Storage Auditing Using Merkle Trees and KZG Commitments

Dr. Chloe I. Avery* Justin Sheek†

July 26, 2024

Contents

1 Overview/Background

This paper presents ideas for how audits of storage Providers could be performed in a decentralized data storage system. We introduce two different possibilities: using Merkle trees and using KZG commitments. KZG commitments are the more efficient way of performing audits of storage providers, and offer many additional benefits. However, Merkle tree audits work much the same way, are easier to understand, and provide a good scaffolding for understanding the KZG case. Therefore, we first discuss Merkle tree audits, and then discuss KZG audits. Although this paper can stand alone, the ideas presented in this paper will be put in proper context if the Orchid Storage lightpaper [1] is read first.

*Author

†Contributor

1.1 Erasure Coding

This paper operates under the assumption that upon upload, Clients *erasure code* their data. Erasure coding is a way to break up data into multiple overlapping pieces, say n pieces, such that for some k (with $k \leq n$), *any* k pieces can be used to reconstruct the data. The following sections don't strictly rely on the use of erasure coding, but it is discussed. For more information about erasure coding and how Orchid uses it for decentralized data storage, see [1], Orchid's storage lightpaper.

2 Performing Audits Using Merkle Trees

In this section we describe how to perform Merkle tree audits both *interactively* (meaning that the Client is online the entire time and able to interact with the storage Provider) and *non-interactively* (meaning that the Client is offline and storage Providers post proofs that are later checked by other storage Providers).

2.1 What is a Merkle Tree?

The definition of a Merkle tree relies on cryptographic hash functions.

Definition 2.1. A *cryptographic hash function* is a map from an arbitrary binary string to a binary string of some fixed length such that:

1. The function is one way. In other words, for any specified output, it is not computationally feasible to find an input that maps to it.

2. The function is collision-resistant. In other words, it is computationally infeasible to find two distinct inputs that have the same output.

The output of a cryptographic hash function is often referred to simply as a "hash", and the function applied to a specific input is often referred to as the hash of that input.

An example of a cryptographic hash function is SHA-256 (SHA stands for "Secure Hash Algorithm"), which outputs a 256 bit hash.

To encode data[1] into a Merkle tree, begin by choosing a number n, then break the data into 2^n *pieces*. That is, thinking of the data as a binary string, we split the data into a sequence of 2^n binary strings. Using the data and this hash function, we build a tree, starting at the bottom level (the leaves) of our tree, which represent these pieces of data and their hashes. We then pair off the leaves and let the parent of a pair of leaves be the hash of the concatenation of this pair. Next, we do the same thing for the parents, pairing them off, and then letting the parent of a pair be the hash of the concatenation of that pair. We repeat this all the way up until we get a single hash at the top. Since the data was broken into 2^n pieces, we are assured that we can pair off pieces and end up with a single hash at the top of our Merkle tree.

Definition 2.2. The *root hash* of a Merkle tree is the hash at the top of the tree.

[1] Although there are many ways to think about data, for this purpose it is best to think of data as a binary string.

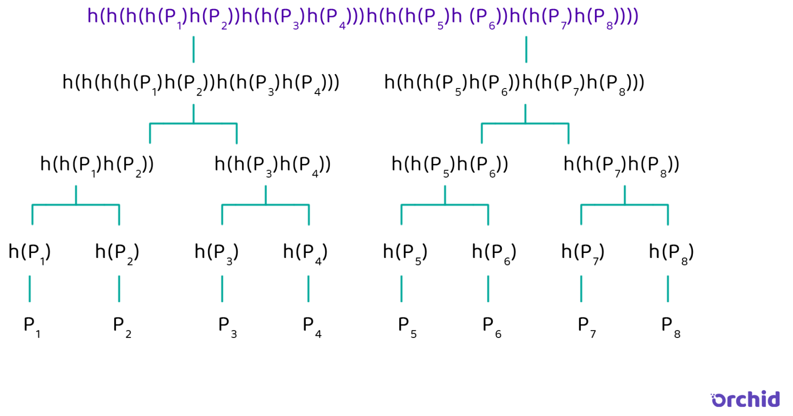

Figure 1: An example of a Merkle tree in which the data consists of 8 pieces, denoted $P_1, \ldots, P_8$, and h denotes our cryptographic hash function. The root hash, at the top of the tree, is in purple.

2.2 Interactive Audits

Once a Client selects a Provider and transmits their encoded data[2], the Client dictates to the Provider how to chunk the data, therefore determining the structure of the Merkle tree. The Provider constructs the tree and then transmits the root to the Client. Meanwhile, the Client constructs the same tree. If the root matches, the Client may delete their data as they only need to retain the root for use in the next stage.

Thereafter, at any time the Client can choose a (random) leaf of the Merkle-encoded erasure block and request that it be sent, along with the necessary metadata from the Merkle tree to prove that that piece was a part of the erasure block that has a matching root. This is called a *Merkle inclusion proof*, and is best described using an example.

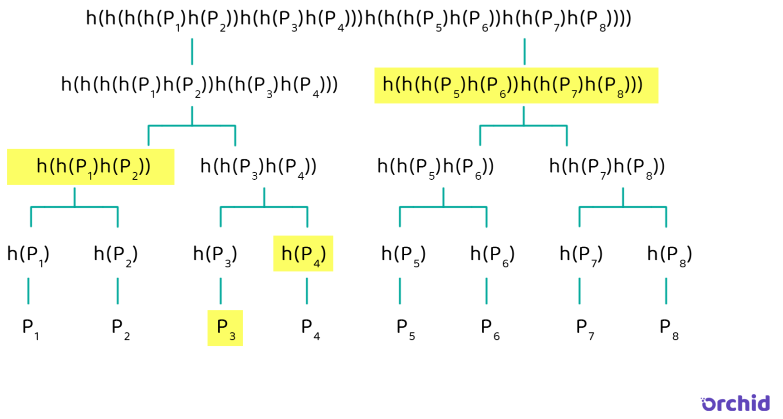

Figure 2: A Merkle tree with the erasure block broken into 8 pieces. If the Client has asked the Provider for a Merkle inclusion proof for piece P_3, then the Provider is expected to send the highlighted data.

The client then checks that

[2] an erasure block

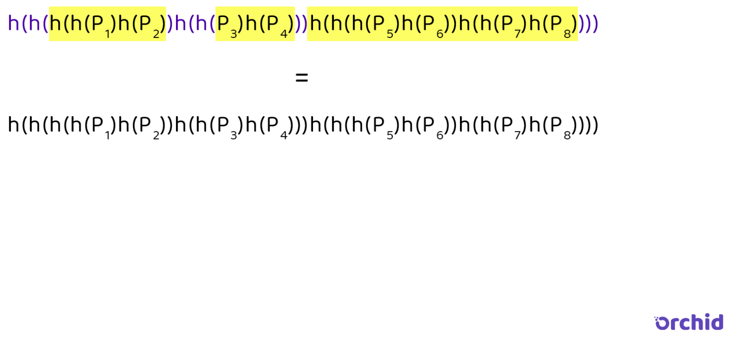
h(h(h(h(P_1)h(P_2))h(h(P_3)h(P_4)))h(h(h(P_5)h(P_6))h(h(P_7)h(P_8))))
=
h(h(h(h(P_1)h(P_2))h(h(P_3)h(P_4)))h(h(h(P_5)h(P_6))h(h(P_7)h(P_8))))

Figure 3: How a Client would check that P_3 is a part of our example Merkle Tree. The purple hashes are ones that the Client computes, and the highlighted data is data that is provided by the Provider. The root hash that the Client checks against is the one they have been storing.

Therefore, with the Client only needing to store the Merkle root, the Provider is able to prove that they have the piece that was requested by showing that it belongs to the erasure block that they are storing.

2.3 Non-Interactive Audits

When the Client is online, they can perform audits of the providers themselves, as described in section 2.2. However, as soon as the Client goes offline, the issue of trust becomes more complicated. Many projects add another actor, often called an Auditor, to perform this task. Adding another party, however, adds a level of complexity to a system with already carefully balanced incentives. Auditors could conspire with storage Providers, conspire with Clients, or simply perform griefing attacks. Since trust remains an issue as the Client does not inherently trust the Auditors, this begs the question: who audits the Auditors?

The idea that there is nothing stopping a storage Provider from also being an Auditor got us thinking: what if instead we start with the assumption that storage Providers are also Auditors.

Once we've abandoned the idea of adding an Auditor, there are two main paths: the Providers can audit each-other, or Providers can self-audit. What we describe in this paper is a mix of these two ideas. Providers are part of a cohort with the other Providers, each holding an erasure block that originates from the same data. Because Providers will already have some information about the other providers in their cohort[3], this enables them to perform self-audits as well as monitor other providers in their cohort for correctness and timeliness of their self-audits.

2.3.1 Fiat-Shamir

We utilize the Fiat-Shamir heuristic, originally described in [4], which is a method for turning an interactive protocol into a non-interactive protocol. The audit itself looks very similar to the interactive audit case, with the main difference being that rather than sending data to the Client, the Provider will post the necessary Merkle tree data on chain at certain pre-specified intervals[4]. These Merkle proofs are then checked by the other providers in the same cohort.

For more information about how Orchid Storage ensures that storage Providers are incentivized to both post proofs themselves and check proofs posted by other storage Providers, see [1].

[3] This is necessary in order to do repairs. For more information on how Orchid Storage approaches repairs, see [1], Orchid's Storage lightpaper.

[4] This could be something like every 10 blocks.

3 Performing (Non-Interactive) Audits Using KZG Commitments

While it is theoretically possible for Providers to perform self-audits using Merkle proofs, posting frequent and large proofs to a blockchain can be very costly. Not to mention that when using the ideas discussed in Section 1, since at every step a piece of the data is posted on-chain, eventually all of the data will be posted on-chain (Then why hire providers in the first place? You might as well have posted your data to a blockchain from the beginning.). KZG commitments reduce both the amount that needs to go on-chain and the frequency with which proofs need to be posted.

We describe these audits in the case where there are two Clients and two Providers. In this example, each Client has one piece of data which is broken into two blocks, and each block is broken into two sub-blocks in a manner which is agreed upon by the Client and the Provider. From there, it should be relatively obvious how to generalize this to more Clients, more Providers, more data, and data broken into more pieces. We provide this simplified explanation in order to save us from having a nasty mess of subscripts.

3.1 KZG Background info

The following sections use KZG commitments[5], which we do not describe in detail, but rather describe how we use them. However, there are some quite well-written explanations out there. We particularly like the explanations in [3] and [6]. The latter gives background for all of the math one might need to understand KZG commitments, including groups, polynomials, and pairings. In particular, we make use of the Section "Kate as a vector commitment" of [3], which uses Lagrange interpolation to create a polynomial that passes through particular points. In the following sections, the commitments described will be to these polynomials (rather than ones where the relevant data is stored as coefficients of the polynomial).

Disclaimer: It is likely that improvements can be made to this scheme for efficiency. If this is something you have ideas for or are excited to talk about, feel free to reach out to the Orchid team!

3.2 Notation

We call our Clients A and B, our Providers α and β. Assume that Client A has one piece of data, d_A, and Client B has one piece of data, d_B. d_A is broken into blocks $b_{A,\alpha}$ and $b_{A,\beta}$ and d_B is broken into blocks $b_{B,\alpha}$ and $b_{B,\beta}$. Provider α stores blocks $b_{A,\alpha}$ and $b_{B,\alpha}$ and provider β stores blocks $b_{A,\beta}$ and $b_{B,\beta}$.

We then further break each block into two sub-blocks. These are the pieces of data that get randomly chosen to be audited. We break $b_{A,\alpha}$ into $s_{A,\alpha,0}$ and $s_{A,\alpha,1}$, $b_{A,\beta}$ into $s_{A,\beta,0}$ and $s_{A,\beta,1}$, $b_{B,\alpha}$ into $s_{B,\alpha,0}$ and $s_{B,\alpha,1}$, and $b_{B,\beta}$ into $s_{B,\beta,0}$ and $s_{B,\beta,1}$.

[5] In the literature you might see KZG commitments being called "Kate commitments", pronounced "kah-tay". These are the same thing. The name KZG comes from Kate, Zaverucha and Goldberg, who originally introduced the concept in [5]

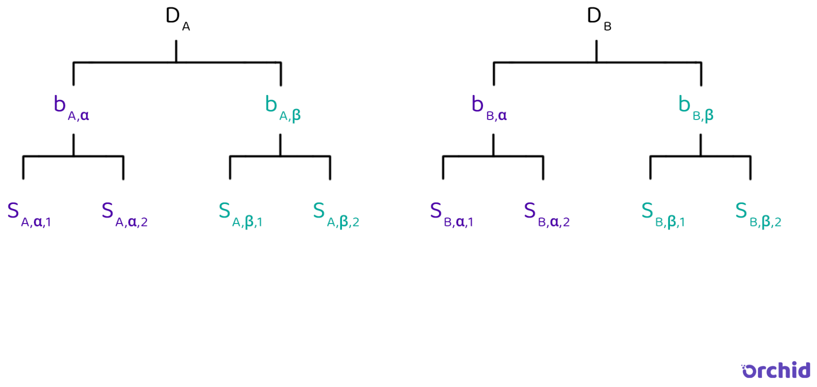

Figure 4: How the data is broken into blocks and sub-blocks. The data held by Provider α is in purple, and the data held by Provider β is in teal.

Let h denote a hash function.

3.3 (Trusted) Setup

Creating a KZG commitment typically involves a trusted third party choosing a random group element and computing public parameters. However, by using Trusted Setups, as outlined in [2], we can eliminate the need for a third party.

3.4 Data Upload

When Client A uploads their data, they will send Provider α the block $b_{A,\alpha}$. Provider α and Client A come to an agreement on how to break this block into sub-blocks $s_{A,\alpha,0}$ and $s_{A,\alpha,1}$. Provider α constructs a polynomial[6] $r_{A,\alpha}$ such that

$$r_{A,\alpha}(0) = h(s_{A,\alpha,0}) \text{ and } r_{A,\alpha}(1) = h(s_{A,\alpha,1}).$$

Provider α then commits to the polynomial $r_{A,\alpha}$ by posting the KZG commitment $C_{A,\alpha}$ on chain. Client A can also compute $C_{A,\alpha}$ to be assured that Provider α is committing to the correct data before potentially deleting the data.

The same thing happens when Client A sends data to Provider β and when Client B sends their data to Providers α and β.

3.5 Self-Auditing/Commitments

We now introduce three functions, f, g, and h, which allow us to utilize the Fiat-Shamir technique using the time as a seed to randomly select a block for auditing. In our case, where there are only two sub-blocks held by a Provider for any given Client, these functions are simply a fancy way of choosing 0 or 1 in a non-anticipatable, deterministic, pseudo-random way.

f takes the current time as input and outputs a non-anticipatable, deterministic, pseudo-random number. This could, for example, be the current block hash on the blockchain that is being used.

g takes as input the Client identification, and outputs the number of sub-blocks that the Provider is holding for that Client.

h takes in the outputs of f and g and returns the output of f mod the output of g. That is, the remainder given when dividing the output of f by the output of g.

[6] Using Lagrange interpolation, as discussed in section 3.1.

Providers post their self-audits at certain intervals. These intervals could, for example, be specified in the Rate Certificate, or could be left up to the Provider to strike a balance between being paid frequently to minimize risk and being paid infrequently to minimize cost. Suppose at time t, Provider α performs a self-audit[7]. Provider α constructs and commits to two polynomials, $q_{\alpha,t}$ and $p_{\alpha,t}$.

$q_{\alpha,t}$ is a polynomial that passes through each of the commitments that Provider α has posted on-chain, one for each data block that they are holding, so by committing to $q_{\alpha,t}$, Provider α is making a claim about what data they are currently holding. In our example, the Polynomial $q_{\alpha,t}$ is constructed such that

$$q_{\alpha,t}(0) = C_{A,\alpha} \text{ and } q_{\alpha,t}(1) = C_{B,\alpha}.$$

Provider α commits to $q_{\alpha,t}$ by posting the KZG commitment $C_{\alpha,q,t}$ on chain. Since the other providers in a cohort can see the commitments to the r polynomial for the cohort they are in, they can later perform the opening and verification phases of the KZG commitment to $q_{\alpha,t}$ at the value of the commitment to the relevant r polynomial. For example, Provider β is in the same cohort as α for data D_A, and Provider β can see that Provider α has posted $C_{A,\alpha}$ on chain. Later, Provider α posts the commitment $C_{\alpha,q,t}$ on chain. Provider β can then request that Provider α "opens" the commitment $C_{\alpha,q,t}$ at the point[8] $C_{A,\alpha}$. Once Provider α sends Provider β the relevant data for the opening, Provider β can perform the verification.

At the same time, Provider α constructs a polynomial $p_{\alpha,t}$, which passes through a randomly[9] chosen sub-block for each block that α is storing. In our example, the Polynomial $p_{\alpha,t}$ is constructed such that

$$p_{\alpha,t}(0) = s_{A,\alpha,h(f(t),g(A))} \text{ and } p_{\alpha,t}(1) = s_{B,\alpha,h(f(t),g(B))}.$$

Provider α commits to $p_{\alpha,t}$ by posting the KZG commitment $C_{\alpha,p,t}$ on chain. Because each sub-block is chosen randomly[10], when the Client later "opens" and "verifies" Provider α's commitment to $p_{\alpha,t}$ and sees that α was storing that sub-block, they can be reasonably convinced that α was storing the entire block.

Note that in the case with more providers, for example if α is in multiple cohorts with β, α can check those relevant commitments all at once (see the Multiproofs section of [3]).

4 Open Questions and Future Directions

In this paper, we present ideas both for interactive audits and non-interactive audits. While having a system that uses non-interactive audits works in both cases: when the Client is online or away, it may be advantageous or more cost-effective (as interactive audits do not require anything to be posted to a blockchain) to be able to switch between interactive and non-interactive audits. However, building this switch into a protocol can be complicated, therefore, this, as well as further efficiency improvements are directions for further research.

5 Acknowledgements

A huge thank you to Dan Montgomery, Patrick Niemeyer, and Dr. Steven Waterhouse for edits, feedback, insights, and advice. And thank you to Chad Harper for making the diagrams.

References

[1] Chloe Avery and Justin Sheek. Orchid Storage: A New Open Source Initiative For Decentralized, Incentivized Data Storage. `https://www.orchid.com/storage-litepaper-latest.pdf`.

[2] Vitalik Buterin. How do trusted setups work? `https://vitalik.eth.limo/general/2022/03/14/trustedsetup.html`.

[7] This works the exact same way when provider β performs a self-audit.

[8] Note that in the literature commitments are usually "opened" at a random point. In this case, Provider β only cares that the polynomial $q_{\alpha,t}$ passes through the point $C_{A,\alpha}$

[9] Here we really mean non-anticipatably, deterministically, and pseudo-randomly

[10] non-anticipatably, deterministically, pseudo-randomly

[3] Dankrad Feist. KZG Polynomial Commitments. `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`.

[4] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[5] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. `https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf`.

[6] Luksgrin. A quick insight on Algebra and KZG Commitments. `https://github.com/luksgrin/opensense-algebra-and-kzg/blob/main/algebra_and_kzg.ipynb`.