swarm

smash-proof

auditable storage for swarm secured by masked audit secret hash

viktor trón, aron fischer, nick johnson

draft version May 2016
ETHERSPHERE ORANGE PAPERS 2

Contents

Abstract

This paper [a] is the result of our research into securing decentralised storage and distribution in the context of implementing an incentive layer for swarm.
Swarm is a distributed storage platform and content distribution service, a native base layer service of the Ethereum web 3 stack. The goal is a peer-to-peer serverless hosting, storage and serving solution that is DDOS-resistant, has zero-downtime, is fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system.
Users of swarm need make sure their content is preserved, even when it is rarely accessed. This kind of ensured archival necessitate integrity audits of documents and data collections without transferring the data itself. We present a family of proofs of custody that has properties ideal to serve these needs and allows flexible trade-offs in terms of compactness, repeatability, third party provability and outsourceability for the various scenarios in which audits are used.

[a] The authors are indebted to Gavin Wood, Vitalik Buterin and Jeffrey Wilcke for their daring vision of the next generation internet and for their original concepts of web 3 and swarm. We thank Vitalik Buterin, Dániel Varga, Anish Mohammed, Martin Becze, Christian Reitwiessner and Dániel A. Nagy for their comments and suggestions. Their names here by no means imply endorsement. All errors are ours.

1 Introduction

Proof of custody [2] is a construct that proves to an auditor that a storer has custody of data without the auditor having it. Such constructs can be used to audit the integrity of remotely stored documents trustlessly. In the context of a distributed file storage system, proof of custody schemes need to consider both storage and network communication overhead and offer flexible trade-offs between security and cost.

In the specific context of swarm, we need to make sure we have a scheme that best adapts to

- the communication protocol used by participating peers: bzz on devp2p
- the storage protocol used: direct content addressed storage
- unit of storage: fixed small-sized chunk
- typical use-cases: long term insured storage of infrequently accessed collections of data
- existing efficient contracting/payment protocols: pairwise accounting
- existing escalation to retributive measures: litigation on the ethereum blockchain

The proof of custody scheme proposed here draws on earlier work based on well understood basic cryptography (*[4]*, *[1]*), but in light of the above considerations a novel approach was warranted (inspired by *[6]* and *[2]*).

2 Auditing chunks with pregenerated secrets

In this section we introduce the concept of audit by challenge and formally define the SMASH proof of custody scheme with special attention to its application to the small fixed-size chunks used as basic units of swarm.

2.1 Audit by challenge and response

Auditing a particular chunk c of data is done via a *challenge*. Who initiates this audit and how the challenge reaches the storer of the chunk is discussed in detail later.

To generate a challenge the original *owner* only needs to reveal a *seed*. Once the storer learns the seed for the challenge, they can use this seed to find the corresponding *secret* by a procedure defined below which requires them to have the data. The secret itself is the response to the challenge in its simplest form.

If the secret is not prematurely revealed by the original owner, the storers must have calculated it themselves, since it cannot be guessed and is cryptographically secure against brute forcing if the set of seeds is large enough. As the calculation relies on the data, storers are incentivised to keep the chunk in full. Once a secret is verified as valid, it is fair to assume that the swarm had the file at some point. Moreover, as the seed had never been revealed until the challenge was issued nor was the seed guessable in advance, and since the seed was also used in the calculation, we know that storers still have the chunk currently, ie., at least they had it when the seed was revealed.

[2] In place of custody, the terms existence, storage, resource, retrievability, integrity are often used in the literature. We see no semantic importance as to this choice of wording, but for consistency, will use custody.

If the storer keeps the data they will always know that they will always be able to calculate a correct response. If a storer chooses not to preserve the data, it is impossible to be 100% sure that they give the right secret to a challange. Given these properties it is valid to say that *responding to a challenge with a valid secret can be considered a reliable positive proof of current custody.*

Whoever initiates an audit need not know the secret only be able to check that it is correct. This is achieved by *masking* the secret by hashing it and make the mask public (*[6]*). Thus the original owner publishes a hash of the secret and any third party can verify that the secret the storer provides in response to a challenge has the correct hash. Since unhashing is cryptogaphically impossible, verifying that the secret hashes to the mask is equivalent to checking the secret itself. If audits are to be repeated, several secrets and their corresponding masks need to be pregenerated by the owner. However not all masks need to be remembered in order for any third party to verify that the secret provided is correct. Instead the pregenerated masks can be organised in a *Merkle tree* (*[3]*) and the correctness of a mask can be proven by a *Merkle proof* of the mask as long as the root hash of the Merkle tree is known and trusted. In other words the owner can pregenerate any number of audits secrets and outsource the storage of the mask to the storer allowing storer-side proofs of the secrets' validity.

In the remainder of this section we formalise this approach. In the context of swarm and the discussion in this paper we use the following terminology:

owner node that produces/originates content by sending a store request

storer node that accepted a store request and stores the content

guardian the first node to accept a store request of a chunk

custodian node that has no online peer that is closer to a chunk address

auditor node that initiates an audit by sending an audit request

insurer node that is commissioned by an owner to launch audit requests on their behalf

2.2 Calculating the audit secret

The simplest non-reversible way to derive a secret from a seed and a chunk is to hash the entire chunk with the seed prepended. Assume third parties have a way to verify that the secret given by the challengee is correct and conclude that the storer has custody of the data. But what do they conclude if the owner and storer disagree on the secret? In this case, an explicit proof is needed to show that the seed and the data derive a secret (not) matching the mask.

The relevant insight here is that we pick a Merkle proof of the data chunk (c) based on the seed (s) and an index (j) specifying a particlar segment of the chunk, and manipulate only that to result in the *audit secret hash* ($\texttt{ASH}(c, s, j)$). By doing this we allow explicit proofs whose length is logarithmic in the chunk size.

The only possible scenario when the proof is not conclusive under this simple version is if a storer node had previously responded to a specific seed, stored the response and discarded the data. In this case if an auditor challenges the same chunk with the same index and seed, the storer can respond correctly even though they no longer have the data stored. On the other hand, if the indexes are not recycled, storers can be absolutely sure they can get rid of those parts of a chunk that the already-used indices referred to. To mitigate this, we propose that segment index for an audit is derived from a fixed slice of bits of the seed itself (essentially random bits), so indexes will be recycled during successive audits. Given the seed s and the number of segments

in the chunk $2^d = n$ we propose that the index can be deduced from the seed as

$$j \equiv s \mod 2^d$$

In other words, the last d bits of the seed map to j.

So given a chunk c, a seed s, we construct the secret the following way.

1. First we make sure all chunks have lengths that are powers of 2 padding shorter chunks as necessary. If chunk $c^{\ll}$ is shorter than the predefined maximum chunk size ($\mathit{MaxChunkSize} = 2^m$) then we append to it some padding to make the length of resulting data blob ($\mathit{pad}(c^{\ll})$) the smallest power of 2 m' such that $2^{m'} > n$. In particular appending hashes until the length exceeds the first power of two and then finally we truncate [3] .

$$\mathit{pad}(c, s, i) \stackrel{\text{def}}{=} \begin{cases} c, & \text{if } i = 0 \\ \mathit{pad}(c, s, i-1) \parallel \mathcal{H}(\mathit{pad}(c, s, i-1) \parallel s), & \text{otherwise} \end{cases}$$

Then we define the padded chunk as

$$\mathit{pad}(c^{\ll}) \stackrel{\text{def}}{=} \mathit{pad}(c^{\ll}, s, i)[0 : 2^{m'}]$$

where i is chosen as the smallest index such that

$$\mathit{len}(c^{\ll}) + i \cdot \texttt{sizeof}(\mathcal{H}) >= m'$$

With this padding process defined, we will from now on assume that all chunks have a length that is a power of 2, therefore the next step is well defined.

2. Chop the chunk into hash sized segments. Assume for convenience that hash size is a power of two: $\texttt{sizeof}(\mathcal{H}) = 2^h$ and $h < m$, then c is a concatenation of n segments (for padded shorter chunks $m < \mathit{MaxChunkSize}$):

$$c = \sigma_0 \parallel \sigma_1 \parallel \ldots \parallel \sigma_{n-1} \text{ where}$$

$$n = 2^{m-h}$$

$$\mathit{len}(\sigma_i) = 2^h \text{ for } 0 \leq i < n$$

We introduce the following notation to project a chunk to its j^{th} segment. This allows us to view a chunk of data as a segment array.

$$c\langle j \rangle \stackrel{\text{def}}{=} \sigma_j$$

$$c\langle j : k \rangle \stackrel{\text{def}}{=} \sigma_j \parallel \sigma_{j+1} \parallel \ldots \parallel \sigma_k$$

3. Now calculate the modified version of the data. Take the j^{th} segment of the chunk and replace it with a modified segment that is the original segment hashed with the seed appended:

[3] $\parallel$ stands for concatenation, and the notation $x[i : j]$ stands for the byteslice $x[i] \parallel x[1] \parallel \cdots \parallel x[j-1]$ where $x[i]$ is the i^{th} byte of x. $\mathcal{H}$ stands for a hash function of choice. To help readability, the variable c always stands for a chunk of data, σ for a segment of a chunk, λ for levels of Merkle trees, s for seed.

$$\Delta(c, s) \stackrel{\text{def}}{=} c\langle 0 : j-1\rangle \parallel \mathcal{H}(\sigma_j \parallel s) \parallel c\langle j+1 : n-1\rangle$$

where

$$j = s \mod 2^d$$

4. Build up a binary Merkle tree over the segments. Since the number of segments is a power of 2, the resulting tree is regular and balanced. Calculate the Merkle root of this Merkle tree to arrive at the audit secret.

We define the tree in this way to ensure that calculating the audit secret hash requires you to have the chunk itself and also that malicious users cannot cheat the audit by precalculating the tree and forgetting the chunk.

Let us now fix notation for the hashes in a generic regular binary Merkle tree. Leaf nodes are at $\lambda = 0$, non-leaf nodes at $\lambda \geq 1$.

$$\mathcal{M}_{2,\mathcal{H}}(c, \lambda, i) \stackrel{\text{def}}{=} \begin{cases} \mathcal{H}(c\langle i\rangle), & \text{if } \lambda = 0 \\ \mathcal{H}(\mathcal{M}_{2,\mathcal{H}}(c, \lambda-1, 2\cdot i) \parallel \mathcal{M}_{2,\mathcal{H}}(c, \lambda-1, 2\cdot i+1)), & \text{otherwise} \end{cases}$$

and we denote the Merkle root of the chunk as

$$\Re(c) = \mathcal{M}_{2,\mathcal{H}}(c, d, 0)$$

We can define the audit secret hash as the Merkle root of the chunk with the modified segment

$$\texttt{ASH}(c, s) \stackrel{\text{def}}{=} \Re(\Delta(c, s))$$

As the other segments $(\sigma_i; i \neq j)$ did not change, if one knows the Merkle proof belonging to segment j of the original chunk then, given the seed, the modified Merkle proof can simply be recalculated in exactly d steps. This essentially means that the number of steps in the proof of correctness is logarithmic in the chunk length.

Let us examine this Merkle proof in detail. We begin by introducing notation for the successive nodes of the Merkle proof for the j^{th} segment of a chunk (see also figure 2):

$$\mathcal{CH}_\lambda(c, j) \stackrel{\text{def}}{=} \mathcal{M}_{2,\mathcal{H}}(c, \lambda, I_C(\lambda, j))$$

$$\mathcal{PH}_\lambda(c, j) \stackrel{\text{def}}{=} \mathcal{M}_{2,\mathcal{H}}(c, \lambda, I_P(\lambda, j))$$

where

$$I_C(\lambda, j) = \frac{j - (j \mod 2^\lambda)}{2^{\lambda+1}}$$

$$I_P(\lambda, j) = \begin{cases} I_C(\lambda, j) + 1, & \text{if } I_C(\lambda, j) \mod 2 = 0 \\ I_C(\lambda, j) - 1, & \text{otherwise} \end{cases}$$

Since $I_C(\lambda, j) \mod 2$ is the λ-th least significant bit in the binary representation of j, the index's bits inform us which order $\mathcal{CH}$ and $\mathcal{PH}$ are concatenated to yield the hash of the next level. Define the directional hash function:

$$\mathcal{H}^{\equiv}_d(x, y, j, \lambda) \stackrel{\text{def}}{=} \begin{cases} \mathcal{H}(x \parallel y), & \text{if the } (d-\lambda)\text{-th bit of } j \text{ is } 0 \\ \mathcal{H}(y \parallel x), & \text{otherwise} \end{cases}$$

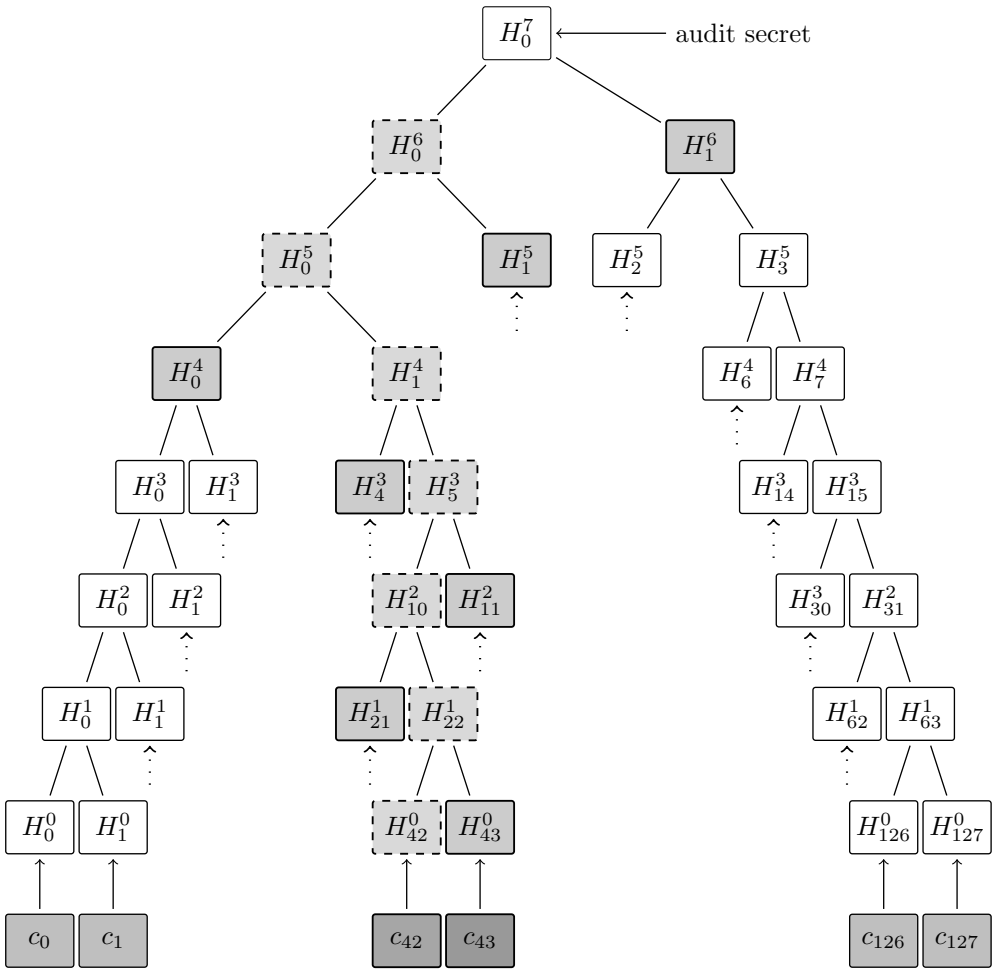

Fig. 1: The figure represents the Merkle tree for a chunk ($H_i^\lambda \stackrel{\text{def}}{=} \mathcal{M}_{2,\mathcal{H}}(c, \lambda, i)$). Shaded in grey in the middle is the Merkle proof for index 42 (7-bit binary represetation is `0011010`). The proof can be verified knowing only the data segments at the given index $j = 42$ and its sister segment (next segment if index is even, previous if odd), plus sister hashes at each level as indicated.

Now, given $j, c\langle j\rangle$ and $\mathcal{PH}_0, \ldots, \mathcal{PH}_{d-1}$, we can calculate $\mathcal{CH}_0, \mathcal{CH}_1, \ldots, \mathcal{CH}_d$

$$\mathcal{CH}_\lambda(c, j) = \begin{cases} \mathcal{H}(c\langle j\rangle), & \text{if } \lambda = 0 \\ \mathcal{H}_d^{\equiv}(\mathcal{CH}_{\lambda-1}, \mathcal{PH}_{\lambda-1}, \lambda - 1), & \text{otherwise} \end{cases}$$

Given a Merkle proof then, both the chunk hash and the audit hash can be verified. For the latter the auditor simply plugs in the salted segment (segment j hashed together with the seed) and calculates the audit secret hash as the root using the same side hashes as the original proof (Figure 2).

$$\mathcal{AH}_\lambda(c, s) = \begin{cases} \mathcal{H}(c\langle s \mod 2^d\rangle \parallel s), & \text{if } \lambda = 0 \\ \mathcal{H}_d^{\equiv}(\mathcal{CH}_{\lambda-1}, \mathcal{PH}_{\lambda-1}, \lambda - 1), & \text{otherwise} \end{cases}$$

and

$$\texttt{ASH}(c, s) = \mathcal{AH}_d(c, s)$$

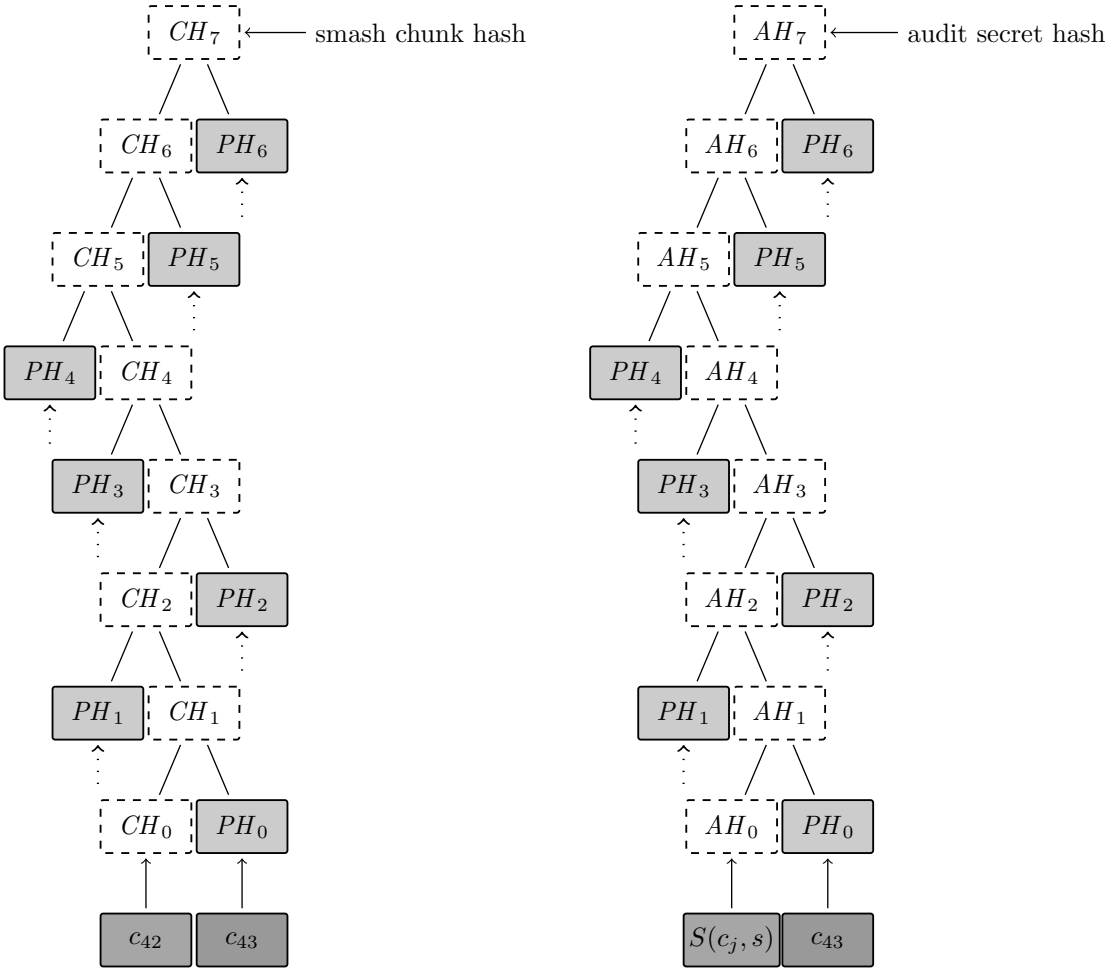

Fig. 2: Given a chunk hash, a seed, and the index, the audit secret hash for $\texttt{ASH}(c, s, j)$ can be calculated and verified using only the Merkle proof for the segment at the index. The left hand side is the Merkle structure of the original segmented chunk, the right hand side represents the corresponding Merkle proof for the audit secret.

If an auditor maliciously published a false ASH, then a storer would find that the ASH they calculated does not match the published one. In this case it is important that the storer can

prove that they are innocent - that it is the published ASH that is fraudulent. The Merkle proof of the segmented chunk (Figure 2) proves that they really are storing the chunk and the corresponding Merkle proof proves that the ASH they calculated is the correct one.

2.3 Masked audit secret hash (MASH) tree

Now we turn to the formal definition of the masked audit secret hash tree. This is relevant for repeatable audits without remembering the secrets themselves. The basic idea is to store all the masked secrets in a Merkle tree (MASH tree) and to remember only the root of this tree (MASH root). A successful response to a challenge contains not just the secret, but also the Merkle proof from the secret to the MASH root.

Assume that we have $k = 2^r$ audit seeds $s_0, \ldots s_{k-1}$ specific to a chunk. Each audit seed allows nodes to launch an independent challenge to the swarm and check that the associated data is preserved. We define r as the *repeatability order* of the audit. Using the audit seeds and the chunk one can construct a *masked audit secret hash* tree (*MASH tree*) as follows:

1. Given a chunk and the n audit seeds, calculate the corresponding audit secrets.
2. Given the n audit secrets, construct n masked audit secrets by taking their hash (MASH).

$$\texttt{MASH}(c, i) = \mathcal{H}(\texttt{ASH}(c, s_i)) \text{ for } 0 \leq i < k$$

3. All of these masked secrets need to be stored by storers in order to prove either the correctness of their secret or incorrectness of some seed. So take the masked secrets in the order of indexes and build the binary Merkle tree of the pieces. The root of this Merkle tree is the MASH root.

$$\Re[\texttt{MASH}](c) = \Re(\texttt{MASH}(c, 0) \parallel \texttt{MASH}(c, 1) \parallel \ldots \parallel \texttt{MASH}(c, k-1))$$

4. Only the MASH root needs to be remembered by the owner and it should always be referenced as part of the challenge.

The *MASH proof* for a particular seed can be verified by only knowing the root mask at the given index and the sister hashes at each level of the proof. The process is entirely analogous to the case of the smash chunk hash.

We assume that the length of the MASH proof $\mathcal{P}[\texttt{MASH}]$ is $l = len(\mathcal{P}[\texttt{MASH}])$ and the MASH index i of the masked secret is given (or derived from the seed, see below).

1. If $l \mod 32 \geq 0$, reject the proof.
2. Set the repeatability parameter $r = l/32$
3. Using the directional hash function $\mathcal{H}^{\equiv}_d(x, y, i)$, the storer's secret can now be calculated using the following recursive definition

$$\mathcal{MH}_\lambda(c, s) = \begin{cases} \mathcal{H}(\texttt{ASH}(c, s)), & \text{if } \lambda = 0 \\ \mathcal{H}^{\equiv}_d(\mathcal{CH}_{\lambda-1}, \mathcal{PH}_{\lambda-1}, \lambda - 1), & \text{otherwise} \end{cases}$$

and

$$\texttt{MASH}(c, i) = \mathcal{MH}_r(c, s)$$

Now if $\texttt{MASH}(c, i) = \mathcal{MH}_r(c, s)$ the MASH proof is valid and one can conclude with certainty that the file is stored in the swarm.

2.4 Responding to a challenge

In the simplest form, the response to the challenge is the secret itself (ASH). Storers are also able to prove that the secret is correct if they know the mask securing the chunk: if the hash of the secret matches the mask in the i^{th} position in the MASH tree, answering a challenge consists of the MASH proof of the i^{th} mask. This is the positive response assuring the integrity of storage of the chunk. Hence the motto: SMASH-proof = *Secured by Masked Audit Secret Hash* proof. We can say the chunk is *smash-proof*.

If on the other hand the hash of the secret does not match the mask at the relevant index, then the storer can give the Merkle proof of the relevant segment of the original chunk. This response is called a *smash proof*, and we can say the (faulty) audit challenge has been smashed by the storer.

Given the usual 256bit Keccak SHA3, $\texttt{sizeof}(\mathcal{H}) = 32$ used in swarm, MASH proof itself is exactly $32(r+1)$ bytes long. For instance if $r = 3$, the proof with the secret takes a mere 128 bytes. A swarm chunk is $4096 = 2^7 \cdot 32$ bytes, so the complete ASH proof of a swarm chunk is $8 \cdot 32 = 256$ bytes.

In the latter case when the challenge is smashed, the smash proof is a little longer since it also involves giving Merkle proofs of segments of the original chunk. In this scenario, the storer calculated the secret from the given seed s and found that it does *not* match the audit mask. The storer then submits a Merkle proof, proving the existence and position of the respective segments in the original chunk and, coincidentally, proving the audit mask faulty. This form of proof can be also used if the auditor wants to make sure the secret is correctly derived from the seed while not knowing the secret or its mask. This will be used as second pass challenge after failed partial verification of a secret which is not 100% conclusive. To clarify: if a storer submits a secret whose hash does not match the audit mask then either the storer submitted a false secret, or the audit mask is wrong. By submitting the storage proof directly the storer can prove that it is the audit mask that was faulty. This proof is also used in conjunction with the MASH proof to prove to a third party that a challenge was invalid. This type is expected to be used very rarely, since the only way they come about is if auditors are sending frivolous false seeds or are publishing incorrect masks, which they are disincentivised to do.

challenge	**input**	**storer knows**	**response**
Merkle proof	smash chunk hash, segment index		Merkle proof
ASH	smash chunk hash, seed		audit secret hash
ASH proof			chunk segment, ASH proof
MASH proof	smash chunk hash, seed, MASH root	mask ok	audit secret hash, MASH proof
		mask not ok	ASH proof, MASH proof

Fig. 3: Challenges and responses: types of challenges, their input and the response storers can give. The first three types of challenge make no claim as to whether the auditor knows the secret. The MASH proof challenge presupposes the storer knows the mask. The storer always responds with the MASH proof, if they find that the mask matches they also include the audit secret hash in their response, otherwise they submit the relevant Merkle proof (from which the ASH can be derived).

3 Repeatability and file-level audits

In this section we expose the problem of scalability that comes with repeated audits of fixed sized chunks. We show that the solution lies in finding larger structures than the chunk which are to be audited directly, essentially auditing many chunks simultaneously. We do this in a way that storage critical audit masks can be reused without compromising security. Incidentally, this same method offers a systemic and rather intuitive way of auditing documents and document collections (the units that are semantic to the users). We propose an algorithm to recursively generate seeds for the successive chunks of a larger collection and provide a partial secret verification scheme that offers error detection and efficient backtracking to identify missing chunks. This *collection-level recursive audit secret hash* (*CRASH*) will provide the basis for collective iterative auditing, an efficient automated integrity protection system for the swarm.

3.1 The problem of scaling audit repeatability with fixed chunks

The choice of the repeatability parameter r has an impact on the length of the Merkle proofs which are needed for MASH proofs. More importantly, though, since someone needs to remember the masks, this scheme has a fixed absolute storage overhead that is independent of the size of the pieces we are proving the storage of. Since it is not realistic to require more than 5-10% administrative storage overhead even for very long storage periods, larger r values only scale if the same seeds can guard the integrity of larger data.

In particular, take the example of a standard swarm chunk size of 4096 bytes ($m = 12$) and assuming standard Keccak 256bit Sha3 hash we have $h = 5, d = 7$. Given the MASH base length of 2^{r+h}, 128 independent audits incurs a 100% storage overhead. Instead for a chunk $r = 0, 1, 2, 3, 4$ seem realistic choices, yielding a storage overhead of $0.8, 1.6, 3.125, 6.3, 12.5\%$ respectively.

Ultimately, repeatability order should reflect the *storage period* (TTL = time to live) of the request, therefore *audit repeatability and fixed chunk-size cannot scale unless we compensate for the overhead by reusing seeds over several chunks.* This problem does not occur with Storj since the shards can be sufficiently big, however with swarm, the base unit of contracting is the chunk. The insight here is that we can reuse the same seed over several chunks if and only if we query the integrity of those chunks at the same time.

Users will probably want to check the integrity of their assets on semantic units like document or document collection, i.e., a solution should be in place to make sure litigation and auditing are easily managed for these units. Incidentally, collection-level recursive audit secret hash solves both problems at one go. This is the topic of the following section.

3.2 Collection-level recursive audit secret hash

In this subsection we define the audit secret hash for collections, i.e., an algorithm to calculate an audit secret hash from a document collection using only a single audit seed. First we define a strict ordering on all chunks in a document collection as follows:

1. Take the manifest describing the document collection and walk through the paths in the order defined by the manifest trie (lexicographic) and define M as the function mapping paths to swarm hashes of the documents they route.

$$M : \mathcal{P} \mapsto \text{Range}(\mathcal{SH})$$

2. Let $\Pi(M) \subseteq \mathrm{Dom}(M)$ be the set of unique paths in the manifest such that if several paths point to the same document take the first one in the order.

$$\pi \in \Pi(M) \overset{\text{def}}{\Leftrightarrow} \pi \in \mathrm{Dom}(M) \text{ and } \nexists \pi' \text{ such that } M(\pi) = M(\pi') \text{ and } \pi' < \pi$$

This defines a unique set of documents and a strict ordering over documents.

For each document, take the chunk tree of a document as defined by the swarm hash chunker. See figure 4.

1. Let $\triangle(\nu)$ be the set of all nodes in the subtree encoded in ν. Now define a strict ordering of nodes in the chunk tree for document d.

$$\nu <_d \nu' \overset{\text{def}}{\Leftrightarrow} \begin{cases} \nu \in \triangle(\nu'), & \text{or} \\ \exists \nu_n, \nu_m, i, j, \text{and } \nu_t & \text{such that} \\ \mathcal{H}(\nu_n) = \nu_t\langle i \rangle & \text{and} \\ \mathcal{H}(\nu_m) = \nu_t\langle j \rangle & \text{and} \\ i < j \end{cases}$$

2. Combine this ordering of nodes and the ordering of unique paths in the manifest, extend the ordering of nodes over the entire document collection as follows:

$$\nu <_M \nu' \overset{\text{def}}{\Leftrightarrow} \begin{cases} \nu <_d \nu', & \text{if } \exists d \text{ such that } \nu, \nu' \in \triangle(d) \text{ or} \\ d <_M d', & \text{if } \exists d, d' \text{ such that } \nu \in \triangle(d) \text{ and } \nu' \in \triangle(d') \end{cases}$$

3. Now define the set of unique nodes $\mathcal{C}(M)$ of the document collection.

$$\nu \in \mathcal{C}(M) \overset{\text{def}}{\Leftrightarrow} \nexists \nu' \text{ such that } \mathcal{SH}(\nu) = \mathcal{SH}(\nu') \text{ and } \nu' <_M \nu$$

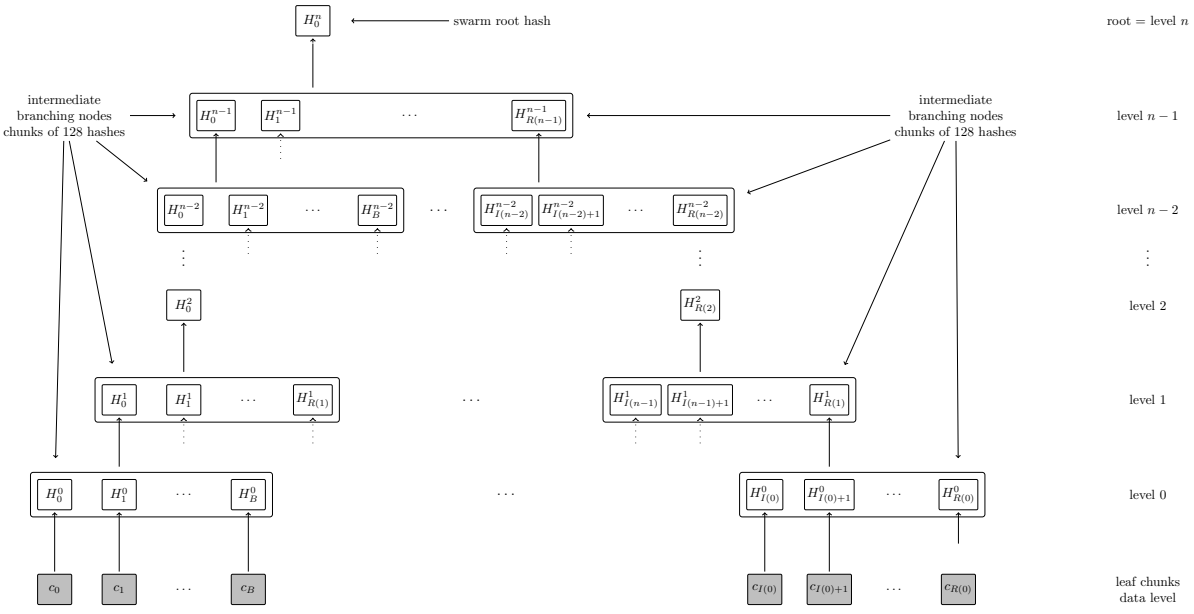

Fig. 4: The swarm hash construct. Hierarchical chunking.

The resulting ordered set of chunks will be used to define the collection-level recursive audit secret hash.

1. Let M be the manifest of a document collection and $\mathcal{C}(M) = \{c_0, c_1, \ldots c_n\}$ be the set of unique chunks such that $c_i < c_j$ for all $0 \leq i < j \leq n$. The last chunk c_n is the root chunk of the manifest.
2. Let s be the seed for M.
3. Define the CRASH function for M at index i as

$$\texttt{CRASH}(M, s, i) \stackrel{\text{def}}{=} \begin{cases} \texttt{ASH}(c_0, s), & \text{if } i = 0 \\ \texttt{ASH}(c_i, \mathcal{H}(\texttt{CRASH}(M, s, i-1) \parallel s)), & \text{otherwise} \end{cases}$$

4. The collection-level recursive audit secret hash for M is defined as

$$\texttt{CRASH}(M, s) \stackrel{\text{def}}{=} \texttt{CRASH}(M, s, n)$$

In practice given a collection the owner wants to store, the secrets can be efficiently generated at the time the files are chunked. As the chunks are uploaded, and guardian addresses and their receipts are stored in a structure parallel to the chunk tree anyway.

This pattern can be applied to document collections covering entire sites or filesystem directories and therefore scales very well. Given the swarm parameters of $m = 12, h = 5$, for a TTL requiring repeatability order r (for 2^r independent audits without ever seeing the files again), the minimum data size to achieve a desired maximum storage overhead ratio k is $k \cdot 2^{r+5}$ Setting *r=128*, so the masks fit into one chunk, a 20-chunk file (80KB) would allow 128 independent audits with a 5% storage overhead [4] .

This audit will not reveal the actual secret to the individual storers of chunks, therefore it can never be used to prove to third parties that a fix limit challenge is invalid. For the same reason it is not used for public litigation.

If we know nothing about the individual secrets used in the recursive formula, and we use ASH challenges to obtain $\texttt{CRASH}(M, s, i)$, the correctness of the secret is only verifiable after we calculate the final $\texttt{CRASH}(M, s)$ and check it against the mask. If it does not match, we have no way of identifying at what index the error occurred. Requiring ASH proofs directly at every index, on the other hand, would incur an order of magnitude more network traffic. However, a reasonable middle ground is possible.

The insight here is that we can use partial verification on the individual secrets. When auditing, every time a new ASH secret is given, ε bits of the secret are checked. If a mismatch is encountered, the audit enters into a second pass backtrack mode and actual Merkle proofs are obtained from the nodes.

Note that the audit secret hash from one chunk determines the seed for the next chunk's audit. Since an incorrect secret yields a new random seed and thus a new subsequent secret, and since secrets thus obtained have a uniform distribution, newly introduced errors can generate false positives on average 1 in 2^{ε} times. As a result, the probability that any error remains undetected for n steps is less than $2^{-n \cdot \varepsilon}$. This property makes it efficient to follow a simple backtracking

[4] If the audit frequencies are dependent of the TTL of chunks only, this scheme will provide a guarantees about the proportion of chunks that are expected lost over any given period of time. With the chunksize (hash and branching parameter) fixed, linear increase of resources provide exponentially more reliability on the individual chunks. If the requirement is full integrity of an entire collection, the frequency of audits need to be adjusted to make up for the (likewise) exponential loss of reliability in the number of chunk tree nodes (i.e., size of the collection). This loss is the consequence of fixing the degree of redundancy as well as the size of the unit that encodes it (see *[5]*). We are curerntly looking at the intricate interplay between integrity audits and redundant encoding and how it can be used in a *divide and conquer* approach to the scaling of erasure codes.

strategy: when a mismatch is encountered on $\texttt{CRASH}(M, s, i)$, proceed by requiring Merkle proofs for past chunks in order of their recency, i.e., $c_i, c_{i-1}, c_{i-2}, \ldots$.

This is all based on the premise that the bits the errors are checked against are precalculated and stored. This creates an extra overhead of ε bits per chunk, modifying our minimum datasize requisite to $k \cdot (2^{r+\varepsilon+8})$ bits.

The exact procedure covering auditing and litigation is detailed in the following section.

3.3 Generating the seeds

Optimising the storage for owners to originate audits it is important that a series of seeds should be deterministic so the seed can be calculated when an audit is initiated.

1. Every node has a *master seed* (*MasterSeed*) that is derived from a ethereum seed account *acc* protected by a password. This master seed is never shown or cached, it only exists in memory.

$$MasterSeed = \mathcal{H}(\texttt{privkey}(acc) \parallel \texttt{address}(acc))$$

2. Using the chunk hash, one can generate the series of base seeds for a chunk.

$$BaseSeed(c, 0) = \mathcal{H}(MasterSeed \parallel \mathcal{CH}(c))$$
$$BaseSeed(c, i) = \mathcal{H}(BaseSeed(c, i-1) \parallel BaseSeed(0))$$

3. The i^{th} transparently indexed seed ($TIS(c, i)$) is obtained by replacing the first r bits of the base seed with the index.

$$TIS(c, r, i) = i \cdots 2^{2^h - r} + (BaseSeed(c, i) \mod 2^{2^h - r})$$

These transparently indexed seeds are used to generate masks to submit together with the store request for a chunk. For entire collections, we use the transparently indexed seeds of the root chunk of the collection manifest [5] .

This indexing scheme allows owners to generate a seed needed for an audit for any chunk without having any information whatsoever. In order to generate a seed in range though, they need to know the repeatability order of a chunk. We will most likely assume that r is the logarithm of the TTL of an insured chunk [6] .

Incidentally, this allows the owner to calculate the index of the previous seed used for the collection from the current time and time of the receipt, so repeated audits with the same seed can be avoided without the need to keep a cursor. Non-automated audits on chunks are expected to occur infrequently and since they count as anomalies, they are likely to be recorded for reasons of reputation etc.

[5] It is rather unlikely that we ever need so high r values that the security of the secret against bruteforcing is compromised.

[6] The base of this log would set the clock tick for automated audits, making it a system constant will allow predictable audit traffic estimates given the size of the swarm.

4 CRASH-proof auditing and litigation

In this section we define an incentive compatible auditing and litigation process that is encoded in the swarm protocol [7] . It relies on CRASH/SMASH challenges for proof of custody for integrity checking which also serve as evidence sent to the blockchain when initiating public litigation.

4.1 Prerequisites for insured storage

Suppose an owner of a chunk wishes to have it stored and insured. The owner communicates directly with a registered peer who will act as *guardian* of this insured chunk. When a store request for an insured chunk is sent from the owner to the guardian, the owner must include the smash chunk hash [8] , as well as the MASH root and sign it together with the swarm hash of the chunk. The smash chunk hash is needed to verify Merkle proofs, while the MASH root is needed to verify MASH proofs. Both are needed in order to provide negative proofs against an auditor sending frivolous audit requests.

When the store request is accepted by the guardian, they provide the owner with a receipt consisting of the store request signed by the author and counter-signed by the guardian. We use a court-case like system of public litigation on the blockchain, so the signatures are important in order for smart contracts to verify if a challenge is valid.

After the owner generates the MASH tree, calculated and remembered all verification bits and uniqueness bits, they have two options. One is to remember the data and store it along with the chunk hash. This allows them to launch and verify simple audit requests which are responded to by the relevant audit secret hash (ASH) value, and check that the hash of the ASH matches the entry in the MASH tree. The other option is not to store the MASH tree, but only to remember the MASH root. They would send off the masked audit secret hashes along with the store request. This enables owners to obtain proofs of custody without having any parts of the data whatsoever beyond the chunk hash, the MASH root and the signature of the receipt.

Even though querying a particular chunk is allowed and can be done manually, the automated audit and litigation process launches audits on document collections and/or files instead.

4.2 Document- or collection-level auditing and litigation

It is expected that auditing should happen not at the chunk-by-chunk level, but at a file or file-collection level that is semantic for the end users. The basic process for this is the following.

- The owner identifies a batch of chunks (document or collection of documents that contains files to be retrieved at similar very low frequencies and stored for the same period) to store.

[7] SWINDLE (Secured With INsurance Deposit Litigation and Escrow) is the part of the bzz protocol that handles the logic and communications relating to auditing and litigation as well as the corresponding smart contract on the blockchain that handles the particular court procedure.

[8] At the time of writing the "swarm hash" used to identify a chunk in the swarm is simply its hash, while the "smash chunk hash" from the Merkle proofs is the Merkle root of a binary tree that treats the chunk as n segments of size 2^h (in our case 128 segments of 32 bytes). The question arises why we do not combine these two. In particular, we could simply use the smash chunk hash (the root of the binary Merkle tree over 32 byte sequences) instead of the simple swarm hash in the swarm chunker. This would have the benefit that smash chunk hashes would not need to be stored separately as part of the audit metadata. However, the smash chunk hash involves 255 hashing operations as opposed to the single one of the swarm hash, therefore, extensive benchmarks are needed before we opt in for this option.

The owner submits store requests for each chunk and collects receipts from the respective guardians.

- The owner stores all the guardians' receipts in a parallel structure.
- The owner generates the base seeds to be used for auditing all the files listed in the manifest and then precalculates the secrets. The owner masks the audit secret hashes by hashing them and proceeds to build the MASH tree [9] .
- Along the way, the owner records the partial verification bits for each intermediate CRASH secret.
- The owner calculates all the smash chunk hashes belonging to the chunks and records them in a parallel structure.
- Finally, the owner records a *uniqueness bit* (a boolean flag) for each chunk. Since it it possible that the same chunk appears multiple times in a document collection, and since we want to avoid uneccessary repeated audits for such chunks, we must store one extra bit of information - this is the uniqueness bit belonging to each chunk in the collection.
- Once all these have been assembled, the owner can put them in a manifest.

Let us assume that all chunks have been stored and the owner obtained a receipt for each from the respective guardians. Once a document collection is assembled, the manifest describing the collection is created. This *collection audit manifest* will contain all the metadata needed for auditing and litigation, notably:

1. the guardian receipts of all the unique chunks,
2. the smash chunk hashes of all the unique chunks,
3. the uniqueness bits of all the chunk tree nodes,
4. the partial verification bits (the last two bits of the expected intermediate secrets) and
5. the MASH-es.

This audit manifest is a special structure that is sold to insurers who are obligated to store the metadata and be prepared to receive seeds from the owner at any time to initiate audit requests on the owners behalf. Alternatively insurers can take on the entire task of issueing seeds [10] (and therefore generating all the metadata).

The *audit request* for the document or collection is a tuple consisting of

1. the swarm root hash of the collection audit manifest.
2. the base seed for this audit round
3. the MASH index (unless derivable from the seed) and
4. common TTL (storage period).

[9] Implementation note: IO and memory allocation being the main bottleneck, the secrets for all seeds are best calculated with a single chunking iteration.

[10] This can be done trustlessly if the insurer generates masks for the seeds themselves and publishes them (put it on the blockchain, or just publishes their own valid receipt). If the seed is leaked before it is due, or not revealed when it is due, the insurer stands to lose their deposit and compensate the owner. To catch the insurer caching results on their own beforehand, they need to collect signed audit responses from all nodes to show the nodes have seen the seed. Nodes are rewarded if they report leaked seeds or incorrect MASH-es. Therefore the auditor can cheat the audit only if they collude with the custodians of each chunk in the collection in advance, and precalculate their secrets. By keeping the reward for leaking significantly higher than what the insurer can afford as bribe will make this line of attack uneconomical.

The audit request is signed by the owner.

Audit request are sent out to the swarm, addressed by the swarm root hash of the collection audit manifest.

Auditing an entire document collection requires audits of many chunks but the main auditor launches an audit of the first chunk only. Once the audit is thus initiated by the main auditor it proceeds automatically until it is complete or an error is found.

If any of the metadata is not available at the time of the audit, the main auditor will not be able to conduct a proper audit and therefore they cannot respond to the owner. If this happens, the owner can escalate the issue and start litigation against them by sending the audit request in a transaction to the blockchain.

Initiating the automated audit process:

1. Anyone that has the collection audit manifest can act as the main auditor and start off the recursive collective audit procedure.
2. The main auditor retrieves the supporting structures (guardian data, smash chunk hashes, partial verification bits, uniqueness bits and the MASH-es).
3. The auditor starts by verifying the MASH root and the signature and checks the integrity of the support data.
4. If all the data checks out, the main auditor then sends out the audit request to the top chunk (hashing to the collection swarm root hash) of the collection.

Recall that a chunk encodes a subtree, in particular a non-leaf chunk consists of 128 swarm hash segments. These are the hashes of chunks on the lower level of the chunk tree, each in turn encoding their subtree. In the initial round (and the only one in case of success) the audit involves sending out audit requests of the simple type. These requests are similar to retrieval requests except that in their response, proximate storers do not send back the chunk itself but instead they calculate the audit secret hash (ASH) and respond with that. Thus during simple audit, audit requests are broadcast from a node to its peers in the swarm and the swarm collectively forwards them all the way to custodian nodes. Responses travel back to parent auditors the same way (see Figure 5).

After the audit has been initiated, the *automated collective audit process* proceeds as follows.

5. The main auditor launches the collection/file audit. This means they send an audit request for the chunks represented by the hash segments in their own chunk one at a time proceeding from left to right skipping chunks that occurred before in the collection (as per the respective uniqueness bit).
6. These audit requests for a chunk are addressed by the swarm hash of the chunk, and get forwarded in the usual way to end up at the custodian of the chunk in question.

In order to accelerate the process we make sure that peers that get involved in the collective audit get forwarded all the relevant data they need:

7. In addition to the audit request as specified above the parent auditors send the partial verification bits and uniqueness bits relevant to the subtree audited by the child auditor.

These storers that have just received an audit request are either custodians of a data chunk (leaves), or they are custodians of an intermediate chunk in the swarm tree just like their parent auditor. This is just one off network traffic and need not be repeated for subsequent audits.

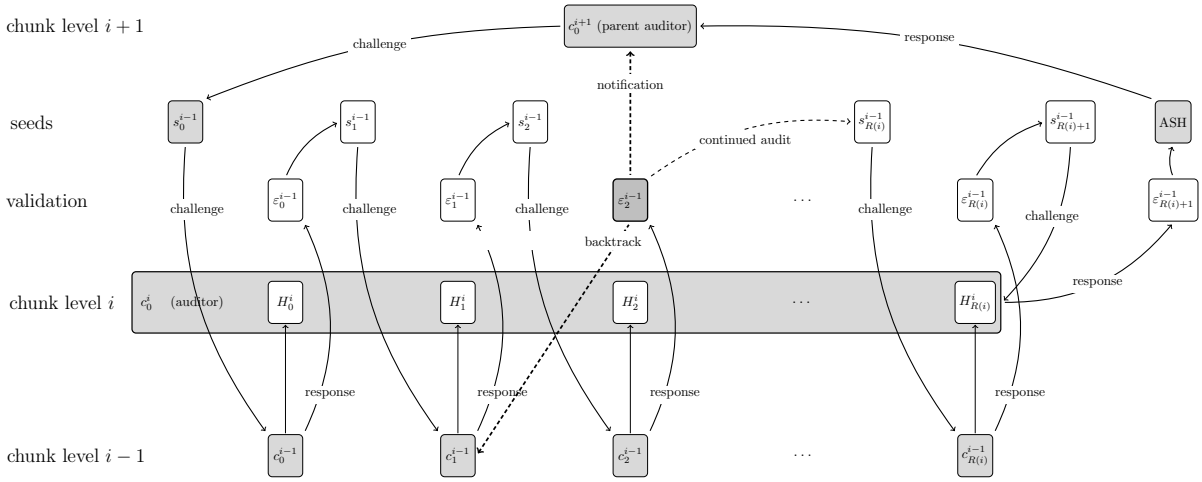

Fig. 5: This figure zooms in on a chunk in a chunk tree of a document. The chunks represent their custodian nodes that act as auditors of the subtree their chunk encodes. The arrows represent the flow of information in the successive steps of calculating CRASH. The custodian of the non-leaf chunk receives a seed and iterates over the hashes of its chunk. It initiates an ASH challenge on their immediate child nodes in succession. After receiving the response from one chunk's custodian, they perform validation against the error bits and calculate the subsequent seed that they then send on to the following child as an ASH challenge. In case the validation fails, the node backtracks and escalates the audit to a Merkle proof challenge (dashed lines). After piping the seed through the children's audits and getting back valid ASH-es, the node performs a self-challenge as if it was a leaf chunk and sends back the resulting audit secret to their parent auditor.

8. Custodians of an intermediate chunk proceed in the same fashion as the top auditor and recursively spawn audit requests on the chunk/subtree defined by the successive hash segments of their chunk one at a time.

9. Custodians of leaf chunks simply calculate the audit secret hash for their chunk using the seed they received and return that if the partial verification bits match. If they do not match then something went wrong and they respond with a complete Merkle proof instead.

10. Upon receiving the secret for a chunk (the simple ASH response) represented by a hash segment of their own chunk, the auditor also checks the secret against the corresponding partial verification bits. If no error is detected, the auditor generates the next seed needed for the audit of the next subtree addressed by the following hash segment. If errors are detected, the auditor starts backtracking to find the source – see point 14.

11. After all subtree secrets are covered, i.e., the ASH for (the chunk hashing to) the rightmost hash segment is received, the auditor then uses the next seed to calculate their own ASH i.e. the secret for their own chunk. They check their verification bits and if that matches they respond to their parent auditor with this secret. If it does not match they know an error occured before, so they start backtracking to find the source – see point 14. This step is not spurious because in case there are skipped subtrees (as per uniquness bits), the ASH of the last child does not prove their possession of the entire chunk, ie. a malicious storer could use the non-unique chunks hashes for storage and still pass the audit any number of times.

It is easy to see that this process follows the order defined in the previous section, and therefore the last secret calculated by the top auditor is the collection-level recursive audit secret (CRASH) for the collection in question.

12. If everybody responds to the audit and if the final secret (CRASH) matches the respective mask (MASH), then the audit is successful. At this point the main auditor can send a MASH proof to any interested party, proving a successful audit.

13. Whoever is interested can verify the MASH proof against the MASH root and if it checks out, they can be fairly certain the collection is preserved in full integrity and promptly retrievable in the swarm.

Failure:

14. If at any time during the audit process there is no response to an audit request about a chunk, the guardian of that chunk is looked up by the responsible auditor and is sent a Merkle proof request. Upon receiving a repsonse, the auditor verifies the proof and calculates the ASH secret and proceeds according to steps 1–8. If there is no response, the audit is escalated and litigation on the blockchain starts: the auditor sends the ASH proof challenge to the blockchain accusing the guardian of having lost the chunk in question. From here on the standard deadline for refutation starts. The exact procedure is discussed in *[5]*.

15. Errors are detected in two ways: either an intermediate auditor finds that one of their children returned an audit secret that does not match the verification bits, or the main auditor finds that the final secret does not match the respective MASH. When this happens we need to find the culprit, i.e., the node that lost the chunk. This is done by sending out successive Merkle proof challenges. Luckily, due to the iterative error coding scheme used (in which one segment's ASH is the input to the seed of the next challenge), once an error occurs the probability of it staying undetected falls exponentially. Therefore the culprit is most likely to be among the most recently audited chunks.

As a consequence of this, the best strategy is to proceed backwards and check the most recently audited chunks directly for proof of custody using a Merkle proof challenge. If a node responds with a correct proof, the previous chunk is queried. Once a node fails to respond with a correct proof we have found the culprit. If a culprit is found, the audit is escalated and litigation on the blockchain begins. The node carrying out this (partial) audit feeds back the information about the error to their parent auditor. Thus the peers know not to pursue litigation themselves against their child auditor [11] .

Note that in our recursive auditing scheme, the intermediate (non leaf) nodes were not only audited themselves, but they also served to initiate audits on the subtrees encoded in their chunk. This offers great efficiency gains because if the entire audit were to be carried out by just one peer, then chunks for each intermediate node would need to be retrieved in order for the main auditor to initiate all the audit requests for subtrees. Collective auditing has the immediate benefit that no intermediate chunks ever need to be actually retrieved, because the audit of subtrees are carried out by peers that store the chunk. This means a successful audit require only one challenge-response message roundtrip per node involved.

4.3 Ensuring correct syncing and distribution

As it turns out, collective auditing has great advantages in policing correct syncronisation. As a result of recursive audits, when audit responses are retrieved, the audit requests come from nodes independent of the owner. This helps nodes identify neighbours that refused to sync. If

[11] In order to protect against offending nodes to simply responding with frivolous litigation notices, the notice needs to contain a transaction hash for the challenge sent to the blockchain. This way parent auditors can rest assured the audit is indeed escalated.

an audit request reaches a node that is most proximate to the target chunk, the node recognizes that it is a chunk that it was supposed to receive while syncing with one of its peers. If it did not, then it sends an audit request to the chunk's guardian and feedback to its parent auditor (see Figure 6).

This can be thought of as a warning to the guardian (or in fact the node that acts as custodian). If they still keep the chunk to themselves, they will lose money as a result of litigation. Even if they are innocent, they are motivated to forward since that is cheaper still than litigation. Therefore they will forward the audit request to their appropriate online peer towards the node that they had forwarded the original store request to. If all nodes delegate and forward, the proximate node will eventually receive the chunk and can act as custodian. Interestingly, this situation could also happen as a result of network growth and latency. We conclude that SWINDLE recursive auditing can repair retrievability [12] .

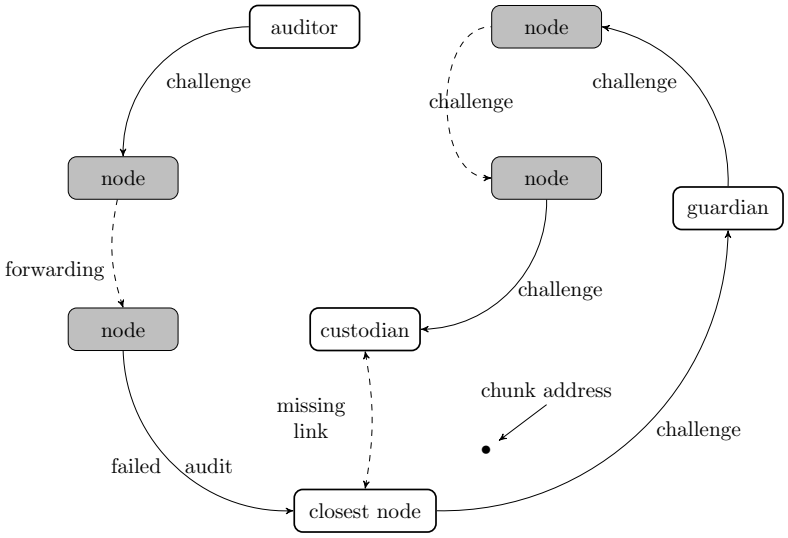

Fig. 6: The arrows represent local transactions between connected peers. After the audit reaches the closest node and the chunk is not found, the closest node challenges the guardian who in turn challenges the node it originally bought a receipt from, and so on until the challenge lands on the current custodian who now has the chance to connect to the node that is actually closest to the chunk address (or at least closer).

If the closest node gets the chunk, it calculates the audit secret and the audit can continue. If there is a delay longer than the timeout, the audit concludes and litigation starts against the current custodian. The initiator includes the address of the known closer node without which the impostor cannot be prosecuted.

The collective audit is also used as a health check, for instance, to repair chunks in erasure coded collections. The repair process itslef is independent of the litigation (see *[5]*).

5 Conclusion

In this paper we presented a simple proof of custody formula inspired by the Wilkinson–Buterik proof of storage used by Storj (*[6]*). The formula offers 3 different types of challenge that

[12] Note that adaptation to network growth and shrinking is taken care of by the syncing protocol. However if network connections are saturated and/or nodes have not yet heard of each other it could happen that they are genuine yet appear not syncronized.

auditors can use in different stages. We specified an auditing and litigation scheme that has ideal properties to secure the swarm against chunk loss.

SMASH/CRASH proofs offer integrity checking for chunks as well as for documents and document collections that

- allows owners to initiate and control audits without storing any information other than the swarm hash of the chunk;
- allows owners to outsource auditing without a trusted third party;
- it provides a seed generation algorithm for securing large document collections with a single audit secret so it scales for both storage and bandwidth;
- the successive secrets matched against verification bits offer a method of error detection which makes it very efficient to find offending nodes without remembering the (masked) secret for each chunk;
- allows easy verification by third parties like smart contracts to serve as evidence when it comes to litigation on the blockchain;
- works without ever writing anything to the blockchain which is thus only used for last-resort litigation;
- enables compact proofs to optimize bandwidth use and prevent blockchain bloating
- offers guardians and custodians ways to refute the challenge, including proof that auditors request is invalid.

We outlined an auditing and litigation protocol which

- offers efficient ways to probe the swarm off-chain with recursive outsourceable collective audits;
- enables prompt incentivised escalation whereby an audit continues as litigation on the blockchain;
- helps nodes identify greedy peers that do not forward chunks;
- offer a way to repair improper syncronisation state;
- offer a method to detect a damage in erasure coded data ideally leading to repair due to redundancy

References

[1] Kevin D Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, 43–54. ACM, 2009.

[2] Vitalik Buterin. Secret sharing and erasure coding: a guide for the aspiring dropbox decentralizer. 2014. URL: https://blog.ethereum.org/2014/08/16/secret-sharing-erasure-coding-guide-aspiring-dropbox-decentralizer.

[3] Ralph C Merkle. Protocols for public key cryptosystems. In *Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society*, 122. IEEE, 1980.

[4] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3):442–483, 2013.

[5] Viktor Tron, Aron Fischer, Daniel Nagy A, and Zsolt Felföldi. Swap, swear and swindle: incentive system for swarm. Technical Report, Ethersphere, 2016. Ethersphere Orange Papers 1. URL: bzz://ethersphere.sw/orange-papers/1/sw^3.pdf.

[6] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj: a peer-to-peer cloud storage network. Technical Report, storj, 2014. v1.01. URL: https://storj.io/storj.pdf.