Cost-effective Data Upkeep in Decentralized Storage Systems

Racin Nygaard
University of Stavanger, Norway
racin.nygaard@gmail.com

Hein Meling
University of Stavanger, Norway
hein.meling@uis.no

John Ingve Olsen
University of Stavanger, Norway
johningveolsen@gmail.com

ABSTRACT

Decentralized storage systems split files into chunks and distribute the chunks across a network of peers. Each peer may only store a few chunks per file. To later reconstruct a file, all its chunks must be downloaded. Chunks can disappear from the network at any time as peers are untrusted and may misbehave, fail or leave the network. Current systems lack a secure and cost-effective mechanism for discovering missing chunks. Hence, a client must periodically re-upload all of the file's chunks to keep it available, even if only a few are missing from the network. Needlessly re-uploading chunks waste significant amounts of the network's bandwidth, takes additional time to complete, and forces the client to pay for unwarranted resources.

To address the above problem, we propose SUP, a novel protocol that utilizes proof-of-storage queries to detect missing chunks. We have evaluated SUP on a large cluster of 1000 peers running a recent version of Ethereum Swarm. Our contributions include the design and implementation of SUP and a study of Swarm's redundancy characteristics. Our evaluation shows that SUP significantly improves bandwidth utilization and time spent on data upkeep compared to the existing solution. In common scenarios, SUP can save as much as 94 % bandwidth and reduce the time spent re-uploading by up to 82 %. While dependent on the storage network's bandwidth pricing policy, using SUP may also reduce the overall monetary costs of data upkeep.

CCS CONCEPTS

• **Computer systems organization** → *Maintainability and maintenance*; **Availability**; *Redundancy*; • **Information systems** → **Distributed storage**; *Cloud based storage*;

KEYWORDS

data upkeep, proof-of-storage, decentralized storage system, re-upload, peer-to-peer

ACM Reference Format:
Racin Nygaard, Hein Meling, and John Ingve Olsen. 2023. Cost-effective Data Upkeep in Decentralized Storage Systems. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3555776.3577728

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SAC '23, March 27-31, 2023, Tallinn, Estonia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577728

1 INTRODUCTION

In large-scale decentralized peer-to-peer storage systems such as *Ethereum Swarm* [38, 43] and *InterPlanetary File System* (IPFS) [7], peers collaborate by storing and serving each other data. These systems aim to pool the storage and computational resources of the peers together to create affordable and reliable storage for everyone. Anyone can upload files to the network, which are split into smaller chunks, as shown in Figure 1. Each chunk is replicated in a small subset of the total peers in the network.

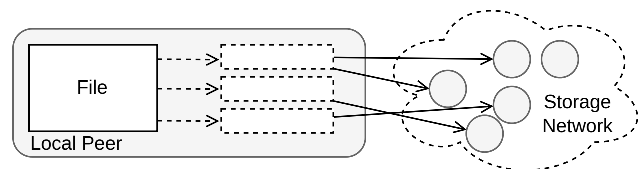


Figure 1: The local peer split files into chunks. The chunks are uploaded to different peers in the storage network.

The main challenge with this idea follows from the fact that the peers are untrusted, and the networks may exhibit significant churn. The peers may intentionally or by accident delete or corrupt the data they are tasked to store. Furthermore, a peer may itself become unreachable by the other peers due to issues with network connectivity, or it may have failed in other ways. To mitigate the unreliability of the peers, the incentives of the network motivate correct behavior with cryptographic tokens.

Even though previous work [29] has proposed solutions for data resilience, both IPFS [37] and Swarm still rely on a high degree of replication to keep data persistent. Without maintenance, as long as there are misbehaving peers and churn, the replication degree of chunks will decrease over time. The minimal replication of a file's chunks is also of particular interest, as the unavailability of a single chunk may cause the de-facto unavailability of the entire file. We have studied chunk availability in the public Swarm network and found that chunks can become unavailable as soon as 6 days after upload.

To maintain a high degree of replication, these systems rely on the client to continuously re-upload their data [20]. In Swarm, this process is called *Data Stewardship*. However, Data Stewardship always re-uploads all chunks of a file. For large files, this consumes unnecessarily large amounts of bandwidth. IPFS does not have a dedicated process for re-uploads [35, 42] and relies solely on its standard upload functionality.

In this work, we propose Storage Upkeep Protocol (SUP), a lightweight proof-of-storage system that uses storage proofs to save bandwidth when re-uploading data. A proof-of-storage system has three distinct actors, the challenger, the prover and the verifier. It

is an essential requirement that the prover cannot pre-compute the proof before seeing the challenge. As such, each challenge is coupled with a unique nonce and thus a verified proof ensures that the prover had the data when the proof was created. SUP is targeted for usage in decentralized storage systems where the data is de-duplicated and peers are untrusted.

We evaluate SUP on a large cluster consisting of 1000 Swarm Bee peers. The results show that SUP can provide up to a 94 % reduction in the bandwidth consumed in the network when re-uploading. It also provides faster re-uploads; in common scenarios, the time spent re-uploading is reduced by up to 82 %.

Our contributions are summarized as follows:

- The design and implementation of SUP, a cost-efficient data upkeep protocol built for decentralized storage systems.
- A performance evaluation of SUP and Data Stewardship on a real-world cluster.
- Monitoring data availability over four weeks in the public Ethereum Swarm network.

2 SWARM OVERVIEW

In this section, we present an overview of the Swarm network and its data upkeep protocol, Data Stewardship.

Swarm [43] is a global decentralized storage and communication system that distributes stored data to a network of peers. According to a monitoring website [17], the public network has more than 2000 active peers in the past month, and over 863 000 total peers. The peers connect to the Swarm network using the Swarm Bee client. Swarm's p2p overlay network used for routing and discovery is based on the Kademlia distributed hash table (DHT) [26].

Each peer in Swarm is required to deploy a smart contract, called checkbook, to an EVM-compatible blockchain, e.g., Ethereum or Gnosis. The smart contract is used to reward peers with BZZ tokens when they provide resources to the network. In the current version of Swarm, providing bandwidth is incentivized. Specifically, peers pay to download chunks and are rewarded for delivering chunks and forwarding messages. After deploying the smart contract, each peer generates a unique peer address used to identify it in the network. The address is generated by hashing the concatenation of the peer's Ethereum public key, the network identifier, and the hash of the block immediately following the one that deployed the peer's checkbook smart contract.

2.1 Data Storage in Swarm

Swarm splits files into 4 KB chunks. A unique *chunk identifier* is derived for each chunk by passing the chunk's content through a cryptographic hash function. Swarm creates a 128-ary Merkle tree where each of the file's chunks is a leaf. The internal nodes and root of the Merkle tree are also stored in 4 KB chunks and contain a concatenation of the chunk identifiers of their children.

Chunk identifiers and peer addresses share the same address space. When a chunk is uploaded to the network it is sent to the *closest peer*, whose address has the greatest proximity to the chunk's identifier. The *proximity* of two addresses is the number of equal prefix bits in both addresses [38].

Each chunk is replicated by its *storer peers*; these are the closest peer and that peer's *nearest neighborhood.* A neighborhood is a set of peers that have the same proximity to an address. A peer's nearest neighborhood is the neighborhood with 8 or more members that have the closest proximity to the peer's address.

Any peer may choose to store any given chunk. However, a chunk may be irretrievable by other peers unless it is stored by its storer peers. This follows from how messages are routed in Swarm, which we discuss next.

2.2 Message Routing in Swarm

The message routing protocol in Swarm, called *forwarding Kademlia*, differs from the original Kademlia description [26]. In the original description, the peer X, which wants to look up a value y in the DHT, starts by asking the closest peer it knows to y. This peer, which we call Z, then returns a set of peers Z' that are closer to y than itself. This process continues until the closest peer in the network, Y, is discovered. Finally, the request for y is sent to Y.

In forwarding Kademlia, instead of Z replying to X with a list of candidates that are closer to y, it will forward the request to one of them. That peer then forwards the request to the closest peer it knows. Eventually, the request reaches a peer Y that knows of no closer peer to y than itself. Then, Y returns y along the same path as the request, if it has y.

Kademlia allows peers to find chunks in logarithmically many steps; if the chunk is stored at the correct peers, that is, its storer peers as defined above. A chunk that is not stored by its storer peers is not guaranteed to be found through Kademlia routing. Peers may, however, choose to cache chunks to improve retrievability.

2.3 Data Stewardship

"It is in the nature of Swarm that data eventually disappears" [20]. Swarm clients use a mechanism called Data Stewardship [20] to mitigate this fact by periodically re-uploading their data to the storage network. This, of course, requires the client to have the data that it wishes to re-upload.

The client initiates Data Stewardship by specifying a chunk identifier. Data Stewardship then uploads the chunk to its storer peers using the *push-sync* protocol. If the chunk is the root of a Merkle tree, it uploads the entire tree.

2.4 Data Availability in Swarm

We conducted a four-week experiment on the public Swarm network to see how quickly chunks would disappear. We uploaded a 5 MB file to the *Swarm gateway* [40] and checked the availability of its chunks once a day. The chunk availability over time is shown in Figure 2. Green bars indicate that the whole file is retrievable, and red bars indicate that some chunks were unavailable. On the 6th day, we observed that some of the chunks were unavailable, but they returned on the 7th day. After the 16th day, the file remained unavailable due to missing chunks.

We count chunks in one of 16 buckets (`0-f`), based on the first four bits of the chunk identifier, each time they are unavailable. Then, we normalized the buckets to account for the fact that there is some variation in how many of the file's chunks have the same 4-bit prefix. Finally, we plot the relative size of each bucket in Figure 3 as the *accumulated* (blue) bars. These bars show the frequency that chunks belonging to each bucket were found to be unavailable. We

also count how many unique chunks fall into each bucket. That is, if the same chunk is unavailable multiple times, we count it only once. We normalize and plot the relative size of these counts also in Figure 3 as the *unique* (orange) bars.

In Figure 3, we see that chunks in buckets c and e were most often found to be unavailable. However, buckets 5 and 4 have the most unique chunks. Almost none of the unavailable chunks were found in bucket 8. This seems to suggest that certain parts of the network are less reliable than others.

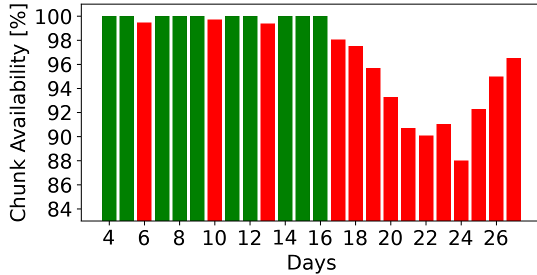


Figure 2: Chunk availability over time.

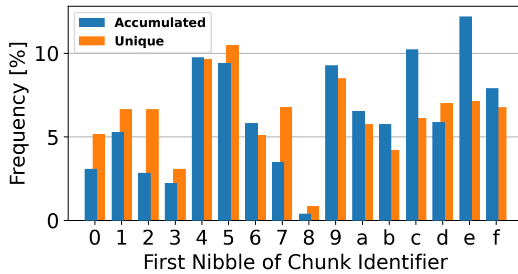


Figure 3: Faulty chunk identifier frequency.

3 SYSTEM MODEL AND REQUIREMENTS

We assume the presence of a p2p network, where each peer is connected to a subset of the total peers in the network. An overlay network is used to route messages between unconnected peers. Each peer generates a private key that is used to digitally sign messages, and a corresponding public key so that other peers can verify the signature. The key distribution and management are out of the scope of this paper.

The public key is further used to generate a unique address for use in the overlay network. The address serves as the baseline for the connectivity graph for the overlay network, such that peers are more likely to be connected to other peers with similar addresses.

There are three distinct actors in the protocol, the *Challenger* which issues proof-of-storage challenges, the *Prover* which receives the challenges and produces the proofs, and the *Verifier* which evaluates the correctness of the proof.

Requirements. 1) We design the protocol to prevent provers from computing valid proofs before receiving challenges. 2) A valid proof must ensure with an overwhelming probability that the prover was in possession of the data when the proof was created. 3) Proofs are bound to a specific prover, and cannot be reused by others. 4) Proving and verification must be efficient. 5) Proofs should be of a small constant size.

3.1 Threat Model

Challengers and provers in SUP may attempt to attack the system. Verifiers only receive proofs and thus cannot attack the system. A malicious challenger may trigger a Sybil attack by manufacturing a large number of challenges and sending these to the provers. The provers would then have to spend computational- and I/O-resources to compute the proofs, which the attacker would discard. Individual provers may attempt to construct proofs for data they do not store. Similarly, a collection of provers may collude to answer proof-of-storage challenges for the same reason. Provers may also generate false proofs that verifiers would spend resources to reject.

We address malicious challengers in Section 4.3 and malicious provers in Section 4.2. We discuss colluding provers in Sections 4.2 and 4.4. Finally, we address false proofs in Section 4.5.

4 THE STORAGE UPKEEP PROTOCOL

SUP is a novel protocol designed to save bandwidth and reduce time spent re-uploading data to decentralized storage networks comprised of untrusted peers. Re-uploading is defined as uploading previously uploaded data to the network and is often necessary to persist data, e.g., due to network churn or unreliable peers. The reduction in bandwidth and time spent comes from SUP's issuing of *storage challenges* and awaiting *storage proofs* before selectively uploading only those chunks that were missing or had invalid proof.

Our design is flexible, and modularized and does not require changes in the system's existing protocols. Instead, we add a new protocol to significantly reduce the amount of data transmitted during the re-uploading process.

SUP relies on the underlying P2P network to route messages between peers. The space-time diagram in Figure 4 illustrates SUP's protocol execution. In the figure, one entity acts as both the challenger and verifier, and three storage peers are labeled P_1, P_2, and P_3. The challenger, P_1 and P_2, all store a set of chunks labeled $\{a, b, c\}$, while P_3 only stores $\{a, b\}$. The protocol starts with the challenger sending a proof-of-storage challenge to the storage peers. Upon receiving the challenge, each storage peer computes a proof for those chunks it is storing and sends that proof to the verifier. The verifier processes each proof, and for P_3's proof, the verifier will detect that P_3 is missing chunk $\{c\}$. The verifier will then enter the re-upload phase and send the missing chunk to P_3.

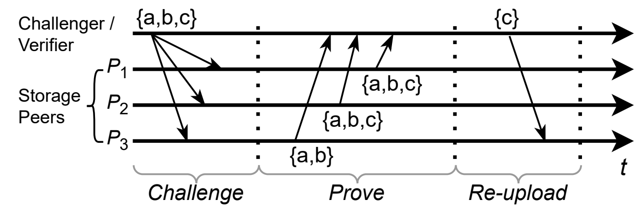


Figure 4: The message flow between the challenger, verifier, and the storage peers.

Each peer has access to cryptographic keys for digitally signing and verifying messages. In the following algorithm description, we let $\Sigma(m)$ denote the digital signature of a message m and pub a peer's public key. The following subsections will detail the most significant actors in the protocol.

4.1 Re-upload

Algorithm 1 gives a high-level pseudo-code for SUP. The client interacts with its local peer and calls `Reupload` with a list of chunk identifiers it wishes to persist in the storage network. If no list is given, the default is to use the identifiers of all chunks stored by the local peer. Given the list of chunk identifiers, the challenger creates a proof-of-storage challenge by calling `CreateChallenge` (Section 4.3). The challenge is then sent via the p2p network to the storage peers responsible for storing the chunks.

Each storage peer awaits challenge messages (Section 4.4) and generates a proof-of-storage proof for those chunks covered by the challenge that the peer is storing. Upon receiving a proof, the verifier will attempt to validate the proof using `VerifyProof` (Section 4.5). For each valid proof, the verifier removes chunk identifiers covered by the proof from the list of chunks to be re-uploaded.

Once the verifier has processed all proofs, the *uploadList* variable will only contain chunk identifiers that are missing from the network and must be re-uploaded. The verifier will iterate over the *uploadList* and upload the chunks to the storage peers.

We leave counting the proven chunk identifiers to get information about the redundancy in the network for future work.

Algorithm 1 Reuploader

```
 1: Local persistent state at challenger/verifier:
 2: C                                              ▷ Set of stored chunks
 3: S                                              ▷ Set of storage peers
 4: uploadList                                     ▷ Identifiers for chunks to upload
 5: remaining                                      ▷ Outstanding storage proofs to verify
 6: func Reupload(chunkIDs)
 7:   if chunkIDs = ∅ then
 8:     chunkIDs ← {c.id ∀ c ∈ C}                  ▷ Identifiers of all stored chunks
 9:   chal ← CreateChallenge(chunkIDs)             ▷ Algorithm 2
10:   uploadList ← chunkIDs
11:   remaining ← |S|
12:   send ⟨ chal ⟩ to S                           ▷ Send chal to storage peers
13: upon receive ⟨ proof ⟩ do                      ▷ proof verified signature
14:   if VerifyProof(proof) then                   ▷ Algorithm 4
15:     for all chunkID ∈ proof do
16:       uploadList ← uploadList \ chunkID
17:   remaining ← remaining − 1
18: upon event remaining = 0 do                    ▷ All proofs processed or timeout occurred
19:   for all chunkID ∈ uploadList do
20:     chunk ← {c : c ∈ C ∧ c.id = chunkID}
21:     send ⟨ chunk ⟩ to S                        ▷ Upload missing chunk
```

4.2 Proof Construction

The primary purpose of our storage proof is to assure the challenger that the peer in question is currently storing the data queried in the challenge. To ensure that the peer is storing the data when generating the storage proof, we need to remove any way to precompute proofs before receiving the challenge. Resistance against pre-computing proofs is achieved by having the challenger embed a unique *nonce* (number used once) into each challenge query. The nonce is unknown to provers before they process the challenge.

The proof construction assumes the existence of a cryptographic hash function that is *preimage-resistant* [34] and *puzzle-friendly* [27]. The preimage resistance property states that for any given hash value h, it is computationally infeasible to find y such that $\mathrm{H}(y) = h$. Puzzle friendliness states that for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $\mathrm{H}(k \mid\mid x) = y$, in time significantly less than 2^n. We define chunk a's chunk proof in (1), as the cryptographic hash of the concatenation of a nonce and a's data.

$$\mathrm{CP}_a = \mathrm{H}(nonce \mid\mid a) \tag{1}$$

Hashing fixes the size of chunk proofs to a small constant. Note that the concatenation order $nonce \mid\mid a$ in (1) is essential, since otherwise the chunk proof may be vulnerable to the *length extension attack* [14]. This weakness is present in well-known iterative hash functions based on the Merkle-Damgård construction, such as MD5, SHA-1, and SHA-2. In the length extension attack, an attacker with knowledge of $\mathrm{H}(m_1)$ and the length of m_1 can calculate $\mathrm{H}(m_1 \mid\mid m_2)$ for an arbitrary message m_2 without knowing m_1. If we flipped the concatenation order in the chunk proof to be $\mathrm{H}(a \mid\mid nonce)$, a dishonest prover could store only $\mathrm{H}(a)$ and the length of a and delete chunk a. Upon receiving a new challenge, a prover could use the length extension attack to calculate $\mathrm{H}(a \mid\mid nonce)$ without knowing a. We recommend computing the chunk proof in (1) using a strong cryptographic hash function without known weaknesses, such as the length extension attack.

The chunk proofs are aggregated to form a full storage proof for the challenge. We chose to aggregate the chunk proofs using XOR. Using XOR reduces memory consumption, as we only need to keep a small fixed amount of data in memory. The prover computes the hash of the concatenation of the aggregated chunk proofs and the prover's public key. Concatenating with the public key prevents other peers from copying the aggregated chunk proof to use it to claim they are storing the chunks. For a peer with the public key pub, the base proof for a set of chunks $\{a, b, c\}$ is given in (2).

$$\mathrm{BP}_{a,b,c} = \mathrm{H}((\mathrm{CP}_a \oplus \mathrm{CP}_b \oplus \mathrm{CP}_c) \mid\mid pub) \tag{2}$$

Since XOR is both associative and commutative, the order in which chunk proofs are aggregated does not impact the final result. We want that our scheme allows a prover to answer subsets of a challenge. For example, a challenger might issue a challenge for the chunks $\{a, b, c, d, e\}$, but the recipient peer can only prove $\{a, b, c\}$. To encode that only the chunks $\{a, b, c\}$ are included in the proof, we introduce a bitmap L. The i-th bit in L represents whether or not the i-th chunk is included in the proof. For example, $L = 11100_2$ means that chunks $\{a, b, c\}$ are included in the proof.

In addition, we need to embed the nonce to allow the verifier to know which challenge is being proven, as the challenger might have issued multiple challenges. Lastly, we require that the prover digitally signs the base proof, providing message integrity and proof that it is authorized to issue proofs for the included public key. This allows the verifier to determine that the proof comes from the intended peer. Given a challenge requesting proofs for $\{a, b, c, d, e\}$, a peer storing chunks $\{a, b, c\}$ should respond with a proof according to (3), where bitmap $L = 11100_2$.

$$\mathrm{Proof}_{a,b,c} = \{\mathrm{BP}_{a,b,c},\ nonce,\ L,\ \Sigma(\mathrm{BP}_{a,b,c}),\ pub\} \tag{3}$$

4.3 Challenger

When a client wants to re-upload data using SUP, we send a challenge to the other storage peers to determine which chunks are missing from the network and must be uploaded. The challenge contains a list of chunk identifiers that the challenge concerns.

To prevent provers from pre-computing proofs, the challenger embeds a nonce in the challenge. The nonce is obtained by generating a random value and then cryptographically committing to the value by hashing it. The purpose of cryptographically committing is to be able to dispute any claim over who was the originator of the challenge, as only the challenger will be able to reveal the preimage of the committed nonce.

Finally, the list of chunk identifiers is packed together with the committed nonce and then digitally signed to create the challenge. By digitally signing the challenge, we can prevent unauthorized or excessive issuance of challenges. In addition, the signature allows provers to verify the integrity of the challenge. Algorithm 2 lists the pseudo-code.

Algorithm 2 Create Challenge

```
1: Local persistent state at challenger:
2: commits                              ▷ Map of nonce pre-images
3: challenges      ▷ Map of chunk identifiers in sent challenges
4: pub                                  ▷ Challenger's public key
5: func CreateChallenge(chunkIDs)
6:    k ← RandomValue()
7:    nonce ← H(k)
8:    commits[nonce] ← k                     ▷ Commit to the nonce
9:    challenges[nonce] ← chunkIDs
10:   chal ← [ nonce, chunkIDs ]
11:   return [ chal, Σ(chal), pub ]
```

4.4 Prover

Algorithm 3 lists the pseudo-code for proving chunk possession after receiving a challenge. The prover initially verifies the integrity of the challenge, and its digital signature to prevent misuse. For each chunk identifier contained in the challenge, the prover checks if it stores the chunk, and if so, it creates a chunk proof for it, as described in (1), and then aggregates it to the base proof described in (2). The base proof is initialized to a null vector with the same length as the hash function.

Including both the committed nonce and the chunk data in the chunk proof means that the prover must know both simultaneously, making pre-computation infeasible. To aggregate the chunk proofs, we use XOR with the base proof. This reduces the algorithm's memory consumption, as the chunk's data can be garbage collected from memory after creating each chunk proof. The bitmap L indicates which chunks are included in the proof. As the chunks are processed in the same order as the challenge, we add 2^i to the bitmap to mark the i-th chunk as included.

After processing all the chunks, we cryptographically hash the concatenation of the base proof with the prover's public key to create the final base proof. Including the prover's public key prevents other peers from eavesdropping on the communication to re-use the proof for themselves. We then create the final proof using the final base proof, the received nonce and L. The final proof is returned to the challenger together with its digital signature.

Algorithm 3 Prove Data Possession

```
1: Local persistent state at prover:
2: C                                       ▷ Set of stored chunks
3: pub                                     ▷ Prover's public key
4: upon receive ⟨ chal ⟩ do             ▷ chal verified signature
5:    baseProof ← [0]^H.length
6:    L ← 0                ▷ Bitmap of chunks included in proof
7:    for i ← 0 to |chal.chunkIDs| do
8:       chunk ← {c : c ∈ C ∧ c.id = chal.chunkIDs[i]}
9:       if chunk ≠ ∅ then
10:         baseProof ← baseProof ⊕ H(chal.nonce ∥ chunk)
11:         L ← L | (1 ≪ i)                          ▷ Add 2^i to L
12:   baseProof ← H(baseProof ∥ pub)
13:   proof ← [ chal.nonce, L, baseProof ]
14:   reply ⟨ proof, Σ(proof), pub ⟩   ▷ Send proof to challenger
```

4.5 Verifier

The pseudo-code for the verifier is listed in Algorithms 4 and 5. Any peer who stores the same chunks the storage proof covers can verify the proof. To verify a storage proof, the verifier calculates a new proof for the same chunks and then compares it to the received proof. When the proof covers fewer chunks than the challenge, we use the bitmap L to know which chunks are covered by the proof. The prover claims that the challenge's i-th chunk is covered by the proof only if L's i-th bit is set. The verifier returns *true* only if the received proof matches the newly calculated proof.

Typically, a challenge is sent to multiple peers, and for verification of multiple proofs, the verifier can leverage caching to amortize the cost of I/O operations to fetch chunks. That is, the verifier caches each chunk proof so that it can verify multiple proofs from different storage peers without accessing its local storage for each of them. When iterating through the proof, each chunk proof is retrieved from the cache or computed on the fly and stored in the cache for future lookups.

Algorithm 4 Verify Proof

```
1: Local persistent state at verifier:
2: C                                       ▷ Set of stored chunks
3: challenges       ▷ Map of chunk identifiers in sent challenges
4: func VerifyProof(proof)
5:    myProof ← [0]^H.length
6:    chunkIDs ← challenges[proof.nonce]  ▷ chunkIDs that may be in proof
7:    for i ← 0 to |proof.L| do         ▷ |proof.L| is ⌊log_2 proof.L⌋ + 1
8:       if proof.L & 2^i ≠ 0 then          ▷ Chunk is included in proof
9:          chunkProof ← GetChunkProof(proof.nonce, chunkIDs[i])
10:         if chunkProof = nil then
11:            return false   ▷ Error: Must store all proven chunks to verify
12:         myProof ← myProof ⊕ chunkProof
13:   myProof ← H(myProof ∥ proof.pub)
14:   return myProof = proof.baseProof
```

Algorithm 5 Get Chunk Proof

```
1: Local persistent state at verifier:
2: C                                       ▷ Set of stored chunks
3: chunkProofs               ▷ Map of all processed chunk proofs
4: func GetChunkProof(nonce, chunkID)
5:    if chunkProofs[nonce ∥ chunkID] = nil then
6:       chunk ← {c : c ∈ C ∧ c.id = chunkID}
7:       if chunk = ∅ then
8:          return nil
9:       chunkProofs[nonce ∥ chunkID] ← H(nonce ∥ chunk)
10:   return chunkProofs[nonce ∥ chunkID]
```

5 IMPLEMENTATION

We have implemented SUP as a package in Swarm Bee version 1.7.0 (released 24 July 2022). The implementation consists of about 700 lines of Go code, plus about 600 lines of code for benchmarking, testing and metrics collection. We have made the source code available at: (https://github.com/relab/sup).

5.1 Adapting SUP to Swarm

We presented SUP in Section 4 as a protocol suitable for use in a generalized decentralized storage network. To implement SUP in Swarm, we had to deviate slightly from the protocol specification. The following paragraphs explain how we implemented SUP in Swarm.

1) Our implementation sends *one chunk identifier per challenge* instead of multiple identifiers batched together in a single challenge. In Swarm, each chunk should be stored by its *storer peers*, as defined in Section 2.1. The challenger cannot accurately predict which chunks will share the same storer peers, due to its incomplete view of the network. Furthermore, the probability that any two chunks will share the same storer peers decreases as the network size increases. A possible way to implement batching is to split the batches if a forwarder notices that some chunk identifiers in the batch have different storer peers. Nevertheless, SUP significantly improves bandwidth use even without this optimization, compared to Data Stewardship. Therefore, we leave it to future work to implement this optimization.

2) We introduce a forwarder role to relay challenges from challengers to provers that are not directly connected. A challenger sends a challenge to the directly connected peer closest to the challenge's chunk identifier. A peer receiving a challenge becomes a forwarder of that challenge if it is directly connected to a peer in even closer proximity to the target chunk identifier; otherwise, the peer becomes the prover. Proofs are returned to the challenger through each forwarder who helped deliver the challenge.

3) The challenger is unlikely to be directly connected to the prover. A mechanism is needed for the challenger to verify that the proof comes from the correct part of the network, that is, the chunk's storer peers. As discussed above, the challenger cannot accurately predict the storer peers for a chunk. However, it can make an informed guess about the storer peers' proximity to the chunk. Peer addresses are uniformly distributed in Swarm. Challengers can expect storer peers to have at least the same proximity to the chunk identifier as the challenger's proximity to its nearest neighbors. Therefore, we require provers to include a *block hash* with each chunk proof. This should be the block hash used to generate the prover's peer address. The verifier can then recover the public key from the signature of the proof and combine it with this block hash to recover the peer address of the prover. The verifier can compute the prover's proximity to the chunk identifier using the recovered peer address. Based on the verifier's view of the network, the verifier can ensure that the prover is at least as close as the chunk's storer peers *should* be.

4) Finally, our implementation omits signatures of challenges. We note that the *request* messages in other Swarm protocols, such as push-sync and the retrieval protocol, do not include such signatures. We believe that including signatures would compromise *sender anonymity*, as the prover must identify the signer to verify a signature. Sender anonymity is a "crucial feature of Swarm" [43]. Therefore, as explained above, we chose to omit signatures on challenges, but they are still necessary on proofs. We believe an incentive system would encourage forwarder peers to forward challenges without modification. However, implementing such an incentive system is beyond the scope of this paper.

5.2 Structure

In our SUP implementation in Swarm Bee, the same peer acting on behalf of the client fulfills the roles of challenger and verifier. Other peers may assume the role of forwarder or prover, depending on their proximity to the target chunk in a challenge.

We expose a private HTTP API endpoint in the Swarm Bee peer as a direct replacement for Data Stewardship. When invoking the API, the client specifies the identifier of the chunk, or the root chunk of a file, to be re-uploaded. This chunk identifier is input into a `Reupload` function, which performs the roles of the challenger and verifier.

SUP registers a new protocol with the *libp2p runtime*. libp2p [25], is the networking library used by Swarm Bee. Whenever a peer receives a new connection using this protocol, the libp2p runtime invokes a handler function specified by SUP. This handler performs the functions of the forwarder and prover.

5.3 Challenger and Verifier

When SUP is invoked through the API, the challenger computes the list of chunks to re-upload. The client can specify an entire file by inputting the identifier of the file's root chunk. In this case, the challenger traverses down the entire Merkle tree that composes the file to collect all the chunk identifiers. Then, for each chunk identifier, the challenger invokes the `reuploadChunk` function.

The `reuploadChunk` function creates a challenge, consisting of a nonce and the chunk identifier, and then sends it to the closest peer and waits for a response. If a proof is received in response, the challenger verifies it and checks the address of the prover, as explained in Section 5.1. If no response was received, or the proof was invalid, `reuploadChunk` invokes push-sync to re-upload the chunk to the network.

5.4 Forwarder and Prover

SUP's libp2p handler function implements the forwarder and prover roles. The handler's job is to receive a challenge and then respond with a proof, if possible. As explained in Section 5.1, it takes on the forwarder role if it is directly connected to a peer closer to the chunk identifier. Otherwise, it takes on the role of prover and generates a storage proof for the chunk if it has the chunk. In either case, the handler eventually sends the proof back to the peer from whom it received the challenge.

If a proof could not be obtained, the handler closes the communication stream instead. This causes a cascade where every previous forwarder in the chain between the challenger and the handler closes their communication streams also. Eventually, the challenger detects that its communication stream with its closest peer was closed and re-uploads the chunk.

6 EVALUATION

In this section, we present our evaluation of SUP. We measured and compared the message sizes of SUP and Data Stewardship to estimate SUP’s theoretical bandwidth savings. We ran experiments with both protocols on a network of 1000 Swarm peers to measure the bandwidth use and re-upload duration in several different scenarios. Our results show that SUP saves up to 94 % bandwidth and uses up to 82 % less time than Data Stewardship. Its bandwidth use and re-upload duration scale linearly with the number of lost chunks.

6.1 Experimental Setup

To run our experiments, we used a cluster of 30 physical machines. Each machine is installed with Ubuntu 18.04.4 LTS and has 32 GB RAM, an Intel Xeon E-2136 3.30 GHz CPU, a 1.5 TB SSD disk, and 1 Gbit/s NIC. To orchestrate the cluster and manage 1000 Swarm Bee peers, we used Kubernetes [23] and Helm [19]. We distributed the Swarm Bee peers on 28 of the machines, used one to host a private Ethereum network, and the last one to manage the experiment execution. In our setup, we used version 1.7.0 [39] of Swarm Bee with our modifications and SUP implementation.

6.2 Evaluation Framework

To measure the practical bandwidth savings of SUP in comparison with Data Stewardship, we ran the two re-upload protocols on files with different chunk loss rates. We developed an evaluation framework that facilitates the uploading of files to Swarm and the random removal of a percentage of chunks.

Our evaluation framework improves upon previous work evaluating Swarm [28]. The previous work required all peers to be terminated before the evaluation tool could modify the state of each peer. Our framework, however, can operate directly on online peers. We accomplish this by adding new features to Swarm Bee’s debug API and integrating our framework with the API.

We used the framework to write a program for running our experiments. The program is given a set of file addresses to re-upload, a range of chunk-loss percentages, and snapshots listing the chunks stored by each peer. For each file, it applies a modified snapshot with a percentage of the chunk identifiers belonging to the file removed. Then, it performs a re-upload for the file using one of the two re-upload protocols. Lastly, it measures the re-upload duration and gathers metrics from each peer to calculate the bandwidth usage.

The evaluation program uses the Kubernetes port-forwarding API to connect to peers from outside the Kubernetes cluster. When running experiments using our framework, we discovered and reported a memory leak in the Kubernetes port-forwarding API client (https://github.com/kubernetes/kubernetes/issues/112032).

6.3 Message Sizes

We measured the message sizes for the request and response messages in SUP and push-sync (used by both SUP and Data Stewardship). SUP’s requests (challenges) are 68 bytes, and its responses (proofs) are 205 bytes. Push-sync’s requests (chunk deliveries) are 4256 bytes, of which 4104 bytes is the chunk itself, and its responses (receipts) are 135 bytes. In Table 1, we use these message sizes to estimate the bandwidth usage of a single instance of SUP and Data Stewardship when the chunk is available and unavailable. We do not consider the unexpected case where an invalid proof is received, in which the transmission of the invalid proof comes in addition to the challenge, delivery, and receipt.

Table 1: Estimated bandwidth usage for one chunk.

Chunk Availability	SUP	Data Stewardship	SUP Savings
Available	273 B	4391 B	93.78 %
Unavailable	4459 B	4391 B	-1.55 %

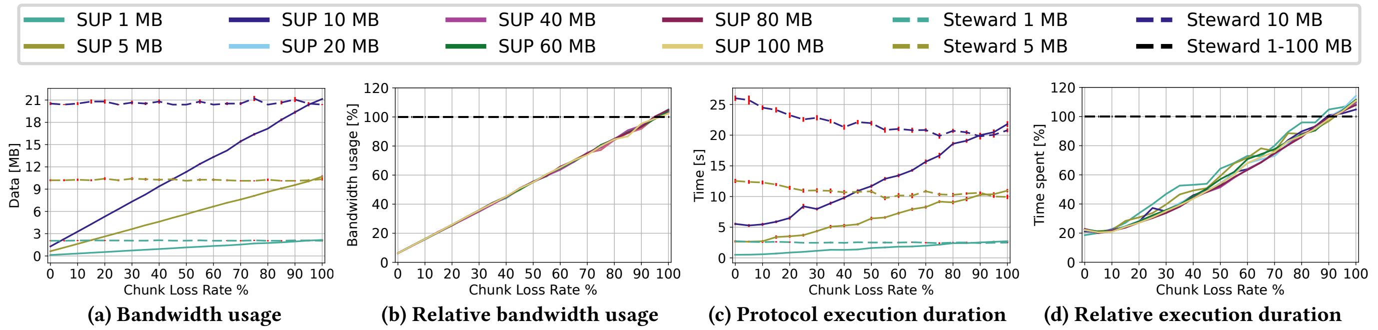
Based on the estimates in Table 1 we derive a linear expression for bandwidth usage in SUP: $273 + 4186l$ bytes, given chunk loss rate $l \in [0, 1]$. From this expression, we calculate that SUP should use less bandwidth than Data Stewardship until 98.38 % chunk loss.

6.4 Cost-effectiveness of SUP

We demonstrate that SUP is cost-effective by comparing it against Data Stewardship in Swarm Bee. The evaluations were made on our cluster with 1000 Swarm Bee peers. We varied the file sizes from 1 to 100 MB, and the chunk loss rates from 0 to 100 %. The chunk loss rate is defined as the percentage of chunks that are missing in the network. Each experiment was repeated 22 times, and the results are presented in Figure 5. As expected, the results show that the benefit of SUP deteriorates as the chunk loss rate increases. Our results show that when the chunk loss rate reaches around 90 %, it is more cost-effective to re-upload the chunk, without checking if it is already stored. A previous study of file availability in Swarm [29] shows that even with a high replication degree, the storage system breaks down long before reaching such extreme rates of chunk loss. Our experiment on the data availability in the public Swarm network conducted over four weeks, presented in Section 2.4, shows that the chunk loss rate peaked at 12 % on a single day and was less than 10 % on all other days.

As forecasted in Section 6.3, we observe in Figure 5a that bandwidth usage in SUP scales linearly for files of 1 MB, 5 MB, and 10 MB. As expected, bandwidth usage is not affected by chunk loss in Data Stewardship for the same file sizes. We observe the same effect in Figure 5b, which shows the relative bandwidth used by SUP compared to Data Stewardship for files ranging from 1 to 100 MB. The eight lines representing SUP are more or less completely overlapping and range from 6.3 % bandwidth usage at 0 % chunk loss to 104 % bandwidth usage at 100 % chunk loss. The bandwidth usage of SUP and Data Stewardship is similar when around 95 % of the network’s chunks are missing.

Figure 5c shows our results when evaluating the protocol execution duration of SUP and Data Stewardship for files of 1 MB, 5 MB, and 10 MB. By protocol execution duration, we mean the total time elapsed since the client initiated the protocol until all missing chunks have been re-uploaded. We see that SUP has a lower execution duration than Data Stewardship until the chunk loss rate reaches 90 %. Interestingly, Data Stewardship is slower when the chunk loss rate is low. We believe this effect is due to storage peers updating their prioritized list of chunks to garbage collect when receiving a chunk they already have. When comparing the relative protocol execution duration, we see in Figure 5d, that SUP only



(a) Bandwidth usage (b) Relative bandwidth usage (c) Protocol execution duration (d) Relative execution duration

Figure 5: Cost-effectiveness for different chunk-loss rates. (a,c) The red vertical lines show the standard error.

uses 20 % of the time that Data Stewardship uses when the chunk loss rate is low, and that SUP remains superior for all file sizes until between 85 to 90 % chunk loss rate.

7 RELATED WORK

This section will discuss how other proof-of-storage (PoS) algorithms relate to the one presented in SUP. The earliest relevant works were published in 2007 [4, 21, 31]. Since that time, there has been considerable work to construct schemes with additional features and improved properties [1–3, 5, 6, 8–13, 15, 16, 18, 22, 30, 32, 33, 41, 44–48, 50–52].

Proof-of-storage algorithms provide a way to outsource storage to a remote server while being able to verify that the peer is correctly storing the data. The verifications are done via three actors, a challenger, a prover, and a verifier. PoS algorithms are closely related to proof-of-retrievability, which not merely asserts that the data is stored but also can be retrieved. The variety of PoS algorithms differs in performance, as summarized in [49]. The three actors must share the computational burden of the PoS algorithm. However, the algorithms divide the computational share differently. Some algorithms also require some pre-processing and additional metadata at one or more of the actors and thus has some storage overhead. Lastly, the number of bits required to transmit a challenge or a proof differs between the algorithms.

The PoS algorithm in SUP stands out in a few ways. First, there is no storage overhead on any of the actors. Second, the proofs generated by our algorithm verify the entire chunk, as opposed to a few specific bits or random samples. Third, our algorithm targets decentralized storage systems where peers are untrusted and unreliable. Lastly, as we have targeted the re-uploading use case, there are several features that our algorithm does not require.

The first feature is *public verifiability*, allowing anyone, not just the data owner, to query the remote server with storage challenges. Such schemes require that the data owner generates some *proving metadata* that other peers can use to generate challenges and verify proofs. Other definitions for public verifiability, sometimes called *public auditability*, allows a peer to assert the correctness of a challenge or proof to a third-party. Typically, such a feature is desired in protocols where peers want to prove the misbehavior of other peers. One example is FileCoin [24], where storage peers must periodically prove data possession and integrity. Other peers can verify the storage peer's proofs, and if found invalid, they can punish the storage peer by excluding it from the network or taking its collateral.

The next feature is *updatable*, which allows metadata to be partially modified on the storage peers. An updatable scheme is well suited for use cases where data updates are frequent and computing the metadata needed for proving is expensive. As previously mentioned, SUP does not need any additional metadata. Moreover, the decentralized storage systems that we target are immutable.

Lastly, some schemes can detect the data's replication degree. Typically, these schemes work by encoding the data differently for each storage peer. For our use case, this is not sufficient, as storage peers are untrusted and may collude in answering storage challenges. In addition, such a design requires additional metadata, which is undesirable for SUP.

8 CONCLUSION

This paper presents SUP, a protocol for cost-effective data upkeep in decentralized storage networks. The need for data upkeep is well documented in both IPFS [36] and Swarm [20]. We monitored the data availability in the public Swarm network over four weeks. We found that clients can be expected to re-upload their files 6 days after the previous upload to keep the file available. Current protocols for re-upload waste resources, as they require clients to upload the entire file, even though only a few chunks may have been lost. SUP employs a novel proof-of-storage algorithm, which is used to determine what is already stored in the network before unnecessarily uploading existing data. In addition, SUP does not incur additional storage overhead at the peers.

We have demonstrated a working solution in a large P2P network with 1000 peers running a recent version of Ethereum Swarm. SUP saves up to 94 % bandwidth and reduces re-uploading time by up to 82 %. This reduction benefits the client as bandwidth consumption is linked to monetary cost, and it also improves the entire network resource utilization. The source code is made available to support the adoption of SUP in other decentralized storage networks.

ACKNOWLEDGMENTS

We would like to thank members of the Swarm team for helpful discussions and technical assistance. This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

REFERENCES

[1] Frederik Armknecht, Jens-Matthias Bohli, David Froelicher, and Ghassan Karame. 2017. Sharing Proofs of Retrievability across Tenants. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 275–287.

[2] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O Karame, Zongren Liu, and Christian A Reuter. 2014. Outsourced Proofs of Retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 831–843.

[3] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. 2011. Remote Data Checking Using Provable Data Possession. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (2011), 1–34.

[4] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable Data Possession at Untrusted Stores. In *Proceedings of the 14th ACM conference on Computer and communications security*. Association for Computing Machinery, New York, NY, USA, 598–609.

[5] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. 2008. Scalable and Efficient Provable Data Possession. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. Association for Computing Machinery, New York, NY, USA, 1–10.

[6] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. 2009. Proofs of Storage from Homomorphic Identification Protocols. In *Advances in Cryptology – ASIACRYPT 2009*. Springer, Berlin, Heidelberg, 319–333.

[7] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *arXiv* abs/1407.3561 (2014), 11.

[8] Kevin D Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 187–198.

[9] Kevin D Bowers, Ari Juels, and Alina Oprea. 2009. Proofs of Retrievability: Theory and Implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. Association for Computing Machinery, New York, NY, USA, 43–54.

[10] David Cash, Alptekin Küpçü, and Daniel Wichs. 2017. Dynamic Proofs of Retrievability Via Oblivious RAM. *Journal of Cryptology* 30, 1 (2017), 22–57.

[11] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. 2008. MR-PDP: Multiple-Replica Provable Data Possession. In *2008 The 28th International Conference on Distributed Computing Systems*. IEEE, New York, NY, USA, 411–420.

[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. 2019. Proofs of Replicated Storage Without Timing Assumptions. In *Advances in Cryptology – CRYPTO 2019*. Springer, Cham, 355–380.

[13] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. 2009. Proofs of Retrievability via Hardness Amplification. In *Theory of Cryptography*. Springer, Berlin, Heidelberg, 109–127.

[14] Thai Duong and Juliano Rizzo. 2009. Flickr's API Signature Forgery Vulnerability. https://dl.packetstormsecurity.net/0909-advisories/flickr_api_signature_forgery.pdf Accessed: 2022-12-29.

[15] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2015. Dynamic Provable Data Possession. *ACM Transactions on Information and System Security (TISSEC)* 17, 4 (2015), 1–29.

[16] Chaowen Guan, Kui Ren, Fangguo Zhang, Florian Kerschbaum, and Jia Yu. 2015. Symmetric-Key Based Proofs of Retrievability Supporting Public Verification. In *Computer Security – ESORICS 2015*. Springer, Cham, 203–223.

[17] Janoš Guljaš. 2022. Network Statistics - Swarm Scan. https://swarmscan.resenje.org Accessed: 2022-12-28.

[18] Zhuo Hao, Sheng Zhong, and Nenghai Yu. 2011. A Privacy-Preserving Remote Data Integrity Checking Protocol with Data Dynamics and Public Verifiability. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1432–1437.

[19] Helm. 2022. The package manager for Kubernetes. https://helm.sh/ Accessed: 2022-12-29.

[20] Rinke Hendriksen. 2021. Data stewardship #1508. https://github.com/ethersphere/bee/issues/1508 Accessed: 2022-12-29.

[21] Ari Juels and Burton S Kaliski Jr. 2007. PORs: Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 584–597.

[22] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology - ASIACRYPT 2010*. Springer, Berlin, Heidelberg, 177–194.

[23] Kubernetes. 2022. Production-Grade Container Orchestration. https://kubernetes.io/ Accessed: 2022-12-29.

[24] Protocol Labs. 2017. Filecoin: A Decentralized Storage Network. https://filecoin.io/filecoin.pdf Accessed: 2022-12-29.

[25] libp2p. 2022. libp2p - A modular network stack. https://libp2p.io Accessed: 2022-12-29.

[26] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*. Springer, Berlin, Heidelberg, 53–65.

[27] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, Princeton, NJ, USA.

[28] Racin Nygaard. 2022. Lessons Learned from a Bare-metal Evaluation of Erasure Coding Algorithms in P2P Networks. *arXiv* abs/2208.12360 (2022), 3.

[29] Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. 2021. Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 236–247.

[30] Hovav Shacham and Brent Waters. 2008. Compact Proofs of Retrievability. In *Advances in Cryptology - ASIACRYPT 2008*. Springer, Berlin, Heidelberg, 90–107.

[31] Mehul A Shah, Mary Baker, Jeffrey C Mogul, and Ram Swaminathan. 2007. Auditing to Keep Online Storage Services Honest. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. USENIX Association, USA, 6.

[32] Jian Shen, Jun Shen, Xiaofeng Chen, Xinyi Huang, and Willy Susilo. 2017. An Efficient Public Auditing Protocol With Novel Dynamic Structure for Cloud Data. *IEEE Transactions on Information Forensics and Security* 12, 10 (2017), 2402–2415.

[33] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. 2013. Practical Dynamic Proofs of Retrievability. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. Association for Computing Machinery, New York, NY, USA, 325–336.

[34] William Stallings. 2017. *Cryptography And Network Security: Principles and Practice, 7th Edition*. Pearson Education, Upper Saddle River, NJ, USA.

[35] IPFS team. 2020. "ipfs add" calculate CID and check for it *before* uploading to API server. https://github.com/ipfs/kubo/issues/7586 Accessed: 2022-12-29.

[36] IPFS team. 2022. Garbage collection. https://docs.ipfs.tech/concepts/persistence/#garbage-collection Accessed: 2022-12-29.

[37] IPFS team. 2022. Persistence, permanence, and pinning. https://docs.ipfs.io/concepts/persistence/#persistence-permanence-and-pinning Accessed: 2022-12-29.

[38] Swarm team. 2021. Storage and Communication Infrastructure for a Self-Sovereign Digital Society. https://www.ethswarm.org/swarm-whitepaper.pdf Accessed: 2022-12-29.

[39] Swarm team. 2022. Bee is a Swarm client implemented in Go. https://github.com/ethersphere/bee Accessed: 2022-12-29.

[40] Swarm team. 2022. Swarm Gateway. https://gateway.ethswarm.org/ Accessed: 2022-12-29.

[41] Hui Tian, Yuxiang Chen, Chin-Chen Chang, Hong Jiang, Yongfeng Huang, Yonghong Chen, and Jin Liu. 2017. Dynamic-Hash-Table Based Public Auditing for Secure Cloud Storage. *IEEE Transactions on Services Computing* 10, 5 (2017), 701–714.

[42] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. 2022. Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web. In *Proceedings of the ACM SIGCOMM 2022 Conference*. Association for Computing Machinery, New York, NY, USA, 739–752.

[43] Viktor Trón. 2020. The Book of Swarm - v1.0 pre-release 7 November 17, 2020. https://www.ethswarm.org/The-Book-of-Swarm.pdf Accessed: 2022-12-29.

[44] Boyang Wang, Baochun Li, and Hui Li. 2014. Oruta: Privacy-Preserving Public Auditing for Shared Data in the Cloud. *IEEE Transactions on Cloud Computing* 2, 1 (2014), 43–56.

[45] Cong Wang, Sherman SM Chow, Qian Wang, Kui Ren, and Wenjing Lou. 2013. Privacy-Preserving Public Auditing for Secure Cloud Storage. *IEEE Trans. Comput.* 62, 2 (2013), 362–375.

[46] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. 2010. Toward Publicly Auditable Secure Cloud Data Storage Services. *IEEE Network* 24, 4 (2010), 19–24.

[47] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. 2012. Toward Secure and Dependable Storage Services in Cloud Computing. *IEEE Transactions on Services Computing* 5, 2 (2012), 220–232.

[48] Hao Yan, Jiguo Li, Jinguang Han, and Yichen Zhang. 2017. A Novel Efficient Remote Data Possession Checking Protocol in Cloud Storage. *IEEE Transactions on Information Forensics and Security* 12, 1 (2017), 78–88.

[49] Anjia Yang, Jia Xu, Jian Weng, Jianying Zhou, and Duncan S Wong. 2021. Lightweight and Privacy-Preserving Delegatable Proofs of Storage with Data Dynamics in Cloud Storage. *IEEE Transactions on Cloud Computing* 9, 1 (2021), 212–225.

[50] Kan Yang and Xiaohua Jia. 2012. Data storage auditing service in cloud computing: challenges, methods and opportunities. *World Wide Web* 15, 4 (2012), 409–428.

[51] Jiawei Yuan and Shucheng Yu. 2013. Proofs of Retrievability with Public Verifiability and Constant Communication Cost in Cloud. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 19–26.

[52] Yan Zhu, Huaixi Wang, Zexing Hu, Gail-Joon Ahn, Hongxin Hu, and Stephen S Yau. 2011. Dynamic Audit Services for Integrity Verification of Outsourced Storages in Clouds. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. Association for Computing Machinery, New York, NY, USA, 1550–1557.