0Chain Storage Protocol
Version 2

Thomas H. Austin*[1,2], Paul Merrill[1], Siva Dirisala[1], and Saswata Basu[1]

[1]0chain LLC
[2]San José State University

August 31, 2018

Abstract

Since its creation by Satoshi Nakomoto, the blockchain has revolutionized cryptocurrencies. The blockchain provides a means of providing consensus, so that a group of machines can act as a trusted third party without trusting any specific machine.

The blockchain is the backbone for many decentralized applications (dApps), providing many services that were traditionally only offered by large, centralized institutions. However, the development of dApps is often held back by the limitations of the blockchain; many mining consensus algorithms are slow, and adding additional functionality to the blockchain can slow it further.

0chain seeks to provide a blockchain that provides rapid consensus, and yet one that still offers the functionality needed for the development of powerful dApps. This paper seeks to address that tension of concerns for storage. By offloading most of the storage work to a different group of machines than the miners, the miners can focus on rapid consensus. The storage providers, knows as blobbers, provide proof that they are storing the correct contents by writing a Merkle root to the blockchain. Special signed receipts called *markers* ensure that the blobbers are rewarded for any work that they do; a challenge protocol verifies that the blobbers continue to store the data.

1 Introduction

Bitcoin [11] introduced the world to decentralized digital currency. It relies on the blockchain that provides decentralized consensus while guaranteeing both immutability and finality of transactions. Critically, the protocol is designed so that each actor's economic incentives are aligned with the proper behavior according to the protocol.

Since its advent, many have realized that the blockchain technology that Bitcoin is built on can be useful in other domains. 0chain promises a blockchain-based architecture for distributed storage and computation. By owning 0chain tokens, clients can build distributed applications (dApps) without relying on a single source for providing their storage. Additionally, in the 0chain economy, tokens are not consumed when employed, allowing its clients to continue to use these tokens indefinitely.

As part of its architecture, 0chain offers data storage, which is the focus of this paper. A key distinction of our system from other blockchain storage solutions is that we divorce the role of mining from that of providing storage. Computers that provide storage are referred to as *blobbers*. Blobbers are neither responsible nor required for mining. In this manner, we lighten the load on our mining network and enable fast transactions on a lightweight blockchain.

*thomas.austin@sjsu.edu

Though the clients and blobbers work off-chain, their interaction is monitored through transactions on the blockchain. The client initially writes a transaction to the blockchain declaring their[1] intent and locking up their storage, ensuring that the blobber knows they won't be cheated by the client.

As the client and blobber interact, the client generates special signed receipts called *markers*. These markers act like checks that the blobber can later cash in with the blockchain. Once the interaction between client and blobber has concluded, the blobber writes an additional transaction to the blockchain, which redeems the markers for 0chain tokens and commits the blobber to a Merkle root [8] matching the data stored. The leaves of the Merkle tree must match markers sent from the client, preventing either the client or the blobber from defrauding each other.

After a file has been stored, a *challenge protocol* ensures both that the blobber continues to store the file and continues to be paid for that work. The mining network posts a transaction challenging the blobber to provide a block of data. The blobber must provide that block of data, the Merkle path, and the client-signed markers to prove that the right data is stored. The blobber is then rewarded or punished accordingly.

With our design, the majority of the work between clients and blobbers happens off-chain. The mining network is only involved enough to ensure that clients pay blobbers for their work and that the blobbers are doing the work that they have been paid to do.

The paper is outlined as follows: Section 2 reviews related work and discusses how our approach fits in this design space. Section 3 gives a high-level overview of our protocol, the entities involved, and some of the attacks that we are concerned with. Section 4 gives a more detailed description about how a client can lock their tokens in exchange for rights to system resources on the 0chain network. Section 5 discusses how a connection between a client and a blobber is initialized; since much of our protocol centers around establishing this connection, we go into particular detail here. Once the connection is established, the client and blobber may use either block level storage (Section 6) or file level storage. We initially focus on block level storage, outlining how reads and writes between a client and blobber happen in Section 6.1, with an optimized read-only mode described in Section 6.2; Section 6.3 shows how the mining network verifies that blobbers are storing the data that they claim, and also discusses how blobbers are compensated for storing data. Section 7 shows the additional changes needed in the protocol to provide file level storage instead. Section 8 discusses how a client can remove data from the network. Section 9 outlines how our network recovers when a blobber fails or otherwise disappears from our network, and Section 10 concludes.

2 Related Work

In Filecoin [4], miners prove that they are storing data through special Proofs-of-Spacetime, which are themselves based on Proofs-of-Replication [14]. Essentially, Filecoin miners store the data in an encrypted format using a modified form of cipher-block-chaining. This design deliberately introduces latency to their "Storage Market" network, preventing cheating miners from being able to produce the data in time. A second "Retrieval Market" relies on a gossip protocol to provide the needed data off-chain. The Filecoin literature discusses many interesting attacks, such as Sybil attacks, outsourcing attacks, and generation attacks. We review these attacks in the context of 0chain storage and outline our defense in Section 3.4.

Both Sia [15] and Storj [16] provide decentralized storage in peer-to-peer networks. These systems rely on Merkle trees made of the hashes of file contents, allowing them to (probabilistically) verify that the storage provider is actually storing the data that they claim to store without verifying the entire file. They guarantee the reliability of their systems by using erasure codes to divide data between the data storage providers.

MaidSafe ties its currency to data, paying data providers whenever data is requested. Its Proof-of-Resource [7] mechanism uses a zero knowledge proof to verify that data is actually stored. Unfortunately, details about the Proof-of-Resource design are scant, making it difficult to evaluate.

Miller at al. [10] discuss how to replace Bitcoin's Proof-of-Work with a Proof-of-Retrievability solution, which they use in a modified version of Bitcoin called Permacoin. Essentially, Permacoin miners prove that

[1] In English, there are a variety of ways to refer to a third person singular of indeterminate gender; all of them are problematic. With apologies to members of the National Socialist Grammar Party, we have elected to use "they" and "their" as the least bad of the terrible options available.

they are storing some portion of archival data. Since the proof itself provides part of the data, miners inherently serve as backup storage. In contrast, with our approach the storage providers do not serve as miners, and hence can specialize their machines for storage.

Ali et al. [1] show how a distributed public key infrastructure (PKI) can be built on top of a blockchain. Their initial experiments use Namecoin [12]. Due to concerns about the low number of miners in the Namecoin network, they migrate their system to the Bitcoin network. Their Blockstack naming and storage system relies on a *virtualchain* to introduce new functionality without changing the underlying blockchain. While their approach is similar to ours in that storage is done off chain, there is much less focus on ensuring that the storage providers are actually providing the data.

Other approaches, such as Burst's Proof-of-Capacity (PoC) [5] and SpaceMint's Proof-of-Storage [13] use storage for their consensus algorithms. The data itself is not intended to be useful outside of consensus, though Burst has plans to store "dual-use" data in their PoC3 consensus algorithm [5].

3 Overview

This section gives a high-level overview of our storage protocol design, including the entities in our system and the attacks of concern.

3.1 Players

Our system involves interaction between clients, who have data that they wish to store, and blobbers who are willing to store that data for a fee. Neither the client nor the blobber necessarily trust one another, so transactions are posted to a blockchain produced by a trusted mining network[2].

The design of our protocol seeks to minimize the load on the mining network, so miners are not directly involved in the data transfer between clients and blobbers. Nonetheless, the transactions posted to the blockchain assures clients that their data is stored and gives blobbers confidence that they will be paid for their service; if either party misbehaves, the blockchain has the information to help identify cheaters.

Each client includes an application responsible for encoding the data. Our system relies on erasure coding, which helps to provide increased resiliency for the data stored. Clients are assumed to have a certain amount of 0chain tokens.

In the context of storage, *miners* are responsible for accepting requests from clients, assigning storage to blobbers, locking client tokens to pay for their storage, and testing that blobbers are actually providing the storage that they claim.

A *blobber* is responsible for providing long-term storage. Blobbers only accept requests to write data that have been approved by the mining network, which helps to prevent certain attacks. Like clients, blobbers are expected to have a certain amount of 0chain tokens. Blobbers are paid in three ways:

- When data is read from them, the clients give them special markers that the blobber can redeem for 0chain tokens. These markers are detailed more in Section 6.1.
- Similarly, when clients write data to them, they give special markers.
- Whenever a blobber is storing data, they are randomly challenged to provide certain blocks of that data; if these challenges are passed, the mining network rewards the blobber with 0chain tokens.

3.2 Protocol Sketch

Our basic storage protocol can be broken into five parts.

First, clients must use tokens to reserve system resources. These resources include the amount of storage, the number of reads, and the number of writes needed for the data. In this stage, blobbers are also assigned to handle the client's allocation of data. The client's tokens are locked for a set period of time. Once the

[2]Note that this is very different from saying that they trust the individual miners that comprise the network.

time has elapsed, the client regains their tokens and loses their storage resources. Of course, a client may decide to re-lock their tokens to maintain their resources, though the number of tokens needed may change depending on the 0chain economy. 0chain's economic protocol is discussed in a separate document.

When a client wants to upload or update a file, they must write a group of transactions to the network declaring their intent. The mining network validates that the client is authorized to store the file and deducts their available free storage allocation.

Once the connection is established, the mining network no longer acts as an intermediary between the client and the blobbers. During this phase, the client generates markers to give to each blobber in exchange for access to system resources. Each blobber collects these markers and redeems them with the mining network once the exchange is complete; this transaction also notifies the mining network that the exchange has finished, and lets the network know that the client and blobber agree on the data that the blobber is expected to store. For read-only exchanges, a *lightning mode* allows clients to read from blobbers without first contacting the mining network; again, special markers ensure that the blobbers are still compensated for their work.

After the blobber has completed a declaring that data is stored, the mining network will periodically challenge the blobber to provide a randomly chosen block of that data. These challenges involve a carrot and stick approach; the blobber is punished if they fail the challenge, but rewarded with additional 0chain tokens when they pass the challenge. This design also ensures that blobbers are paid even when the data is not frequently accessed.

When the client no longer wishes to store the data, they issue a transaction to the network. Once it is finalized, the blobber deletes the data and the client may reuse their storage allocation.

3.3 Repair

Of course, things sometimes go wrong, and our system needs a way to handle these cases. The repair protocol arises when a blobber is identified as malicious, drops from the network, or is no longer considered suitable for storing the data that it has. When needed, the client can read the data from the network, reconstruct the missing fragment of data, and re-upload it to the network. If the client is online and monitoring the blockchain, the recovery process can be automated; that approach is discussed in Section 9.

3.4 Attacks

Filecoin [4] discusses many interesting attacks that can arise in blockchain-based storage systems. Here we provide a brief overview of these attacks to help the discussion on our protocols.

An *outsourcing attack* arises when a blobber claims to store data without actually doing so. The attacker's goal in this case is to be paid for providing more storage than is actually available. For example, if Alice is a blobber paid to store `file123`, but she knows that Bob is also storing that file, she might simply forward any file requests she receives to Bob.

Since the cheater must pay the other blobbers for the data, this attack is not profitable. Additionally, the mining network's blockchain gives some accounting information that can be analyzed to identify potential cheaters.

Another attack may occur if two blobbers collude, both claiming to store a copy of the same file. For example, Alice and Bob might both be paid to store `file123` and `file456`. However, Alice might offer to store `file123` and provide it to Bob on request, as long as Bob provides her with `file456`. In this manner, they may free up storage to make additional tokens. In essence, *collusion attacks* are outsourcing attacks that happen using back-channels. A *Sybil attack* in the context of storage is a form of collusion attack where Alice pretends to be both herself and Bob. The concerns are similar, but the friction in coordinating multiple partners goes away. Note that blobbers are assigned randomly, helping to further reduce the chance of collusion. Additionally, since we use erasure coding, two blobbers are unlikely to be storing identical data.

Following the approach used by Sia and Storj, we use erasure codes to help defend against unreliable blobbers in our network. Furthermore, we make more demands on the capabilities of our blobbers; if a

blobber repeatedly underperforms expectations, their reputation may suffer, and they risk being dropped from our network.

Finally, generation attacks may arise if a blobber poses as a client to store data that they know will never be requested or that can be regenerated easily. By doing so, they hope to be paid[3] for storing this data without actually needing the resources to do so.

Our primary defense against generation attacks is our economic protocol; essentially, blobbers would end up paying themselves and several other blobbers for the generated data. And since blobber selection is determined randomly by the mining network, the attacker has no guarantee that their machines will be selected. Our challenge protocol, discussed in Section 6.3, offers further protection. Essentially, blobbers are periodically required to provide the files that they store. Moreover, the mechanism for challenging them is also the mechanism that pays them; if a blobber found a way to avoid being challenged, they would never be paid.

4 Locking system resources

In order to store files, clients must use their tokens to purchase a certain amount of storage for a set amount of time; during this period, the clients' tokens are locked and cannot be sold. Likewise, to access or update their data, clients must purchase a certain number of reads and writes.

To lock tokens, the client posts a transaction to the mining network. The full details of the mining network transactions are beyond the scope of this document, but the transaction includes:

- The id of the client (`client_id`).
- The amount of storage (`amt_storage`).
- The number of reads (`num_reads`).
- The number of writes (`num_writes`).
- A `params` field for any additional requirements. We have kept this field deliberately vague to allow for future flexibility.

Different packages are available to the client to acquire the `amt_storage`, `num_reads`, and `num_writes` values desired. Note that `num_reads` may be used to access other clients' data, as long as the data owner permits the access.

At this time, an initial list of blobbers are elected to store the client's data allocation; this list may change over time as the blobber's availability and the client's needs change. This transaction also initializes a `read_counter` and `write_counter` for the client and blobber to use in their interactions. Section 6.1 reviews this counter's utility in both paying the blobber and charging the client.

5 Creating a connection

The core of our protocol relies on establishing a secure connection for data upload from the client to the blobber, curated by the mining network. Since neither the client nor the blobber trust one another, this protocol is designed so that both parties acting in their own best interest nonetheless benefit each other. Any transgressions can be identified by the mining network, which can punish the misbehaving party.

The goals of this protocol are:

- To ensure that the mining network knows what data the blobber has stored, with a public commitment from the blobber.

[3] In Filecoin, the authors seem to be more concerned with miners exploiting this approach to increase their voting power on the blockchain.

- To allow an optional handshake between the client and blobber when possible, meaning that the client and the blobber can both publicly commit to the same data.

Essentially, the client writes an initial transaction to the blockchain requesting a secure connection, which *might* include a Merkle root committing to the data[4]. After the interaction between the client and blobber, the blobber writes an additional transaction to the blockchain closing the exchange and committing to the data. A handshake happens when both the client and blobber specify the same Merkle root by the end of the second transaction. However, if the two are not in agreement, we rely on the blobber's Merkle root.

An alternate design would be to cancel the transaction when the client and blobber did not agree on the Merkle root. However, in the case of a client going offline unexpectedly in mid exchange, all of their updates would be lost. The Merkle trees help the client and blobber to identify blocks that do not match and negotiate a correction of the data once the client is back online.

In our system, data allocations are identified by a combination of the client's unique id (`client_id`) and the client-chosen `data_id`. Individual transactions are named by the triple of these two fields and a timestamp (`T`). In addition to creating unique ids for transactions, the timestamp also ensures that each request is fresh and helps defend against replay attacks.

In the interest of brevity, we may refer to the triple of `client_id`, `data_id`, and `T` as the transaction id (`trans_id`).

Figure 1 shows the process for establishing a connection between clients and blobbers. Note that all content read from the blockchain (indicated by dashed lines) is verified and can be trusted, though we omit the full details of the transaction from our figure for the sake of brevity. By the point this protocol begins, clients are assumed to have used their tokens to lock the necessary storage, number of reads, and number of writes that they need for their request.

Note that clients will typically create a group of transactions for their requests, depending on their erasure code settings, but these transactions are coordinated by the client. Though Figure 1 only shows a single transaction, it is important to remember that there are many of these transactions whenever data is stored in the network.

The steps of the protocol are as follows:

1. The client initializes the process by committing a transaction[5] to the blockchain. The transaction includes:
   - `client_id`.
   - `data_id`.
   - `T`.
   - (Optional) The Merkle root [8] of the data. Note that the client's Merkle root is only used to help establish a handshake between the client and blobber. If there is a discrepency, the blobber's Merkle root is used.
   - `upload_size_max`, indicating the maximum data that the client can upload to the blobber.
   - Additional parameters (`params`) for this request.
   - The client's signature, `cl_sig`. The full details of the signature are described by our mining protocol, but all fields listed here are included in the signature.
2. The client monitors the blockchain until the transaction has been finalized.
3. The client and blobber set up a secure connection[6].

[4]The client's commitment is optional in order to facilitate interactive updates, where the client might not know the Merkle root in advance.

[5]Each transaction is part of a transaction group, allowing the mining network to coordinate the transactions in the group in order to assign the best blobbers for the task. For simplicity's sake, we assume that each transaction is separate. However, the blockchain could write all of these details in a single transaction if desired.

[6]At the time of this writing, our implementation relies on HTTPS.

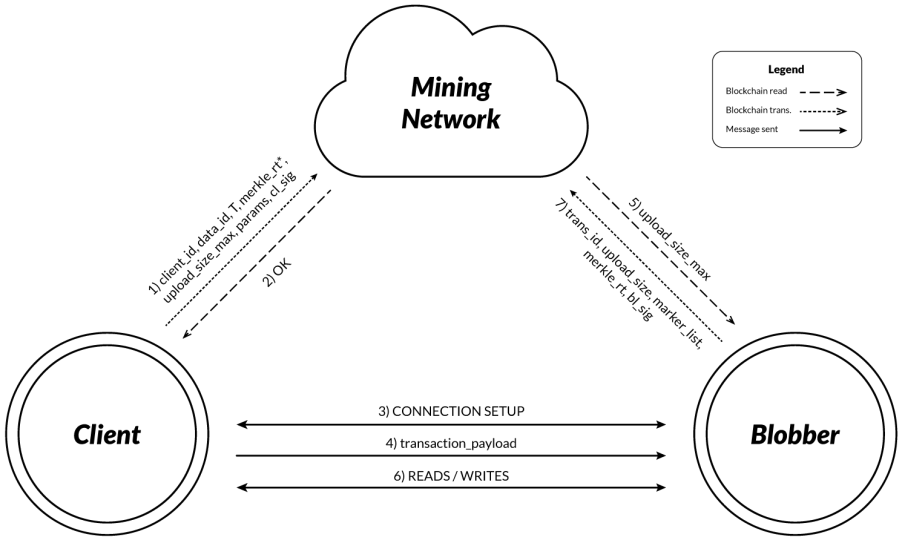

Figure 1: Establishing a client-blobber connection

4. Once the connection has been established, the client sends the transaction payload to the blobber, allowing the blobber to quickly lookup the transaction on the blockchain and verify that the contents of the transaction are valid.

5. The blobber reads the transaction from the blockchain verifying that the request is valid. Additionally, the blobber learns the maximum size of storage that the client is expecting to use; the blobber will not be paid for storage beyond this amount, and therefore will discard additional data uploaded beyond this amount.

6. The client may now read and write data from the blobber, as outlined in Section 6.1.

7. Once the session has completed, the blobber writes a transaction to the blockchain that both pays the blobber and commits the blobber to storing the data matching the Merkle root chosen by the blobber. The blobber's signature `bl_sig` includes all fields listed in the transaction. Note that `trans_id` in this transaction is the id of the client's initial transaction. The `upload_size` is also included in the transaction; if `upload_size` is less than `upload_size_max` from message 1, the client regains the rights to store that additional amount of data. If the blobber does not report this value accurately, they run the risk of being caught during the challenge protocol (discussed in Section 6.3).

Note that the client's storage rights are considered to be used as soon as the initial transaction is written, though they may regain some of that storage after the blobber commits their transaction.

Once the final transaction is written, a certain portion of the blobbers tokens are locked for the period that they are committed to provide storage. These tokens are the blobber's "stake", which may be seized if the blobber fails to fulfill any of their duties.

The information stored in the `params` field in message 1 depends upon the nature of the transaction. If this is a new file storage request, the client must specify the number erasure code slices needed to reconstruct data (`k`) and the total number of erasure code slices available (`n`).

The Merkle tree that the blobber stores is derived from the markers created by the client, detailed in the next section. Assuming that the blobber and client agree on the size of the file, the blobber is not able to forge a Merkle tree for a new file without risk of being detected in the challenge protocol. However, there are some possible attacks that the blobber could do on updates to existing files; we review these briefly at the end of Section 6.1.

After these messages are complete, the client and blobber may exchange data.

Attacks: It is possible that a client might create a transaction on the mining network, but never send the data to the blobber, either as an attempt to damage a blobber's reputation or to prevent a blobber from being paid by other clients. Three factors mitigate this attack:

- The client must lock up tokens to perform this attack. In essence, they would be paying for storage without using it.
- Blobbers are not challenged by the mining network until they post a transaction to finalize the data exchange (detailed in Section 6.1).
- Blobbers periodically monitor the blockchain for transactions involving them; if they notice this transaction, they can write a transaction to the blockchain closing the connection.

Similarly, a blobber might not respond to the client and refuse to complete the connection[7]. Again, several factors make this attack unlikely:

- Once the connection is established, the client is expected to send markers as detailed in Section 6.1. The blobber redeems these markers for 0chain tokens, and hence has a vested interest in completing the connection.
- If the client becomes dissatisfied, they can delete their data from the blobber and reassign it to a different blobber. When this happens, the blobber is no longer paid for storing the data.

6 Block Level Storage System

Our protocol is generic enough to accept either block level storage or file level storage. The outline of our protocol differs very little, although certain minor adjustments are needed.

In this section, we consider block level storage, since it is simpler conceptually. In block level storage, there is no concept of files or objects. Data blocks are instead organized by arbitrary identifiers, and any file organization is applied on the client's end.

6.1 Reads and Writes

Once a secure connection has been established between the client and the blobber, the client can choose to read data from the blobber, upload new data to the blobber, or update data currently stored with the blobber.

The 0chain model for uploading or downloading files requires that the client compensate the blobber. This process is done through the use of special `read_marker` and `write_marker` values created by the client. Each marker is a pair of a counter value (either `read_ctr` or `write_ctr`) and a signature.

The format of the signature portion of a `read_marker` is

$$\texttt{[READ,owner\_id,data\_id,blobber\_id,block\_num,read\_ctr]}_{client}$$

where `READ` is a constant and `owner_id` is the ID of the client owning the data; that is, it is the owner's `client_id`, which may or may not be the same as the client requesting the data. The format of the signature of a `write_marker` is

$$\texttt{[WRITE,trans\_id,blobber\_id,hash(data),block\_num,write\_ctr]}_{client}$$

where `WRITE` is a constant and `hash(data)` is the hash of the current block of data being sent. Note that the `write_marker` includes `trans_id` instead of `owner_id` and `data_id`; this field contains the same information, but furthermore includes the timestamp of the transaction authorizing the write.

[7]Of course, a blobber might be temporarily offline and unable to respond. Critically, if the blobber is online, they are economically motivated to respond.

The blobber collects these markers, and when the transaction has either completed or timed out, the blobber writes a transaction to the blockchain effectively cashing in the markers in exchange for 0chain tokens. The difference between the `read_ctr` and `write_ctr` values on the blockchain (representing the reads and writes currently consumed from the client's purchased resources) and the `read_ctr` and `write_ctr` values in this transaction determine the blobber's pay and the "bill" to the client.

This transaction has the following effects:

- The blobber is paid in 0chain tokens.
- The client loses the corresponding number of reads and writes.
- If the client is using less storage than initially specified, they regain the rights to the additional amount of storage.
- The Merkle root of the data (if it has been updated) is specified by the blobber. At this point, the blobber may be challenged to provide the data that they store. Since the blobber is also compensated for passing these challenges, they have a vested interest in completing the operation.

After cashing in markers, the blobber may discard all `read_marker`s. However, they must keep the latest `write_marker` for each block position. The write markers are used in the challenge protocol to ensure that the blobber is storing data that the client actually wished to store.

In case of a discrepancy between the client and blobber, the client may request the Merkle tree or any other meta-information stored by the blobber. This request requires an open connection, but does not require the client to send a `read_marker`.

Double-spending markers: In some sense, markers serve as an additional currency, and hence the double-spending problem is a concern here. Blobbers might attempt to redeem a marker multiple times, or a client might attempt to pay different blobbers with the same marker.

Since a unique counter is established between the client and blobber, and the blobber is rewarded based on the difference between the last marker stored on the blockchain and the counter value of the current marker, redeeming the same marker multiple times has no benefit for the blobber. A client might attempt to pay multiple blobbers with the same marker. However, since `blobber_id` is included in the marker, this attack would likewise fail.

Generation attack: If blobbers pose as clients, it is possible that they could generate markers without reading the data solely as a mechanism to get 0chain tokens. However, since the blobber would have to lock tokens to acquire reads, they would in some sense be paying themselves with their own money. Our economic protocol is carefully crafted so that this attack is not profitable.

Additionally, clients are expected to interact with many blobbers on any given connection. A client only interacting with a single blobber is strongly indicative of this attack, and the client and blobber may be punished for the interaction.

Bad markers: Clients might create more markers than the number of reads and writes they have purchased, essentially writing checks that they cannot cash. However, blobbers are aware of the client's current allotted number of reads and writes, and will not respond if this allotment is exceeded.

Other attacks: A blobber might ignore a client's request for data and simply cash the markers sent by the client. However, in this case the client would quickly stop sending markers to the blobber, preventing the blobber from receiving additional payment. Furthermore, the client would report an error to the network, and might decide to delete their data from the blobber.

The blobber might send invalid data to the client in lieu of the data previously stored by the client; however, the client might have the Merkle tree, in which case they would quickly spot the problem and report an error. Regardless, the blobber is expected to store the Merkle tree and the `write_marker`s for

each block position and can be asked to provide those details. The mining network stores the Merkle root, preventing the blobber from providing a false tree.

A blobber might simply ignore a client's attempts to upload data. However, since the blobber is paid for all `write_marker`s redeemed, the blobber is strongly motivated to complete the exchange. In cases where the client's updates lead to a drop in the total allocation, the blobber might be tempted to ignore the update in order to continue to be paid for the larger, old allocation. However, the client can delete data without involving the blobber, as outlined in Section 8.

Instead, if a client simply writes data, the blobber might not store the data. However, when redeeming markers, the blobber must confirm the new Merkle root. Therefore, the mining network would be able to catch the blobber's cheating with the challenge protocol (Section 6.3).

A client might send different data to the blobber that does not match the Merkle root specified in the blockchain. However, the blobber's Merkle root is used in the event of any discrepancy, minimizing the damage of this attack.

6.2 Read-only Exchanges

When uploading or updating data on the network, it is important that the network has visibility into that interaction. This feature allows the miners to determine if there is an ongoing update to data, and enables auditors to spot potentially malicious activity.

However, a downside of this approach is that it introduces a delay; until the session is finalized, data upload cannot commence.

For reads, quick access is both a higher priority and less of a risk. Therefore, we provide a *lightning mode*, where the client does not need to write an initial transaction to the network.

We assume that the client knows the identity of the blobbers storing the data they desire. They may contact the blobber directly. In their initial message, they provide:

- Their own `client_id`.
- The `trans_id` of the transaction where they initially locked their tokens to get `read_marker`s. This transaction allows the blobber to quickly verify the validity of the client's request.
- The `client_id` and `data_id` corresponding to the data that they wish to read. Note that this `client_id` is the ID of the owner, and may be different than the ID of the client currently requesting the data.

After the connection is initialized, the client sends `read_marker`s, following the process outlined in Section 6.1. Once the exchange is complete, the blobber writes a transaction to the blockchain to redeem the `read_marker`s.

Since there is no initial transaction to the blockchain, there is a greater chance that the client may exceed their number of reads, particularly if they read from a large number of blobbers simultaneously.

When the blobber redeems the markers they have received, if they exceed the client's allocation of reads, the client's stake is seized to pay the blobber. However, once the client's stake has been exhausted, the blobber is not paid for any additional reads. This approach motivates blobbers to cash in their markers quickly, reducing the client's chances of success. Furthermore, it prevents the client and blobber from colluding to generate new tokens that exceed the client's seized stake.

6.3 Challenge Request

In order to verify that a blobber is actually storing the data that they claim, our protocol relies on the miners periodically issuing challenge requests to the blobbers. This protocol is also how blobbers are rewarded for storing files, even if the files are not accessed by any clients. When the blobber passes the challenge, they receive newly minted 0chain tokens.

At a high level, the challenge protocol involves three phases:

1. The mining network randomly selects the data to be challenged. This phase also preemptively punishes the blobber for failing to provide the data.

2. Once a blobber has been challenged, the blobber sends their data to a subset of a randomly selected group of *validators*. This request includes a signed authorization from the blobber.

3. The validator writes a transaction to the blockchain indicating whether the test passed. This transaction also pays the validator and rewards the blobber if the test was successful.

The selection of validators is a particular challenge. We consider two possible attacks:

- In a *cronyism attack*, a blobber sends the data to a friendly validator who approves all challenges without attempting to validate the data.

- In an *extortion attack*, a validator demands additional compensation from the blobber in exchange for passing the challenge.

We defend against these attacks by randomly selecting a set of V validators. The first A validators to approve or reject the challenge are rewarded for their work. The setting of A and V help to tune the defenses against these attacks. The difference between these values must be narrow enough to make a successful cronyism attack unlikely, but high enough to prevent an extortion attack.

Challenge Issuance Phase The mining network must initially post a transaction to the network randomly challenging a blobber providing storage. This transaction must include:

- The ID of the blobber challenged.

- The list of eligible validators.

In order to avoid any bias or collusion between miners and blobbers, the challenge is determined through the use of a verifiable random function (VRF) [9]. Conceptually, we can envision this challenge as a roulette wheel, where a blobber with more data would have more slices on the wheel. VRFs avoid the *last actor problem*, preventing any single miner from skewing the results unfairly.

The initial transaction preemptively punishes the blobber by deducting X coins. This approach removes any incentive for the blobbers to perform a denial-of-service attack against the enforcement mechanism.

Additionally, this phase establishes a `challenge_ctr` initially set to A. With each challenge, the value of A is reduced by one. Once this counter reaches zero, additional validations will be ignored. This setup limits the overall payout to both the blobber and validators in case of collusion between the two parties.

Justification Phase Once challenged, the blobber must initialize a connection with each validator to conduct the challenge. The validator randomly selects a block of data[8] that the blobber must provide. The blobber then provides the validator with

- The challenged block of data.

- A matching `write_marker` proving that the owner of the data desired this block of data to be stored.

- (Optional) The $log_2 n$ nodes of the Merkle tree needed to reconstruct the Merkle root.

- A signature from the blobber authorizing the challenge, called a *challenge authorization receipt*, which includes the challenge transaction and the id of the validator.

[8] If desired, the validator can insist on many different blocks of data.

When the blobber contacts the validator, the validator verifies the blobber's signature and verifies that the data matches both the `write_marker` and the Merkle root. By knowing the number of blocks, the validator knows the path that should be provided, preventing the blobber from adjusting the tree shape to their advantage.

We note that the Merkle path might be omitted in the block-storage case, since the marker and block of data are sufficient proof. However, if the client specified a Merkle root, the Merkle path could be used as an alternate proof form; this feature might be useful if the blobber either lost or did not wish to store the markers.

Judgment Phase In our design, we wish to ensure that the validator is economically incentivized to carry out the challenge, but indifferent to the result of the challenge.

Once each validator has performed the test, they write a transaction to the blockchain containing:

- A bit specifying whether the challenge was successful.
- Payment to the validator for their work.
- A reward for the blobber, if the challenge was successful.
- The blobber's challenge authorization receipt to conduct the challenge.

A validator might be tempted to automatically fail the blobber's challenge, since that gives an advantage in writing the first transaction to the blockchain. However, since the validator cannot be paid without the challenge authorization receipt from the blobber, the blobber can prevent the validator's attack in future transactions. If the validator becomes known for this attack, other blobbers may refuse to authorize challenges with that validator.

The blobber's total reward is $X + Y$ tokens, where X is the amount of coins previously taken from the blobber and Y is their additional reward. Therefore, each transaction rewards the blobber with $(X + Y)/A$.

6.3.1 Attacks

A few attacks arise in the challenge protocol. Extortion attacks and cronyism attacks have been discussed already. In this section, we discuss a few additional challenges.

Outsourcing attacks: A blobber might attempt to pass challenges by outsourcing requests for the data to other blobbers. However, since the data is erasure coded, the cheating blobber would need to pay the other blobbers in markers in order to get the data. By tuning the payouts and locking amounts, we can make this attack unprofitable for the blobber.

Collusion attacks: This attack is similar to an outsourcing attack, except that the blobber relies on free, back-channel connections to other blobbers to provide fragments of the data when needed to reconstruct the blobber's own fragment of data.

Collusion attacks are a little more challenging, but we do have some defenses.

It is unlikely that a blobber would be able to recover all fragments of the data via back-channels, so they would still need to request the data from the mining network. If the blobber acts like a normal client making a request, then they would request data from all of the blobbers returned; at this point, there is no advantage to collusion compared to outsourcing.

On the other hand, if the blobber does *not* get data from all blobbers, we can detect this abnormal behavior, and potentially reassign the blobber's storage to a different blobber. Over time, this pattern can be used to identify colluding parties.

7 File level storage

Section 6 outlined our protocol when block level storage is used. However, it may be desirable to use file level storage instead. The concepts of our design are similar, but additional complexities arise.

Our architecture is similar to Git [6]. In the case of any errors between the client and blobber, following Git's structure facilitates negotiation of the corrections between the two parties. Also, if a file is updated while a client still happens to be reading from it, the blobber is able to easily provide the original file without interruption.

Git's design follows a structure somewhat reminiscent of Merkle trees. Files are named according to the hash of their contents. Directories contain the hashes of their files, effectively acting like the inner nodes of a Merkle tree. Each commit is the root of a tree giving the current view of the file system.

The fundamental difference between a Git commit and the Merkle trees used for block storage is that the Git trees do not have a regular shape. This lack of uniformity complicates the challenge protocol and requires some additional details to be stored. Furthermore, the leaves of the tree are files, which also need a corresponding Merkle tree for their contents. In this section, we review the related changes.

For a deeper understanding of Git, Chacon [3] provides an excellent overview, and may be helpful for understanding the design of our system.

7.1 File System Structure

The organization of the file system is the prerogative of the blobber, but there are certain restrictions that we enforce.

We expect that each file is stored with a name matching the cryptographic hash of its contents, analogous to how Git stores its *blobs*; we will use that terminology for non-directory files. An interesting side effect of that approach is that all identical copies of the file are only stored once; each reference to a copy of the file in the file system points to the same hash value.

An obvious question then arises – is the blobber paid for each unique file, or for each copy?

Since the blobber would naturally store only a single file, we structure our compensation in this manner. The choice also protects against certain trivial generation attacks.

For each directory, a file is stored that references the blobs that it contains and any subdirectories. For each file the directory contains, the directory's metafile specifies:

- The file name.
- The type of the file: directory or blob.
- The hash of the file contents, needed to refer to the corresponding blob/directory.
- The size of the file.
- The Merkle root of the file contents (blobs only).

For each blob, a corresponding `write_marker` must be stored as well. With file storage, additional information about the path of the file needs to be specified in the marker. The format of the signature of the new `write_marker` is

$$\texttt{[WRITE,trans\_id,blobber\_id,}\mathbf{file\_size,merkle\_rt(file\_data)}\texttt{,write\_ctr]}_{client}$$

The format of the `read_marker` also needs to be updated. The `file_path` of the file must be included. In addition, the `block_num` field is still needed, though it now refers to the block position in the file; this design facilitates a client reading just a portion of a large file, if desired. The `read_marker` signature's format is

$$\texttt{[READ,owner\_id,data\_id,blobber\_id,}\mathbf{file\_path}\texttt{,block\_num,read\_ctr]}_{client}$$

7.2 Data upload

The connection process works as described in Section 5, except that the hash committed to the blockchain is no longer the Merkle root of the data. Instead, both the client and blobber write:

`hash(blob_list,file_root)`

The `blob_list` is a list of all blobs stored and their size, sorted by the blob's name where the name is determined by the hash of the blob's contents. By specifying this file, we can easily identify a single block to challenge with a single random number, where all blocks are equally likely to be selected. For example, if a blobber were storing files names `ae34`, `f03e`, and `08c7` of 64KB, 147KB, and 2,429KB, the blob_list would be:

`ae34`,64 `f03e`,147 `08c7`,2429

The `file_root` is the hash matching the root directory of the file system. By committing the hash of both `blob_list` and `file_root`, the blobber simultaneously commits to both the blobs' contents and the directory structure.

7.3 Challenge Protocol

With this information, the challenge protocol works in a similar manner as described in Section 6.3. The only difference is that the challenged blobber must provide:

- The challenged block of data.
- The `write_marker` for the challenged file[9].
- The Merkle path from the block to the Merkle root of the file.
- The `file_size`.
- The `blob_list`.
- A valid file path from the blob to its root.
- The directory meta_files needed to calculate the `file_root`.

From this information, the validators are able to verify that the data provided matches with the hash on the blockchain.

8 Deleting data

For deleting individual blocks or files, the client can directly contact the blobber through the normal process. However, if the client becomes dissatisfied with the performance of a specific blobber, they may elect to delete and reassign their entire allocation of data stored with that blobber.

To delete their data allocation, the client posts a transaction to the blockchain. Once the transaction is finalized on the blockchain, the client regains the storage allocation. The mining network reassigns the data allocation to a different blobber.

Blobbers are expected to poll the blockchain for these transactions. Once a blobber notices that the allocation has been deleted, they delete the locally stored data.

Nothing in our protocol enforces that the blobbers actually delete the data when asked, though they have little economic incentive to keep it. If the client is concerned about the confidentiality of their data, they are expected to encrypt their data before storage.

[9] Alternately, a `write_marker` could be stored for every block of the file, which would allow for a blobber to get paid even if a client failed to upload the complete file. We have elected to keep the `write_markers` at the file level to reduce the total number of markers.

9 Recovering data

When a blobber disappears unexpectedly from the network, the data needs to be regenerated and stored with another blobber. In our design, we assume that the client is online and able to monitor relevant challenge transactions from the blockchain.

We outline the process below.

1. The mining network challenges a blobber ($B1$) to provide a block of data.
2. After time T, the blobber is assumed to have failed the challenge.
3. Once time T has passed, the client who owns the data can post a transaction to the blockchain to reassign the data to a different blobber. This transaction is treated as a delete; that is, the blobber is no longer expected to store the data nor rewarded for storing it.
4. If needed, the client can read data from each blobber needed to reconstruct the missing data. This process works as a normal read operation, except that the blobber's providing the data are paid for their work out of $B1$'s stake of coins. The client must include the transaction id from step 3 as part of the `PARAMS` field.
5. The client reconstructs the missing slice of data on their end.
6. The client uploads their data to a new blobber, following the usual process. Again, if the client includes the transaction id from step 3 in the `PARAMS` field, they do not have to pay for the work performed by the blobber receiving the data; the new blobber's pay is taken from $B1$'s seized stake.

10 Conclusion and Future Work

0chain provides a framework for distributed data storage where neither the client nor the blobber need to trust one another. A key trait of our design is that the mining network is not directly required to work with the files, which reduces the load on our miners. Through our protocol, both client and blobber agree on the data that the blobber is expected to store, with an easily verified Merkle root stored on the blockchain.

In future work, we plan to improve the permissioning scheme of our storage protocol. Currently, all data is assumed to be public, though it may be encrypted by the client. In some cases, however, it may be desirable for clients to restrict their data to a subset of users, or perhaps to get compensation for access to their content. Proxy re-encryption (PRE) [2] can allow more sophisticated sharing of data. PRE allows a semi-trusted third party to convert data encrypted with a public key to data encrypted with a different public key; with this design, we can release data to select parties without decrypting it for all to see.

Acknowledgments We would like to thank our advisers for their valuable feedback on this paper, especially Jonathan Katz and Sarath Ambati.

References

[1] Muneeb Ali, Jude C. Nelson, Ryan Shea, and Michael J. Freedman. Blockstack: A global naming and storage system secured by blockchains. In *USENIX Annual Technical Conference*, pages 181–194. USENIX Association, 2016.

[2] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, pages 127–144. Springer, 1998.

[3] Scott Chacon. *Git Internals: Source code control and beyond*. PeepCode Press, 2008.

[4] Filecoin: A decentralized storage network. Technical report, Protocol Labs, August 2017.

[5] Seán Gauld, Franz von Ancoina, and Robert Stadler. The burst dymaxion: An arbitrary scalable, energy efficient and anonymous transaction network based on colored tangle. `https://dymaxion.burst.cryptoguru.org/The-Burst-Dymaxion-1.00.pdf`, accessed July, 2018, 2017.

[6] Git – fast version control. `https://git-scm.com/`, accessed August, 2018.

[7] Nick Lambert, Qi Ma, and David Irvine. Safecoin: The decentralised network token. `https://docs.maidsafe.net/Whitepapers/pdf/Safecoin.pdf`, accessed July, 2018, 2015.

[8] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[9] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 120–130. IEEE Computer Society, 1999.

[10] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *IEEE Symposium on Security and Privacy*, pages 475–490. IEEE Computer Society, 2014.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[12] Namecoin homepage. `https://namecoin.org/`, accessed August 2018.

[13] 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.

[14] Proof of replication. Technical report, Protocol Labs, July 2017.

[15] David Vorick and Luck Champine. Sia: Simple decentralized storage. Technical report, Nebulous Inc., November 2014.

[16] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, James Prestwich, Gordon Hall, Patrick Gerbes, Philip Hutchins, and Chris Pollard. Storj: A peer-to-peer cloud storage network. Technical report, Storj Labs Inc., December 2016.

A Revisions

Version 1	June 6, 2018	Initial draft
Version 2	August 31, 2018	Allow blobber to specify Merkle root. Added additional details to challenge protocol. Removed error reporting protocol. Expanded related work discussion. Self-repair protocol added. Added file level storage. Added lightning-mode read-only support.