Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization

Racin Nygaard
racin.nygaard@uis.no
University of Stavanger
Stavanger, Norway

Vero Estrada-Galiñanes
vero.estrada@epfl.ch
EPFL
Lausanne, Switzerland

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

ABSTRACT

In cryptographic decentralized storage systems, files are split into chunks and distributed across a network of peers. These storage systems encode files using Merkle trees, a hierarchical data structure that provides integrity verification and lookup services. A Merkle tree maps the chunks of a file to a single root whose hash value is the file's content-address.

A major concern is that even minor network churn can result in chunks becoming irretrievable due to the hierarchical dependencies in the Merkle tree. For example, chunks may be available but can not be found if all peers storing the root fail. Thus, to reduce the impact of churn, a decentralized replication process typically stores each chunk at multiple peers. However, we observe that this process reduces the network's storage utilization and is vulnerable to cascading failures as some chunks are replicated 10× less than others.

We propose *Snarl*, a novel storage component that uses a variation of alpha entanglement codes to add user-controlled redundancy to address these problems. Our contributions are summarized as follows: 1) the design of an entangled Merkle tree, a resilient data structure that reduces the impact of hierarchical dependencies, and 2) the Snarl prototype to improve file availability and storage utilization in a real-world storage network. We evaluate Snarl using various failure scenarios on a large cluster running the Ethereum Swarm network. Our evaluation shows that Snarl increases storage utilization by 5× in Swarm with improved file availability. File recovery is bandwidth-efficient and uses less than 2× chunks on average in scenarios with up to 50 % of total chunk loss.

CCS CONCEPTS

• **Computer systems organization** → **Availability**; **Redundancy**; Reliability; • **Information systems** → **Storage recovery strategies**; *Distributed storage*; *Storage replication.*

KEYWORDS

storage utilization, entanglement codes, erasure codes, cryptographic decentralized storage system, file availability

Middleware '21, December 6–10, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8534-3/21/12.
https://doi.org/10.1145/3464298.3493397

ACM Reference Format:
Racin Nygaard, Vero Estrada-Galiñanes, and Hein Meling. 2021. Snarl: Entangled Merkle Trees for Improved File Availability and Storage Utilization. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3464298.3493397

1 INTRODUCTION

Large-scale decentralized peer-to-peer (p2p) storage systems, such as *InterPlanetary File System* (IPFS) [2] and *Ethereum Swarm* [28], aim at providing novel services to satisfy current and future storage and communication needs. In such p2p systems, integrity verification plays a pivotal role [23], as peers are untrusted and the network may exhibit significant churn. Further, each peer typically only stores a subset of each file in fixed-sized chunks, and each chunk is given a content address for later retrieval. A content address is unique; it is the cryptographic hash of the chunk's data. To assist in this integrity verification and retrieval, a Merkle tree [18] data structure is typically used, where the tree's nodes are distributed across the peers of the p2p network based on their content address.

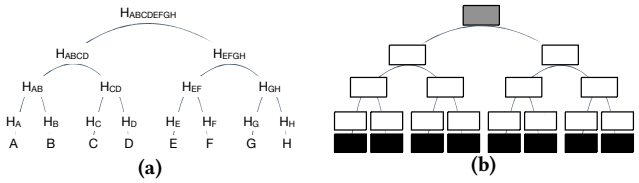

Figure 1: Merkle tree: (a) Binary Merkle tree built with 8 data chunks (A-H); (b) content addressing application.

A Merkle tree is a cryptographic data structure that maps multiple elements (nodes) to a single root node, as seen in Figure 1a. Internal nodes are obtained by hashing their children recursively using a bottom-up approach, so the root hash value covers the entire tree. The tree's integrity can be verified using the root hash value as any modification is propagated up. Merkle trees are widely used as they minimize audit costs and facilitate a wide range of applications, as mentioned in §2.

Figure 1b shows a file in a content-addressing system, such as the aforementioned Swarm and IPFS, where all content is placed at the lowest level of the tree, while the internal nodes and root are metadata required for lookup. Any given file will be represented by a unique Merkle tree, where the chunks of the file are the content at the lowest level. Internal nodes and the root are also stored as chunks in the p2p network, and contain a concatenation of the content addresses of their children. Thus, to retrieve the content,

we must first request the root using its content address and then recursively retrieve internal nodes until we reach the leaves. Consequently, the loss of the root or an internal node causes a cascading failure as it is impossible to request their children without the content address.

The data loss in real systems is further exacerbated because they use k-ary Merkle trees with a large branching factor, k. To illustrate the problem, Figure 2 depicts a radial visualization of the 128-ary Merkle tree for a 100 MB file stored in our 1000-peer Swarm cluster. The tree contains 24426 leaves, 191 internal nodes at level 1, two internal nodes at level 2, and the root at level 3. This figure also captures information about imbalances caused by the decentralized replication process in Swarm. Specifically, the replication factor (R) for each chunk is in the range of $R \in [9, 160]$. The root, labeled "1", has $R = 132$, with its two children labeled "2" and "3" having $R = 108$ and $R = 31$, respectively. This means that all data below internal node "3" is far more vulnerable than the data below "2".

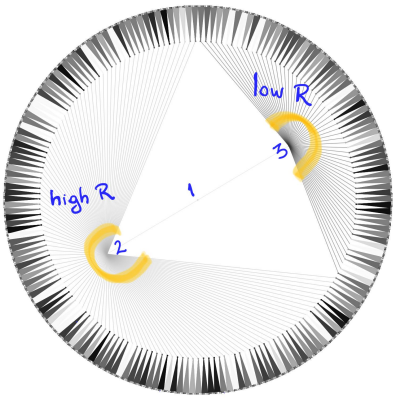

Figure 2: The 128-ary Merkle tree for a 100 MB file obtained in our cluster. The leaves are along the outer orbit and the root in the center. Edges connecting parents with children are colored according to their replication factor, with darker edges corresponding to weakly replicated parents.

1.1 Motivations for This Work

The Merkle trees used in decentralized storage systems introduce *hierarchical dependencies* between the content and the *metadata*, i.e., the content addresses used to locate the content. Hence, the root and internal nodes must also be available to ensure that the content is available. In other words, any replication added to the leaves is futile if the pointers to the leaves are not reachable. Previous experiences support our claim. Wilcox-O'Hearn admitted that about 90 % of all the content that has ever been stored in the Tahoe-LAFS [32] grids might have died despite using "erasure coding parameters with massive fault tolerance" [31]. He highlighted that administrators get a false impression by computing reliability under the assumption of independent failures because the numbers lead to many nines of reliability. In particular, for the standard 3-out-of-10 erasure code and $p = 0.90$ used in Tahoe-LAFS, we get 6 nines of reliability. He concluded that monitoring is more important than the redundancy parameters used in the system.

The assumption of independent failures is widely accepted despite that researchers have shown its pitfalls [22]. However, to the best of our knowledge, the risk of cascading failures in Merkle trees has not been studied. Previous research on cryptographic file systems focuses on the intersection between integrity and confidentiality. While some works combine Merkle tree and erasure coding or replication, the availability of the root and internal nodes has not been considered. Hence, we observe that the intersection between integrity and availability has not been sufficiently studied.

Coding algorithms are more storage efficient than replication [14, 21, 29], but they require *repair metadata* to decode. Decoding algorithms need some clue to start decoding, e.g., the index number of each coded block. A common solution is to use a metadata file or manifesto. For instance, Tahoe-LAFS treats the metadata as an additional erasure-coded file distributed on k servers [24]. Another approach is to store metadata in a local manifest. The caveat is that a local manifest is not publicly accessible and thus limits user collaboration. Hence, having to manage an additional metadata file is undesirable, especially in decentralized storage systems.

Some decentralized systems include incentive mechanisms to improve availability [6]. Currently, the Swarm Foundation is about to deploy an incentive layer that will create liabilities for chunk availability in the Ethereum Swarm network [26]. Still, redundancy mechanisms are critical for content survivability. However, it is well-known that systems built for fault tolerance in networks with untrusted peers use hefty amounts of redundancy [16, 20, 31]. Thus, *we posit that a method imposing low redundancy yet achieving high fault tolerance will improve the network's storage utilization.*

1.2 Contributions

We propose a novel solution that combines Merkle trees with alpha entanglement (AE) codes [7] to address the gaps above, aiming at high fault tolerance, low bandwidth, and low storage requirements. To the best of our knowledge, this is the first work that addresses the intersection between integrity and availability when using Merkle trees.

As with other codes, AE codes also require repair metadata. However, we observe that by exploiting synergies between the Merkle tree structure and the encoding/decoding algorithm, we can eliminate the need for a local or remote metadata file. Therefore, our solution only requires the content address of the tree's root to locate content even if it is encoded and distributed in a p2p network.

We present Snarl, another way of saying entangle. Snarl does not require any modification to the underlying storage system and seamlessly integrates with a local peer in a p2p storage network. Users interact with Snarl when uploading or downloading content. The content is encoded in an entangled Merkle Tree (eMT) structure before being uploaded to the network. Further, Snarl offers user-controlled redundancy in the sense that the encoding configuration can be adjusted to match the behavior of the storage network, i.e., higher network churn requires more redundancy.

We have implemented Snarl and deployed it in our 1000-peer cluster running Ethereum Swarm. Our results demonstrate that data integrity and availability for decentralized storage systems can be obtained without significant storage or bandwidth overhead. Our evaluation shows that Snarl is capable of simultaneously improving

the storage utilization and file availability in Swarm. Specifically, with the Snarl-14 configuration, **five times** the amount of data could be stored in the network with failure-resiliency comparable to Swarm. We show that file recovery is bandwidth-efficient in the presence of failures and uses less than 2.08× the chunks on average in scenarios with up to 50 % of total chunk loss.

2 PRELIMINARIES

Integrity and redundancy are interrelated. Systems introduce some form of redundancy to ensure data integrity [25]. Integrity checking requires comparing data with some derived piece of information to offer a certain level of assurance. For example, disk mirroring and checksums can detect integrity violations but cannot recover the original data. Assurance can be increased in a system that stores $N > 2$ copies using a majority vote, i.e., the majority of the copies are accepted as valid with some degree of confidence. Other widespread techniques are RAID parity and error detection and correction algorithms, which can recover data up to some level. For example, a (n, k) MDS erasure code encodes k data symbols into n coded symbols, and any k out of these n coded symbols can be used to decode the k data symbols. MDS stands for maximum distance separable codes and, in practice, it means that a (n, k) MDS code can recover data up to $n - k$ failures.

Hash functions generate a derived piece of information with negligible overhead. This property made them suitable to check correctness in memory stacks and queues [3]. Merkle trees [18] leverage the benefits of hash functions. Their original application was a digital signature system, where the growth of the signature is logarithmic with the number of messages signed. With its efficient verification of large amounts of data, it found its way into several other areas—particularly p2p-based systems, where it is necessary to verify the integrity of data received from untrusted parties. Often, cryptographic file systems, e.g., SiRiUS [10], TDB [15], Tahoe-LAFS [24], use Merkle trees to guarantee integrity. It is also used in blockchains like Bitcoin [19] and Ethereum [33], decentralized storage systems like IPFS [2] and Swarm [28], revision control systems like Git [4], and protocols like BitTorrent [5].

2.1 Swarm Overview

Swarm is a decentralized storage system that distributes stored data to a network of peers. The peer-to-peer network is based on Kademlia [17] for discovery and routing. By following the protocol, storing, and delivering content, peers are rewarded BZZ tokens through a smart contract on the Ethereum blockchain. According to a monitoring website [1], the public network has more than 270 000 peers.

Connecting to the Swarm network requires running a local peer or using a public gateway. Peers runs the Bee client, which is under active development at the time of writing. The Bee client will incorporate incentive mechanisms to reduce network churn. In this work, we used the more stable Swarm client, v0.5.8. We run a cluster with 1000 peers isolated from the public Swarm network.

Swarm splits files into 4 KB chunks and places them as leaves in a 128-ary Merkle tree. The internal nodes and root of the Merkle tree are also 4 KB chunks and contain a concatenation of the content addresses of their children. Chunks in Swarm are given a unique content address derived from a cryptographic hash function over the chunk's data. All chunks are distributed to peers whose address has the same prefix as the chunk's content address, also known as an address space neighborhood.

Redundancy is of significant importance in Swarm. It is deeply embedded in its design to provide fault tolerance, censorship resistance, DDoS resistance, and zero downtime.

Currently, the redundancy in Swarm is provided by a decentralized replication process called *syncing*. Syncing is separated into three mechanisms. First, *push-sync* involves transferring chunks from the uploader to storage peers, i.e., from the network's entry point to each chunk's corresponding address space neighborhood. Once the chunk reaches the neighborhood, it is replicated to all the peers within it. Second, when a new peer enters the network, it initiates *pull-sync* with its neighbors. When pull-syncing, the new peer queries its connected peers for a list of chunks they are storing and then proceeds to request those within its address space. Lastly, each time a chunk is transferred in the network, all peers that act as relays between the sender and receiver may choose to replicate the chunk.

2.2 System Requirements

Our primary design goal for Snarl is to increase the resilience of data stored in a cryptographic decentralized storage system without adding storage overhead. Snarl should be optional and user-controlled, meaning that the user should be able to specify the resilience level and choose how much of the resilience will be disclosed to the public. Snarl should enable file recovery despite the failure of a large part of the underlying storage system. Further, any peer should be able to repair efficiently, without overhead when there are no failures.

Snarl requires no modification to the underlying storage system. However, we specifically target storage systems that connect the stored data in a graph with a hierarchical dependency between nodes, typically a Merkle tree. We assume the presence of APIs to upload and download content.

3 ENTANGLED MERKLE TREE

Let $k\text{-}MT(d_1, \ldots, d_n)$ denote a *k-ary Merkle tree*, where data items $d_1, \ldots, d_n$ are at the leaves of the tree, and k is the branching factor limiting the number of children of a node. An internal node of the tree i_h at height h with children $c_1, \ldots, c_k$ is the hash of the concatenation of its children, i.e., $H(c_1 || \ldots || c_k)$, for H a collision-resistant hash function. If the concatenated leaves form a file f, we can refer to its Merkle tree as $k\text{-}MT_f$.

Cryptographic decentralized storage systems often split the content into chunks and provide integrity via a Merkle tree. The Merkle tree maps chunks to a single root hash value used as a content identifier and entry point for retrieval. However, a serious concern is that the hierarchical dependencies between chunks can render several chunks irretrievable, even though they are stored at available peers. This can happen if a critical storage peer fails, resulting in a logic failure cascade. To reduce the impact of these dependencies within Merkle trees, we aim to transform the Merkle tree into a failure-resilient structure.

To fulfill our aims, we define a *k-ary entangled Merkle tree* (*k-eMT*) data structure, constructed from a *k-MT* using these functions in sequence:

```
k-MT | mapper | swapper | entangler | k-eMT
```

Snarl implements these functions. The original *k-MT* contains information (the user file or any arbitrary content) at the leaf level. However, the mapper, swapper, and entangler consider all the nodes of the *k-MT* as information. The entangler generates α *k-eMT*s as output. These carry redundant information to recreate all nodes in the original *k-MT*.

We start by describing *k-eMT* in §3.1 and §3.2, followed by the *entangler* in §3.3 and §3.4. The *mapper* is described in §3.5, and lastly, the *swapper* in §3.6.

3.1 The *k-eMT*: Challenges and Solutions

Let *k-eMT*$(p_1, \ldots, p_m)$ denote a *k-ary entangled Merkle tree*, where items $p_1, \ldots, p_m$ are at the leaves of the tree, and m is the total number of nodes in the original *k-MT*. An item p_i is a parity obtained by the entanglement algorithm and is a shorten for a parity $p_{i,j}$, more details are explained in §4.4. An internal node of the tree i_h is computed as in *k-MT*.

Our *k-eMT* design solves the problem of handling metadata for decoding mentioned in §1. We avoid the use of extra metadata by carefully crafting the entangled chunks in a forest formed by the original *k-MT* and a few *k-eMT*s created with the entanglement. By leveraging content-addressing and AE codes, the decoding process only needs to know the roots. In fact, some roots may be kept private and released only when needed via a smart contract or some other medium.

Moreover, the forest reduces the hierarchical dependencies of the original *k-MT*, as with the addition of *k-eMT*s there are alternative paths for traversal. In addition, as will be detailed in §4, the new structure can recover from missing and corrupt nodes.

We highlight that the design of *k-eMT* is non-trivial. It involves mapping the elements of a hierarchical structure into a helical lattice structure constructed by the AE encoding algorithm. As we explain later, the process needs to remove specific dependencies between chunks, as otherwise, the protection offered by AE codes would be reduced due to the appearance of additional irrecoverable failure patterns.

Our solution is designed for *k-MT* with $k \gg 2$, see §3.6 for more details. This is consistent with real-world systems, e.g., Swarm uses $k = 128$ and IPFS uses $k = 174$. Large k generates a shallow tree that accelerates the lookup in content address networks.

3.2 Canonical Naming

To aid in the reconstruction of the tree, we devise a canonical naming scheme. A similar scheme was proposed by Merkle [18] for a 2-*MT* with the root node labeled 1, the left child of node i labeled $2i$ and right child of node i labeled $2i+1$. Our scheme supports k-ary branching, and also allows us to assign labels to nodes during initial chunking, without knowing the size of the data beforehand. The scheme is based on a post-order traversal algorithm, with naming starting at the leaf level (see §3.5 for an example).

3.3 Overview of Alpha Entanglement Codes

AE codes [7] are designed to tolerate a large number of failures with low computation and bandwidth requirements. The encoding algorithm produces *entangled chunks* embodying parities to be disseminated across the system. Assuming the chunks are distributed to different storage peers, the decoding algorithm can use multiple "paths" in the lattice structure to decode the content. Each path requires the availability of a set of distinct peers. Paths are formed by an n-length combination of data and entangled chunks, with $n \geq 2$. The shortest path to repair a single failure has length two. Multiple paths improve the access and recovery likelihood for temporarily and permanently unavailable content, respectively. As such, paths provide for an efficient recovery mechanism for high churn scenarios.

The entanglement process creates a graph of interdependent chunks. In its simplest form, where $\alpha = 1$, the graph is a path, usually referred to as a chain or strand. It starts with $e_{\_,1} = 0$, a dummy chunk and continues with $e_{1,2} = d_1$, and then alternates unencoded data chunks and entangled chunks. The strand is constructed by encoding the entangled elements $e_{i,j}$ according to: $e_{n,\_} = e_{n-1} \oplus d_n$, with $n > 1$ where d_i are data chunks presented as input to the encoding algorithm, and $e_{i,j}$ is adjacent to d_i and d_j. Note that d_i are the ordered chunks obtained by chunking a file f, or the chunks of any arbitrary data stream.

The Cylindrical Helical Lattice. With $\alpha > 1$, the graph becomes a *lattice* composed of intertwined strands, where each data chunk d_i belongs to α strands.

Let $LAT_\alpha(d_1, \ldots, d_n)$ denote an $AE(\alpha, s, p)$-*lattice* for $\alpha > 1$, where data items $d_1, \ldots, d_n$ are the lattice vertices v. LAT_α is a regular graph of degree 2α with d_i's position in the lattice according to: *top* v for $i \equiv 1 \pmod{s}$, *central* v for $i \not\equiv 1 \wedge i \not\equiv 0 \pmod{s}$, or *bottom* v for $i \equiv 0 \pmod{s}$. LAT_α is composed of $s + (\alpha - 1) \cdot p$ intertwined strands. Each strand can be constructed independently using the equations in §3.3; therefore, lattice construction can be parallelized for efficiency.

For $\alpha \in [2, 3]$, the lattice can be thought of as a weaker version of graph embedding on a cylinder, i.e., we relax the embedding definition by omitting the non-intersection condition for edges [12]. In LAT_3, p strands are cylindrical double-helix, denoted *RH- and LH-strands* (right-handed and left-handed helical strands), and the remaining s strands are in parallel with the cylinder axis, hence, denoted *H-strands* (horizontal strands). In LAT_2, the strands are not a double-helix since there are only $s + p$ intertwined strands, choosing RH- or LH-strand does not yield any difference. Helical strands revolve around the imaginary central axis of a cylinder. RH-strands connect vertices in cycles, alternating a sequence of a top vertex, $s - 2$ central vertices, and a bottom vertex. LH-strands connect vertices in cycles, alternating a sequence of a bottom vertex, $s - 2$ central vertices, and a top vertex. H-strands connect vertices of the same type.

Let $LW(d_i, \ldots, d_{i+s \cdot p-1})$ denote a *lead window* for $\alpha = 3$, where data items $d_{i+j \cdot (s+1)}$ with $j \in [0, p-1]$ are connected to the same helical strand. The lead window describes the interval it takes a helical strand to revolve around the cylinder's axis. The concept is similar to the pitch of a helix, i.e., the height of one complete

helix turn, measured parallel to the helix's axis. Given a flat representation of the lattice, we can say that if d_i is a top vertex, a lead window spans an area defined by s rows and p columns with its top-left vertex d_i.

3.4 A Variation for AE codes: A Toroidal Lattice

We propose a variation for AE codes based on closed entanglements [8]. Closed entanglements are entanglements with $\alpha = 1$ that generate a closed path with a slight modification to the algorithm to connect d_n with d_1 by recomputing $e_{1,2}$ and replacing the dummy chunk with $e_{n,1}$. The objective of closing the path is to better protect the elements at the extreme of the path. The original AE codes, however, used an "open" lattice. We modified the original design by closing each path that forms the lattice. As a result, the cylindrical helical lattice is transformed into a toroidal lattice. The subtleties of the closing are related to the number of nodes, N, in the tree. If N is multiple of $s \cdot p$ (the number of vertices in a lead window), the closing is straightforward. For other cases, the connections added to close the lattice prioritize connecting vertices from the same path. Figure 3 illustrates the toroidal lattice for a case where N is not multiple of $s \cdot p$.

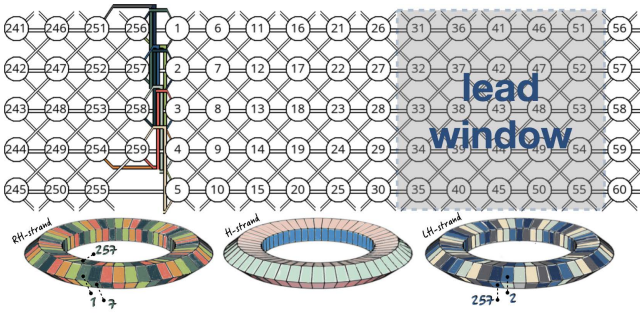

Figure 3: Toroidal lattice for a Merkle tree with 259 nodes.

3.5 Mapping the Tree Into a Lattice

Mapping is the first of two steps needed to prepare the input for the entanglement. The encoding algorithm (entangler) takes an input stream of "ordered" chunks (information), i.e. entangler is deterministic and produces an output that depends on the order of the input chunks. Internally, the entangler creates LAT_α, where its vertices $d_1, \ldots, d_m$ correspond to the input stream (m is the total number of k-MT nodes and not just the leaves). All nodes are treated as information in our redundancy scheme in order to create redundancy for the internal nodes and the root too. Each vertex represents a distinct chunk. The order of the input chunks is reflected in the way the lattice unfolds.

To map the tree into the lattice, the *mapper* uses a post-order traversal algorithm that reads the tree and generates the input stream. To illustrate, reading the 2-MT showed in Figure 1a, $MT_f(A, \ldots, G)$ produces the ordered input: $A \rightarrow B \rightarrow H_{AB} \rightarrow C \rightarrow D \rightarrow H_{CD} \rightarrow H_{ABCD} \rightarrow E \rightarrow F \rightarrow H_{EF} \rightarrow G \rightarrow H \rightarrow H_{GH} \rightarrow H_{EFGH} \rightarrow H_{ABCDEFGH}$. Using our canonical naming, the elements of the sequence are labeled $1, 2, 3, \ldots, 15$. Therefore, this example creates a LAT_α with 15 vertices. The number of strands in LAT_α is defined by the entangler (obtained by the coding parameters). Before, we need the swapper to finish preparing the input for the entangler.

3.6 Swapping

Swapping is the second and final step to prepare the input for entanglement. Nodes that have a parent-child relationship cannot be adjacent or in a close neighborhood in the LAT_α. The reason for this is to avoid that a parent is entangled with its children or with their neighbors. If that occurs, and the chunk represented by the parent node is missing, then it is not possible to retrieve the elements in the lattice that are used to recover the missing chunk. The swapping algorithm (swapper) moves the parents at least one LW away from their children. The swapper looks for a candidate vertex position where none of the vertices in its neighborhood are children of the swapped vertex.

4 SNARL

The high-level architecture comprises three components: Snarl, a proxy, and the underlying storage system. Figure 4 shows the architecture and the main packages of Snarl.

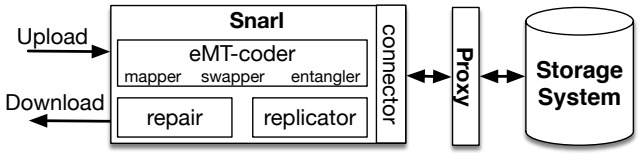

Figure 4: Snarl architecture.

Snarl is designed to be a general-purpose tool for providing user-controlled redundancy to any Merkle tree, and requires no modification to the underlying storage system. Snarl uses a modular approach and comprises four packages: (i) *eMT-coder* implements encoding and decoding algorithms for the eMT (§3). (ii) *repair* contains the algorithms needed to recover from failures. (iii) *replicator* combines eMTs and replication to further improve storage utilization. (iv) *connector* abstracts the low-level details of the storage system.

The repair algorithm locates parity chunks in a bandwidth-efficient manner and is only applied when failures are detected. When downloading content not encoded with an eMT, Snarl reverts to the retrieval mechanism of the underlying storage system. Likewise, a non-Snarl user can download content encoded with an eMT, as the original Merkle tree is untouched and reachable.

Snarl comprises 4800 lines of Go code and 2200 lines for testing and benchmarking purposes. We have made the code available in our GitHub repository (https://github.com/relab/snarl-mw21).

4.1 User Interaction With Snarl

The user interacts with Snarl using the command-line. To encode content – without interacting with the storage system, we provide the *entangle* command. The entangle command takes as input the content that needs protection and the desired level of protection. It will generate α files, one for each eMT, and no other metadata.

To interact with the storage system, the commands *download* and *upload* are used. Upload calls entangle internally before uploading the original Merkle tree and α eMTs to the storage system. After

each upload, the storage system replies with the content address of the root. The user must persist the content addresses of the roots for the original Merkle tree and the eMTs, as they are necessary to retrieve the content later. Each content address is small; usually, 32 bytes, depending on the underlying storage system.

To download content from the storage system, a user must supply the original Merkle tree root's content address. When downloading through Snarl, a user may also pass the content address of the eMTs root, allowing Snarl to repair the original content if chunks are missing. The repair process is seamless for the user, downloading parity chunks on-demand and reconstructing the content, despite widespread data loss.

4.2 Snarl Interaction With the Storage System

This section explains how Snarl interacts with the underlying storage system before detailing how we successfully deployed Snarl using Swarm as the storage system.

Snarl does not require any change to the underlying storage system. To achieve this, we assume a *proxy* that exposes APIs to upload and download content. In a decentralized storage system, the proxy will be a peer participating in the network. As we expect the various storage systems' APIs to be slightly different, we provide a set of interfaces in our *connector* package. This allows a separation of concerns in the other packages, as all storage system-specific details are implemented in separate packages.

To satisfy the connector package's interfaces, specific details such as the Merkle tree branching factor, its balancing algorithm, and the maximum chunk size must be implemented. By designating this to its own package, the behavior can be mirrored either by calling the proxy, importing a library, or as a separate implementation of its specification.

Swarm as the Storage System. Swarm uses a 128-*MT* construction for each file. A chunk's maximum payload size is 4096 bytes, with an extra 8 bytes describing the accumulated size of all its leaves if it is an internal node, otherwise its length. The tree is constructed from left to right ensuring that all non-leaves, except the righter-most, are of the same size and height. These 8 bytes also subverts the *length extension attack* [30] and can be used to determine the canonical index of the node without exploring the tree. At the same time, the 8 bytes create some difficulty, as our eMT is limited by the same 4096 bytes. Thus, we cannot encode the chunk size information in the eMT, as we would have to fit 4104 bytes inside 4096. Instead, we observe that since the tree's construction is deterministic, the chunk size can be derived from the size of the eMT and the chunk's position.

4.3 Local Repair Information

The edges of the toroidal lattice outlined in §3.4 are the parity chunks needed for repair, and knowledge of their content address is the only way to retrieve them.. The mapping of parities to content addresses is called *repair metadata* and is used by Snarl when requesting parities.

As the size of repair metadata grows linearly with the content's size, it could be impractical to store locally for large files. Further, it hinders *collaborated repairs*, as only those with access to the metadata can partake in repairs. Storing the metadata in the storage system itself is also a bad idea, as the original content's resilience would be reduced to the single failure of the metadata.

In Snarl, we instead implement the eMT from §3, which allows requesting correct parities knowing only the root chunk's content address. An eMT has the same number of leaves as there are chunks in the original Merkle tree. The leaves are ordered such that the left parity for the chunk with canonical index n is leaf n in an eMT. To find the leaf index for the right parity for the chunk, we implemented an algorithm according to §3.3.

We now illustrate how to retrieve the desired leaves from the $k\text{-}eMT(d_1, \ldots, d_n)$. To retrieve a leaf d_i from the $k\text{-}eMT$, we first request its root to know the content addresses for their children. Each child is a leaf or an internal node that points to a subtree. Every subtree of a parent is of equal size, apart from the right-most. To calculate the subtree's size, we realize that their size must be a power y of the branching factor k, where y is the chunk's level, starting at 0 on the leaves. Thus, given a parent at level h, each parent's child will have size k^{h-1}, apart from the right-most. Therefore, to find a path to leaf x, we traverse through the g-th sibling defined by $g = \lceil \frac{x}{k^{h-1}} \rceil$. This continues recursively by redefining $x = x - (g-1) \cdot k^{h-1}$ for each step, until we reach the desired leaf. The pseudo-code is given in Algorithm 1.

Algorithm 1 Parity retrieval from an entangled Merkle tree

```
1: func GetParity(parentAddr, leafId)
2:   parent ← Download(parentAddr)
3:   childSize ← parent.branchFactor ∧ (parent.level − 1)
4:   if childSize = 1 then                    ▷ All children are leaves
5:     return Download(parent.child[leafId])
6:   next ← math.Ceil(leafId / childSize)     ▷ g
7:   leafId ← leafId − (next − 1) · childSize ▷ x
8:   return GetParity(parent.children[next], leafId)
```

4.4 Repairing Failures

Snarl can detect and repair failures due to both corrupt and missing chunks. It is agnostic to the reason for a failure, whether it is due to data loss, network partition, malicious behavior, or churn. Integrity verification of chunks is done by comparing the cryptographic hash of the received data with the requested content address. Therefore, we treat corrupt chunks the same way as missing chunks.

A repair begins as soon as a failure is detected during a download and runs concurrently with the remaining download. Snarl uses the same algorithm to repair any node of the Merkle tree. It is intelligently aware that the loss of an internal node prevents downloading any of its children and will thus give priority to repairing internal nodes first.

We illustrate the repair algorithm in Figure 5 with three example scenarios. For illustrative simplicity, we use a Merkle tree with three levels and a branching factor of 14.

Snarl starts by requesting the root of the original Merkle tree, labeled "25" in Figure 5d. Then it proceeds through the internal nodes, labeled "15" and "24", to the leaves, which hold the content. If a chunk loss or corruption is detected, even at the root chunk, Snarl immediately requests parity chunks from the eMTs, also shown

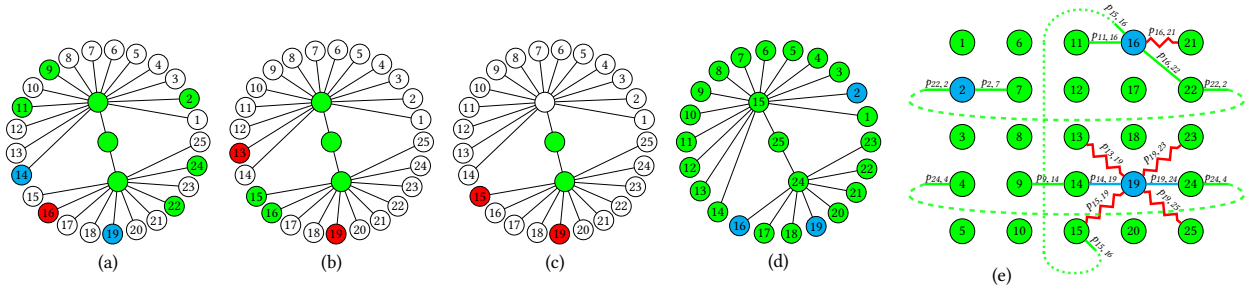

Figure 5: Repair algorithm: (a,b,c) horizontal (H-eMT), right (RH-eMT) and left (LH-eMT) entangled Merkle trees, (d) original Merkle tree, (e) lattice. Failed downloads are colored with red, successful downloads are colored in green. Repairs are colored with cyan. White is not requested. Users can collaborate to maintain the system by re-uploading repaired chunks.

in Figure 5 with three strand types ($\alpha = 3$), (a) H-eMT, (b) RH-eMT, and (c) LH-eMT. The lattice shown in Figure 5e is a virtual data structure used to coordinate repairs. Each vertex in the lattice represents a chunk from the original Merkle tree. The parities in the lattice are illustrated as edges labeled $p_{X,Y}$, where X is the vertex connected on the left, and Y is the vertex connected on the right. When downloading a parity, we request the n-th leaf chunk of an eMT of the correct strand type, where n is the left-connected vertex's index.

Using the toroidal structure of the lattice, Snarl can repair in many failure scenarios in which the underlying storage system cannot. After successful repair, the user can re-upload the repaired data and parity chunks to the storage system, thus aiding system-wide maintenance.

We consider three example failure scenarios. First, the most basic scenario, where a single data chunk is lost. As soon as Snarl cannot download data chunk "2", it requests parities $p_{2,7}$ and $p_{22,2}$ from H-eMT. Both are available, and data chunk "2" is repaired with two parity downloads.

In the second scenario, data chunk "16" is lost, and when attempting to download the parities $p_{11,16}$ and $p_{16,21}$ from H-eMT, Snarl discovers that $p_{16,21}$ is unavailable. Snarl then requests the two adjacent parities from the RH-eMT, i.e., parities $p_{15,16}$ and $p_{16,22}$. Both are available, and data "16" is repaired with three parity downloads.

Data chunk "19" is lost in the last scenario, including all its adjacent parities in the H-eMT, RH-eMT and LH-eMT. Snarl can recover from such a scenario by repairing one of the parity pairs. It requests parity $p_{9,14}$ and uses that together with data "14" to repair $p_{14,19}$. Similarly, parity $p_{24,4}$ and data "24" are used to repair $p_{19,24}$. Thus, with a complete parity pair, data "19" can be repaired with only two parity downloads.

During repair, Snarl will expand both vertically and horizontally in the lattice until either the chunk is repaired or an irrecoverable pattern is detected. The expansion is recursive and follows the order; H, RH, and LH, referring to the eMT from which parity chunks are requested first.

4.5 Replicating Entangled Merkle Trees

It has been shown that a combination of erasure codes and replication [9] generally achieves better storage utilization and lower repair overhead than the two methods separately. With Snarl, we propose a similar combination of eMTs and replication.

We distinguish between replication of internal and leaf chunks. These must be weighed against each other, as a higher replication factor for internal chunks improves the likelihood that a leaf chunk can be located but does not contribute to the repair algorithm. On the other hand, a higher replication factor for leaf chunks does contribute to the repair algorithm. However, a loss of their parent results in the de facto loss of all leaves.

We use (1) as a measure to balance the replication of internal versus leaf chunks. Equation (1) captures the likelihood that a chunk replicated r times, with each replica placed on a distinct peer, would still be available after f peers failed in a network with n peers. We observe that adding replicas is an effective measure to increase recovery likelihood if the replication factor is low. However, at higher replication factors, the gains of adding another replica approach 0. In fact, with only 45 replicas, the recovery likelihood is more than 99.9 %, with up to 86 % peer failure.

$$P(x) = 1 - \prod_{i=0}^{n-f-1} \frac{n-r-i}{n-i} \tag{1}$$

Snarl can scale the replication factor, allowing a user to specify the total storage consumption of the encoded data to be proportional to uniformly replicating the original Merkle tree. Further, the weighting of the replication factor for internal and leaf chunks can also be adjusted. By default, we cap the number of internal chunk replicas at 45. To differentiate between Snarl configurations, we label them with the ratio of storage consumption compared to the original Merkle tree. For example, with Snarl-5, we use the same storage consumption as having 5 copies of the original Merkle tree.

5 EVALUATION

We evaluate Snarl using the toroidal lattice variant of closed entanglements described in §3.4. The encoding parameters are $\alpha = 3$, $s = 5$, $p = 5$, as it has been previously studied in the literature [7]. It's worth noting that Snarl allows the user to adjust these parameters to fine-tune the performance and redundancy to its requirements. We compare Snarl with full uniform replication, where every chunk has the same number of copies, and with Swarm's default redundancy.

In our first evaluation, we studied how files in Swarm are replicated in our cluster. We then evaluate the chunk distribution of files encoded with Snarl. Next, we study Snarl's file availability on our cluster of 1000 Swarm peers to show how Snarl can increase storage utilization by over 80 %. We show that the encoding speed of Snarl is linear, requiring only 3.6 seconds for a 1 GB file. The effectiveness of our repair algorithm is evaluated by examining bandwidth overhead incurred in different failure scenarios.

5.1 Experimental Setup

We ran our experiments on a cluster of 30 machines running Ubuntu 18.04.4 LTS. Each machine is equipped with Intel Xeon E-2136 3.30 GHz CPU, 32 GB RAM, 1.5 TB SSD disk, and 1 Gbit/s NIC. We used Helm [11] and Kubernetes [13] to distribute 1000 Swarm peers on 28 machines. We use the remaining two machines to host the Snarl client, the bootstrapping peer, and manage the experiment execution.

The bootstrapping peer allows the Swarm peers to discover each other and to achieve their desired connectivity. Swarm's *maxpeers* parameter is left unchanged, allowing a peer to connect with up to 50 peers in the network.

Each chunk's replication factor is highly variable in Swarm, as we discuss in §5.2. To compensate for this variability and make the comparisons fair, we need to adjust each chunk's replication factor to match the evaluated coding scheme. To facilitate these adjustments, we have developed a set of tools.

Our first tool, *listchunks*, lists the content addresses of all the chunks of each file in our storage network. The second tool, *deletelist*, determines which chunks must be deleted from which peer. To evaluate files of different sizes, we must run listchunks and deletelist for each file. To obtain an aggregated delete list, we use a third tool, *combinelist*. Finally, the aggregated list is passed to *deletechunks*, to delete the chunks on the peers in the storage network.

Swarm has three mechanisms for chunk distribution, also called *syncing*. Two of these mechanisms, *push-syncing*, and *pull-syncing*, can be disabled with the *no-sync* command-line option when starting a peer. However, the syncing process for chunks delivered through the Kademlia [17] DHT cannot be disabled. This posed a challenge for us, as the increased chunk replication would unduly skew the results for the different coding schemes.

Hence, to ensure an identical system state across experiment runs, we must counteract this built-in syncing process. To that end, we create a snapshot of the entire storage network and recover from a snapshot between each experiment run. Further, we ensure that each peer is well-connected before each experiment run. To reach sufficient connectivity, the peers must first discover each other. We monitor the discovery process by periodically polling Swarm's inter-process communication file, *bzzd.ipc*. As soon as the desired connectivity is reached, we start the next experiment run.

5.2 Replication in Swarm

Our study reveals that the replication factor (R) in the Swarm network is not uniform at the chunk level, meaning that some chunks are more replicated than others. However, the file size does not appear to impact R notably. For file sizes 1 MB, 10 MB, and 100 MB, we found that each chunk was in the range $R \in [9, 162]$, with an average R of 72. The relatively high average R of 72 means that for every gigabyte of original data stored in the network, the collective resources of the storage peers will consume at least 72 gigabytes.

We plot the relative chunk replication for a 100 MB file in Figure 6. The file consists of 25803 unique chunks, with a R that ranges from 9 for 26 chunks (0.1 %) to 162 for 3 chunks (0.01 %), with an average R of 72.36. The horizontal axis shows the replication factor, and the vertical axis shows the relative frequency of occurrence.

Figure 7 shows how the chunks are distributed on the storage peers by plotting the number of chunks each peer stores for the same 100 MB file. We have sorted the peers in ascending order of the number of chunks stored. Interestingly, the values appear to follow a power-law distribution. The number of chunks stored ranges from 105 on peer 1 to 12865 on peer 1000, with an average number of chunks stored 1867.

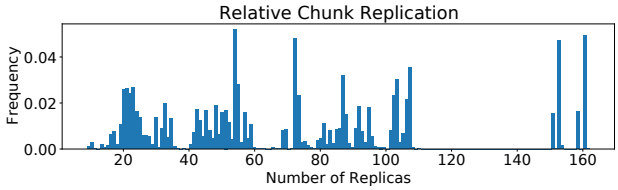

Figure 6: Relative chunk replication for a 100 MB file stored in a Swarm network consisting of 1000 peers.

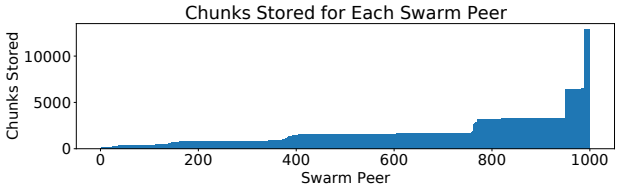

Figure 7: Storage consumption for a 100 MB file in a Swarm network consisting of 1000 peers. Peers are enumerated on the horizontal axis, sorted by the number of chunks stored.

5.3 Chunk Distribution

We observe that a file encoded with Snarl occupies a broader range of the address space than with full replication.

By mapping the first two bytes of the content address for each chunk, we get an idea of how a file would be distributed in the network, as chunks are placed at peers that share the same prefix. We repeated the experiment 10000 times with randomly generated files and report the average result.

For a 1 MB file in Swarm, there are 259 unique chunks with Swarm's full replication, with 258 distinct prefixes, meaning that 2 chunks share the same two first bytes in their address. Snarl encoding generates 1048 unique chunks, with 1039 distinct prefixes.

A file encoded with Snarl is evenly spread and occupies 1.59 % of the address space, while a replicated file occupies only 0.39 %. Consuming more address space is advantageous to mitigate an

attack attempting to make a file unavailable by monopolizing a small section of the address space.

5.4 File Availability

Next, we empirically estimate Snarl's file availability from the recovery likelihood for different failure scenarios. We compare the results of Snarl to full uniform replication, where each chunk in the Merkle tree is replicated the same number of times. We also compare with Swarm, which has a chunk distribution similar to that shown in Figure 6. In the case of replication, it is clear that the file can be recovered as long as at least one replica of each chunk that composes the file is available. We conduct two types of evaluations for file availability. In the first experiment, we obtain the recovery likelihood by artificially marking some percentage of *chunks unavailable* and trying to recover the file. The second experiment is similar, except that we mark some percentage of *storage peers unavailable*. Our evaluations show that Snarl provides a higher recovery likelihood for a lower storage consumption than the case for full replication. We list the storage consumption used by each scheme in Table 1.

5.4.1 Chunk Loss. We compare files of sizes 1, 10, and 100 MB encoded with Snarl-5, Swarm with replication factor 5 (R-5), and replication factor 10 (R-10). We run 10000 iterations of each experiment, with new random input data for each. Snarl-5 used for this evaluation has the same storage consumption as R-5; thus, we use this as the baseline for our evaluations.

Because we are only interested in recovering the entire file, a single chunk loss will make the file unavailable. Hence, the recovery likelihood is expected to decrease as the file size increases. As the number of chunks that compose a file increases, given the same chunk loss percentage, the likelihood of an irrecoverable error also increases. From Figure 8, we can observe that the impact on recovery likelihood for larger files is less for Snarl than for full replication for all three file sizes.

From Figure 8, we can see that Snarl-5 has a significantly higher recovery likelihood than both R-5 and R-10, with up to 50 % chunk loss. Given 45 % chunk loss, Snarl-5 has a recovery likelihood of 99 % for a 1 MB file, while R-5, in comparison, has 1 % recovery likelihood, and R-10 92 %.

Table 1 summarizes the maximum chunk loss percentage that the redundancy scheme can tolerate and still provide 99 % recovery likelihood.

5.4.2 Peer Failure. We now evaluate Snarl deployed with 1000 peers, where each peer stores some of the chunks that compose the file. For this evaluation, we use Snarl-5 and Snarl-14. We choose Snarl-14 to evaluate the improved storage utilization Snarl offers, as Snarl-14 has a storage consumption roughly 81 % lower than regular files in Swarm. For each experiment we run 10000 iterations.

We can see from Figure 9 that Snarl-5 outperforms both R-5 and R-10, even considering that R-10 requires twice the storage. Interestingly, Snarl-14 outperforms the redundancy provided by Swarm while using less than 20 % of the storage. In other words, if every file in Swarm were encoded with Snarl, storage utilization would increase so that five times the amount of data could be stored in the network, with improved resiliency against failures.

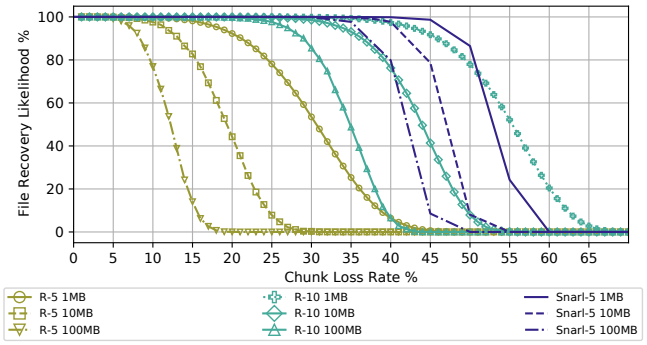

Figure 8: Recovery likelihood for various chunk loss rates.

The recovery likelihood of replicated files for R-5, R-10, and Swarm, seems to be largely unaffected by the file size. This is because we need all chunks that compose the Merkle tree to recover the file, combined with the fact that chunks in Swarm are not uniformly distributed over the entire network. In other words, chunks are placed at peers with similar addresses, and therefore we are evaluating the likelihood that all peers in at least one address region are failing. Because chunks for various file sizes in Swarm are distributed to the same address region, the recovery likelihood is independent of file size, at least for files 1 MB and up.

Table 1 summarizes the maximum peer failure percentage that the redundancy scheme can tolerate and still provide 99 % recovery likelihood.

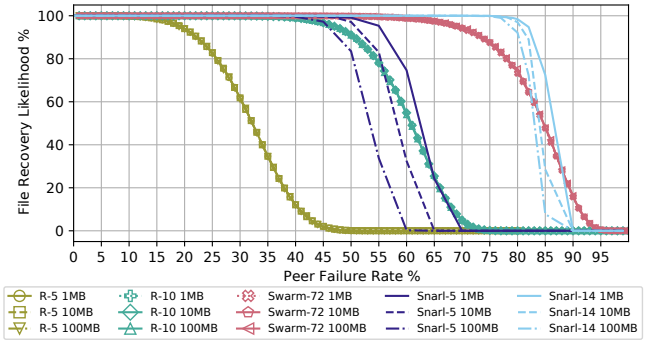

Figure 9: Recovery likelihood for various peer failure rates.

5.5 Encoding Speed

We evaluate the encoding performance of Snarl by measuring the time for encoding files from 1 MB to 1 GB. We ran this experiment on a single machine in our cluster.

Since encoding is based on lightweight XOR operations, it can be implemented efficiently. We iterate over the file, and for every chunk, we XOR with the previously added chunks to create the parity chunks. We keep the last parity chunks in memory to perform the XOR operation with the newly added chunk to create the new parity. The cumulative time it takes to execute the XOR operations is the most significant factor for larger file sizes. As expected,

Table 1: Snarl performance when encoding files of different size: storage consumption given in MB and total chunks number, the number of internal nodes in the *k-MT* and the α *k-eMT* unique and total nodes including replicas, the maximum chunk loss/peer failure for 99 % recovery likelihood per redundancy scheme. Higher values are better for max. chunk loss/peer failure.

	Storage Consumption: MB::>total chunks			Internal Nodes: unique::>total			Max. Chunk Loss			Max. Peer Failure		
Scheme	1 MB	10 MB	100 MB	1 MB	10 MB	100 MB	1 MB	10 MB	100 MB	1 MB	10 MB	100 MB
R-5	5.06::>1295	50.41::>12905	503.96::>129015	3::>15	21::>105	203::>1015	14 %	8 %	5 %	14 %	14 %	14 %
R-10	10.12::>2590	100.82::>25810	1007.93::>258030	3::>30	21::>210	203::>2030	37 %	29 %	22 %	40 %	40 %	40 %
Snarl-5	5.06::>1295	50.41::>12905	503.96::>129015	15::>165	87::>2784	818::>33518	45 %	38 %	34 %	50 %	47 %	43 %
Snarl-14	13.15::>3626	131.07::>36134	1310.31::>361242	15::>330	87::>3915	818::>36810				79 %	78 %	77 %
Swarm-72	72.60::>18585	723.69::>185264	7292.91::>1866986	3::>104	21::>1548	203::>14258				59 %	59 %	59 %

the encoding time is linear with the file size; a file of 1 MB takes 3.8 milliseconds, a 10 MB file takes 37 milliseconds, a 100 MB file takes 360 milliseconds, and a file of 1 GB takes 3.6 seconds.

Decoding is also based on XOR, where two parity chunks are used as input to the XOR operation to reconstruct a data chunk. Thus, decoding a file requires using the same number of XOR operations as chunks in the file, resulting in linear time complexity. The actual time required for decoding, however, will largely depend on the network latency.

5.6 Network Overhead

We have previously outlined the repair algorithm in §4.4. As long as there are no failures, there is no need for any parity chunks. Only when failures occur will Snarl start requesting necessary parity chunks from the network. Each data chunk is connected with three parity pairs in the lattice, as shown in Figure 5e. At best, a single failure or missing data chunk can be repaired with only a single pair. Both chunks in the pair are requested in parallel, and if either of them is unavailable, the next pair will be requested. However, in the case none of the three pairs can be downloaded, Snarl will expand outwards in the lattice, requesting additional parities to repair the parity pairs. This process is recursive and terminates only when the data chunk is repaired or an irrecoverable pattern is detected.

Therefore it is crucial that Snarl is bandwidth-aware and only requests the minimal number of parities necessary for successful retrieval. In addition, Snarl should be able to traverse the eMT to find the correct parities needed for each type of failure.

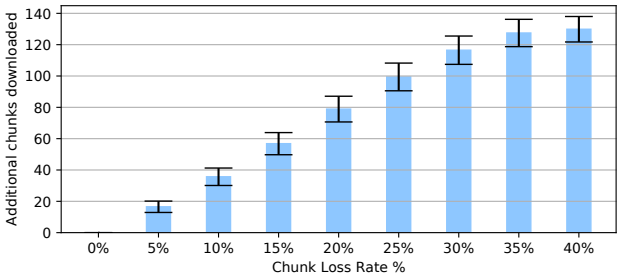

Figure 10: Download overhead for a 1 MB file in Snarl with increasing levels of chunk loss.

We measure the bandwidth efficiency of Snarl by counting how many additional chunks are downloaded compared to Swarm with no failures. Figure 10 shows how many additional chunks were downloaded for various amounts of chunk loss in a 1 MB file. Each experiment was executed with 10000 iterations. The error bars show the standard deviation. We can see that Snarl does not retrieve any parities when there are no failures, and with failures present, the number of chunks downloaded grows linearly with the failure rate.

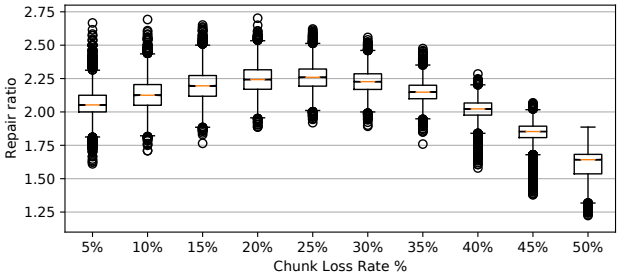

Figure 11: Repair overhead for a 1 MB file in Snarl with increasing levels of chunk loss.

To evaluate the repair efficiency of Snarl, we define the *repair ratio* as the number of parities retrieved, divided by the number of lost data chunks. Figure 11 plots our findings, showing that the repair ratio is slightly higher than 2 until there is a high number of chunk losses, after which it starts to decrease. It decreases because each parity is connected to two data chunks in the lattice, and thus if both of these data chunks are lost, Snarl will only retrieve this parity from the network for the first repair operation and then from the local storage for the second one.

6 DESIGN ALTERNATIVES

We now discuss two interesting alternatives based on erasure coding to add resiliency to the Merkle tree.

6.1 Swarm Tree

Swarm tree is a design alternative outlined in [27, §5.1], which to our knowledge, has not yet been implemented. In Swarm tree, erasure coding is added at the system level to protect the tree from data loss. The tree's non-leaf nodes are encoded using a (128,112) MDS erasure code, i.e., each node can have 112 child nodes, with 16 entries for parities.

Since Swarm tree is a system-level redundancy scheme, it does not allow users to control the replication parameters of the coding scheme. Further, since each internal node in the Swarm tree is

part of a stripe with 112 data chunks and 16 parity chunks, users *must* retrieve parities, even if there are no failures. By encoding all redundancy information in the same Merkle tree as the original file, users *must* disclose resiliency levels publicly. As with any (n, k) MDS code, we need any k-of-n chunks to recover the stripe. Thus, to repair a single chunk, we first need to retrieve k chunks to recover the stripe and then reconstruct the chunk.

An unknown replication factor protects the root chunk, and thus we ignore this apparent weakness in our evaluations. Swarm tree's description states that there should be 16 parity blocks in each stripe, independent of data elements.

We can calculate the probability of an irrecoverable error for chunk losses by realizing that this scheme is equivalent to a composition of a sequence of MDS-stripes, i.e., a 1 MB file would be split into 4 stripes; (128, 112)-(128, 112)-(48, 32)-(19, 3), where the first number inside each bracket is the stripe length n and the second number is the data elements k. The first three stripes contain the 256 leaf nodes, while the last stripe contains the internal nodes. Thus, an irrecoverable error in any of these stripes effectively renders the entire composition, or file, unavailable.

To evaluate the irrecoverable likelihood, we have developed equation (2), where c is the total number of chunk losses, S_p is the number of parity blocks in a stripe, S is the set of stripes and d is the set of chunk losses per stripe. The set D captures all possible combinations of chunk losses per stripe, e.g., given $|S| = 3$, we can have x chunk losses in stripe 1, y losses in stripe 2, and z losses in stripe 3, where $x + y + z = c$. Let L be a subset of D, as described by (2), containing those scenarios that lead to critical failure. The product $\prod_{n=1}^{|S|} \binom{S_n}{d_n}$ gives the number of combinations of the given failure scenario using multivariate geometric distribution, i.e., all possible ways to select x chunks from stripe 1, y from stripe 2, and z from stripe 3.

$$P(c) = \frac{\sum_{n=1}^{|L|} \prod^{|S|} \binom{S_n}{d_n}}{|D|}, \; where \; |D| = \binom{\sum S}{c}, c \leq \sum S, \qquad (2)$$
$$L \leftarrow \{\forall d \in D : |d| = |S| \wedge \sum d = c \wedge \exists d_n \in d : d_n > S_p\}$$

Further, if we assume that the probability of chunk loss (b) is independent, all stripes are of identical length (n), and with the same parity parameter (n-k), we can simplify (2) to (3). For a stripe to be available, we need at least k out of n elements, thus the sum $\sum_{k=0}^{n-k}$ the probability that we have up to k losses for a given chunk loss b. The product $\prod^{|S|}$ gives the recovery likelihood for the entire chain.

$$P(X) = 1 - \prod^{|S|} \sum_{k=0}^{n-k} \binom{n}{k} b^k \times (1-b)^{n-k} \qquad (3)$$

Equation (3) shows the pitfall of this approach—*recovery likelihood decreases drastically with file size*. This is because Swarm's branching factor is constant at 128, and thus the only way to accommodate a larger file is to add branches, resulting in more stripes and higher $|S|$.

File availability given chunk loss is shown for various file sizes in Figure 12. In the legend, [sample] denotes results from random sampling; similar for [eq (2)] and [eq (3)]. Interestingly, results from (3) with stripes 20 and 200 fit nicely with results obtained from random sampling for file sizes 10 MB and 100 MB, respectively. From this, we can deduce that with a constant parity parameter, empty branches contribute little to the number of irrecoverable errors.

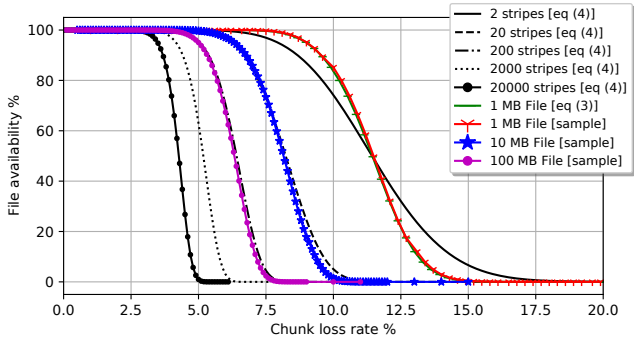

Figure 12: Availability of Swarm tree for various file sizes.

6.2 Coded Merkle Tree

Coded Merkle tree [34] is similar to Swarm tree in that they extend each level of the Merkle tree with additional erasure-coded blocks. There are two main differences between them. First, in Swarm tree, the input data is segmented into stripes of length 112 and extended with 16 parity blocks to 128 in total. In coded Merkle tree, all input data is placed in the same stripe and then extended with the appropriate number of parity blocks, depending on the coding rate ($r = k/n$). Secondly, Swarm tree always extends a stripe with 16 parity blocks, even if the stripe is not complete. In coded Merkle tree, the length of the extension depends solely on the coding rate, leading to several different coding settings.

7 CONCLUSION

This paper introduced Snarl, a user-controlled storage component that lets users control how to store data redundantly. Snarl can also improve storage utilization in cryptographic decentralized storage systems. These systems typically split the content into chunks and provide integrity verification and lookup services via a Merkle tree. The root and internal nodes are used to locate data in content-addressed storage and must thus be stored redundantly.

We have designed an entangled Merkle Tree, a resilient data structure that protects all nodes in the Merkle tree.

Based on our evaluation on Ethereum Swarm, we conclude that if every file was encoded with Snarl, five times as much data could be stored in the network, with a comparable failure-resiliency. We believe that these findings may prompt developers to incorporate Snarl into their systems.

ACKNOWLEDGMENTS

We thank Leander Jehl for providing valuable feedback on earlier versions of this paper, Rodrigo Q. Saramago for technical assistance, and the members of the Swarm Foundation for helpful discussions. This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

REFERENCES

[1] Bee nodes live. https://beenodes.live/. Accessed: 2021-06-01.
[2] Juan Benet. IPFS-Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.
[3] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the Correctness of Memories. *Algorithmica*, 12(2):225–244, 1994.
[4] Scott Chacon and Ben Straub. *Pro Git*, chapter Git internals - Git objects. Number 10.2. Apress, 2 edition, 2014.
[5] Bram Cohen. Incentives Build Robustness in BitTorrent. https://www.bittorrent.org/bittorrentecon.pdf, 2003. Accessed: 2021-10-26.
[6] Erik Daniel and Florian Tschorsch. IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks. 2021.
[7] Vero Estrada-Galiñanes, Ethan Miller, Pascal Felber, and Jehan-François Pâris. Alpha Entanglement Codes: Practical Erasure Codes to Archive Data in Unreliable Environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 183–194. IEEE, 2018.
[8] Vero Estrada-Galinanes, Jehan-François Pâris, and Pascal Felber. Simple Data Entanglement Layouts With High Reliability. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2016.
[9] Roy Friedman, Yoav Kantor, and Amir Kantor. Replicated Erasure Codes for Storage and Repair-Traffic Efficiency. In *14-th IEEE International Conference on Peer-to-Peer Computing*, pages 1–10. IEEE, 2014.
[10] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *NDSS*, volume 3, pages 131–145. Citeseer, 2003.
[11] Helm. The package manager for Kubernetes. https://helm.sh/.
[12] Naoki Katoh and Shin-ichi Tanigawa. Enumerating Constrained Non-crossing Geometric Spanning Trees. In Guohui Lin, editor, *Computing and Combinatorics*, pages 243–253, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
[13] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/.
[14] WK Lin, Dah Ming Chiu, and YB Lee. Erasure Code Replication Revisited. In *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings.*, pages 90–97. IEEE, 2004.
[15] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, 2000.
[16] Petros Maniatis, Mema Roussopoulos, Thomas J Giuli, David SH Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems (TOCS)*, 23(1):2–50, 2005.
[17] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
[18] Ralph C Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
[19] Satoshi Nakamoto et al. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
[20] Sean C Rhea, Patrick R Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y Zhao, and John Kubiatowicz. Pond: The OceanStore Prototype. In *FAST*, volume 3, pages 1–14, 2003.
[21] Rodrigo Rodrigues and Barbara Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *International Workshop on Peer-to-Peer Systems*, pages 226–239. Springer, 2005.
[22] Bianca Schroeder and Garth A Gibson. Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You? *ACM Transactions on Storage (TOS)*, 3(3):8–es, 2007.
[23] Thomas SJ Schwarz and Ethan L Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 12–12. IEEE, 2006.
[24] Mennan Selimi and Felix Freitag. Tahoe-LAFS Distributed Storage Service in Community Network Clouds. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 17–24. IEEE, 2014.
[25] Gopalan Sivathanu, Charles P Wright, and Erez Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36, 2005.
[26] V Trón, A Fischer, DA Nagy, Z Felföldi, and N Johnson. Swap, swear and swindle: incentive system for swarm, may 2016.
[27] Viktor Trón. The Book of Swarm - v1.0 pre-release 7 November 17, 2020. https://www.ethswarm.org/The-Book-of-Swarm.pdf. Accessed: 2021-10-26.
[28] Viktor Trón et al. Swarm documentation - release 0.5. https://swarm-guide.readthedocs.io. Accessed: 2021-05-25.
[29] Hakim Weatherspoon and John D Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002.
[30] Wikipedia contributors. Length extension attack — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Length_extension_attack&oldid=934998209, 2020. [Online; accessed 26-October-2021].
[31] Zooko Wilcox-O'Hearn. [tahoe-dev] erasure coding makes files more fragile, not less. https://tahoe-lafs.org/pipermail/tahoe-dev/2012-March/007185.html. Accessed: 26.10.2021.
[32] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe – The Least-Authority Filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 21–26, 2008.
[33] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
[34] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded Merkle Tree: Solving Data Availability Attacks in Blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 114–134. Springer, 2020.