Writeup for ZeroPool's Sharded Storage Solution: Building the Protocol Bottom Up

Ivan Oleynikov
ivan.oleynikov95@gmail.com

ZeroPool
https://zeropool.network/

April 8, 2024

In this document, we present a walkthrough of ZeroPool's Sharded Storage protocol [5, 6]. This protocol lets some blockchain nodes rent out their disk space to other ones. Compared to existing approaches, it offers good scalability and the blowup rate for a given security level.

The main novelty of this protocol is being ZK-native: all of its operations — polynomial commitments, FFT — can be efficiency verified in a zkSNARK. As a consequence, Sharded Storage can enable rollups to store their blocks with little overhead, and combine the zkSNARK proof of data being stored (and available for nodes to read) with the zkSNARK proof of rollup's application logic. This allows nesting rollups with little overhead, and can enable unparalleled scaling of blockchain applications. The impact of Sharded Storage is not limited to rollups, it can also be used for other blockchain applications that require to store large amounts of data with strong security.

For instance, Sharded Storage can be configured to achieve more than 100 bits of statistical security — against an advesary controlling half of the network — with blowup rate of only 8. In other words, each 1 GB of payload will cause nodes to store 8 GB of shards. This approaches the cost of Web2 where this 1GB would be stored directly, but provides drastically stronger security guarantees.

In this walkthrough, we present the protocol step by step, starting with a very simple setting, then gradually adding requirements and showing how to satisfy them one by one. We provide a rather informal exposition, with emphasis on intuitive understanding over formal details. For a full formal presentation, see the original article [5].

Contents

Notation

In this section, we briefly review the notation for some of the primitives that we use.

Polynomial Commitment Scheme A Polynomial Commitment Scheme consists of three algorithms: deterministic $c := \mathsf{commit}(m)$ to commit to a polynomial $m(x)$, $(b, \pi) := \mathsf{prove}(m, a)$ to evaluate $b := m(a)$ and produce the proof π of this, $\mathsf{verify}(c, a, b, \pi)$ returning 1 if proof π of $b = m(a)$ is correct (where $c = \mathsf{commit}(m)$ for a polynomial $m(x)$) and 0 otherwise.

This notation simplifies some nuances like public-keys and decommitment information that some PCS have. FRI [4] is an example PCS that can be expressed in terms of the notation we give here.

Reed–Solomon Code with Erasures [3] We denote Reed–Solomon encoding with

$$w := \mathsf{encode}(m, (1, 2, \dots, n)),$$

which produces a codeword $w = (w_1, w_2 \dots w_n)$. The input word is a polynomial $m(x)$ of degree $l - 1$ with $l \leq n$.

The input word can be recovered back from any l symbols of w. Say, if w was transmitted over a lossy channel and only the symbols $w_{i_1}, w_{i_2} \dots w_{i_l}$ got through, one can run

$$m := \mathsf{decode}((i_1, w_{i_1}), (i_2, w_{i_2}), \dots, (i_l, w_{i_l})).$$

In practice, encode is a simple evaluation of $m(x)$ at the points $g^1, g^2, \dots, g^n$ (for a pre-defined element g). While decode is Lagrange interpolation [1] of $m(x)$ from provided points. In this form, it is suitable for efficient implementation using Fast Fourier Transform (FFT).

We assume that Reed–Solomon Code and PCS used work over the same field $\mathbb{F}$.

Step #1: Proof of Data Availability

Consider the following problem: given Alice, Bob and Charlie, Alice wants Bob to store some data for her. She wants to give her data to Bob, go offline, and later come back and claim her data. While Alice is gone, we want Charlie to periodically query Bob and verify that Bob has not erased Alice's data.

In the final solution (built in the following sections), Alice will model the data owner who pays for storage, Bob will be a node that provides storage for a fee, and Charlie will be the Rollup that coordinates them and manages the fees.

We want to ensure that if Bob decides to erase part of the data that Alice handed him (to save his disk space), Charlie will catch him during the next check with some good probability. Below is the protocol that achieves this. We assume that Alice's data is public and can be disclosed to Bob or Charlie. (If privacy is needed, Alice can add an extra layer of encryption to her data on top of the solution we present here.)

Alice's Dealing Protocol

Suppose that Alice has a message m that she wants Bob to store for her. She performs the following steps:

1. Alice uniquely encodes her message as a polynomial $m(x)$ of the appropriate degree over the field $\mathbb{F}$ used by commit.
2. Alice computes commitment $c := \text{commit}(m)$. Then she sends m to Bob, and c to Charlie.
3. Bob recomputes Alice's commitment $c := \text{commit}(m)$, and sends it to Charlie.
4. Charlie checks that the commitments he got from Alice and Bob are the same. If not, Charlie returns an error to Bob.

Observe that commitment c is very short, while message m may be quite long. Now Charlie holds the short c which binds the long message m stored by Bob — in the sense that Bob can not find another message m' that would produce the same commitment c.

Charlie's Challenge Protocol

Charlie can periodically run the following protocol to check if Bob is still storing m. This protocol runs only after a successful execution of Alice's Dealing Protocol above. At this point, Charlie knows the commitment c he got from Alice; Bob knows the message m he got from her.

1. Charlie picks a random point a from the field $\mathbb{F}$, sends it to Bob.
2. Bob computes $(b, \pi) := \text{prove}(m, a)$, where $b = m(a)$ and π is the proof certifying this. The m is treated as a polynomial here the same way as in Alice's Dealing Protocol.
3. Bob sends b and π to Charlie.
4. Charlie verifies the proof by doing $\text{verify}(c, a, b, \pi)$.

If Bob did not have m at the beginning of the protocol above, he would not be able to reliably compute correct b and produce a proof π that will pass Charlie's test.

If he tries to guess the correct b and make the proof pass, the probability of him succeeding will not exceed $1/q$, where q is the number of elements in the field $\mathbb{F}$.

Suppose Alice tries to frame Bob and give him a message m' (in Dealing Protocol) that is different from the message she committed to. In that case, Charlie will see this at the last step, notify Bob and will not run the Challenge Protocol with him.

Step #2: Mutliple Messages

The above solution can be easily extended to support multiple messages. Below are the same two phases, but now Alice stores multiple messages.

Alice's Dealing Protocol

Let Alice have messages $m_0, m_1, \ldots, m_{l-1}$, all of which she encodes as polynomials $m_j(x)$.

1. Alice computes commitments $c_j := \mathsf{commit}(m_j)$. Then she constructs a Merkle tree with values c_j in the leaves. Denote the root hash of this tree as r.
2. Alice sends m to Bob and r to Charlie.
3. Bob recomputes Alice's commitment values c_j and the Merkle tree root r, then sends r to Charlie.
4. Charlie checks that he got the same r from both Alice and Bob. If not, he returns an error to Bob.

Charlie's Challenge Protocol

The Challenge Protocol between Charlie and Bob also extends to mutliple messages as shown below.

1. Charlie picks a random seed s suitable for a PRG and sends it to Bob.
2. Bob computes a PRG stream from s and uses it to sample random indices $i_1, i_2, \ldots, i_k$ and random points $a_1, a_2, \ldots, a_k$.
3. Bob evaluates each stored polynomial $m_{i_j}(x)$ at the point a_j (for $j \in \{1, 2, \ldots, k\}$), and produces a proof of having done so correctly: $(b_j, \pi_j) := \mathsf{prove}(m_{i_j}, a_j)$.
4. Bob sends the computed values b_j and π_j to Charlie.
5. Charlie produces the same keystream from s as Bob, computes the same indices i_j and points a_j. Then he verifies the proofs by doing $\mathsf{verify}(c_{i_j}, b_j, a_j, \pi_j)$.

Extensions

Non-Interactive Challenge Protocol The last two steps of Charlie's Challenge Protocol can alternatively be done non-interactively, using a zkSNARK. Bob acts as a zkSNARK Prover and Charlie as a Verifier. We briefly outline the conditions checked by the zkSNARK proof.

Public inputs: seed s and Merkle tree root r.

Private inputs: polynomial evaluation proofs π_i, tree leaves c_i, input points a_i, evaluations b_i.

Conditions checked by the zkSNARK: (1) a_j are correctly generated from s, (2) c_j are correct leaves of tree r, (3) π_j prove that $b_j = m_{i_j}(a_i)$.

Furthermore, the PRG seed s can be derived by Bob himself without Charlie. For example, using hashes with Fiat–Shamir heuristics and Verifiable Delay Functions on the blockchain. This eliminates all the messages Charlie was sending and makes the whole Challenge Protocol non-interactive: Bob can produce the proofs without any input from Charlie, and Charlie will still verify that they are correct.

Modifying Messages This solution allows Alice to replace her message m_i with a new value m_i'. For that, she computes a new commitment $c_i' := \mathsf{commit}(m_i', k)$, sends it to Bob and Charlie. Bob, in response to that, sends Charlie the new tree root r' with the Merkle proofs showing that it differs from r in exactly one leaf.

With this, we can essentially implement a read-write disk for Alice. If she splits her data into blocks and assigns each block to a separate message m_j, she can read and write to any of such blocks independently from the others.

Discussion

This solution checks only k of the messages m_0, m_1, ..., m_{l-1}. Suppose that Bob erases an $\varepsilon \in [0, 1)$ fraction of messages $\{m_j\}_j$; then he gets caught if and only if at least one of those k checked messages happens to be one he has erased. The probability of this is $\rho = 1 - (1 - \varepsilon)^k$. For a very small ε, $\rho \approx k\varepsilon$.

Step #3: Fees and Penalties for Bob

Now, we extend our model from #2 with one more piece of complexity: Bob will be paid tokens for storing Alice's data. And now he is not guaranteed to strictly follow the steps of the protocol we made, but may deviate from it if that helps him earn more tokens. In other words, we still assume that Bob will not try to break the system for its own sake, but will only do so if he believes it will earn him more tokens.

Consider the following incentive policy for Bob: Charlie will pay Bob A tokens each time Bob successfully proves that he keeps Alice's data, and he will slash Bob for B tokens if Bob fails to prove that he stores the data. Our goal is to set parameters A and B in such a way that Bob's earnings are maximized when $\varepsilon = 0$, i.e. Bob has not erased any messages.

If Bob erases ε fraction of Alice's messages, he could employ the freed space for some other use (like renting it out to someone else) and earn $C\varepsilon$ amount of tokens for that. When Charlie performs the Challenge Protocol with Bob, Bob will be rewarded with A tokens with probability ρ, or shashed for B tokens with probability $1 - \rho$. This yields the following formula for expected earnings of Bob per a round of Challenge Protocol with Charlie. Note that, when designing the protocol, we do not set the parameter C, but rather estimate how much a malicious Bob could earn using certain amount of disk space.

$$\begin{aligned} C\varepsilon + \rho A - (1 - \rho)B &= C\varepsilon + (1 - k\varepsilon)A - k\varepsilon B \\ &= A - (A + B - C/k)k\varepsilon. \end{aligned}$$

For $\varepsilon = 0$, this value collapses to A, which is strictly positive. If we set B so that $B > C/k - A$, this will ensure that increasing ε will only make Bob's earnings smaller, and demotivate him from erasing any data.

For normal operation, when Bob is rewarded with A tokens, his reward is payed from the tokens Alice has deposited as her payment for rented disk space.

Step #4: Sharing Data Among Mutliple Bobs

In this section, we relax the assumption we made on step #3 about Bob following economic incentives. Now, we consider a setting with one Alice, one Charlie and a pool of many Bobs. Some Bobs are honest, they follow economic incentives and try to maximize their earnings and avoid being slashed — like on step #3. Other Bobs are malicious, they are colluding together and try to cause DoS on Alice's data even if Charlie slashes them for it.

Such malicious Bobs account for some actor who is willing to DoS Alice's data despite all the fees they will have to pay to Charlie. This could be someone who has special interest in Alice's data being lost — due to reasons outside of our protocol.

Suppose there are n Bobs in total, denoted as $\text{Bob}_1 \ldots \text{Bob}_n$. Some Bobs may be malicious, we do not know which ones exactly, and we do not know how many of them are malicious. But we are guaranteed that at least t of them are honest. The following are Alice's Dealing Protocol and Charlie's Challenge Protocol that allow to store Alice's data on Bobs and periodically verify that they are still keeping it.

Alice's Dealing Protocol

Suppose that Alice has messages m that she wants Bob to store for her. She performs the following steps.

1. Alice encodes her message m into n shares using Reed–Solomon code:

$$(\mathsf{sh}_1, \ldots \mathsf{sh}_n) := \mathsf{encode}(m, (1, \ldots n)).$$

2. Let H be a hash function that takes two polynomials over $\mathbb{F}$, and produces an element of $\mathbb{F}$.
   (a) Alice generates random seeds σ_j for plot generation;
   (b) computes the plots $p_j := \mathsf{plot}(\sigma_j)$;
   (c) computes special values $\alpha_j := H(\mathsf{sh}_j, p_j)$, treating sh_j and p_j as polynomials over $\mathbb{F}$;
   (d) combines shares with the plots using $v_j := \mathsf{sh}_j + \alpha_j \cdot p_j$.
3. Alice, Charlie and Bob_j run the Dealing Protocol from step #1 to store $(\sigma_j, \alpha_j, v_j)$ on Bob_j (for $j \in \{1 \ldots n\}$). As a result, Bobs get the triples $(\sigma_j, \alpha_j, v_j)$, Alice and Charlie get the commitments to these triples.

   Note that these protocol runs can be done in parallel, since they do not depend on each other.

Charlie's Challenge Protocol

Charlie runs n challenge protocols from step #1, verifying that (for $j \in \{1 \ldots n\}$) each Bob_j is still storing $(\sigma_j, \alpha_j, v_j)$ given to him by Alice.

Alice's Data Recovery Protocol

After Alice has dealt the data to Bobs, went offline for some time, she may come back later and want to get her data back.

Below, we describe the protocol that achieves this. At the beginning of this protocol, Bobs hold the triples (σ, α, v) Alice has asked them to store. Alice holds the commitments to these triples.

1. Each Bob_j sends the values $(\sigma_j, \alpha_j, v_j)$ he stored back to Alice. Bobs that are malicious may choose not to do this here. But since at least t Bobs are honest they will send the data back to Alice, and

she will get at least t such triples. Suppose that the values Alice received are $(\sigma_{i_j}, \alpha_{i_j}, v_{i_j})$, where $i_1, i_2 \dots i_t$ are the indices of at least t Bobs that have sent her the data and j is ranging over $\{1 \dots t\}$.

2. Alice verifies that the values she received match the commitments she has stored (from Dealing Protocol).
3. Alice recomputes the plots $p_{i_j} := \mathsf{plot}(\sigma_{i_j})$.
4. Then she recomputes the shares $\mathsf{sh}_{i_j} := v_{i_j} - \alpha_{i_j} \cdot p_{i_j}$.
5. Alice decodes her message from the shares $m := \mathsf{decode}((i_1, \mathsf{sh}_{i_1}) \dots (i_t, \mathsf{sh}_{i_t}))$.

Discussion

The protocols we described in this section, allow Alice to distribute her data to multiple Bobs, have Charlie periodically ensure that the Bobs are storing the data, and later let Alice ask for her data back. We assume that at least t out of n Bobs follow economic incentives, and with the fees and penalties from step #3, the most optimal strategy for such Bobs would be to faithfully keep the data and give it back to Alice.

The way we use Reed–Solomon code ensures that if at least t Bobs return their data to Alice (she she asks), Alice will be able to recover it. And our use of plotting ensures that multiple honest Bobs can not collude together to deduplicate their data and store only one copy in a centralized manner. Plotting ensures that if they do so, then their cost of responding to Charlie's challenges will require orders of magnitude more CPU time.

One can apply the Merkle tree technique from step #2 here to have Alice store multiple messages on Bobs. We have not included it to simplify the exposition. Another possible optimization is to collect the commitments Alice and Charlie are storing for each Bob into a Merkle tree, so they store only one commitment for all Bobs instead of commitment per each Bob.

Another simplification we have made in the description above is having only one Alice. But in practice we will use a version where many Alices are storing their messages on the same pool of Bobs and have one Charlie run the challenge protocols for all of them.

The protocol we described in this section allows multiple Alices deal their data to the same Charlie and a pool of Bobs associated with him. In the following sections we will use the version of step #4 where one Charlie and the pool of Bobs associated with him store messages of many Alices (and using the Merkle tree technique).

Step #5: Allowing Mutliple Alices and Mixing Bobs

Consider a setting with the following participants: one Dave, multiple Charlies, multiple Alices, and multiple Bobs. Bobs are arranged into pools, one pool for each Charlie.

Roles Charlies, Alices and Bobs play the same roles as in step #4. Each Charlie has a pool of Bobs associated with him, and a number of Alices that are renting the disk space in this pool. Dave will be coordinating the other participants, distributing Alices and Bobs between pools controlled by Charlies.

Assumptions We assume that at least half all Bobs is honest and are incentivised to maximize their earnings in this protocol. Dave and Charlies are all honest, they always follow the steps of the protocol we give here. The pools and Charlies stay static throughout the protocol execution, while Alices and Bobs can come and go dynamically.

Overview In this section we describe the algorithm for Dave to handle: new Alices coming and wanting to store their data in some pool, existing Alices quitting the protocol (and removing their data), new Bobs coming online and asking to assign them to a pool (where they can start earning rewards for renting out their disk space), existing Bobs going offline. Will employ a special mixing technique to continuously rearrange Bobs between pools, so that each pool (with very high probability) contains at least t honest Bobs as assumed in step #4.

Dave's Management Algorithm

Dave maintains the list of pools. For each pool, Dave knows how much data is stored in it and the list of Bobs supplying it with space. All his data is public, other participants can always read it.

This list briefly outlines the way Alices, Bobs and Charlies interact with Dave and form pools. Once assigned to a pool, they perform the protocols described in step #4.

Alice. When a new Alice joins the protocol, she looks at the current (public) configuration stored by Dave, picks whichever pool she likes and directly contacts Charlie and Bobs of that protocol to store her data. When Alice wants to quit the protocol, she contacts the Charlie and Bobs of her pool.

Alice also deposits the necessary amont of tokens as a payment for storing her data. The rewards of Bobs will be payed from this amount.

Bob. When a new Bob joins the protocol, he registers himself with Dave and waits for further orders from Dave. Dave may order an unassigned Bob to join some pool, or order a Bob who is in a pool to leave that pool (becoming unassigned) or move to another pool. When a Bob wants to quit the protocol, he notifies Dave and Charlie managing his pool (if he is assigned to one).

While a Bob is unallocated, he continuously performs space mining like in Chia [2]. Dave does not register those Bobs who fail to prove that they have enough free space. This prevents DoS of Dave by nodes that do not have enough space: just getting registered with Dave costs node more resources than for Dave to handle that registration.

When Dave assigns Bob_u to a pool, Bob_u needs to get his shares of that pool's data to store somewhere. He can't get it from Alices via the Dealing Protocol, since Alices may not be online at this moment. Therefore, Bob_u will ask t of the Bobs who are already in this pool to send him their shares, recover the data from these shares (the same way as Alice would do it), and recompute his share (using Alice's Dealing Protocol with the same seed locally). In some cases this can be optimized: when Bob_u is taking the place of another Bob_v who is being moved out of this pool at the same time, he can directly get his share from Bob_v (and verify that it's correct using commitments stored by Charlie).

Charlie. Each Charlie notifies Dave of any relevant changes in his pool: Bobs quitting (or being kicked by Charlie due to failing the Challenge Protocol), the amount of used space changing (due to Alices joining or leaving).

The list below describes Dave's actions in response to events happening in the protocol.

When a new Bob joins: mark him as unassigned to any pool.

When a Bob_v **assigned to a pool** Π **leaves:** pick a random Bob_u that is unallocated or in another pool, move Bob_u to Π in place of Bob_v. If Bob_u was taken from another pool Π', pick an unallocated Bob_w and assign him to the pool Π'.

Perform the following mixing m times: pick a random bob Bob_i from Π and random Bob_j not from Π (i.e. from unallocated or another pool), swap Bob_i and Bob_j.

Discussion

The evolution of Bobs' distribution in the pools can be modelled by a Markov process. We do not present the analysis here, but the original Sharded Storage article shows that one can pick some moderate parameter values for this protocol to ensure that this process is extremely unlikely to arrives at a state where some pool has less than t honest Bobs (and Alice's data is lost). For example, if each pool has $n = 512$ Bobs in total, any $t = 64$ of whom can recover the data, each time a Bob leaves we perform $m = 3$ mixings, and half of all Bobs in the protocol are malicious, then the probability of data being lost is below 2^{-115}. The blowup rate for the parameters in this example is $512/64 = 8$; i.e. if an Alice stores 1 GB of payload in some pool, that will cause the Bobs of that pool to store 8 GB of shares in total.

The description here makes a simplifying assumption of pools being static. In practice, the number of pools may need to go up and down to account for changes in supply (from new Bobs) and demand (from new Alices) of disk space to rent. Such logic can be implemented by Dave, but we do not go into detail about it here.

Another detail we left out in this description is fees paid to Bobs that are unassigned to any pool (to make sure that such Bobs also get paid), as well as the cost of storing the data for Alices (to accomodate for changing supply and demand). There are multiple promising approaches here (see the original article [5]), and the concrete approach can be picked after the more essential details are sorted out.

Step #6: Running on Blockchain

In the sections above, we took a rather abstract approach and did not specify how the roles of the participants we described map to blockchain. We address it in this section and explain what role each of Alice, Bob, Charlie and Dave play in blockchain setting. For details, refer to the original article [5, 6].

Alice is a user who wants to pay to have her data stored.

Bob is a miner who earns tokens for renting out his disk space.

Charlie is an L3 rollup who verifies the actions of Bobs, publishes the necessary updates to L2 Dave together with zkSNARK proofs of these updates being correct.

Dave is an L2 rollup who collects updates from L3 Charlies, and publishes the commitment to its own state with zkSNARK proofs on blockchain.

All operations this protocol performs can be efficiently verified with a zkSNARK. This radically separates Sharded Storage from previous approches to decentralized storage on blockchain.

One particularly interesting application of Sharded Storage is to let an L2 rollup store its own blocks in a pool of Bobs. The rollup must make its blocks available for any nodes that may want to join it in the future. Sharded Storage can store these blocks, and produce a ZK-proof of them being stored and available for anyone, achieving better performance than duplication or on-chain storage.

If now the same L2 rollup also runs our algorithm, it can take the roles of Dave and Alice simultaneously and pay Bobs for storing its blocks, then create a ZK proof of this being the case (proof of data availability) and push that proof up level above it (which may be another rollup or L1 chain). Furthermore, the proof of data availability can be combined with the L2 rollup's (application-specific) proof of state transition, and verified in a single run. This way, we encode that the rollup's state was modified correctly and its blocks were stored reliably using a single proof.

This approach can be extended further: if we build a hierarchy of rollups, each of them can prove that its data is safely stored on Bobs, and then incorporate similar proofs from the underlying rollups into it. It can drastically improve the scalability rollup architecture.

Summary

The end result we have built on step #6 is a protocol with the following levels:

- L2 rollup is assigning the disk space providers to the pools.
- Each pool is managed by its own L3 rollup which continuously checks that providers are indeed storing the data they were given.
- And the provider nodes are storing the data and generating proofs of it to claim rewards.

The main contribution of this protcol is being ZK-native and enabling the rollups to produce zk-SNARK proofs of their data being safely stored on provider nodes. This allows to seamlessly incorporate this solution into an existing logic that rollups are proving with zkSNARK.

With an appropriate reward system (which we do not detail here), this solution can both help end-applications scale and also improve the architecture of rollups themselves.

References

[1] Lagrange polynomial, Wikipedia. `https://en.wikipedia.org/wiki/Lagrange_polynomial`.

[2] Proof of space, Chia documentation. `https://docs.chia.net/proof-of-space/`.

[3] Reed–solomon error correction, Wikipedia. `https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction`.

[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[5] Igor Gulamov. Blockchain sharded storage: Web2 costs and Web3 security with Shamir secret sharing. `https://ethresear.ch/t/blockchain-sharded-storage-web2-costs-and-web3-security-with-shamir-secret-sharing/18881`.

[6] Igor Gulamov. Minimal fully recursive zkDA rollup with sharded storage. `https://ethresear.ch/t/minimal-fully-recursive-zkda-rollup-with-sharded-storage/19020`.

Contacts

With questions and comments for this document, write to one of the following community channels.

- `https://discord.gg/TNesyTqFdf`
- `https://t.me/ZeroPoolCommunity`