2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)

Lock and Load: A Model for Free Blockchain Transactions through Token Locking

Paul Merrill and Thomas H. Austin
Research Group / Department of Computer Science
0Chain LLC / San José State University
San Jose, United States

Jenil Thakker and Younghee Park
Department of Computer Engineering
San José State University
San Jose, United States

Justin Rietz
Department of Economics
San José State University
San Jose, United States

***Abstract*—Bitcoin introduced the world to blockchain-based cryptocurrencies, and Ethereum highlighted their value in building distributed applications (dApps). However, the development of blockchain-based applications has been held back by high transaction fees.**

In this paper, we introduce a model for free transactions on the blockchain. Rather than spending tokens for transaction fees, a token owner (known as a *client*) locks tokens to generate new tokens as a reward for the miner who includes the transaction in a block. This *token-locking reward model* eases congestion on the blockchain in the same manner as fees do in protocols like Bitcoin, but without forcing clients to sacrifice their tokens.

This same design can be used to incentivize service providers. We show how a client can lock their tokens to generate new tokens for storage providers, and how this reward mechanism can help to facilitate an audit of the storage provider.

***Index Terms*—blockchain, storage, cryptocurrency economics**

I. Introduction

Our blockchain introduces a *token-locking reward model*; rather than spending tokens, clients lock tokens to pay for services. This act creates more tokens, which can be used to reward miners or other entities for their work.

This model is similar to Bitcoin's design [1]. Early in Bitcoin's history, miners were primarily incentivized to generate blocks through *coinbase transactions* that rewarded miners with newly created bitcoins. As interest in Bitcoin has skyrocketed, clients must offer transaction fees to motivate miners to include their transactions. This design also serves to ease congestion; when demand for transactions increases, clients can offer greater rewards for including their transactions.

Our model can be seen as a blend of these two mechanisms. Clients lock tokens to generate new tokens for miners, similar to coinbase transaction rewards; but as with transaction fees, a client may offer to lock a greater amount of tokens to ensure that their transaction is accepted by the mining network.

When clients lock tokens, they can give the newly generated tokens to any other client, facilitating a market for creating distributed applications (dApps). In this manner, tokens in our network can be used to buy services for "free", in the sense that the client does not lose their tokens, but still gives something of value to the service providers.

Using our token-locking reward model, we build a sample dApp for storage, the blockchain-observable storage system (BOSS). With BOSS, storage providers are rewarded for their work by clients who lock tokens. We also show how BOSS can ensure that neither clients nor storage providers can cheat one another; this process relies on special, signed *markers* that ensure public agreement between the two parties. Additionally, this agreement can be publicly validated; third parties can verify that the storage provider is storing the agreed-upon data using nothing more than public transactions on the blockchain and signed messages from the client.

A key property of our design is that the clients may give themselves the newly generated tokens (hereafter referred to as interest). Alternatively, we could restrict a client to only reward service providers. However, that approach would incentivize clients to feign services in order to mint new tokens; we legitimize this behavior and eliminate such shenanigans.

An interesting economic consequence of this design is that it reduces the opportunity cost of holding a token versus holding a fiat currency in an interest bearing bank account. The interest paid at least partially offsets a possible reduction in the token value. If the level of interest paid moved inversely with the token price the interest payment might substantially offset changes in token value, which could be a stabilizing factor. If the token price decreases, people will lock more tokens in expectation of receiving a higher interest rate, and this locking of tokens in effect reduces supply, creating upwards pressure on the token price.

Our paper makes the following contributions:

- We present our token-locking reward model, which enables clients to reward service providers by locking tokens, without needing to sacrifice their tokens.
- We demonstrate how our model can be used to incentivize miners to accept transactions and generate blocks.
- We use our token-locking reward model to build a storage dApp, allowing clients and storage providers to negotiate an agreement for service. As with incentivizing miners, we can reward storage providers for their work without requiring the client to sacrifice tokens.
- We show how signed markers and the blockchain can be used to validate a storage provider's work.
- We provide an economic analysis of our system, showing how our model can reduce the price instability typically associated with cryptocurrencies.

978-1-7281-1264-0/19/$31.00 ©2019 IEEE
DOI 10.1109/DAPPCON.2019.00013

IEEE computer society

II. Token-Locking Reward Model

In our model, clients can generate new tokens by locking their existing tokens via a transaction to the blockchain. These tokens are then unspendable for a period of time[1]. However, this transaction immediately creates new tokens, in essence a form of prepaid interest. Clients reward the newly generated tokens to other clients, which may be a miner or the clients themselves. We refer to the client who holds the tokens as the *owner*, and the client receiving the reward as the *recipient* or the *reward recipient*, noting that the recipient and the owner might be the same client. After the locking period has elapsed, the client regains the use of the locked tokens.

A *pool* is an account holding tokens, where a smart contract restricts when and how the tokens may be spent. When a client locks tokens, they are set aside in a pool that makes them unspendable for 90 days (enforced by the smart contract). Other pools may have different restrictions, depending on how different service providers are paid. A pool may contain *subpools*; that is, within a pool, the smart contract can enforce that certain funds are earmarked for different entities.

A. Locking Tokens

To lock tokens, a client writes a special transaction called a *lock transaction*, which specifies:

- The amount (A_{lock}) of tokens to lock in a pool.
- The amount (A_{spend}) of tokens that the client wishes to spend; these tokens will be removed from the client's account and added to the interest paid out to the recipients.
- The recipients (clients or pools) of the reward R.

The reward R paid to the recipients is based on an interest rate multiplier M, the amount of tokens locked (A_{lock}), and the amount of tokens the client chose to spend (A_{spend}). The formula is given below:

$$R = (M * A_{lock}) + A_{spend}$$

The A_{spend} parameter allows a client to pay an additional reward beyond the interest ($M * A_{lock}$) earned from locking tokens. For instance, Alice might own 102.5 tokens. She wishes to purchase a service from Bob, who demands 3.5 tokens for his total reward. If the interest rate multiplier M is set to 0.025, then $R = 2.5625$, which does not meet Bob's requirement. Instead, Alice can lock 100 tokens to create 2.5 tokens of interest, given to Bob. She adds an additional token to the reward, giving Bob the 3.5 tokens he requires. Alice retains 1.5 unlocked tokens, and a pool of 100 tokens that will be unlocked after 90 days.

This design allows a new client to experiment with our system more cheaply before committing to buying a larger amount of tokens. However, it will be more profitable for the client to pay for services solely through locking tokens, as we show in Section IV.

[1]Currently this period is set to 90 days in our implementation.

B. Writing Transactions

Our system is designed to be largely agnostic with regards to the underlying blockchain. However, the miners must be compensated in some manner for their work. Here we review how our token-locking reward model can still incentivize the miners to produce new blocks[2].

When clients wish to write transactions to the blockchain, they specify a transaction reward in terms of locked tokens and spent tokens. The transaction will have a size in bytes S. Miners maximizing their profit will select from these transactions to maximize R/S.

This design allows a client to designate transaction rewards as needed, locking as many tokens as required for its transaction to be accepted. Our approach also allows the blockchain ecosystem to seamlessly transfer between a free and paid model, permitting the client to spend tokens instead of locking them in case transaction fees rise to particularly high levels.

C. Staking Tokens

A service provider can be required to *stake* tokens as part of its service guarantee. This process is similar to locking tokens: the principal is locked in a pool payable to the service provider, and the service provider immediately receives the interest for the locked tokens.

A key difference, however, is that the smart contract permits (some portion of) the staked tokens to be seized if the terms of service are not met. The terms and conditions of this smart contract can be specific to the service provided.

An important property of our design is that honest service providers do not have an economic disincentive to stake their tokens. They receive the interest on their staked tokens just as if the tokens had been locked, and earn *additional* tokens for providing their service.

III. The Blockchain Observable Storage System

With the economic design of our ecosystem established, we review how a service can be offered on the blockchain and validated by the mining network. Our system is named the Blockchain Observable Storage System (BOSS), emphasizing that the blockchain is able to provide evidence allowing the storage providers' work to be validated. Storage providers in our system are known as *blobbers*.

A key distinction from other blockchain storage solutions is that we divorce the role of mining from that of providing storage. Blobbers are neither responsible nor required for mining, allowing us to lighten the load on our mining network and enable fast transactions on a lightweight blockchain.

Our design assumes that the client is using erasure codes [3] to ensure greater resiliency. While this is not a strict requirement, it does enable a client to recover if a blobber proves to be

[2]Our blockchain also includes the role of *sharders*, responsible for the long-term storage of the blocks. We omit them from our analysis, but note that they can be rewarded for their work by requiring a percentage of the mining rewards to be paid to them. Assuming that the economic considerations for the sharders are similar to those of miners, the miner's actions should reasonably satisfy the needs of the sharders. For more details on sharders, we refer the interested reader to a related technical report [2].

unreliable. From the perspective of the blobbers, they neither know nor care if the data is erasure coded. In our discussion, we will discuss erasure codes only as far as it affects any potential attacks that could arise.

A. Relevant Attacks

Filecoin [4] discusses many interesting attacks that can arise in blockchain-based storage systems, such as Sybil attacks, outsourcing attacks, and generation attacks. We provide an overview of these attacks to aid 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 a file, but she knows that Bob is also storing it, 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 likely to be profitable, particularly if erasure codes are used. For instance, if the erasure code settings were 10 by 16, then to be paid for a single read, the cheater would need to pay 10 other blobbers for their shares of the data.

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.

Finally, generation attacks may arise if a blobber poses as a client to store data that can be regenerated easily or that they know will never be requested. By doing so, they hope to be paid for storing this data without actually needing the resources to do so. (Filecoin's authors are concerned with miners exploiting this approach to increase their voting power on the blockchain, which is not a concern in our design.)

Our primary defense against generation attacks is our economic protocol; essentially, blobbers would end up paying themselves for their work. Since they could lock tokens and pay themselves anyway, this attack gives them no advantage.

B. BOSS Overview

First, the client and blobber negotiate the price and terms of service, establishing a *read price* and *write price* that determines the tokens required per gigabyte of data read or written, respectively. At the end of their interaction, the client has an *agreement*[3] signed by both the client and blobber. The client writes a transaction to the blockchain committing both the client and the blobber to their agreement.

[3] *Contract* would be a more appropriate term, but we wish to avoid confusion with smart contracts.

The client locks tokens, giving the interest to a *blobber reward pool* earmarked specifically for the blobber. The blobber will not provide service if the corresponding pool has insufficient funds. Since no other entity has access to the tokens in this pool, the blobber may cache its information about the pool, avoiding the need to check the blockchain for the balance of the pool.

To upload a file, a client contacts the blobber and uploads a file, including a signed *write marker* that authorizes the blobber to invoke the storage smart contract to transfer tokens from the blobber reward pool to a *challenge pool*; when the blobber later proves that it is providing storage, the tokens are paid from the challenge pool to the blobber. This transaction also serves to commit the blobber to the storage contents, so that the act of arranging payment also makes it possible to catch any cheating on the blobber's part.

File downloads work similarly: the client sends a signed *read marker* to a blobber as part of their request for data. The read marker authorizes the blobber to draw tokens from the blobber reward pool. Unlike with writes, the blobber receives the pay for providing reads immediately. An additional difference is that the client reading the data might not be the owner of the data.

The mining network will periodically challenge the blobber to provide randomly chosen blocks of that data. These challenges involve a carrot and stick approach; the blobber is punished if they fail the challenge, but receive their token rewards when they pass the challenge.

C. File System

Our protocol is agnostic regarding how the blobber stores data. We review our file system, but note that clients and blobbers could agree to use other systems, or even opt for block-level storage over file-level storage.

Our architecture is similar to Git [5]. 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. (Chacon [6] provides an excellent overview of Git, and may be helpful for understanding the design of our system.)

We assume that a client uses erasure coding to provide greater resiliency for their data. Entities in the file system include directories, files, and *fragments*; a file's metadata contains information about the original file, while the corresponding fragment's metadata includes details about the blobber's share of the original file.

Directories are named by the hash of their content. A directory stores a list of the files it contains, tracking:

- `type`, either "file" or "directory".
- `name`, the name of the file in the directory.
- `hash`, the cryptographic hash of the file contents; this hash identifies the corresponding metadata file or file contents on the system.

For files, additional information is stored about the fragment:

- `merkle_root`, the Merkle root [7] of the fragment.
- `size`, the size of the fragment in bytes.

In addition, the blobber must keep special *write markers*, which are signed commitments from the client as to the contents of the file system. The client can request the write marker in order to verify the file system contents, freeing the client from needing to store any data on their own system. Write markers also serve to track version information, and to authorize payment to the blobber. Write markers are discussed in more detail in Section III-E3.

D. Storage Agreement Negotiation

The client and blobber must negotiate a price for writes and a price for reads, both in terms of tokens/gigabyte of data. Other criteria may be negotiated between the client and blobber as needed, allowing the blockchain to serve as a public record of their agreement.

Once terms have been established, the client writes a transaction to the blockchain with the terms of their agreement. We refer to this transaction as the *storage agreement transaction*. This transaction includes:

- The id of the client (`client_id`).
- The id of the blobber (`blobber_id`).
- The `allocation_id` identifying this storage *allocation*, referring to the data that the blobber stores for the client. This globally unique ID is a function of `client_id`, `blobber_id`, and a timestamp.
- The tokens of reward paid to the miner per gigabyte read (`read_price`).
- The tokens of reward paid to the miner per gigabyte uploaded (`write_price`).
- A `params` field for any additional requirements.
- The signatures of both the client and blobber.
- Offer expiration time, to ensure that the client does not invoke an old agreement that is no longer profitable for the blobber.
- Storage duration, determining how long the blobber needs to provide storage. After this period has elapsed, the blobber no longer needs to store the client's files; of course, the client and blobber can negotiate to extend the storage period[4].

This transaction also initializes a `read_counter` and `write_counter` for the client and blobber to use in their interactions, both initially set to 0. These values increase with each transaction depending on the amount of data uploaded or downloaded. By calculating the last counter value with the new counter value, the amount of reward the blobber has earned is determined easily. Section III-E and Section III-F review these counters' utility in rewarding the blobber on behalf of the client.

This transaction also creates two new pools:

[4]From the perspective of the blockchain, the renewal is treated as a completely new agreement – no special support is needed. The client can generate a write marker (discussed in Section III-E3) to pay the blobber for files that the blobber is already storing.

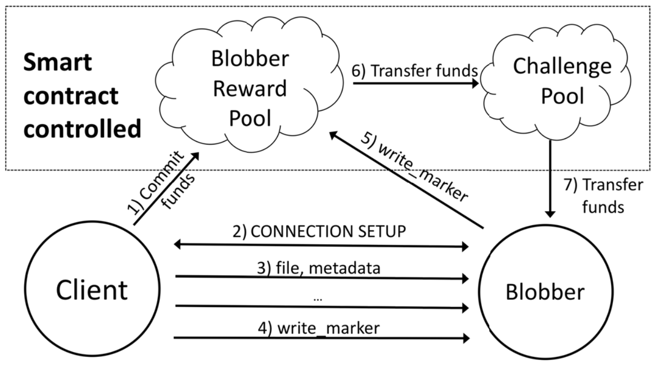

Fig. 1. Storage Process

1) The *blobber reward pool*, containing the interest that the client generated as the rewards for the blobber to store and serve data.
2) The *challenge pool*; when the blobber verifies that it is storing the data, it may receive some portion of the reward stored in this pool.

When the funds in the blobber reward pool are depleted, the client may lock additional tokens to add funds to them. The challenge pool is initially empty, but gains tokens with every write that the client does. (Reads, in contrast, are paid to the blobber directly.)

E. Pay for Writes

In the storage contract transaction, the client locks tokens and pays the interest to the blobber reward pool. These tokens represent the blobber's pay for storing the client's data. (A portion of these funds are allocated for the validators, discussed in Section III-G.)

Blobbers are paid for every file uploaded, and they are expected to store the files until the end of the contract period negotiated with the client. (A client can elect to delete files stored with the blobber, but does not receive any refund for doing so).

Note that they are not paid immediately. The funds are set aside in the challenge pool; the blobber receives tokens from this pool upon satisfying a challenge to prove that they are actually storing the data.

Figure 1 gives an overview of the process for a client uploading file to a blobber. The steps are as follows:

1) Before uploading data to the blobber, the client must commit tokens to the blobber reward pool.
2) The client and blobber set up a secure connection. At the time of this writing, our implementation relies on HTTPS.
3) The client transfers files and the corresponding metadata. This step may be repeated until all files have been uploaded.
4) The client uploads a signed write marker, which serves as the client's commitment to the file system contents.
5) The blobber calls the storage smart contract to claim its reward. This call commits the blobber to the file system

contents, and also stakes some of the blobber's tokens to guarantee its performance.
6) A portion of the funds in the blobber reward pool are transferred to the challenge pool, according to the write price specified in the agreement.
7) Some time later, the blobber is challenged to prove that the files are stored correctly. If the challenge is successful, the blobber receives a portion of the tokens in the challenge pool. This process is repeated until the end of the storage agreement. Section III-G reviews the challenge protocol in detail.

A key property of the BOSS design is that the blobber's mechanism to get paid (by writing a transaction in step 5) also commits the blobber to the contents of the file system; the challenge protocol can then detect if the files are not actually stored on the system.

The write marker is another important part of our design. It contains a hash of the root of the file system, and hence the client's signature serves as the client's endorsement of the file system contents. Therefore, even though the blobber and client do not necessarily trust each other, their agreement to the file system's expected contents is publicly verifiable. The format of the write marker is discussed in more detail in Section III-E3.

1) Blobber Reward Pool: Step 1 in Figure 1 allows the client to allocate funds for storing or reading data from the blobber. Typically, this step is done during the contract negotiation, outlined in Section III-D, though the client may decide to lock additional tokens to replenish this pool when it has been depleted.

The blobber can verify the funds in this pool, and may refuse to accept data if the client does not have sufficient tokens locked in this pool. Note that the blobber is not directly paid for writes by this pool. Instead, the funds are payable to the challenge pool, discussed in Section III-E2. The blobber will receive any funds sent to this pool, but must periodically prove possession of the file system contents to earn these tokens.

It is possible that the tokens in the blobber reward pool are not used by the client before the negotiated end of the terms of service. Once the terms of service have expired, any tokens remaining in this pool are transferred to the client. The client may also write a special *cancellation transaction* to end the terms of the contract early; the blobber reward pool is then emptied out, with all tokens transferred to the client. This mechanism gives the client an escape in case the blobber fails to live up to the terms of service.

We note that the client could abuse this cancellation mechanism in order to cheat the blobber, though the attack is unlikely to be very profitable for storage. A blobber does not commit the client's uploaded file contents until payment has been accepted by the blockchain. On reads, cancellation would mean that the data could no longer be read, making the benefit of this attack very questionable.

Nonetheless, we anticipate that our design may be useful for other services. We address the timing issues that arise by delaying the unlocking of funds for 24 hours. In that time, the blobber may redeem any outstanding markers for its rewards as per normal operation.

2) Challenge Pool: When the blobber writes a transaction (in step 5 of Figure 1), funds are transferred from the blobber reward pool to the challenge pool. The amount of tokens transferred is dictated by the write marker, which includes the amount of data uploaded, and the negotiated write price stored in the agreement.

Funds in this pool are paid out to the blobber during the challenge protocol, detailed in Section III-G. Once the terms of service have elapsed, and the blobber is no longer required to store the data, any remaining funds are paid to the blobber. (Note, however, that any funds connected to unsatisfied challenges are not released; this design prevents a blobber from "running out the clock" to avoid passing a challenge). A small portion of the tokens in the challenge reward pool is set aside for the *validators*, a separate group of machines who verify that the blobber is providing the storage that it claims to be providing.

The transfer to the challenge pool also stakes some of the blobber's own tokens, as described in Section II-C. The blobber earns the interest on the staked tokens, but runs the risk that the staked tokens could be burned if any challenges are failed.

3) Write Markers: Write markers serve as the client's commitment to the file system contents, the blobber's proof of storing data required by the client, and a "cashier's check" which the blobber can redeem to get paid. A write marker contains:

- The client id, blobber id, and allocation id.
- A timestamp, to ensure freshness.
- `file_root`, the hash of the root directory on the file system.
- `prev_file_root`, used for versioning.
- `write_counter`, which serves to give the total size to date of the data that the client has uploaded to the blobber for this allocation.
- The signature of the client owner verifying the other fields in this marker.

The client will not send any additional write markers until the latest write marker has been committed to the blockchain.

We note that the client might refuse to upload the write marker after uploading data. In this case, the blobber will simply discard the data after a timeout period.

4) Levels of Trust: The protocol that we have detailed assumes no trust between the client and blobber. However, if the client and blobber have an established relationship they might elect to use a more relaxed trust mode.

Normally, a client will not send additional write markers until the blobber has redeemed the previous write marker. However, the client can relax this requirement and send write markers before older markers have been redeemed, which allows for more graceful recovery should the client crash while uploading data. This approach risks that the blobber then delay redeeming markers until near the agreement's expiration, in the hopes of avoiding being identified as a cheater. Gradations of

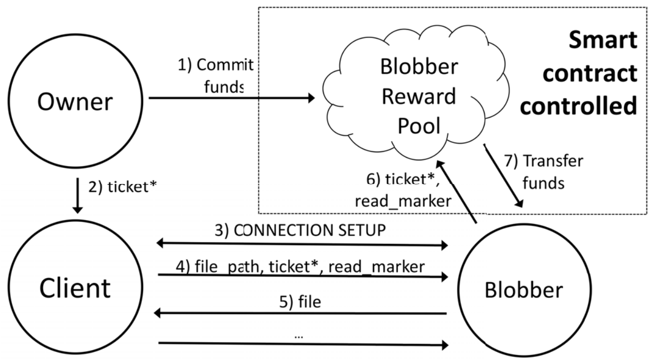

Fig. 2. Reading from Blobbers

this trust might be used as well; for instance, a client might send multiple write markers during an active session, but not start a new session until all markers from the previous session have been redeemed.

In a yet more relaxed model, the client might avoid validation of storage altogether. With this approach, the funds in the blobber reward pool are made directly payable to the blobber, and the challenge protocol is not performed. The advantage of this setup is that no funds need to be allocated for the validators (discussed in Section III-G), and hence the blobber might offer the service more cheaply. However, if the blobber fails to live up to their terms, the client has no recourse. The blobber's cheating is still publicly verifiable, but with no incentive for the validators and no staked funds from the blobber, the blobber will suffer no consequences. This trust model is only appropriate if the client and blobber have a well-established relationship.

F. Pay for Reads

The storage contract transaction includes a negotiated *read price*. However, unlike with writes, the client reading the data might not be the owner of the data.

Figure 2 shows the process for a client reading previously uploaded data from the blobber. The steps are as follows:

1) Before uploading data to the blobber, the owner must commit tokens to the blobber reward pool.
2) The owner of the data gives the client reading the data an *authorization ticket* permitting the read.
3) The client and blobber set up a secure connection.
4) The client requests a file and sends a signed read marker. If the client is not the owner of the data, the client must also include a "ticket" authorizing the read.
5) The blobber responds with the requested file. (Step 4 and step 5 may be repeated for different files.)
6) The blobber calls the smart contract once the reads are completed, including the read marker and the ticket (if the client is not the owner). The smart contract validates the read marker and checks the authorization ticket (if required); if the read is valid, the smart contract authorizes the release of the owner's funds stored in the blobber reward pool.
7) The smart contract releases tokens from the blobber reward pool and rewards the blobber for the work performed.

In the request for data, the client must send a signed read marker, authorizing the blobber to draw rewards from the blobber reward pool. This process is similar to how writes are rewarded, except that the blobber is paid directly, rather than having those tokens transferred to a challenge pool. The format of read markers is detailed in Section III-F1.

We note that a blobber might redeem the read markers without providing the data. However, the client will stop providing read markers if the blobber is not providing service.

Our design permits data owners to pay for other clients to read their data; The authorization ticket (signed by the owner) grants the client the authority to read data, but also specifies any desired restrictions. The authorization tickets prevent a blobber from reading the data as a mechanism to drain the blobber reward pool. The design of these tickets is discussed in Section III-F2.

1) Read Markers: Similar to write markers, read markers are used for clients to pay blobbers for their work off chain. The blobbers can redeem these markers on the blockchain.

A read marker contains:

- The `client_id` of the client requesting the data. This field ensures that the client is authorized to read the data.
- The `client_id` of the owner of the data.
- The `blobber_id` and `allocation_id`.
- `timestamp`, to ensure freshness.
- The path of the file being read, used to verify that the client is authorized to read the file.
- The data block range of the blocks of the file that the client requests.
- `read_counter`, giving the size of all data read to data.
- The client's signature of the other content of the read marker.

Unlike with write markers, the blobber is not expected to store the read markers after redeeming them. Also, many read markers may be redeemed by cashing in the latest read marker. The smart contract will release funds corresponding to the difference between the `read_counter` in the marker and the latest `read_counter` redeemed on the blockchain.

2) Authorization Tickets: Authorization tickets allow the owner of data to share it with other clients, and to pay for their reads. It also specifies the restrictions on the client's ability to read data. The ticket includes:

- The owner ID, blobber ID, client ID, and allocation ID
- The expiration time (`expiration`)
- The amount of data that can be downloaded (`max_data`)
- The path of files that the client can download (optional)
- The owner's signature

The storage smart contract only accepts read markers that match with the terms given by the authorization ticket. The `expiration` field ensures that the client does not use the ticket indefinitely; the smart contract will not accept read markers after this point. The `max_data` field specifies the maximum data that can be downloaded during the specified period of time.

Lastly, the owner can limit the client's access to a whitelisted subset of files. However, this last restriction depends on the blobber and client not colluding. Otherwise, the client can generate a read marker for a different file as a way of paying the blobber to provide restricted content.

G. Challenge protocol

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 mechanism 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 tokens.

The actual verification is done by a group of machines called the *validators*. The validators can be any group of machines, depending on what makes sense in the blockchain ecosystem. Validators are mutually untrusting. In this discussion, we will assume that the validators are a distinct group of machines from the miners and blobbers.

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

1) The mining network randomly selects the blobber data allocation to be challenged. This process also specifies the validators who will verify the challenge and provides a random seed to be used for the challenges. We refer to this stage as the *challenge issuance*.
2) In the *justification phase*, the blobber broadcasts the data to the validators along with the metadata needed to verify the challenge.
3) Finally, in the *judgment phase*, the validators share their results. Once the validators have agreed on the results of the challenge, they write a transaction to the blockchain indicating whether the test passed. This transaction also pays the validators and rewards the blobber.

Selecting validators is a particular challenge. In a *cronyism attack*, a blobber sends the data to a friendly validator who approves all challenges without validating 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 having the mining network randomly select a set of V validators[5]. For a challenge to pass, at least N miners must approve the results of the challenge. The difference between these values must be narrow enough to make a successful cronyism attack unlikely, but high enough to prevent an extortion attack.

An additional concern is that the validators actually do the verification work. We refer to a validator who does not do the work but who attempts to collect the reward as a *freeloader*.

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

[5]Specifically, the mining network use a *random beacon process* [8] to ensure that no single miner can unduly influence the random number selection.

- The allocation of data challenged, identified by `allocation_id`. Note that this should implicitly identify which blobber is challenged.
- The list of eligible validators.
- A random seed, which determines the indices of the data blocks in that allocation that the blobber must provide.
- The latest write marker at the time of the challenge.

Conceptually, we can envision this challenge as a roulette wheel, where the number of tokens currently due to the blobber from its challenge pool dictates its number of slices on the wheel. (An alternate approach would be to use the size of the data allocation instead, but this can lead to a subtle attack. A blobber could post an agreement for a negligible price with itself as the client, and then commit to storing large amounts of easily regenerated data. With a commitment to a large enough amount of data, other blobbers would be challenged only with a low probability. By tying the probability of being challenged to the amount of tokens in the challenge pool, we make this attack prohibitively expensive to carry out.)

The initial transaction essentially locks a portion of the blobber's stake and reward in a sub-pool specific to this challenge. We follow a "guilty until proven innocent" approach, which prevents a blobber from attempting a denial-of-service attack against a validator in order to avoid punishment. If the blobber never satisfies the challenge, the tokens are effectively burned.

2) Justification Phase: When the blobber observes the challenge issuance on the blockchain, it broadcasts its proof of storage to the validators with

- The file system metadata.
- The write marker proving that file system contents match what is stored on the blockchain.
- The challenged blocks of data, chosen pseudorandomly using the miner's random seed.
- The Merkle paths of those data blocks.

Once the validators receive the blobber's data, they each verify the data that they have been sent.

1) The validator verifies that
    - The file system metadata is valid.
    - The file system root hash matches the write marker.
    - The write marker matches the most recent commitment to the blockchain.

   At this point, the validator has established that the blobber's metadata is valid and matches the blockchain.
2) The validator then calculates the number of blocks on the system from the allocation size.
3) Using the random seed, the validator verifies that the blobber's blocks correspond with the pseudorandom sequence. (This serves to make every block of data on the system equally likely to be challenged, and ensures that the blobber did not try to game the results).
4) For each data block, the blobber verifies that the Merkle path matches up to the file metadata. As part of this process, the validator stores the two penultimate hashes of the Merkle tree; that is, it stores the two hashes that

can be hashed together to give the file's Merkle root. We refer to these hashes as the *validation proof.*

At most one of the hashes in the validation proof should have been provided by the blobber. (To ensure this behavior, the inclusion of additional hashes on the Merkle path is an automatic failure.) Therefore, the validator must have done the work to calculate at least one of the two hashes. This validation proof can be verified easily by the other validators. These proofs are an important part of our defense against freeloaders.

3) Judgment Phase: After the validator has completed its work, it broadcasts the signed hash of its results. We refer to this signed hash as the *pre-commit*. The hash format is:

$$H = hash(validationProofList, R)$$

where $validationProofList$ is a list of the hash pairs serving as validation proofs for each file, and R is a randomly chosen nonce selected by the validator.

The validator then waits to collect the pre-commits for the other validators. Once the timeout period has been reached, it broadcasts its $validProofList$ and its R value to publish its results. No additional pre-commits are accepted at this point. (If less than the minimum number of required signatures is received, it will rebroadcast and wait again).

The validator accepts the signatures of all other validators with valid proofs. provided that the other validators submitted valid pre-commits. Since the results are not publicly observable until after the results are completed, a freeloader is not able to provide a valid pre-commit.

Each validator submits a transaction to the blockchain with its results. The smart contract accepts the first transaction it receives, and only the first. At this point, the blobber receives its reward and the validators receive payment for their work.

The payout amount is pro-rated to match the total payout and the length of the contract. For instance, if blobber Bob's challenge pool contains 12 tokens from Alice for her storage paid over a contract period of 90 days, and the first challenge happens at day 45, Bob will receive 6 tokens for passing the challenge. If Bob is again challenged at day 60, he will receive an additional 2 tokens. On day 90, he will receive the remaining balance of 4 tokens.

The validators are paid in a pro-rated manner similar to how the blobber is rewarded. An equal portion of the reward is set aside for every validator, *even those that did not participate in the validation*. However, the rewards are only distributed to validators who participated in the validation process; the reward for non-participating validators is burned. This design ensures that validators have no incentive to exclude each other, but have a strong incentive to perform the validation work.

4) Challenge Failures: With the challenge protocol, we are concerned that blobbers could be the victim of a denial-of-service attack when they are challenged. An attacker interested in damaging the network could target challenged blobbers, potentially destroying the blobbers' rewards and staked tokens, until no blobbers are willing to provide the service.

As a result, there is no time-out period for challenges. The blobber could contact the validators to complete the challenge at any time. This choice creates a misalignment of incentives, since validators are only paid on successful challenges, and hence might be motivated to collude with a cheating blobber.

To address this concern, we allow blobbers to broadcast a signed *confession* indicating that they are unable to satisfy the challenge. The validators can then write this message to the blockchain and receive their rewards for their validation work. This transaction releases a portion of the blobber's stake back to them. The client owning the data then receives back the token rewards and the rest of the challenged blobber's stake. With this design, a blobber caught cheating that acts in its own best interests will reward the other parties in the system.

H. Challenge Protocol Experimental Results

In order to validate the effectiveness of our challenge protocol and help determine the optimal settings, we ran experiments to determine the number of challenges needed to identify a cheater. We simulate a blobber requesting a single block of data from a single file. Via information gained from the blockchain and the signed write marker, the validator knows the number of blocks in the file and the Merkle root of the file. We simulate a blobber that attempts to cheat by not storing all blocks of data. Each setting was tested with 21 rounds (in order to simplify calculation of the median).

Table I shows our results for a blobber that is expected to store a 32-block file, but who only stores a smaller number of blocks (column 1). Our results show that a blobber failing to store small amounts of data might be able to evade detection for several challenges (requiring a median 27 challenges if they stored all but 1 block of the data). However, as their cheating increases fewer and fewer challenges are needed.

IV. Economic Analysis

The process of locking tokens and earning interest on locked tokens is similar to purchasing a certificate of deposit ("CD") or non-transferable bond with a maturity equal to the locking period. A key difference, however, is that traditional short term CDs or bonds pay out interest at the time of maturity, whereas the interest paid on locking tokens is paid upfront[6].

Holding tokens has an opportunity cost: rather than having d U.S. dollars (USD) in tokens, a client may hold d USD at a bank and earn some relatively risk-free interest rate r^f over time period t. Assuming a constant token price over t, by holding tokens, the client would give up $d * r^f$ dollars they otherwise might have earned. However, if tokens pay a rate r where $r \geq r^f$, this opportunity cost is offset if not dominated, assuming no change in the token exchange rate.

One benefit to clients of upfront interest payments is the reduction in exchange rate risk. Over relatively short periods of time (say the several minutes needed to complete the locking transaction and receive the interest payment), expected token price changes would be small in comparison to expected price changes over longer time periods (say the 90 day proposed

[6]Bonds with longer term maturities often have periodic "coupon" payments. There are some relatively less common CDs that do pay interest upfront.

TABLE I
CHALLENGE PROTOCOL EXPERIMENTAL RESULTS

Blocks Stored	Avg. # Challenges	Median # Challenges
31 / 32	35.57	27
30 / 32	16.67	13
29 / 32	10.10	6
28 / 32	9.24	6
27 / 32	7.62	6
26 / 32	5.76	4
25 / 32	5.52	4
24 / 32	4.81	4
23 / 32	4.62	4
22 / 32	3.48	2
21 / 32	3.29	3
20 / 32	2.67	2
19 / 32	2.95	3
18 / 32	2.00	2
17 / 32	2.14	2
16 / 32	2.00	1
15 / 32	2.14	2
14 / 32	1.67	1
13 / 32	1.90	2
12 / 32	1.29	1
11 / 32	1.33	1
10 / 32	1.62	1
9 / 32	1.19	1
8 / 32	1.43	1
7 / 32	1.10	1
6 / 32	1.19	1
5 / 32	1.14	1
4 / 32	1.10	1
3 / 32	1.14	1
2 / 32	1.10	1
1 / 32	1.00	1

locking period). This helps in part to reduce the exchange rate risk assumed by clients by locking tokens.

To date, most cryptocurrencies and tokens have experienced relatively high exchange rate volatility versus the USD. However, by setting $r > r^f$, clients would in part be compensated for taking on this exchange rate risk, making the tokens relatively more attractive than other cyrptocurrencies or tokens, all else being equal. Moreover, the ability to pay interest on tokens opens up other possible economic design options. For example, an increase in r during times of downward token exchange rate movement may put upwards pressure on the token price by a) making holding tokens more attractive, and b) reducing the readily available supply of tokens due to increased token locking. However, this ultimately increases the token supply more quickly, which may put some longer term downwards pressure on the token value.

A simple example illustrates the benefit of upfront interest payments to clients. Assume a client with 400 tokens wishes to purchase storage that will cost 30 tokens, and needs to pay a miner 10 tokens to write the transaction to the blockchain. Also assume the interest rate is 10% for the locking period. The client has two options[7]. 1) Lock 400 tokens, earn the 40 tokens in interest, and from this interest pay the blobber and the miner. At the end of the locking period, the client still has 400 tokens. 2) Directly pay the 30 tokens to the blobber and 10 tokens to the miner, leaving the client with 360 tokens. As part of this same transaction, the client may lock the 360 tokens at the 10% rate, and would then have, at the end of the locking period, 396 tokens[8]. Thus, the client is better off paying for services with interest earned upfront on locked tokens.

[7]These are arguably the simplest two options. Other more complex transactions are possible

V. RELATED WORK

Other cryptocurrencies have explored locking tokens as a form of Sybil resistance. Tendermint [9] uses a coin-deposit approach; coins are locked for some set period of time in exchange for the right to propose a block and the right to vote on valid blocks. To spend the coins, the miner must post a special *unbonding transaction* and then wait for a set period of time for the coins to be released. Peercoin [10] uses a mix of proof-of-work and proof-of-stake based on coin age. Essentially, miners still compete in a proof-of-work protocol, but the difficulty varies depending on the amount of coin age consumed. While Peercoin does not explicitly lock tokens, its design of encouraging tokens to be inactive in order to gain new tokens bears a resemblance to our design.

The central distinction between our approach and these protocols is that, rather than requiring miners to take tokens out of circulation, clients lock tokens to produce new rewards for the miners. This design allows us to use that same mechanism for rewarding other service providers.

Basis [11] is one of the few cryptocurrencies to address token price volatility. Their design monitors the Basis-USD exchange rate, issuing new tokens when the price of Basis exceeds 1 USD, and issuing special *bond tokens* when the price falls below 1 USD. Their bond tokens are similar to our model: they remove tokens from circulation by promising token rewards. However, their design pays rewards at the end of the locking period. While price stability is not one of the central goals of our design, we note that the locking mechanism has a similar effect on the economics of our token.

Previous work has also explored storage systems and the blockchain. In Filecoin [4], miners prove that they are storing data through special proofs-of-spacetime [12], which are themselves based on proofs-of-replication [13]. 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.

Both Sia [14] and Storj [15] 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 [16] mechanism uses zero knowledge proofs.

[8]If the client could continuously and instantaneously lock earned interest, the client would earn 40 tokens ($\sum_{t=0}^{\infty} 36i^t$). However, the client would incur a 10 token transaction fee each time tokens were locked.

Miller at al. [17] discuss how to reduce Bitcoin's proof-of-work overhead with a proof-of-retrievability solution, which they use in a modified version of Bitcoin called Permacoin. Essentially, Permacoin miners prove that they are storing some portion of archival data. Since the proof itself provides part of the data, miners inherently serve as backup storage.

Ali et al. [18] show how a distributed public key infrastructure (PKI) can be built on top of a blockchain. While their approach is similar to ours in that storage is done off-chain, there is less focus in their work on verifying the storage itself.

Other approaches, such as Burst's proof-of-capacity (PoC) [19] and SpaceMint's proof-of-storage [20] 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 [19].

Verifying that a storage provider is actually providing the storage remains a challenging problem. Juels and Kaliski [21] develop proofs-of-retrievability (POR) for archival data. PORs embed special sentinel values in the data (hidden by encryption) which the verifier can use to probabilistically verify the data storage. Shi et al. [22] develop a dynamic POR scheme, meaning that instead of storing static, archival data, the data with this scheme can be updated. Shacham and Waters [23] discuss how to make these proofs more compact. Provable data proofs (PDPs) [24] are an alternate strategy for verifying data storage. The client maintains secret challenges and responses, which can be kept to a constant amount through the use of homomorphic encryption. As with PORs, work has been done to support dynamic data [25] and to improve efficiency [26]. The reliance on secret data to verify storage makes many of these approaches unsuitable for our system. Since the blockchain has no support for secret data, and neither the client nor the blobber can be trusted, there is no way for the validators to determine whether the blobber is cheating or the client lied about the data being stored. However, there are *publicly verifiable* [22], [24] variants of both PORs and PDPs that avoid this limitation.

VI. Conclusion and Future Work

In this paper, we have reviewed our token-locking reward model and shown how it can incentivize miners to accept transactions without forcing clients to sacrifice their tokens. We have further shown how this model can incentivize service providers by developing a storage system, and demonstrated how our design can leverage the blockchain to ensure that storage providers live up to their requirements.

Our analysis shows that our design can help to reduce the volatility typically associated with cryptocurrencies, and that our model of paying interest to token holders reduces the opportunity cost of holding tokens compared to holding US dollars. We also show that our model of rewarding service providers with interest generated from locked tokens is more profitable for clients compared to paying for services directly.

One interesting aspect of our design is that locked tokens are temporarily out of circulation. In contrast, money in financial bonds can be actively used by the borrower. In future work, we will explore ways that locked tokens could still be in active use, and thereby possibly help to stimulate the token economy.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] P. Merrill and T. H. Austin, "0chain token reward protocol," 2018.

[3] J. S. Plank, "Erasure codes for storage systems: A brief primer," *;login:*, vol. 38, no. 6, 2013. [Online]. Available: https://www.usenix.org/publications/login/december-2013-volume-38-number-6/erasure-codes-storage-systems-brief-primer

[4] "Filecoin: A decentralized storage network," Protocol Labs, Tech. Rep., August 2017.

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

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

[7] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO, A Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1987.

[8] T. Hanke, M. Movahedi, and D. Williams, "DFINITY technology overview series, consensus system," *CoRR*, vol. abs/1805.04548, 2018. [Online]. Available: http://arxiv.org/abs/1805.04548

[9] J. Kwon, "Tendermint: Consensus without mining," https://tendermint.com/static/docs/tendermint.pdf, 2014.

[10] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," http://primecoin.org/static/primecoin-paper.pdf, 2012.

[11] N. Al-Naji, J. Chen, and L. Diao, "Basis: A price-stable cryptocurrency with an algorithmic central bank," Intangible Labs, Tech. Rep., June 2017.

[12] T. Moran and I. Orlov, "Proofs of space-time and rational proofs of storage," *IACR Cryptology ePrint Archive*, vol. 2016, 2016. [Online]. Available: http://eprint.iacr.org/2016/035

[13] "Proof of replication," Protocol Labs, Tech. Rep., July 2017.

[14] D. Vorick and L. Champine, "Sia: Simple decentralized storage," Nebulous Inc., Tech. Rep., November 2014.

[15] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, and C. Pollard, "Storj: A peer-to-peer cloud storage network," Storj Labs Inc., Tech. Rep., December 2016.

[16] N. Lambert, Q. Ma, and D. Irvine, "Safecoin: The decentralised network token," https://docs.maidsafe.net/Whitepapers/pdf/Safecoin.pdf, 2015.

[17] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz, "Permacoin: Repurposing bitcoin work for data preservation," in *Symposium on Security and Privacy*. IEEE Computer Society, 2014.

[18] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *USENIX Annual Technical Conference*. USENIX Association, 2016.

[19] S. Gauld, F. von Ancoina, and R. 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, 2017.

[20] S. Park, K. Pietrzak, A. Kwon, J. Alwen, G. Fuchsbauer, and P. Gazi, "Spacemint: A cryptocurrency based on proofs of space," *IACR Cryptology ePrint Archive*, vol. 2015, 2015.

[21] A. Juels and B. S. K. Jr., "Pors: proofs of retrievability for large files," in *Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 2007.

[22] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.

[23] H. Shacham and B. Waters, "Compact proofs of retrievability," *J. Cryptology*, vol. 26, no. 3, 2013.

[24] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song, "Provable data possession at untrusted stores," in *Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 2007.

[25] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, 2015.

[26] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," *IACR Cryptology ePrint Archive*, vol. 2008, 2008. [Online]. Available: http://eprint.iacr.org/2008/114