheme

We design a blockchain based peer-peer storage system. The blockchain is used to record the storage provided by the host and storage required by the renter, payments made for storage and penalties (if any) incurred by the host. It also keeps track of storage audits, so that neither the host nor the renter can cheat.

BlockStore makes use of a special structure called **Space Wallet** SW. The space wallet keeps track of storage space available to renters in BlockStore. Earlier papers on decentralized storage like Storj, Sia and [15] did not explicitly mention how storage was allocated to users. Space wallet maintains a list of storage space available (similar to UTXO in Bitcoin) and is maintained at all the peer nodes. This information helps in assigning host to renter. Now we describe the main entities in BlockStore.

Host: BlockStore consists of a set of hosts $\mathcal{L}$ who provide storage. A host might have multiple storage devices. Corresponding to each storage device d is a public key/secret key pair (used for verifying/signing transactions), an address (which identifies the storage space) a_d, pointer to the storage space (this could be anonymized) p_a and the size of the storage s_d. When a new host joins, it registers the storage device. The space is now available for use. The space wallet records this. The registration transaction is recorded by the blockchain. Anyone can run periodic audits on the data. The host is responsible for maintaining the integrity of the data stored by the renter failing which it is charged a penalty.

Renter: There is a set of renters $\mathcal{R}$. Each renter B can have multiple addresses (identities). Corresponding to each address there are two pairs of public keys, secret keys. One of them is a public key/secret key (used to generate addresses in the blockchain and verifying/signing transactions) the other for auditing. A renter is assigned (or can choose) a host from the space wallet and store its data (possibly after encryption). It might use erasure coding and distribute the data on multiple hosts. The host and renter then enter into a contract for a certain period of time for a certain fee. This is recorded on the blockchain. Any node can be both renter and host at different times.

Miners: Miners create blocks after validating storage, payment transactions and performing audits and receive rewards. In order to maintain consistency, consensus algorithms needs to be used. We leave the choice of consensus algorithm to the designer. Bitcoin's proof-of-work is a default option.

Peers/Auditors: Peers validate transactions and blocks. Peers also behave as auditors who validate the integrity of storage. These verifiers (auditors) can be users who either rent or host storage. They will have a natural motivation to audit because they are a part of the system.

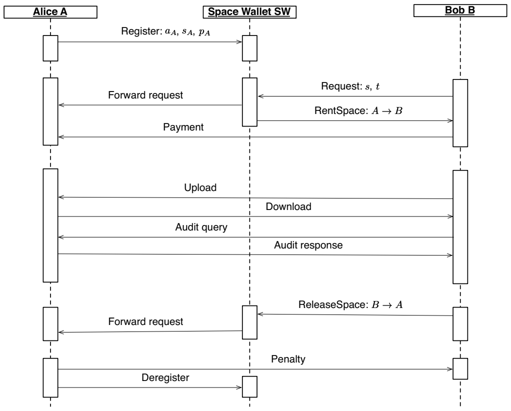

Fig. 1: Basic protocol

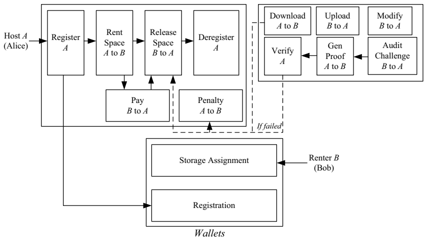

Fig. 2: On-chain and Off-chain transactions

Figures 1 and 2 show the BlockStore protocol. Details are given in Appendix A. The BlockStore framework consists of on-chain and off-chain transactions. `Register`, `RentSpace`, `ReleaseSpace`, `Audits` and `Deregister` are on-chain. File transfers are done off-chain. A BlockStore smart contract considers all the above phases and has different types of functions or transactions (as we will see next). All transactions are signed by the creator. Multiple transactions are put in one block and signed by the miner. Miners have addresses and accounts associated with them to collect payments. A consensus algorithm helps in maintaining a consistent blockchain across the peers.

B. Assumptions

We make the following assumptions: (1) Once the rent transaction is executed, a secure channel is established between the host and the renter. (2) During auditing, the channel between host and auditor is secure. (3) The renter is solely responsible to protect the confidentiality of data. It might choose to encrypt the data or decide otherwise. (4) The renter might choose to use erasure codes to enable to recover data if some data blocks are lost. It might also choose to keep multiple copies of the same data. (5) The host can choose to use fault tolerant measures like erasure coding to protect data at its end. (6) During audits, the host and auditor is online.

TABLE I: Smart Contract of BlockStore

```
Register
Host $A$ submits request $Reg_A = (a_A, p_A, s_A)$
$\gamma \leftarrow PoS.SpaceCommit(G, H, a_A, B_j)$
$\pi_1 \leftarrow PoS.CommitVerify(G, H, \gamma, pk_A, B_j)$
$\pi_2 \leftarrow PoS.ProveSpace(G, H, \gamma, pk_A, B_j)$
Transaction $Tx_{Reg} \leftarrow (Reg_A, \gamma, B_i, \pi_1, \pi_2, t_s, t_e)$
$Payment(a_A, SC, u + c_r)$
$\sigma_{Tx_{Reg}} \leftarrow Sign_{sk_A}(Tx_{Reg})$
If($Verify(Tx_{Reg}, \sigma_{Tx_{Reg}} = 1$ )) then,
    If ($\pi_1 = 1$ and $\pi_2 = 1$ ) then
        $SW \leftarrow SW \bigcup \{\langle a_A, p_A, s_A, t_s, t_e \rangle\}$
        $(Tx_{Reg}, \sigma_{Tx_{Reg}})$ written on blockchain
Else
    $Payment(SC, A, u)$
```

```
Audits
Auditor $AD$ submits a request $Audit = (Tx_{Rent})$
If $(Tx_{Rent}.\tau < T_{curr} - Tx_{Reg}.t_s)$
    $(i, \nu_i)_{i \in Q} \leftarrow AuditQuery(Q, p_A, s_A, B_j)$
    $\Pi \leftarrow AuditResponse((i, \nu_i)_{i \in Q}, a_A, p_A)$
    If($AuditVerification(\Pi, p_A, pk_B) \neq 1$)
       $Tx_{Audit} \leftarrow (Tx_{Audit}, B_j, \Pi)$
       $\sigma_{Tx_{Audit}} \leftarrow Sign_{sk_{AD}}(Tx_{Audit})$
       Trigger ReleaseSpace
       $Payment(SC, a_{AD}, c_a)$
       $(Tx_{Audit}, \sigma_{Tx_{Audit}})$ written on blockchain
```

```
Deregister
Host $A$ submits $DeReg_A = (a_A, p_A, s_A, t_s, t_e)$
If $(a_A, p_A, s_A, t_s, t_e) \in SW$
    Transaction $Tx_{DeReg} \leftarrow (DeReg_A, t_{curr})$
    $\sigma_{Tx_{DeReg}} \leftarrow Sign_{sk_A}(Tx_{DeReg})$
    If $(Verify(Tx_{DeReg}, \sigma_{Tx_{DeReg}}) = 1)$
      $SW \leftarrow SW \setminus (a_A, p_A, s_A, t_s, t_e)$
      $Payment(SC, a_A, d)$
         // $d$ is the remaining deposit
      $(Tx_{DeReg}, \sigma_{Tx_{DeReg}})$ written on blockchain
```

```
RentSpace
Renter $B$ submits request $Rent_B = (a_B, s, \tau)$
$(a_A, p_A, s_A, t_s, t_e) \leftarrow SWLookup(s, \tau)$
Compute fees $F \leftarrow v + c_l + c_b + c_a$
$TC \leftarrow GenContract(a_A, a_B, s, \tau, v, c_b, c_l, w)$
$\sigma_{TC} \leftarrow Sign_{sk_A, sk_B}(TC)$
If$(Verify(TC, \sigma_{TC}) = 1)$
    $Payment(a_B, SC, F)$
    $Tx_{Rent} \leftarrow (TC, H(TC), Tx_Pay, t_s, t_e)$
    $\sigma_{Tx_{Rent}} \leftarrow Sign_{sk_B}(Tx_{Rent})$
    If($Verify(Tx_{Rent}, \sigma_{Tx_{Rent}} = 1$ )) then,
       $SW \leftarrow SW \setminus \{(a_A, p_A, s, t_s, t_e)\}$
        $(Tx_{Rent}, \sigma_{Tx_{Rent}})$ written on blockchain
    Else
       $Payment(SC, a_B, v + c_l)$
```

```
ReleaseSpace
This transaction is triggered by $Tx_{Audit}$,
when audits fails, or by $Tx_{Rent}$ when
the time $\tau$ has elapsed.
$(i, \nu_i)_{i \in Q} \leftarrow AuditQuery(Q, p_A, a_A, B_j)$
$\Pi \leftarrow AuditResponse((i, \nu_i)_{i \in Q}, a_A, p_A)$
$SW \leftarrow SW \bigcup (a_A, p_A, s_A, t_{curr}, t_e)$
$Tx_{Release} \leftarrow (a_A, p_A, s, (i, \nu_i)_{i \in Q}, \Pi, B_j, pk_B)$
$\sigma_{Tx_{Release}} \leftarrow Sign_{sk_B}(Tx_{Release})$
If($AuditVerification(\Pi, p_A, pk_B) = 1$), then
    $Payment(SC, a_A, v)$
    $(Tx_{Release}, \sigma_{Tx_{Release}})$ written on blockchain
Else
    $Payment(SC, a_B, v + w)$, // penalty
    Release $v$ to $B$
     $(Tx_{Release}, \sigma_{Tx_{Release}})$ written on blockchain
```

IV. BlockStore Smart Contract

We present the BlockStore smart contract in Tables I. It consists of the following transactions.

Register: In the Register transaction, a host A registers a storage space of size s pointed to by p_A. The space is available from time t_s to t_e. The Space Wallet SW adds this information, if not already added. Before registering, the host has to prove that it has the desired amount of storage. This is done using $Pos.SpaceCommit(G, H, a_A, B_j)$ algorithm as discussed in Section II-A and Appendix A. The size of the graph depends on the size of the storage space and given by $2.n.L$, where L is the size of output of the hash function and n is the number of nodes in the graph G. γ is the Merkle Tree root hash. In order to ensure that a renter does not cheat by precomputing γ and sending a valid proof Π, the hash of the recent block B_i is used along with pk_A to randomly generate γ. A pays a deposit of $u + c_r$, u serves as deposit in case of malicious behavior and c_r is given to miner who mines a block containing the transaction. The payments are locked in the smart contract SC and later released. This storage space can be rented from A using the transaction RentSpace, that we will discuss next. Tx_{Reg} is signed by sk_A and the signature is denoted by $\sigma(Tx_{Reg})$. If the transaction is valid, the Space Wallet SW adds this new storage information, if not already added. Then a miner who puts this transaction $(Tx_{Reg}, \sigma(Tx_{Reg}))$ in the valid block, receives c_r.

RentSpace: A renter B wishing to rent space of size s for a duration τ, sends a request $Rent_B = (a_B, s, \tau)$. On receiving this request, the SW is looked up to find matching storage. Let v be the cost of storage. Then the renter has to pay $F = v + c_l + c_b + c_a$, where c_l, c_b and c_a are the payments for miners who put the ReleaseSpace, RentSpace and Audits transactions on the blockchain, respectively. A smart contract TC is signed between A and B. We use multisignatures for signing. The smart contract contains the terms and conditions of the contract, including the size of storage, start time t_{curr} and time of use τ, and penalty w to be paid in case host defaults. In the smart contract give in Table I, we have considered that B pays F to A, which is locked in the smart contract. The payment is made in stage RentSpace, when the host has returned back the files to B. If the host defaults (which is detected during audits), then the host does not receive any amount and the amount v is returned back to the renter in penalty phase. However, TC can be more complex, like paying in installments, or variable amount of penalty. The renter stores the data with host.

ReleaseSpace: This transaction is trigger in the following ways:

1) By the renter, if it wants back the files before the expiration time τ,
2) By the smart contract, when the time τ expires, or
3) If an audit fails.

In Cases (1) and (2), when the audit request is made, randomness of the latest block B_j is used to compute $(i, \nu_i)_{i \in Q}$. $\Pi \leftarrow AuditResponse((i, \nu_i)_{i \in Q}, a_A, p_A)$ is the audit response. If the Audit is successful, then the transaction is successful. The space is released and added to the SW. The payment v made by B that was locked in the smart contract is released to A. If the audit fails, then the penalty is triggered. In such a case, B receives the amount v back plus the amount w (a certain percentage of A's deposit u). In either case, the transaction $Tx_{Release}$ is recorded on the blockchain. $Tx_{Release}$ contains information about the storage (a_A, p_A, s), the query $((i, \nu_i)_{i \in Q}, pk_B, B_i)$ and response Π. $Tx_{Release}$ is signed by renter B, who receives the space back.

Audits: Audits can be performed by both the renter or any auditor during any time for a transaction Tx_{Rent}. If the transaction is active (meaning that the time τ has not elapsed), then $AuditQuery$ is called. In order to prevent collusion between auditors an host, the audit query has to be completely random. This randomness is achieved using the hash of most recent block $H(B_j)$. Since the recent block cannot be predicted, the query is random and cannot be stored by the host. The host returns the response Π as $\Pi \leftarrow AuditResponse((i, \nu_i)_{i \in Q}, a_A, p_A)$. If the audit verification fails, then this information is recorded in the blockchain and a ReleaseSpace transaction is triggered. A audit transaction is represented by $Tx_{Audit} \leftarrow (Tx_{Audit}, B_j, \Pi)$. This transaction is signed by the auditor AD as $\sigma_{Tx_{Audit}}$. $(Tx_{Audit}, \sigma_{Tx_{Audit}})$ are recorded on the blockchain. AD receives reward of c_a. If the audit is successful, it is not logged on the blockchain. This prevents DoS attacks, where auditors can flood with audit requests and unnecessarily increase the number of transactions.

Deregister: A host might deregister a storage that it had allocated earlier, if the space is not being used by a renter (it is in SW). The host creates a transaction $DeReg_A = \langle DeReg_A, t_{curr} \rangle$ and signs it $\sigma_{Tx_{DeReg}} \leftarrow Sign_{sk_A}(Tx_{DeReg})$. The remaining deposit is returned. A miner checks if this request is valid and puts it on the blockchain.

Payment: The payment can be made in any native cryptocurrency on which BlockStore is built. If the blockchain is designed in Ethereum, then the payment can be made in ETH, if it is constructed using Bitcoin then the payment can be made in BTC etc.

A. Allocating hosts to renters

To check for available space, and to pair up a renter with a suitable host and algorithm is executed. The design of this algorithm is an independent research problem. A simple way is to sort all SW according to the size of storage and assign a storage of least size greater or equal to s, if available.

For a more sophisticated solution, the location of the host, network parameters of the host, reputation of the host, pricing policy of the host, reputation of the renter will be taken into consideration. We will discuss this in our future follow-up work.

V. Verifying storage integrity

Audits are run by miners to check ReleaseSpace transactions. Audits can optionally be run by peers anytime during the leasing period in Audit. Both auditor and host have to be online during the audit. In both the cases challenge queries are sent to the host and the host send the response. The auditor (miner or renter) can verify based on the response received. We assume that there exists secure channels between the auditor and the host.

A. Choice of auditing schemes

If the renter does not wish to update the data during the storage period, then it uses a public auditing scheme for static data. Some schemes that can be used are [13], [21], [23]. The scheme of [23] support privacy-preserving audits meaning that the miner will not be able to derive the data blocks from the audit response. We present the scheme of [21] for its simplicity and use to to derive the time for verification of transaction.

It has to be noted here, that if the renter updates the data then a dynamic auditing scheme like [12] or [22] can be used. In such a case an authenticated data structure needs to be stored along with the data and refreshed during updates to maintain data freshness. Merkle trees are good authenticated data structures for static data but not for dynamic data. Rank based skip lists [10] or authenticated B-trees [12] can be used.

We use the compact PoR scheme by Shacham and Waters [21] in our current version of BlockStore. For completeness we present it next. Though pairings are quite expensive operations, the proofs are short.

B. Auditing scheme in BlockStore

- *KeyGen:* Let there be an algorithm BLSetup(1^λ) that outputs (p, g, G, G_T, e) as the parameters of a bilinear map, where G and G_T are multiplicative cyclic groups of prime order $p = \Theta(\lambda)$, g is a generator of G and $e : G \times G \rightarrow G_T$. The renter chooses $x \overset{R}{\leftarrow} \mathbb{Z}_p$ as her secret key. The public key of the client is $v = g^x$. Let $\alpha \overset{R}{\leftarrow} G$ be another generator of G and $H_1 : \{0,1\}^* \rightarrow G$ be a hash function.
- *Upload:* Let the renter have a file F_0 with t blocks/segments which it wants to upload to the host. The renter encodes F_0 with an erasure code to form a file F with t segments. Let each segment of the file F be an element of $\mathbb{Z}_p$, that is, $F[i] \in \mathbb{Z}_p$ for all $1 \leq i \leq t$. The renter computes $\sigma_i = (H_1(i) \cdot \alpha^{F[i]})^x$ for all $1 \leq i \leq t$

and uploads the file F along with the tags $\{\sigma_i\}_{1\leq i\leq t}$ to the host.

- *AuditQuery:* During an audit, the verifier (renter or miner) generates a random query $Q = \{(i, \nu_i)\}$ using $H(B_j)$ as input and sends it to the host which acts as a prover.
- *AuditResponse:*Upon receiving Q, the host computes $\sigma = \prod_{(i,\nu_i)\in Q} {\sigma_i}^{\nu_i}$ and $\mu = \sum_{(i,\nu_i)\in Q} \nu_i F[i] \mod p$. The host responds to the verifier with (σ, μ).
- *AuditVerification:*Then the verifier verifies the integrity of renter's data by checking the verification equation

$$e(\sigma, g) \overset{?}{=} e\left(\prod_{(i,\nu_i)\in Q} H_1^{\nu_i} \cdot \alpha^{\mu}, v\right),$$

and outputs 1 or 0 depending on whether the equality holds or not. The correctness of the scheme can be proved as below.

Correctness: $$\sigma = \prod_{(i,\nu_i)\in Q} \sigma_i^{\nu_i} = \prod_{(i,\nu_i)\in Q} (H_1 \cdot \alpha^{F[i]})^{\nu_i x}$$
$$= \left(\prod_{(i,\nu_i)\in Q} H_1^{\nu_i} \cdot \alpha^{\mu}\right)^x$$

VI. Security

BlockStore has the following properties: (1) **No dual letting:** This means that Alice cannot host the same storage space to two users at the same time. This is similar to double spending. (2) **Proofs of storage of host:** The host should prove that it possesses the storage space of size s at location pointed to by p. (3) **Universal audits:** Anyone can run audits any time during the renting process. (4) **Audits on dynamic data (optional):** Anyone can run audits any time during the renting process to verify that updates requested by renters are made by the host. (5) **Confidentiality of data (optional):** The renter can protect the privacy of the data by encryption if it so desires.

A. Auditing Data

A user Bob who rents storage can use erasure coding for fault tolerance. It might also want to run audits to verify the integrity of data. Moreover, Bob can use the storage only in the append only mode or might modify/delete content. Depending upon the use, Bob can either static auditing schemes or dynamic auditing schemes as discussed earlier. The storage structure will depend on the auditing scheme and can be either a Merkle tree or Skiplist or an array of blocks.

B. Preventing dual letting

This is similar to double spending. The case of dual letting arises when Alice gives the same storage space to two users say Bob and Rob. `RentSpace` is a transaction from Alice to Bob Tx' or Alice to Rob, Tx''. Let us assume that Tx' has been included in the blockchain. This means that Alice's storage space is being used by Bob. If it receives enough confirmation and is in the longest blockchain, then Tx'' will not be executed. Once it receives enough confirmations, it is removed from the space wallet. Users can wait for sometime (to validate the transaction) before they upload the data. Thus, dual letting is prevented in a similar way as double spending.

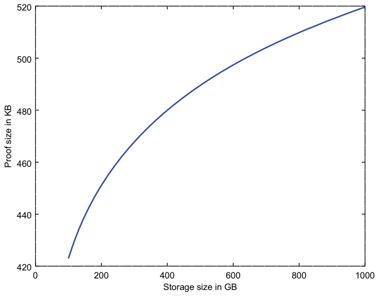

Fig. 3: Storage versus Proof Size for `Register` Transaction

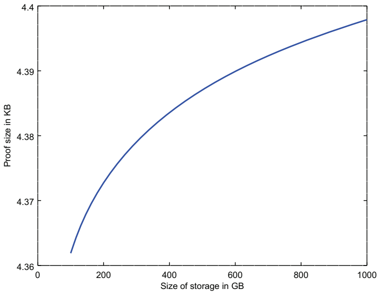

Fig. 4: Storage versus Proof Size for `ReleaseSpace` Transaction

VII. Performance

We first estimate the time taken for the verification of each of the transactions. We consider the information in Appendix B to estimate the time for each operation.

To dedicate a storage of size S, a host has to store a hard-to-pebble graph of size $n = S/2L$, where L is the size of the output of the hash function. We can choose $L = 256$. To prepare for registration, it has to compute the label l_i for each node i using Equation (1). This takes n hash computations. For verification, the verifier has to open $c = \lambda \log n$ nodes and their parents. Observing from the Fig 4 of [18], we find that for 100 GB of data, the time for preparation for registration is 1.5 hours, time to verify is 0.05 ms. Since this step occurs only when the host first steps up in the system, the preparation step can be done offline. The verification cost of the miner is approximately 0.05ms.

To prove space a miner has to send $c = \lambda \log n$ Merkle Tree proofs. Each proof is of size $256 \log n$ bits, if SHA-256 is used as a hash function for the Merkle tree. Therefore number of bits sent across is $256.\lambda(\log n)^2$. Let $L = 256$ bits and $\lambda = 128$. Hence, $n = S/(2 * 256 * 10^{-9})$. The figure shows the plot of storage size (S) vs proof size. For a storage of 100 GB, number of vertices in a hard-to-pebble graph is $n = (10^9 * 8 * 100/512) \approx 100 * 2^{25}$. Thus, the proof size is $128 * 256 * (\log n)^2 \approx 422KB$. Since registration is done only once, this overhead is amortized. It should be noted that the [18] considers $\lambda = 30$, we believe for enhanced security $\lambda = 128$ can be considered. Figure 3 shows the storage space (in GB) versus the proof size (in KB).

Verification of Eqn. V-B that involves $|Q|$ hashes to G_1, one multi-scalar multiplication in G_1, one scalar multiplication in G_1, one addition in G_1, two pairings and one comparison in G_T. A multi-scalar multiplication computes $|Q|$ scalar multiplications followed by $|Q|-1$ additions in G_1 in an efficient manner. For the timing analysis of these operations, we use PandA [8]. In Table II, we mention the cycle-counts of these operations for a 128-bit secure, Type-3 pairing framework involving a pairing-friendly BN curve. We take $|Q| = \lambda = 128$.

Summing up over required number of these operations, we get the total cycle-counts for Eqn. V-B as around $(128*83168+14605364+288240+2468+2*3832644+8320) = 33215184$. These many cycles take approximately 13.29 milliseconds (ms) on this processor. The verification of in line (9) of `ReleaseSpace` transaction takes less than 2 ms [17]. Therefore, the total cost of verifying `ReleaseSpace` transaction is less than 15.29 ms.

The size of the audit proof is $|G_1| + |\mu| + |Q|(|\nu_i| + \log t)$, sizes of σ is $|G_1|$, and that of ν_i and μ are respectively $|\nu_i|$ and $|\mu|$. t is the number of segments. For the value of the security parameter λ up to 128, Barreto-Naehrig (BN) curves [7], [11] are suitable. Thus p, σ, μ ν_i are of size 256 bits each and $|Q| = 128$. Assuming a segment of size $4KB$, $t = |F| * 10^9/4$, where $|F|$ is the storage space in GB. For a storage space of 100 GB, proof size is about 4.36 KB. Figure 4 shows the storage space (in GB) versus the proof size (in KB).

VIII. Possible Extensions and Future Directions

A. Concept for BlockStore Economics

Before renting storage space from Alice (*A*), Bob (*B*) exchanges real money to virtual money via a BlockStore service provider. BlockStore has three exchange rates: (a) R_M from real money to virtual money, (b) R_S from virtual money to storage size, and (c) R_V from renting storage size to virtual money. The difference $R_S - R_V$ can be realized as a commission by the BlockStore Service provider. The exchange rate R_M is used for dynamic control of the total demand for storage sizes in BlockStore. A decreasing R_M indicates a decreasing demand due to higher unit price of storage. The rates R_S and R_V change according to total storage size that can be rented. The total storage size is expected to be increasing over time, due to increase in the number of users and the drop in the unit price of storage. Users are incentivised with virtual money through the time-dependent changes in the rates R_S and R_V. For instance, a user who previously rented a storage space in exchange of virtual money can, in future, borrow a larger storage space using the same amount of the virtual money. New economic models can be defined for the BlockStore service based on the above concept. We will explore the details and analyses of the economics as future work.

B. Future Directions

In the future we would like to address the following issues: (1) Design an incentive mechanism for storing files. Designing a payment system that will reward a host for good service and penalize a host for malicious behavior. (2) Address reputation management for both host and renters. (3) Design efficient algorithms and data structures for the space wallet to optimize resources of the users. (4) Design efficient data auditing schemes to reduce latency and bandwidth, by fewer computation and smaller proof sizes. (5) Use proof of space instead of proof of work during mining for efficient verification. (6) Extend our framework to design a universal framework for sharing apartments, digital media and other goods and services. (7) Implement the framework on decentralized platforms like Ethereum.

IX. Conclusion

In this paper, we present BlockStore, a framework for decentralized storage with Blockchain. It enjoys many features like data integrity, confidentiality and transparency. The transactions are explained in details. The framework is an initial setup and can be extended in many directions to include renting multimedia, apartments and other goods and services. We believe that this paper will open up many research challenges in resource sharing on blockchain.

Acknowledgement

The work was partially done when the first author was visiting KDDI Research (erstwhile KDDI R&D Labs), Japan, Japan under the Japan Trust International Cooperation program by NICT. This work is also partially supported by a University Research Grant by Cisco Inc. USA.

References

[1] BitTorrent. http://www.bittorrent.com/.

[2] Gnutella. https://www.gnu.org/philosophy/gnutella.en.html.

[3] Sia: Simple decentralized storage. http://sia.tech/assets/globals/sia.pdf/ (Whitepaper).

[4] Storj. https://storj.io/.

[5] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 598–609, 2007.

[6] Giuseppe Ateniese, Michael T. Goodrich, Vassilios Lekakis, Charalampos Papamanthou, Evripidis Paraskevas, and Roberto Tamassia. Accountable storage. *ACNS*, 2017.

[7] PauloS.L.M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer Berlin Heidelberg, 2006.

[8] Chitchanok Chuengsatiansup, Michael Naehrig, Pance Ribarski, and Peter Schwabe. PandA: Pairings and arithmetic. In Zhenfu Cao and Fangguo Zhang, editors, *Pairing-Based Cryptography - Pairing 2013*, volume 8365 of *Lecture Notes in Computer Science*, pages 229–250. Springer International Publishing, 2014.

[9] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 585–605, 2015.

[10] C. Christopher Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security*, 17(4):15, 2015.

[11] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[12] Chaowen Guan, Kui Ren, Fangguo Zhang, Florian Kerschbaum, and Jia Yu. Symmetric-key based proofs of retrievability supporting public verification. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, pages 203–223, 2015.

[13] Ari Juels and Burton S. Kaliski, Jr. PORs: Proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security, CCS 2007*, pages 584–597, 2007.

[14] Henning Kopp, Christoph Bösch, and Frank Kargl. Koppercoin - A distributed file storage with financial incentives. In Feng Bao, Liqun Chen, Robert H. Deng, and Guojun Wang, editors, *Information Security Practice and Experience - 12th International Conference, ISPEC 2016, Zhangjiajie, China, November 16-18, 2016, Proceedings*, volume 10060 of *Lecture Notes in Computer Science*, pages 79–93, 2016.

[15] Henning Kopp, David Modinger, Franz Hauck, Frank Kargl, and Christoph Bosch. Design of a privacy-preserving decentralized file storage with financial incentives. In *IEEE Security & Privacy on the Blockchain (IEEE S&B)*, 2017.

[16] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 475–490, 2014.

[17] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies. http://tinyurl.com/j3r6qza.

[18] Sunoo Park, Krzysztof Pietrzak, Albert Kwon, Joël Alwen, Georg Fuchsbauer, and Peter Gazi. Spacemint: A cryptocurrency based on proofs of space. *IACR Cryptology ePrint Archive*, 2015:528, 2015.

[19] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game of graphs. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*, pages 149–160, 1976.

[20] Binanda Sengupta, Samiran Bag, Sushmita Ruj, and Kouichi Sakurai. Retricoin: Bitcoin based on compact proofs of retrievability. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*, pages 14:1–14:10. ACM, 2016.

[21] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of Cryptology*, 26(3):442–483, 2013.

[22] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security, CCS 2013*, pages 325–336, 2013.

[23] Cong Wang, Sherman S. M. Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

[24] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network, 2016. https://storj.io/storj.pdf.

APPENDIX A: THE BLOCKSTORE FRAMEWORK

Our framework consists of the following steps:

1) *Setup*(1^λ)*:* This algorithm initializes wallet SW. The algorithm also initializes the system parameters and generates a set of public parameters PK and a master secret key MSK for use during off-chain audits.
2) *KeyGen*$(1^\lambda, PK, MSK)$*:* This algorithm proceeds in two steps: Onchain.KeyGen which is used by host/renter i to generate secret key sk_i, public key pk_i and address a_i and Audit.KeyGen which creates secret keys for the audit process and to generate proofs of space.
3) *Register*(a_A, p_A, s_A)*:* When a user Alice $A \in \mathcal{L}$ wants to host its storage space, it registers with the space wallet SW. It sends its address a_A, pointer to storage p_A and size of storage s_A. A transaction is recorded in the blockchain to verify that A has a storage of size s_A as claimed by using a proof of space (PoS).
4) *PoS.SpaceCommit*(G, H, a_A, B_j)*:* This takes as input a hard-to-pebble graph G [19] and a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^L$. The host generates labels of all the nodes using Equation (1). Here $\mu = pk_A || H(B_j)$. B_j is the recent block and $H(B_j)$ is a randomness and cannot be precomputed by any host. The host sends the Merkle tree commitment γ to the verifier (miner).
5) *PoS.CommitVerify*(G, H, γ, pk_A)*:* The verifier (miner) verifies if γ is a valid Merkle tree hash using a challenge response protocol.
6) *PoS.ProveSpace*$(G, H, \gamma, pk_A, B_j)$*:* The verifier can verify the space by sending a challenge and checking the response. $H(B_j)$ is a randomness and cannot be precomputed by any host. The difference between this step and that of PoS.CommitVerify is that all openings of the nodes and the parents are required for the latter and thus more expensive.
7) *RentSpace*(a_B, s, τ)*:* This is executed when renter $B \in \mathcal{R}$ requests for space of size s for time τ. It might request the space from a host with address a_A or it might not specify a host. In the first case, the SW requests a_A and if the host is willing to provide space, then a communication between A and B is established. In the second case, the wallet runs a $SWLookup$ algorithm to pair up a host, say A with B. A transaction is recorded in the blockchain to verify that A (having address a_A) is registered with the SW and is hosting its space of size s_A at location pointed to by p_A for time τ to B. B has to agree on the conditions of A in order to store data. Both parties should agree on the terms and conditions of the contract, including pricing, payment, time of lease etc before starting data transfer.
8) *Payment*(B, A, v)*:* B makes a payment of value v to A. A and B can either be entities or the smart contract.
9) *Upload*(s, sk_B, p_A)*:* The algorithm is run by the renter B, who divides the file F into blocks, and converts it to erasure coded file F', creates authentication tags of the blocks using the secret key sk_B and uploads data at location p_A. Erasure coding is optional and decided by the renter.
10) *AuditQuery*(Q, p_A, B_j)*:* A verifier sends a challenge set Q of cardinality $l = O(\lambda)$ to the storage p_A held by a host at a_A. λ is the security parameter. B_j is used as a randomness.
11) *AuditResponse*(Q, a_A, p_A)*:* The host at a_A, after receiving the challenge set Q, computes a proof of storage Π at location p_A and sends to the verifier.
12) *AuditVerification*(Π, p_A, pk_B)*:*The verifier checks if Π is a valid proof of the data stored at p_A using the public key pk_B of B and outputs 1, if true and 0 otherwise.
13) *ReleaseSpace*(a_B, a_A, s, τ)*:* This is triggered when a renter with address a_A releases the space to host a_A or when time τ expires or an audit fails. If the audit is correct, then the payment locked in the smart contract is released to the host A. If this transaction is triggered due to audit failure, then a penalty is charged from the initial deposit of the host. This transaction is recorded on the blockchain.
14) *Deregister*(a_A, s_A, p_A)*:* A user with address a_A deregisters space from SW when it no longer wants to host.

APPENDIX B: BENCHMARK DATA

API function	Cycle-counts
Hash 59-bytes message into G_1	83168
Multi-scalar multiplication in G_1 for $\lambda = 128$	14605364
Scalar multiplication in G_1	288240
Addition in G_1	2468
Ate pairing	3832644
Comparison in G_T	8320

TABLE II: Cycle-counts on a 2.5 GHz Intel Core i5-3210M processor. [20]