Efficient Proof-of-Replication and Proof-of-Spacetime

Anonymous Author(s)

ABSTRACT

This is an abstract part.....No VDE;

TODO: Implementation(evaluation), Section alignment, Paper writing

WIP: Security, Scheme, Market protocol – SH

Do we have to describe the detail of ecosystem? Reward stake to worker...miner... this can prevent collude... – SH

CCS CONCEPTS

• **Security and privacy** → Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper;

KEYWORDS

template; formatting; pickling

1 INTRODUCTION

In this part, we explain what the motivation is, how existing works approach, what competitive power do we have. Compared with VDE based PoRep, we don't adopt VDE, instead we use tricky encryption.

motivation **: FileCoin, the most popular Decentralized Storage Network, has several problems. Generally, we think that node who have storage power should be able to participant DSN as miner(storage provider) node. But in order to become a fileCoin storage provider node, node must execute computation and memory expensive sealing operation(VDE: verifiable delay encoding) to make $\mathcal{R}eplica$. So, ordinary nodes who have normal computational power are difficult to participant fileCoin Network as storage provider. this process is too inefficient. So instead of using computational expensive VDE, in our efficient DSN system will define and use asymmetric sealing operation to make $\mathcal{R}eplica$ – KB**

Using asymmetric key setting, we designed Asymmetric sealing operation that is only data(or sk) owner executable. Before sending Data to storage server, client execute Asymmetric sealing operation using own secret key. – KB

1.1 Our Work

Additional contribution: In PoSt, Filecoin cannot guarantee that the server is storing continuously. More in detail, server is required to prove that it stores the data at some point recursively. That is, it doesn't guarantee that the data has been stored during the time between each epochs. In contrast, our construction let the server prove that it is storing the data for whole duration: If the proving is failed, not storing the data, server cannot generate replica since it doesn't have secret key of the client. – KB

The system itself dedicates for the decentralization. Since the sealing process of existing work, especially for Filecoin, takes many computational resources and time, it might be possible most of the storage(replica) become centralized with a few entities which have high computation resources. – SH

2 TECHNICAL OVERVIEW

Technical overview? Protocol overview?. Some parts in below description will be moved to more appropriate section – SH

In the existing Proof-of-Storage(PoS) researches using Verifiable Delay Encoding(VDE), VDE is an essential tool since "delay encoding" can prevent from generation attack. Without delayed encoding, it is easy for cheating server to generate replica directly even if it does not store in fact. However, delay encoding causes overhead to server since the server must prove that the encoding is executed *correctly* with *exact data* which client asks to store. Thus, we avoid delay encoding and instead we leave encoding to client. In the proposed protocol, client encodes the data by herself and gives the replica to server. Through encoding by client, delay is no longer essential. In this state, however, there is no difficult for server to generate replica quickly. Encoding therefore should be done privately. In contrast, decoding is required to be public which means that retrieval request . Otherwise, 1) malicious client can retrieve public data such as Non-Fungible Token(NFT) and delete it or 2) malicious client does not give her secret key after retrieval request comes. Therefore, the data is encoded with the private key, and decoded with public key. For this, we adopt asymmetric encoding. With asymmetric encoding, client encodes the data to replica with its secret key and send it to server. Server just stores the replica and executes PoRep and PoSt subsequently. Server cannot generate replica arbitrarily since the given replica is encoded with client's secret key. In this protocol, however, it is impossible for server to participate newly when the client disappear.Or even if the client still exist, it is impractical that the client encodes the data whenever new client comes. This can be solved by adopting threshold cryptosystem based on Secret Sharing(SSS) technique []. **TODO : worker..committee.. simple structure description of delegation will be written – SH**

3 BACKGROUND

The order of subsection will be adjusted later – SH Building blocks are introduced in this part. **TODO : Define what property is required for Asymmetric sealing(encoding) – SH**

3.1 Possible attacks

Existing PoS protocol [1] guarantees that a prover provides dedicated storage for the data which client asks by responsing for the challenge at some time. Further, Filecoin [4] proposes stronger notion of possible attacks that malicious provers could abuse to earn reward for the storage they are not providing.

- Sybil attack : Malicious server cheats as it stores more copies than it actually stores, further gets reward by creating multiple sybil identities.
- Outsourcing attack : Malicious server commits to store data more than they actually can store and responses for the challenge by quickly bringing it from other(outsourced) servers.

- Generation attack : Malicious server pretends to store data where they are not storing the data, instead it generates whenever challenge comes.

In addition to above attacks, we should consider possible attack of malicious client since the client encodes by itself in our construction. **Maybe we should mention encoding is on client's side. – SH**

- Denying attack : Malicious client stores public data and remove the data after decoding, or further it doesn't respond to retrieval request.[1] **Tentative – SH**

3.2 Threshold Encryption

Threshold public key encryption [2] is a public key encryption where the private key is shared to decryption servers. In the (t, k)-threshold encryption system, k decryption servers are given ciphertext and distributed private key and send partial decryption piece to *combiner* which gathers k out of t decryption pieces and combine those in order to get the whole decryption. We give informal definitions for using it as cryptographic primitive in our construction. **Further detail will be added – SH**

Definition 3.1 (Threshold Encryption). The (t, k)-threshold encryption for security parameter 1^λ, and a message M is a tuple of algorithm KeyGen, Encrypt, Share, Decrypt, Combine with fololwing syntax:

- $\mathsf{KeyGen}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$ generates a key pair of public key and secret key $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Encrypt}(\mathsf{pk}, M) \rightarrow C$ encrypts a message M with public key pk and outputs ciphertext C.
- $\mathsf{Share}(\mathsf{sk}) \rightarrow [\mathsf{sk}_i]_1^k$ distributes secret key among k decryption servers.
- $\mathsf{Decrypt}(C, \mathsf{sk}_i) \rightarrow M_i$ outputs a piece of decryption M_i with ciphertext C and shared secret key piece sk_i.
- $\mathsf{Combine}(\vec{M}) \rightarrow M$ reconstructs the original message M from $\vec{M}$ which is the set of decryption pieces from t number of decryption servers.

3.3 Set Membership Proof

3.4 Decentralized Storage Network(DSN)

Filecoin [4]

Our DSN is implemented on the blockchain environment. Simple Explanation will be here

3.4.1 DSN participant. Main entities involving the proposed DSN are as below.

- Client : The node who asks server to store data in DSN
- Worker(Committee) : The node who works computation of *Sealing* on behalf of client. It can earn reward if sealing works correctly. We call the group of workers as *Committee* when the workers are on same client and data.
  Detailed description would be better in the below. – SH
- Server : The node who stores the data which some clients send to store. A server should respond correctly for the challenge with the proof proving that she is storing the data. It can get reward when the proof is correct. Otherwise, it gets penalty.

We also refer to worker and server as seal miner and storage miner respectively. **Tentative – SH**

3.4.2 Data structure. In this section, we describe the data structures used in our DSN construction and its notation.
which can be added? Sector..? – SH

Replica: A replica is an output of encoding. A data is transformed into a replica after encoding, and server stores the replica in its local storage. When the retrieval request occurs, a replica is restored to original data.

Order: An order is a data structure sustaining the statement of requesting/providing services. An order O includes who makes the order, the amount of data(or storage), bid price, and other auxiliary information. There are three types of order in our construction.

- $O_{\text{client}} := \{clientID, \text{sizeD}, \text{period}, \text{price}, \text{commD}\}$: Order that client bids. It has client id $clientID$, the size of data sizeD, period how long to store, bid price, and the commitment of data commD.
- $O_{\text{server}} := \{serverID, \text{minSize}, \text{period}, \text{price}\}$: Order that server bids. It has server id $serverID$, period how long a server can provide storage, and bid price.
- $O_{\text{deal}} := \{clientID, serverID, \text{sizeD}, \text{period}, \text{price}, \text{commD}\}$: Order for done deal.

3.4.2 DNS Ledger $\mathcal{L}$: Tx, OrderBook, Block else? – KB

Orderbook: An orderbook is a bunch of orders. Orderbook gathers all kinds of order, and it is publicly opened. Note that with public orderbook, a client can observe which O_{server} is the lowest, and the market refer which O_{server} a client make a contract with.

Slice: A slice is a piece of the data which client is storing in DSN. A data is divided into the number of slices. A data is handled as slice unit rather than whole data.

Accumulator: An accumulator is authenticated data structure where the slices of data are accumulated. A server generates an accumulator for set membership proof proving that it is storing the data which client asks. We can instantiate as Merkle tree, or RSA accumulator.

4 ASYMMETRIC SEALING: ENCODING SCHEME FOR PROOF-OF-REPLICATION

Our focus in this work is on designing the PoRep protocol concisely and efficiently. For the purpose, we devise a new encoding scheme, *Asymmetric Sealing*, which can prevent possible attacks in section 3.1 without delay. For making protocol concise, we shift the subject of encoding from server to client. In Filecoin, a miner which we call server encodes a data for generating replica. Simple encoding is a vulnerable for the possible attacks such as generation attack. Thus, a encoding should be delayed enough so that malicious miner cannot generate a replica arbitrarily in time without storing it. However, the delay lets the protocol more complicated. Since a miner is responsible for the encoding, it should convince a client that the replica is encoded with the data which client requests to store. Server therefore proves for the encoding in the circuit in case of Filecoin. By letting the encoding be done by client, delay

[1] Malicious client couldn't decode the replica even though the retrieval request comes.

is not required no longer and proving for encoding is skipped consequently.

Asymmetric sealing consists of two phases largely : 1) Encoding phase which is called with store request, and 2) Decoding phase which is called with retrieval request. In the encoding phase, data is transformed into replica, and the replica is sent to the server. When the retrieval request occurs, server sends the replica to client and client recovers(decodes) data from the replica. Roughly, the encoding of asymmetric sealing is executed with the clients' secret key so that malicious server cannot make replica herself. In contrast, decoding is executed with public key. Denying attack can be prevented with public decoding. For example, assume that some client stores Non-Fungible Toekn(NFT) of other's to the server. After time goes on, the client brings the NFT back and simply removes the NFT. However, the danger of such monopolizing data of others or public data is highly reduced by public decoding. That is, if anyone can decode the data, the data can float in distributed manner. Surely, this is under the premise that recovering exactly the original data should be available with decoding.

Although, seemingly this naive approach is plausible enough, it still has consideration. It is desirable that a new server can participate in DSN after the first deal is done. However, after the first deal is done, it is difficult for a server to join newly for the following reasons. After the client encodes the data and send a replica to the participating server, it could empty out. Even in the situation where the client maintains the replica, it is contrary to storing the data in the DSN which is the original purpose. That is, if the client stores the replica, it doesn't have to put the data to DSN. Rather, it just needs twice storage comparing with storing the data in its own storage, not entrusting to DSN.

Therefore, we improve naive approach by adopting threshold technique. As explained in 3.2, t-out-of-k shared secret can reconstruct whole secret. A client plays a role as dealer, and we call the nodes getting a shared secret as worker(3.4.1). A client generates a key pair $(\mathsf{pk}, \mathsf{sk})$. Then, it shares its secret to k workers(committee). When involving a new server, t-out-of-k workers(committee) combine together and recovers the whole secret. At last, the committee encodes the data and send replica to the server. It is no longer problem that a new server comes after the replica is sent at first. Additionally, corruption of some workers is tolerated with threshold cryptosystem. For achieving robustness of threshold cryptosystem, each shared computation is verified with the proof of worker.

Definition 4.1. (*Asymmetric Sealing*) Asymmetric sealing AS for security parameter 1^λ, the data $\mathcal{D}$, and the replica $\mathcal{R}$ is a tuple of algorithm KeyGen, Share, Seal, Combine, Retrieve that works as follows and satisfies the notions of *recoverability, unforgeability, and robustness*.

- $\mathsf{Setup}(1^\lambda) \rightarrow (\mathsf{pk}, \mathsf{sk})$ generates a key pair of public key pk and secret key sk.
- $\mathsf{Share}(\mathsf{sk}) \rightarrow ([\mathsf{sk}_i]_1^k)$ distributes pieces of secret key among k workers.
- $\mathsf{Seal}(\mathcal{D}, \mathsf{sk}_i) \rightarrow (\mathcal{R}_i, \pi_i)$ generates a slice of replica $\mathcal{R}_i$ and its proof π_i taking the data $\mathcal{D}$ and shared secret sk_i as input.
- $\mathsf{VfySeal}(\mathcal{R}_i, \pi_i) \rightarrow 0/1$ verifies whether replica slice $\mathcal{R}_i$ is correct or not. It outputs 1 if the proof for π_i is valid, 0 otherwise.
- $\mathsf{Combine}(S_\mathcal{R}) \rightarrow \mathcal{R}$ combines complete replica $\mathcal{R}$ taking the set of replica slices $S_\mathcal{R}$.
- $\mathsf{Retrieve}(\mathsf{pk}, \mathcal{R}) \rightarrow \mathcal{D}$ retrieves the replica $\mathcal{R}$ and recovers the original data $\mathcal{D}$.

Recoverability. For any data $\mathcal{D}$, and its valid replica $\mathcal{R}$, decoding $\mathcal{R}$ should be able to recover original data $\mathcal{D}$. Formally, for valid key pair $(\mathsf{pk}, \mathsf{sk})$, data $\mathcal{D}$, and its valid replica $\mathcal{R}$, it holds:

$$Pr\left[\mathcal{D} = \mathsf{AS.Retrieve}(\mathcal{R}, \mathsf{pk})\right] = 1$$

Unforgeability. For any data $\mathcal{D}$ and valid key pair $(\mathsf{pk}, \mathsf{sk})$, a valid replica $\mathcal{R}$ cannot be generated arbitrarily if fewer than t workers participate in the protocol. Formally, for valid secret key pieces $[sk_i]_1^k$, a subset of valid replica slices S_R and arbitrary replica subset $S_{R'}$, it holds:

$$Pr\left[\mathcal{R} = \mathcal{R}' \middle| \begin{array}{c} \mathcal{R} \leftarrow \mathsf{AS.Combine}(S_\mathcal{R}) \\ \mathcal{R}' \leftarrow \mathsf{AS.Combine}(S_{\mathcal{R}'}) \\ |S_\mathcal{R}| \geq t,\ |S_{\mathcal{R}'}| < t \end{array}\right] \leq negl(\lambda)$$

Robustness. No ones less than t-out-of-k cannot generate a valid replica for the data $\mathcal{D}$. Formally, for threshold t, and the size of the entire worker set k, it holds:

$$Pr\left[\mathsf{VfySeal}(\mathcal{R}_i{}', \pi_i{}') = 1 \middle| \begin{array}{c} (\mathcal{R}_i', \pi_i' \leftarrow \mathsf{AS.Seal}(\mathcal{D}, \mathsf{sk}_i)) \\ \mathsf{sk}_i \notin [sk]_1^k \end{array}\right] \leq negl(\lambda)$$

4.1 Building Blocks

4.1.1 Public Key Encryption(PKE). As defined in *definition* 4.1, the form of the encoding/decoding in asymmetric sealing is similar with public key encryption in terms that sender encrypts the original message to ciphertext and receiver decrypts to recover plaintext from the ciphertext. Therefore, it seems that the public key encryption(PKE) can be used as a building block for asymmetric sealing(AS) if the public key encryption scheme satisfies the notions of asymmetric sealing. However, we cannot make use of it completely identical way since it is vulnerable for *attacks* in 3.1. If we adopt public key encryption for encoding, replica is generated with client's public key. Since the public key is literally public, a malicious server could have power to generate replica even if it doesn't store the replica in fact.

Instead, we use public key encryption slightly different way. For the encoding, decryption of PKE is used since it is same in terms of computing with secret key. When the retrieval request occurs, encryption of PKE is called. Note that both retrieve of AS and encryption of PKE are dealt with public key. At first glance, first decryption-later encryption concept seems peculiar but it is reasonable if we look closely. How we use public key encryption is the computation form, not the concept. What it means is, what we want from public key encryption is that the computation on some input is handled with different key pair, public key and secret key. Therefore, any public key encryption scheme meeting the notions of asymmetric sealing can be applied to our construction.

4.1.2 Zero-Knowledge Proofs(ZKP). For satisfying robustness, each worker has to prove that it computes correctly. That is, worker proves that it computes replica correctly with shared secret and the data given by client. Replica slices can be combined only if verification for each slice is passed. It would be better that server verifies the proof since server is more powerful node than client, which is usually light node. Thus, mitigating computation cost for client is more beneficial to broaden the system. However, simply recomputing replication have to be banned. Server should have no power to encode itself for preventing generation attack. Secret key pieces are leaked through verification in this case. ZKP is suitable to prove sealing while hiding secret key pieces.

Or zk-SNARKs? succinctness – SH

Figure 1 shows the asymmetric sealing construction using public key encryption and ZKP(Π) as a building block. Later, we describe an instantiation of asymmetric sealing in section 4.3.

4.2 Security Proof

Theorem 4.2. *Let $\mathcal{D}$ be the data, and $\mathcal{R}$ be the replica. The construction in figure 1 is a secure asymmetric sealing:reoverable, unforgeable, robust under sercer public key encryption scheme PKE=(KeyGen, Enc, Dec) and zero-knowledge proof scheme* $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$.

4.3 Instantiation

Here we represent an instantiation of our Asymmetric Sealing with RSA encryption and zk-SNARKs. Recall that public key encryption scheme is used to encode/decode and zero-knowledge proof is applied to verification of replica slices. Likewise, we apply RSA encryption as a method of encode/decode and zk-SNARKs for verification of sealing. Let us describe in more detail.

Client shares its secret key sk through following steps. Client selects a random polynomial $A(x)$ of degree t where $A(0) = \mathsf{sk}$ and set $\mathsf{sk}_i \leftarrow A(i)$. Given data $\mathcal{D}$ and shared secret sk_i, worker encodes replica slice $\mathcal{R}_i \leftarrow \mathcal{D}^{\mathsf{sk}_i} \bmod N$. Then, worker makes the proof for convincing server that $\mathcal{R}_i$ is encoded exactly with $\mathcal{D}$ and sk_i. If server accepts, workers combine t replica slices to make whole replica and send it to server. The proof system is constructed with zk-SNARKs. When retrieval request occurs, server sends $\mathcal{R}$ to client, and client recover original data by $\mathcal{D} \leftarrow \mathcal{R}^{\mathsf{pk}}$.

A full description of asymmetric sealing is in figure 2.

5 CONSTRUCTION

As mentioned in section 2, our protocol does not depend on VDE, instead we design and adopt a unique encoding scheme : Asymmetric Sealing. **Just tentative – SH** Note that the asymmetric sealing is unique, but it can be constructed from general cryptographic tools.

Why do we need SSS? New validator participation can be smooth; otherwise, whenever new validator try to participate, client should encode again, or client delete the data(Cannot participate then) – SH

Algorithm ?? represents instantiation of asymmetric sealing, we may add generalized scheme(in the previous subsection or ...) – SH

The overview of our protocol is in figure 3

$\underline{\mathsf{Setup}(1^\lambda, R)}$:
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$
$\mathsf{crs} \leftarrow \Pi.\mathsf{Setup}(1^\lambda, R)$

$\underline{\mathsf{Share}(\mathsf{sk})}$:
Select a random polynomial $A(x)$ with degree t such that $A(0) = \mathsf{sk}$
$\mathsf{sk}_i \leftarrow A(i)$ for all $i \in I$
return $([\mathsf{sk}]_1^k)$

$\underline{\mathsf{Seal}(\mathsf{crs}, \mathcal{D}, \mathsf{sk}_i)}$:
$\mathcal{R}_i \leftarrow \mathsf{PKE.Dec}(\mathcal{D}, \mathsf{sk_i})$
$pi_i \leftarrow \Pi.\mathsf{Prove}(\mathsf{crs}, \mathcal{D}, \mathsf{sk}_i)$
return $(\mathcal{R}_\mathrm{i}, \pi_\mathrm{i})$

$\underline{\mathsf{VfySeal}(\mathsf{crs}, \mathcal{R}_i, \pi_i, S_\mathcal{R})}$:
assert if $\Pi.\mathsf{Verify}(\mathsf{crs}, \mathcal{R}_i, \pi_i) = 0$
$S_\mathcal{R} \leftarrow S_\mathcal{R} \bigcup \{\mathcal{R}_i\}$
return $\mathsf{S}_\mathcal{R}$

$\underline{\mathsf{Combine}(S_\mathcal{R}, T)}$:
$\lambda_i \leftarrow \prod_{j \in T - \{i\}} \frac{z-j}{i-j}$
$a_i \leftarrow \lambda_i(0)$
$\mathcal{R} \leftarrow \prod \mathcal{R}_i^{a_i}$
return $\mathcal{R}$

$\underline{\mathsf{Retrieve}(\mathsf{pk}, \mathcal{R})}$:
$\mathcal{D} \leftarrow \mathsf{PKE.Enc}(\mathcal{R}, \mathsf{pk})$
return $\mathcal{D}$

Figure 1: Asymmetric sealing: Our encoding scheme for DSN. We denote T as the index set of worker. We should modify security parameter λ. It is used as 1^λ and coefficient $\lambda_i(0)$ – SH

5.1 Proof-of-Replication(PoRep)

In this section, we describe the PoRep scheme based on *Asymmetric Sealing(AS).*

Setup(1^λ):

$\text{crs}_1 = (\mathbb{G}, N) \leftarrow \mathcal{G}(1^\lambda)$
$\text{pk} \xleftarrow{\$} [1, \phi(N))$
$\text{sk} \leftarrow \text{pk}^{-1} \bmod \phi(N)$
$\text{crs}_2 = \Pi.\text{Setup}(1^\lambda, R^{seal})$
return $(\text{pk}, \text{sk}, \text{crs}_1, \text{crs}_2)$

Share(sk):

Select a random polynomial $A(x)$ with degree t over $\mathbb{Z}_N$ such that $A(0) = \text{sk}$
$\text{sk}_i \leftarrow A(i)$
return sk_i

Seal($\text{crs}_1, \text{crs}_2, \mathcal{D}, \text{sk}_i$) :

$\mathcal{R}_i \leftarrow \mathcal{D}^{\text{sk}_i} \bmod N$
$\pi_i \leftarrow \Pi.\text{Prove}(\text{crs}_2, \mathcal{D}, \mathcal{R}_i; \text{sk}_i)$
return $(\mathcal{R}_\text{i}, \pi_\text{i})$

VfySeal($\text{crs}_2, \mathcal{R}_i, \pi_i, S_\mathcal{R}$):

Assert if $\Pi.\text{Verify}(\text{crs}_2, \mathcal{R}_i, \pi_i) = 0$
$S_\mathcal{R} \leftarrow S_\mathcal{R} \bigcup \{\mathcal{R}_i\}$
return $\text{S}_\mathcal{R}$

Combine($S_\mathcal{R}, T$):

$\lambda_i \leftarrow \prod_{j \in T - \{i\}} \frac{z-j}{i-j}$
$a_i \leftarrow \lambda_i(0)$
$\mathcal{R} \leftarrow \prod \mathcal{R}_i^{a_i}$
return $\mathcal{R}$

Retrieve(crs_1, pk):

$\mathcal{D} \leftarrow \mathcal{R}^{\text{pk}} \bmod N$
return $\mathcal{D}$

Figure 2: Instantiation of Asymmetric Sealing: Our encoding scheme with RSA encryption and zk-SNARKs. Π denotes zk-SNARKs such as Groth16 [3].

5.2 Proof-of-Spacetime(PoSt)

In this section, we describe PoSt with above scheme

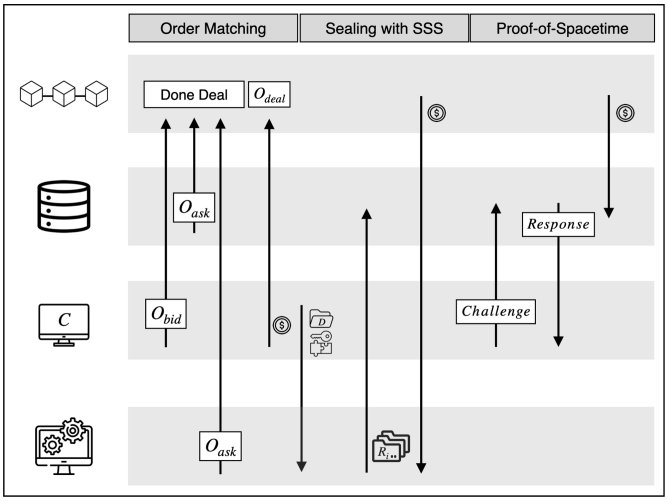

Figure 3: Protocol Overview

Algorithm 1 Proof-of-Replication

Setup($1^\lambda, \mathcal{D}$) :

$(\text{pp}, \text{pk}, \text{sk}) \leftarrow \text{AS.Setup}(1^\lambda)$
$(\mathcal{D}, [\text{sk}_i]_1^k) \leftarrow \text{AS.Share}(\text{pp}, \text{sk}, \mathcal{D})$
for all $\mathcal{W}_i \in \vec{\mathcal{W}}$ **do**
  Send $(\mathcal{D}, \text{sk}_i)$ to all workers $\mathcal{W}_i$
  $\mathcal{R}_i \leftarrow \text{AS.Seal}(\mathcal{D}, \text{sk}_\text{i})$
end for
$\hat{\mathcal{R}} \leftarrow \text{AS.Combine}(\vec{\mathcal{R}}, a_i)$
return $\hat{\mathcal{R}}$

Store(pp, $\hat{\mathcal{R}}$) :

Parse $\hat{\mathcal{R}}$ as $\hat{\mathcal{R}}_1 | \hat{\mathcal{R}}_2 | \cdots | \hat{\mathcal{R}}_n$
$g \xleftarrow{\$} \mathbb{G}$
$\text{pp}_\text{mem} \leftarrow (\text{pp}, g)$
$\mathcal{S} \leftarrow \mathcal{S} \cup \{\hat{\mathcal{R}}_1, \hat{\mathcal{R}}_2, \cdots, \hat{\mathcal{R}}_n\}$
$\text{acc} \leftarrow \text{Mem.accumulate}(\text{pp}, \text{g}, \mathcal{S})$
return $(\text{pp}_\text{mem}, \mathcal{S}, \text{acc})$

Prove($\text{pp}_\text{mem}, \hat{\mathcal{R}}, \text{acc}, \text{c}$) :

$\pi_{mem} \leftarrow \text{Mem.prove}(\text{pp}_\text{mem}, \text{acc}, \text{c})$
return π_mem

Verify ($\text{pp}_\text{mem}, \text{acc}, \pi_\text{mem}, \text{c}$) :

Reject if $\text{Mem.verify}(\text{pp}_\text{mem}, \text{acc}, \pi_\text{mem}, \text{c}) \neq 1$

Algorithm 2 Proof-of-Spacetime(PoSt)

Setup($1^\lambda, \hat{\mathcal{R}}$) :

Parse $\hat{\mathcal{R}}$ as $\mathcal{R}_1, \mathcal{R}_2 \cdots \mathcal{R}_n$ where $|\mathcal{R}_i|$

5.3 Decentralized Storage Network

Decentralized storage network is block chain system that uses storage resources. we construct two phase. Deal phase and Storage phase. In Deal phase, client make deal in order to storage client's data to storage provider's local storage.

- *Deal phase* : to make O_{deal}, client make O_{client} and storage providers make O_{server}
- *Storage phase* : storage providers make PoSt proof to prove they storage specific Replica

Deal protocol

O_{client} := {*clientID*, sizeD, period, price, $\mathcal{N}$, commD}
O_{server} := {*serverID*, minSize, period, price}
O_{deal} := {*clientID*, *serverID*, sizeD, period, price, commD}

Client algorithm :

We explain algorithms to be executed by the client in Deal protocol. **There is a possibility that clients and servers can collude to obtain mining power without storing data. To solve this problem, client leave match order algorithm to the committee. most matched order's price will be different, so to prevent collude, The amount of money that differs is sent to the Committee. until O_{deal} is created by committee and storage server, the client doesn't know who will save data. client and storage server can't collude. – KB**

makeOrder($\mathcal{D}$, $Info$) :
  $commD \leftarrow H(\mathcal{D})$
  Parse $Info$ as {*clientID*,*sizeD*,*period*,*price*,$\mathcal{N}$:*number of copies*}
  $O_{\text{client}} \leftarrow$ {*clientID*, sizeD, period, price,$\mathcal{N}$, *commD* }
  register the O_{client} in the *OrderBook*
  return Success if O_{client} is registered well, else **False**

send($\mathcal{D}, O_{\text{deal}}$) :
  Parse O_{deal} as {*clientID*,*serverID*,*sizeD*,*period*,*price*,*commD*,$\mathcal{W}$}
  (pp, pk, sk) $\leftarrow AS.\text{Setup}(1^{\lambda})$
  $(\mathcal{D}, [\text{sk}_i]_1^k) \leftarrow AS.\text{Share}(\text{pp}, \text{sk}, \mathcal{D})$
  $r \xleftarrow{\$} Z_p^*$
  for $C_i \in \mathcal{W}$ **do**
    $ind := \text{sk}_i's$ index i
    transfer [$\mathcal{D}$, r ,sk_i, ind, *serverID*] to C_i
  end for

Committee algorithms :

Committee : who have a lot of token. when they act maliciously, their network will lose credibility and token price fall down. so they should act honest.

matchOrder(*OrderBook*) :
  for $O_i \in OrderBook$ **do**
    if O_i is not O_{client} then, do next step
    $List_{O_i} \leftarrow []$
    n $\leftarrow O_i.N$
    for $O_j \in OrderBook - \{O_i\}$ **do**
      check $O_j == O_{\text{server}}$
      check O_i.price $\leq O_j$.price $*$ *param*
      check O_i.period $\leq O_j$.period
      check O_i.sizeD $\leq O_j$.sizeD
      If all checks are passed, $List_{O_i}.append(O_j)$
    end for
    (a) select committee nodes set $\mathcal{W}$ who will participant work algorithm. Size of set $\mathcal{W}$ is defined system param.
    (b) select random **n** distinct O_{server} from the $List_{O_i}$
    (c) mapping O_i to **n** distinct O_{server}
    (d) Make n distinct O_{deal} and send to *client*, n *storage servers*
    (e) delete O_i and update mapped O_{server} in *OrderBook*
  end for

work($\mathcal{D}, r, \text{sk}_i, ind, serverID$) :
  $\mathcal{D}' \leftarrow \mathcal{D}_1 \oplus r \,||\, \mathcal{D}_2 \oplus r \,||\, \cdots ||\, \mathcal{D}_k \oplus r$
  $\mathcal{R}_i \leftarrow AS.Seal(\mathcal{D}', \text{sk}_i)$
  transfer [$\mathcal{R}_i, r, ind$] to *serverID*

Storage Server algorithms :

storage sever nodes are storing client's data to get storage fee and mining power. when they make O_{deal}, need to lock *collateral* to prevent act maliciously.

makeOrder($Info$) :
  Parse $Info$ as {*serverID*, sizeD, period, price}
  $O_{\text{server}} \leftarrow$ {*serverID*, sizeD, period, price}
  register the O_{server} in the *OrderBook* and lock *collateral*
  return Success if O_{server} is registered well, else **False**

receive(O_{deal}) :
  Parse O_{deal} as {*clientID*,*serverID*,*sizeD*,*period*,*price*,*commD*,$\mathcal{W}$}
  for $C_i \in \mathcal{W}$ **do**
    receive [$\mathcal{R}_i$,r, ind_i] from C_i
  end for
  $\vec{\mathcal{R}} \leftarrow (\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_k)$
  $\vec{Ind} \leftarrow (Ind_1, Ind_2, \cdots, ind_k)$ where k = $|\mathcal{W}|$
  $\hat{\mathcal{R}} \leftarrow AS.Combine(\vec{\mathcal{R}}, \vec{Ind})$
  $\text{pp}_{\text{mem}}, \mathcal{S}$, acc $\leftarrow$ PoRep.Store(pp, $\hat{\mathcal{R}}$)
  Check decrypt($\hat{\mathcal{R}}$) == $\mathcal{D}$?? – KB
  return $\hat{\mathcal{R}}$, pp_{mem}, $\mathcal{S}$, acc

Storage protocol

when protocol is finished or storage server get mining power, storage server must prove that they store $\mathcal{R}$ without loss
Suppose honest client never send his secret key to Storage Server. Storage Server delete client's $\mathcal{R}$ then server can't reseal $\mathcal{R}$ and make PoSt proof. – KB
It May not be necessary

Client algorithms :

When client want to retrieve own data or other client's data, client make and deploy retrieve Transaction.

retrieve($ServerId$, $commD$) :

transfer $commD$ to $ServerId$ and wait respond
recieve $flag$ from $ServerId$
if $flag$ == ***True***
run file transfer protocol

Storage Server algorithms :

Because of asymmetric encoding properties, only sk owner can make $\mathcal{R}$. That means if storage provider delete his own $\mathcal{R}$ then he can't re produce his $\mathcal{R}$ and proof. so we need to check Storage server run prove algorithm when they get mining powers or at the end of protocol. verify algorithm is verify proofs other storage provider's proof

prove($\hat{\mathcal{R}}$, $\mathsf{pp}_{\mathsf{mem}}$, $\mathcal{S}$, acc, c) :

$\pi_{mem} \leftarrow \mathsf{PoRep.Prove}(\mathsf{pp}_{\mathsf{mem}}, \hat{\mathcal{R}}, \mathrm{acc}, \mathrm{c})$
return π_{mem}

response($clientId$, $commD$) :

$\mathcal{S}_{O_{\mathbf{deal}}}$:= storage Server's set of $O_{\mathbf{deal}}$
for $O_{\mathbf{deal}} \in \mathcal{S}_{O_{\mathbf{deal}}}$ **do**
If $commD == O_{\mathbf{deal}}.commD$:
$\mathcal{D} \leftarrow AS.retrieve(\mathsf{pp}, \hat{\mathcal{R}}, \mathsf{pk})$
transfer ***True*** to $clientId$
run file transfer protocol
return
end for
transfer ***False*** message to $clientId$

6 EVALUATION

Implementation environment,, microbenchmark..benchmark

7 SECURITY

7.1 Security proof for our construction

Theorem 7.1. *Let $\mathcal{D}$ be the data, and $\mathcal{R}$ be the replica. The construction in figure 1 is a secure asymmetric sealing: recoverable, unforgeable, robust under secure public key encryption scheme* $\mathsf{PKE} = (KeyGen, Enc, Dec)$.

7.2 Attack-resistance for our construction

7.3 Generation Attack-Resistant

Only for RSA-like protocol is suitable for our construction, asymmetric sealing. Security depends on RSA Encryption – SH

Theorem 7.2. *For any threshold public key encryption TPKE with security parameter 1^λ and data $\mathcal{D}$, asymmetric sealing is generation attack-resistant.*

Proof. First, we start from review the generation attack. Generation attack is that the malicious server doesn't store the replica, instead it generates replica quickly whenever challenge comes. In Filecoin[4], since server encodes the data(sealing) itself, it has possibility of the generation attack. Thus, [4] adopts delay encoding. However, asymmetric sealing is executed by client, not server, viewpoint is slightly different from [4]. In asymmetric sealing, server cannot produce a replica normally since it just gets replica from client. However, if the TPKE is not secure, a malicious server can generate a replica without having secret key of the client. Therefore, Generation attack resistance depends on TPKE security. □

7.4 Denying Attack-Resistant

Theorem 7.3. *For any threshold public key encryption TPKE with security parameter (1^λ) and data $\mathcal{D}$, asymmetric sealing is denying attack-resistant.*

Proof. □

7.5 Security for Other Properties

Theorem 7.4. *For any threshold public key encryption TPKE with security parameter (1^λ) and data $\mathcal{D}$, asymmetric sealing is resistant against sybil attack and outsourcing attack.*

Proof. □

Unforgeability. we demonstrate our AS scheme hold Unforgeability. Suppose adversary who can break our AS scheme with non negligible probability, then using that adversary we can break OAEP-RSA's semantic security. we will define $Game^{OAEP-Unforgeability}_{A,\mathcal{A}_0,\mathcal{A}_1}$, Adversary wants to break OAEP-RSA semantic security and other adversary wants to break our scheme. – KB

$Game^{OAEP-Unforgeability}_{A,\mathcal{A}_0,\mathcal{A}_1}$

(a) Adversary run q OAEP-RSA.Enc() query
(b) Adversary select m_0, m_1 and send to OAEP-RSA.Enc()
(c) select random bit $b \leftarrow \{0,1\}$ and run $c \leftarrow OAEP-RSA.Enc(m_b)$. send ciphertext c to Adversary.
(d) $\mathcal{D} \leftarrow c$ and send $\mathcal{D}$ to $\mathcal{A}_0$
(e) $\mathcal{A}_0$ select random sk$'$ ans send to $\mathcal{A}_1$
(f) $\mathcal{A}_1$ return $\mathcal{R}'$ to Adversary. If $\mathcal{R}'$ is valid then Adversary can break OAEP-RSA semantic security.

REFERENCES

[1] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*. 598–609.
[2] Yvo Desmedt and Yair Frankel. 1989. Threshold cryptosystems. In *Conference on the Theory and Application of Cryptology*. Springer, 307–315.
[3] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 305–326.
[4] Protocol Labs. 2014. Filecoin: A Decentralized Network. https://filecoin.io/filecoin.pdf. (2014).