Walrus: An Efficient Decentralized Storage Network

George Danezis*†, Giacomo Giuliari*, Lefteris Kokoris Kogias*, Markus Legner*,
Jean-Pierre Smith*, Alberto Sonnino*†, Karl Wüst*

*Mysten Labs, †University College London (UCL)

***Abstract*—Decentralized storage faces a fundamental trade-off between replication overhead, recovery efficiency, and security guarantees. Current approaches either rely on full replication, incurring substantial storage costs, or employ trivial erasure coding schemes that struggle with efficient recovery, especially under high churn. We present Walrus, a novel decentralized blob storage system that addresses these limitations through multiple technical innovations.**

At the core of Walrus is Red Stuff, our first contribution. Red Stuff is a two-dimensional erasure coding protocol that achieves high security with only 4.5x replication factor, while providing self-healing of lost data. This means that recovery is done without centralized coordination and requires bandwidth proportional to the lost data. Finally, Red Stuff is the first protocol to support storage challenges in asynchronous networks, preventing adversaries from exploiting network delays to pass verification without actually storing data. This allows Red Stuff to be deployable in cryptoeconomic systems that go beyond the classic honest-malicious setting.

However, Red Stuff on its own is not sufficient for Walrus as it is designed with a static set of participants in mind. To further support decentralization, we also introduce a novel multi-stage epoch change protocol that efficiently handles storage node churn while maintaining uninterrupted availability during committee transitions. Our system incorporates authenticated data structures to defend against malicious clients and ensures data consistency throughout storage and retrieval processes. Experimental evaluation demonstrates that Walrus achieves practical performance at scale, making it suitable for a wide range of decentralized applications requiring high-integrity, available blob storage with reasonable overhead.

I. Introduction

Blockchains support decentralized computation through the State Machine Replication (SMR) paradigm [1]. However, they are practically limited to distributed applications that require little data for operation. Since SMR requires all validators to replicate data fully, it results in a large replication factor ranging from 100 to 1000, depending on the number of validators in each blockchain.

While full data replication is practically needed for computing on state, it introduces substantial overhead when applications only need to store and retrieve binary large objects (blobs) not computed upon[1]. Dedicated decentralized storage [2] networks emerged to store blobs more efficiently. For example, early networks like IPFS [3] offer robust resistance to censorship, enhanced reliability and availability during faults, via replication on only a small subset [4].

Decentralized blob storage is invaluable to modern decentralized applications. We highlight the following use-cases:

- Digital assets, managed on a blockchain, such as non fungible tokens (NFTs) need high integrity and availability guarantees provided by decentralized blob stores. The current practice of storing data off-chain only secures metadata, while the actual NFT data remains vulnerable to removal or misrepresentation depending on the browser[2].
- Digital provenance of data assets is also increasingly important in the age of AI: to ensure the authenticity of documentary material; to ensure training data sets are not manipulated or polluted; and to certify that certain models generated specific instances of data [5]. These applications benefit from authenticity, traceability, integrity and availability decentralized stores provide.
- Decentralized apps, whether web-based or as binaries, need to be distributed from decentralized stores. Today, the majority of decentralized apps rely on traditional web hosting to serve their front ends and client-side code, which offers poor integrity and availability. Decentralized stores may be used to serve web and dapps content directly while ensuring its integrity and availability. Similarly, decentralized stores can ensure binary transparency for software and support the storage needs of full pipelines of reproducible builds to support the strongest forms of software auditing and chain of custody [6], [7].
- Decentralized storage plays a critical role in ensuring data availability for roll-ups [8], the current scaling strategy of Ethereum. In this setting, storage nodes hold the data temporarily allowing blockchain validators to recover it for execution. As a result, the system imposes replication costs solely on the netted state of the roll-up, rather than the full sequence of updates (e.g. transactions).
- Finally, the integration of decentralized storage with encryption techniques marks a significant paradigm shift [9]. This approach offers users comprehensive data management aligned with the Confidentiality, Integrity, and Availability (CIA) triad, eliminating the need to rely on cloud

[1] A recent example includes 'inscriptions' on bitcoin and other chains, see https://medium.com/@thevalleylife/crypto-terms-explained-exploring-bitcoin-inscriptions-51699dc218d2.

[2] A recent proof of concept attack is described here: https://moxie.org/2022/01/07/web3-first-impressions.html

services as fiduciaries. This integration unlocks numerous promising applications, including sovereign data management, decentralized data marketplaces, and computational operations over encrypted datasets. Although this paper does not focus on these applications, WALRUS, can naturally function as the storage layer for encrypted blobs. This approach provides a structured, layered framework that allows encryption overlays to focus on creating a secure and efficient Key Management System (KMS) without worrying about data availability.

In brief, secure decentralized blob stores are critical for all applications where data is relied upon by multiple mutually distrustful parties and needs to be stored in a credibly neutral store that provides high authenticity, integrity, auditability and availability.

A. Approaches to Decentralized Storage

Protocols for decentralized storage generally fall into two main categories. The first category includes systems with *full replication*, with Filecoin [3] and Arweave [10] serving as prominent examples. The main advantage of these systems is the complete availability of the blob on the storage nodes, which allows for easy access and seamless migration if a storage node goes offline. This setup enables a permissionless environment since storage nodes do not need to rely on each other for file recovery. However, the reliability of these systems hinges on the robustness of the selected storage nodes. For instance, assuming a classic $1/3$ static adversary model and an infinite pool of candidate storage nodes, achieving "twelve nines" of security – meaning a probability of less than 10^{-12} of losing access to a file – requires storing more than 25 copies on the network[3]. This results in a 25x storage overhead. A further challenge arises from Sybil attacks [11], where malicious actors can pretend to store multiple copies of a file, undermining the system's integrity.

The second category of decentralized storage services [12] uses *Reed-Solomon (RS) encoding* [13]. RS encoding reduces replication requirements significantly. For example, in a system similar to blockchain operations, with n nodes, of which $1/3$ may be malicious, and in an asynchronous network, RS encoding can achieve sufficient security with the equivalent of just 3x storage overhead. This is possible since RS encoding splits a file into smaller pieces, that we call *slivers*, each representing a fraction of the original file. Any set of slivers greater in total size to the original file can be decoded back into the original file.

However, an issue with erasure coding arises when a storage node goes offline, and needs to be replaced by another. Unlike fully replicated systems, where data can simply be copied from one node to another, RS-encoded systems require that all existing storage nodes send their slivers to the substitute node. The substitute can then recover the lost sliver, but this process results in $O(|\text{blob}|)$ data being transmitted across the network. Frequent recoveries can erode the storage savings achieved through reduced replication, which means that these systems need a low churn of storage nodes and hence be less permisionless.

Regardless of the replication protocol, all existing decentralized storage systems face an additional challenges: the need for a continuous stream of challenges to ensure that storage nodes are incentivized to retain the data and do not discard it. This is crucial in an open, decentralized system that offers payments for storage and goes beyond the honest/malicious setting. Current solutions always assume that the network is synchronous such that the adversary cannot read any missing data from honest nodes and reply to challenges in time.

B. Introducing WALRUS

We introduce WALRUS, a new approach to decentralized blob storage. It follows the erasure codes type of architecture in order to scale to 100s of storage nodes providing high resilience at a low storage overhead. At the heart of WALRUS, lies a new encoding protocol, called RED STUFF that uses a novel two-dimensional (2D) encoding algorithm that is **self-healing**. Specificaly, it enables the recovery of lost slivers using bandwidth proportional to the amount of lost data ($O(\frac{|\text{blob}|}{n})$ in our case). Moreover, RED STUFF incorporates authenticated data structures to defend against malicious clients, ensuring that the data remains consistent.

One unique feature of RED STUFF is its ability to work in an asychronous network while supporting storage challenges, making it the first of its kind. This is only possible thanks to the two-dimensional encoding that allows for different encoding thresholds per dimension. The low-threshold dimension can be used from nodes that did not get the symbols during the write flow to recover what they missed, whereas the high-threshold dimension can be used for the read flow to prevent the adversary from slowing down honest nodes during challenge periods and collecting sufficient information to reply to challenges.

One final challenge for WALRUS, and in general, any encoding-based decentralized storage system is operating securely across epochs each managed by a different committee of storage nodes. This is challenging because we want to ensure uninterrupted availability to both read and write blobs during the naturally occurring churn of a permissionless system, but if we keep writing data in the nodes about to depart, they keep needing to transfer them to the nodes that are replacing them. This creates a race for the resources of those nodes, which will either stop accepting writes or fail to ever transfer responsibility. WALRUS deals with this through its novel multi-stage epoch change protocol that naturally fits the principles of decentralized storage systems.

In summary, we make the following contributions:

- We define the problem of Asynchronous Complete Data-Sharing and propose RED STUFF, the first protocol to solve it efficiently even under Byzantine Faults (Section III)

[3]The chance that all 25 storage nodes are adversarial and delete the file is $3^{-25} = 1.18 \times 10^{-12}$.

TABLE I: Comparing Decentralized Storage Systems & Algorithms

	Replication for 10^{-12} Security	Write/Read Cost	Single Shard Recovery Cost	Asychronous Challenges	Non-Blocking Epoch Change
Replication	25x	$O(n\|blob\|)$	$O(\|blob\|)$	Unsupported	Unsupported
Classic ECC	3x	$O(\|blob\|)$	$O(\|blob\|)$	Unsupported	Unsupported
Walrus + Red Stuff	4.5x	$O(\|blob\|)$	$O(\frac{\|blob\|}{n})$	Supported	Supported

- We present Walrus, the first permissionless decentralized storage protocol designed for low replication cost and the ability to efficiently recover lost data due to faults or participant churn (Section IV).
- We show how Walrus leverages Red Stuff to implement the first asynchronous challenge protocol (Section IV-F)
- We provide a production-ready implementation of Walrus and deploy a public testnet of Walrus. We then measure its performance and scalability (Section VII).

II. Models and Definitions

Walrus relies on the following assumptions.

A. Cryptographic assumptions

Throughout the paper, we use $hash()$ to denote a collision resistant hash function. We also assume the existence of secure digital signatures and binding commitments.

B. Network and adversarial assumptions

Walrus runs in epochs, each with a static set of storage nodes. At the end of the epoch $n = 3f + 1$ storage nodes are elected as part of the the storage committee of the epoch and each one controls a storage *shard* such that a malicious adversary can control up to f of them.

The corrupted nodes can deviate arbitrarily from the protocol. The remaining nodes are honest and strictly adhere to the protocol. If a node controlled by the adversary at epoch e is not a part of the storage node set at epoch $e + 1$ then the adversary can adapt and compromise a different node at epoch $e + 1$ after the epoch change has completed.

We assume every pair of honest nodes has access to a reliable and authenticated channel. The network is asynchronous, so the adversary can arbitrarily delay or reorder messages between honest nodes, but must eventually deliver every message unless the epoch ends first. If the epoch ends then the messages can be dropped.

Our goal is not only to provide a secure decentralized system but to also detect and punish any storage node that does not hold the data that it is assigned. This is a standard additional assumption for dencentralized storage system to make sure that honest parties cannot be covertly compromised forever.

C. Erasure codes

As part of Walrus, we propose Asynchronous Complete Data Storage (ACDS) that uses an erasure coding scheme. While not necessary for the core parts of the protocol, we also assume that the encoding scheme is *systematic* for some of our optimizations, meaning that the source symbols of the encoding scheme also appear as part of its output symbols.

Let $\text{Encode}(B, t, n)$ be the encoding algorithm. Its output are n symbols such that any t can be used to reconstruct B. This happens by first splitting B into t symbols of size $O(\frac{|B|}{t})$ which are called *source* symbols. These are then expanded by generating $n - t$ repair symbols for a total of n output symbols. On the decoding side, anyone can call $\text{Decode}(T, t, n)$ where T is a set of at least t correctly encoded symbols, and it returns the blob B.

ACDS shares some similarities with Asychronous Verifiable information Dispersal (AVID) [14], [15], given that the main goal of both protocol is to distribute data. However, they also have significant differences most notably the lack of completeness in AVID protocols which is critical for Walrus. A more in depth discussion is provided in Section VIII.

D. Blockchain substrate

Walrus uses an external blockchain as a black box for all control operations that happen on Walrus. A blockchain protocol can be abstracted as a computational black box that accepts a concurrent set of transactions, each with an input message $Tx(M)$ and outputs a total order of updates to be applied on the state $Res(seq, U)$. We assume that the blockchain does not deviate from this abstract and does not censor $Tx(M)$ indefinitely. Any high-performance modern SMR protocol satisfies these requirements, in our implementation we use Sui [16].

III. Asynchronous Complete Data Storage (ACDS)

We first define the problem of Complete Data Storage in a distributed system, and describe our solution for an asynchronous network which we refer to as Asynchronous Complete Data Storage (ACDS). Secondly, we show its correctness and complexity.

A. Problem Statement

In a nutshell a Complete Data Storage protocol allows a writer to write a blob to a network of storage nodes (*Write Completeness*), and then ensures that any reader can read it despite some failures and byzantine behaviour amongst storage nodes (*Validity*); and read it consistently, despite a potentially byzantine writer (*Read Consistency*). More formally:

Definition 1 (Complete Data Storage)**.** *Given a network of $n = 3f + 1$ nodes, where up to f are byzantine, let B be a blob that a writer W wants to store within the network, and*

share it with a set of readers R. A protocol for Complete Data Storage guarantees three properties:

- *Write Completeness: If a writer W is honest, then every honest node holding a commitment to blob B eventually holds a part p (derived from B), such that B can be recovered from $\mathcal{O}\left(\frac{|B|}{|p|}\right)$ parts.*
- *Read Consistency: Two honest readers, R_1 and R_2, reading a successfully written blob B either both succeed and return B or both return $\perp$.*
- *Validity: If an honest writer W successfully writes B, then an honest reader R holding a commitment to B can successfully read B.*

B. Strawman Design

In this section, we iterate through two strawman designs and discuss their inefficiencies.

Strawman I: Full Replication: The simplest protocol uses full replication in the spirit of Filecoin [3] and Arweave [10]. The writer W broadcasts its blob B along with a binding commitment to B (e.g., $H_B = hash(B)$), to all storage nodes and then waits to receive $f + 1$ receipt acknowledgments. These acknowledgments form an availability certificate which guarantees availability because at least one acknowledgement comes from an honest node. The writer W can publish this certificate on the blockchain, which ensures that it is visible to every other honest node, who can then request a Read(B) successfully. This achieves Write Completeness since eventually all honest nodes will hold blob B locally. The rest of the properties also hold trivially. Notice that the reader never reads $\perp$.

Although the Full Replication protocol is simple, it requires the writer to send an $\mathcal{O}(n|B|)$ amount of data on the network which is also the total cost of storage. Additionally, if the network is asynchronous, it can cost up to $f + 1$ requests to guarantee a correct replica is contacted, which would lead to $\mathcal{O}(n|B|)$ cost per recovering storage node with a total cost of $\mathcal{O}(n^2|B|)$ over the network. Similarly, even a read can be very inefficient in asynchrony, as the reader might need to send $f + 1$ requests costing $\mathcal{O}(n|B|)$.

Strawman II: Encode & Share: To reduce the upfront data dissemination cost, some distributed storage protocols such as Storj [17] and Sia [18] use RS-coding [13]. The writer W divides its blob B into $f + 1$ slivers and encodes $2f$ extra repair slivers. Thanks to the encoding properties, any $f + 1$ slivers can be used to recover B. Each sliver has a size of $\mathcal{O}(\frac{|B|}{n})$. The writer W then commits to all the slivers using a binding commitment such as a Merkle tree [19] and sends each node a separate sliver together with a proof of inclusion[4]. The nodes receive their slivers and check against the commitment; if the sliver is correctly committed, they acknowledge reception by signing the commitment. The writer W can then generate an availability certificate from $2f + 1$ signatures and post it on the blockchain.

[4]Writer W could prove consistency among all slivers, but this is overkill for ACDS.

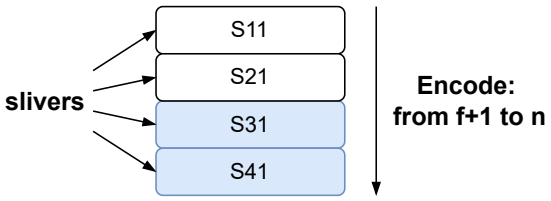


Fig. 1: Encoding a Blob in one dimension. First the blob is split into $f+1$ systematic slivers and then a further $2f$ repair slivers are encoded

A reader continuously requests slivers from the nodes until it receives $f + 1$ valid replies (i.e., replies that are verified against the commitment). The reader is guaranteed to receive them since at least $f + 1$ honest nodes have stored their sliver. The reader then reconstructs blob B from the slivers and then additionally, re-encodes the recovered value and recomputes the commitment [19], [20]. If writer W was honest, the recomputed commitment will match the commitment from the availability certificate and the reader outputs B. Otherwise, writer W may not have committed to a valid encoding, in which case the commitments do not match and the reader outputs $\perp$.

As before, the nodes that did not get slivers during the sharing phase can recover them by reading B. If the output of the read operation is $\perp$, the node returns $\perp$ on all future reads. Otherwise, the node stores their encoded sliver and discards the rest of B. Note this recovery process is expensive: recovery costs $\mathcal{O}(|B|)$ even if the storage cost afterwards is $\mathcal{O}(\frac{|B|}{n})$.

This second protocol reduces the dissemination costs significantly at the expense of extra computation (encoding/decoding and committing to slivers from B). Disseminating blob B only costs $\mathcal{O}(|B|)$[5], which is the same cost as reading it. However, complete dispersal still costs $\mathcal{O}(n|B|)$, because as we saw the process of recovering missing slivers requires downloading the entire blob B. Given that there can be up to f storage nodes that did not manage to get their sliver from writer W and need to invoke the recovery protocol, the protocol has $\mathcal{O}(n|B|)$ total cost. This is not only important during the initial dispersal, but also in cases where the storage node set changes (at epoch boundaries) as the new set of storage nodes need to read their slivers by recovering them from the previous set of storage nodes.

C. Final design: Red Stuff

The encoding protocol above achieves the objective of a low overhead factor with very high assurance, but is still not suitable for a long-lasting deployment. The main challenge is that in a long-running large-scale system, storage nodes routinely experience faults, lose their slivers, and have to be replaced. Additionally, in a permissionless system, there is

[5]There may be an extra $\mathcal{O}(\log n)$ cost depending on the commitment scheme.

some natural churn of storage nodes even when they are well incentivized to participate.

Both of these cases would result in enormous amounts of data being transferred over the network, equal to the total size of data being stored in order to recover the slivers for new storage nodes. This is prohibitively expensive. We would instead want the system to be self-healing such that the cost of recovery under churn is proportional only to the data that needs to be recovered, and scale inversely with n.

To achieve this, Red Stuff encodes blobs in two dimensions (2D-encoding). The primary dimension is equivalent to the RS-encoding used in prior systems. However, in order to allow efficient recovery of slivers of B we also encode on a secondary dimension. Red Stuff is based on linear erasure coding (see section II) and the Twin-code framework [21], which provides erasure coded storage with efficient recovery in a crash-tolerant setting with trusted writers. We adapt this framework to make it suitable in the byzantine fault tolerant setting with a single set of storage nodes, and we add additional optimizations that we describe further below.

Encoding: Our starting point is the second strawman design that splits the blobs into $f+1$ slivers. Instead of simply encoding repair slivers, we first add one more dimension to the splitting process: the original blob is split into $f+1$ primary slivers (vertical in the figure) into $2f+1$ secondary slivers (horizontal in the figure). Figure 2 illustrates this process. As a result, the file is now split into $(f+1)(2f+1)$ symbols that can be visualized in an $[f+1, 2f+1]$ matrix.

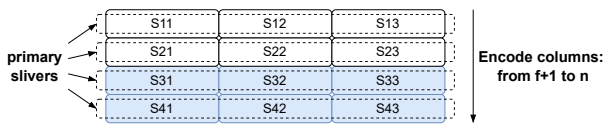
Given this matrix we then generate repair symbols in both dimensions. We take each of the $2f+1$ columns (of size $f+1$) and extend them to n symbols such that there are n rows. We assign each of the rows as the *primary sliver* of a node (Figure 2a). This almost triples the total amount of data we need to send and is very close to what 1D encoding did in the protocol in Section III-B. In order to provide efficient recovery for each sliver, we also take the initial $[f+1, 2f+1]$ [6] matrix and extend with repair symbols each of the $f+1$ rows (of size $2f+1$) and extend them to n symbols (Figure 2b) using our encoding scheme. This creates n columns, which we assign as the *secondary sliver* of each node, respectively.

Write protocol: The Write protocol of Red Stuff uses the same pattern as the RS-code protocol. The writer W first encodes the blobs and creates a sliver pair for each node. A sliver pair i is the pair of i^{th} primary and secondary slivers. There are $n = 3f+1$ sliver pairs, as many as nodes.

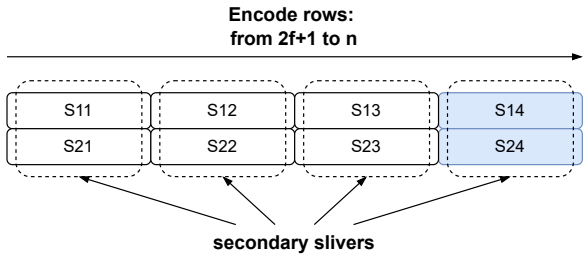
Then, W sends the sliver commitments to every node, along with the respective sliver pair. The nodes check their own sliver pair against the commitments, recompute the blob commitment, and reply with a signed acknowledgment. When $2f+1$ signatures are collected, W generates a certificate and posts it on-chain to *certify* the blob's availability.

In theoretical asynchronous network models with reliable delivery the above would result in all correct nodes eventually receiving a sliver pair from an honest writer. However, in practical protocols the writer needs to stop re-transmitting. It is safe to stop the re-transmission after $2f+1$ signatures are collected, leading to at least $f+1$ correct nodes (out of the $2f+1$ that responded) holding a sliver pair for the blob.

[6]We do not expand the expanded primary silver matrix, only the initial one. This is why the replication is $4.5x$. This also means that during recovery nodes might need to locally expand their slivers to get $3f+1$ symbols.



(a) Primary Encoding in two dimensions. The file is split into $2f+1$ columns and $f+1$ rows. Each column is encoded as a separate blob with $2f$ repair symbols. Then each extended row is the primary sliver of the respective node.



(b) Secondary Encoding in two dimensions. The file is split into $2f+1$ columns and $f+1$ rows. Each row is encoded as a separate blob with f repair symbols. Then each extended columns is the secondary sliver of the respective node.

Fig. 2: 2D Encoding / Red Stuff

Handling Metadata: During the write protocol W computes vector commitments of all slivers and as a last step creates a commitment over the list of these sliver commitments, which serves as a *blob commitment*. These commitments for each sliver form the blob *metadata*. Using these, nodes can later, when queried for a single symbol, prove that the symbol they return is the symbol originally written. This allows for partial reads of data when the systematic symbols are available as well as for efficient recovery. However, these proofs require the opening of the commitments for the respective sliver as well as of the blob commitment w.r.t. the respective sliver commitment.

A node that holds all of their slivers can easily recompute the sliver commitment and its openings, but to open the blob commitment, all sliver commitments from all nodes are required. If we naively replicate this metadata to every single storage node to enable secure self-healing, we create a large overhead that is quadratic in the number of nodes, since each node needs to store the sliver commitments of all nodes. Especially for small blobs, this can make a large difference in the relative overhead. For example, using 32B hashes in a system of 1000 nodes would require storing an additional 64kB on each node, or 64MB in total.

To reduce the overhead, storage nodes maintain an encoded version of the metadata. Since all storage nodes need to get the metadata in full when the write is in progress, there is no need for the client to perform the encoding or to do a 2D encoding. Instead, storage nodes can simply locally encode the metadata with an 1D (f+1)-out-of-n encoding and

keep the shard assigned to them[7]. This reduces the overhead to a constant per node, i.e., from quadratic to linear system-wide overhead.

Read Protocol: The Read protocol is the same as for RS-codes. In order to allow for asychronous challenges, nodes only use their secondary sliver. If this is not necessary, we can use the primary sliver and have a faster reconstruction threshold of $f + 1$.

The Read process starts with R collecting the metadata, i.e., the list of sliver commitments for the blob commitment. To do so, R requests the 1D encoded metadata parts from its peers along with the opening proofs.

After the metadata is decoded, R checks that the returned set corresponds to the blob commitment. Then R requests a read for the blob commitment from all nodes and they respond with the secondary sliver they hold (this may happen gradually to save bandwidth). Each response is checked against the corresponding commitments in the commitment set for the blob. When $2f + 1$ correct secondary slivers are collected R decodes B and then re-encodes it to recompute the blob commitment and check that it matches the blob commitment. If it is the same with the one W posted on chain then R outputs B, otherwise it outputs $\perp$.

Sliver Healing: The big advantage of Red Stuff compared to the RS-code protocol is its self-healing property. This comes into play when nodes that did not receive their slivers directly from W try to recover them. Any storage node can recover their secondary sliver by asking $f+1$ storage nodes for the symbols that exist in their row, which should also exist in the (expanded) column of the requesting node (fig. 3b and fig. 3c). This means that eventually all $2f + 1$ honest nodes will have secondary slivers. At that point, any node can also recover their primary sliver by asking the $2f + 1$ honest nodes for the symbols in their column (Figure 3d) that should also exist in the (expanded) row of the requesting storage node. In each case, the responding node also sends the opening for the requested symbol of the commitment of the source sliver. This allows the receiving node to verify that it received the symbol intended by the writer W, which ensures correct decoding if W was honest.

Since the size of a symbol is $\mathcal{O}(\frac{|B|}{n^2})$ each, and each storage node will download $\mathcal{O}(n)$ total symbols, the cost per node remains at $\mathcal{O}(\frac{|B|}{n})$ and the total cost to recover the file is $\mathcal{O}(|B|)$ which is equivalent to the cost of a Read and of a Write. As a result by using Red Stuff, the communication complexity of the protocol is (almost[8]) independent of n making the protocol scalable.

D. Red Stuff is an ACDS

Section VI provides proofs that Red Stuff satisfies all properties of a ACDS. Informally, Write Completeness is ensured by the fact that a correct writer will confirm that at least $f+1$ correct nodes received sliver pairs before stopping re-transmissions. And the sliver recovery algorithm can ensure that the remaining honest nodes can efficiently recover their slivers from these, until all honest nodes eventually hold their respective sliver, or can prove that the encoding was incorrect. Validity holds due to the fact that $2f + 1$ correct nodes wil eventually hold correct sliver pairs, and therefore a reader that contacts all nodes will eventually get enough slivers to recover the blob. Read Consistency holds since two correct readers that decode a blob from potentially different sets of slivers, re-encode it and check the correctness of the encoding. Either both output the same blob if it was correctly encoded or both output $\perp$ if it was incorrectly encoded.

[7]They should also compute a commitment and an opening of their sliver.
[8]Depends on the vector commitment scheme used.

IV. The Walrus Decentralized Secure Blob Store

In the previous section we presented Red Stuff a necessary component to build a truley permissionless decentralized storage system that allows for cryptoeconomic players. However, it is not sufficient on its own. In this section, we present Walrus which intergrates a blockchain as a control plane for meta-data and governance, with an encoding and decoding algorithm run by a separate committee of storage nodes handling blob data contents and combined them through a novel epoch-change algorithm that allows for permisionless participation of 100s of storage nodes.

Our practical implementation of Walrus uses the Red Stuff encoding/decoding algorithm described in section III-C, Merkle trees [19] as vector commitments, and the Sui blockchain [16] . Walrus can, however, be generalized to any blockchains and encoding/decoding algorithm that satisfies the minimal requirements described in Section II.

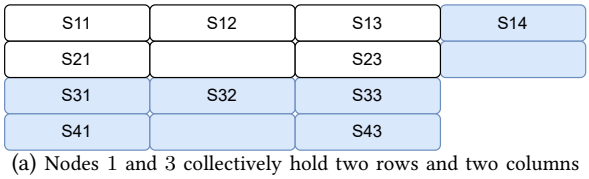
We first describe Walrus flows in a single epoch and then we discuss how we allow for storage node dynamic availability through reconfiguration. Finally, we look into going beyond honest-malicius and providing storage challenges. During an epoch, the interactions of Walrus with the clients is through (a) writing a blob and (b) reading a blob.

A. Writing a Blob

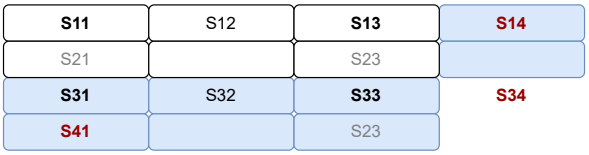
The process of writing a blob in Walrus can be seen in Algorithm 3 and Figure 4.

The process begins with the writer (❶) encoding a blob using Red Stuff as seen in Figure 2. This process yields sliver pairs, a list of commitments to slivers, and a blob commitment. The writer derives a *blob id* id_B by hashing the blob commitment with meta-data such as the length of the file, and the type of the encoding.

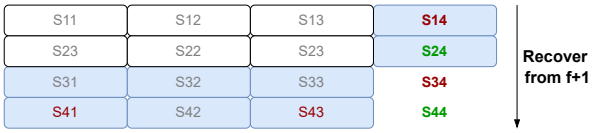
Then, the writer (❷) submits a transaction on the blockchain to acquire sufficient space for the blob to be stored during a sequence of epochs, and to *register* the blob. The size of the blob and blob commitment are sent, which can be used to rederive id_B. The blockchain smart contract needs to secure sufficient space to store both the encoded slivers on each node, as well as store all metadata associated with the commitments for the blob. Some payment may be sent along with the transaction to secure empty space, or empty space



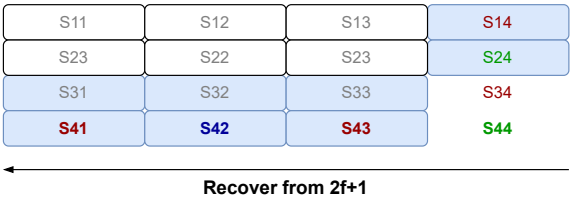
(a) Nodes 1 and 3 collectively hold two rows and two columns



(b) Each node sends the intersection of their row/column with the column/row of Node 4 to Node 4 (Red). Node 3 needs to encode the row for this.



(c) Node 4 uses the $f+1$ symbols on its column to recover the full secondary sliver (Green). It will then send any other recovering node the recovered intersections of its column to their row.



(d) Node 4 uses the $f+1$ symbols on its row as well as all the recovered secondary symbols send by other honest recovering nodes (Green) (which should be at least $2f$ plus the 1 recovered in the previous step) to recover its primary sliver (Dark Blue)

Fig. 3: Nodes 1 and 3 helping Node 4 recover its sliver pair

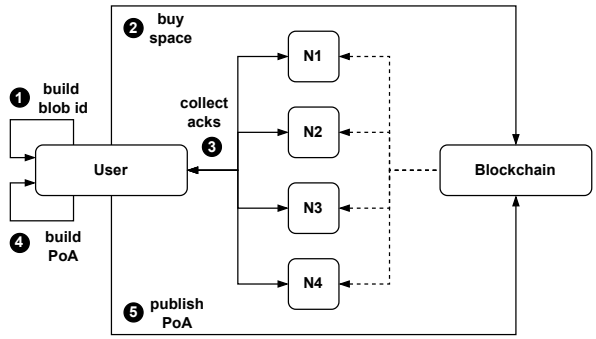


Fig. 4: Walrus write flow. The user generates the blob id of the file they wish to store; acquire storage space through the blockchain; submit the encoded file to Walrus; collect $2f+1$ acknowledgements; and submit them as proof of availability to the blockchain.

over epochs can be a resource that is attached to this request to be used. Our implementation allows for both options.

Once the register transaction commits (❸), the writer informs the storage nodes of their obligation to store the slivers of the blob identified by id_B, sending them the transaction together with the commitments and the primary and secondary slivers assigned to the respective storage nodes along with proofs that the slivers are consistent with the published id_B. The storage node verifies the commitments and responds with a signed acknowledgment over id_B once the commitments and the sliver pairs are stored.

Finally, the writer waits to collect $2f+1$ signed acknowledgments (❹), which constitute a write certificate. This certificate is then published on-chain (❺) which denotes the *Point of Availability* (PoA) for the blob in Walrus. The PoA signals the obligation for the storage nodes to maintain the slivers available for reads for the specified epochs. At this point, the writer can delete the *blob* from local storage, and go offline. Additionally, this PoA can be used as proof of availability of the *blob* by the writer to third-party users and smart-contracts.

Nodes listen to the blockchain for events indicating that a blob reached its PoA. If they do not hold sliver pairs for this blobs they execute the recovery process to get commitments and sliver pairs for all blobs past their PoA. This ensures that eventually all correct nodes will hold sliver pairs for all blobs.

B. Reading a Blob

In the read path, a reader may ask any of the storage nodes for the commitments and secondary sliver (1) for a blob by id_B. Once they collect $2f+1$ replies with valid proofs against id_B (2) they reconstruct the blob. Then (3) the reader re-encodes the blob and re-computes a blob id id'_B. If $id_B = id'_B$ it outputs the blob, otherwise the blob is inconsistent and the reader outputs $\perp$.

Reads happen consistently across all readers thanks to the properties of Red Stuff. When no failures occur, reads only require downloading sliver data slightly larger than the byte length of the original blob in total.

C. Recovery of Slivers

One issue with writing blobs in asynchronous networks or when nodes can crash-recover is that not every node can get their sliver during the Write. This is not a problem as these protocols can function without completeness. Nevertheless, in Walrus we opted to use a two-dimensional encoding scheme because it allows for completeness, i.e., the ability for every honest storage node to recover and eventually hold a sliver for every blob past PoA. This allows (1) better load balancing of read requests all nodes can reply to readers, (2) dynamic availability of storage nodes, which enables reconfiguration without needing to reconstruct and rewrite every blob, and (3) the first fully asynchronous protocol for proving storage of parts (described in Section IV-F).

All these benefits rely on the ability for storage nodes to recover their slivers efficiently. The protocol closely follows the Red Stuff recovery protocols illustrated in Figure 3.

When a storage node sees a certificate of a blob for which they did not receive slivers, it tries to recover its sliver pair from the rest of the storage nodes. For this, it requests from all storage nodes the symbols corresponding to the intersection of the recovering node's primary/secondary sliver with the signatory nodes' secondary/primary slivers. Given that $2f + 1$ nodes signed the certificate, at least $f + 1$ will be honest and reply. This is sufficient for all $2f + 1$ honest nodes to eventually hold their secondary slivers. As a result, when all honest nodes hold their secondary slivers, they can share those symbols corresponding to the recovering nodes' primary slivers, who will then get to the $2f + 1$ threshold and also recover their primary slivers.

D. Handling Inconsistent Encoding from Malicious Writers

One last challenge for Walrus is dealing with a malicious client who uploads slivers that do not correspond to the correct encoding of a blob. In that case, a node may not be able to recover a sliver that is consistent with the commitment of the symbols that it received. Walrus does not require certified blobs to be recoverable (which is the primary goal of AVID [14]) as the writer can always encode recoverable garbage data. Instead, it is guaranteed that unrecoverable blobs will have a third party verifiable proof of inconsistency, associated with id_B, after a read fails.

The read process executed by a correct reader rejects any inconsistently encoded blob by default, and as a result, sharing this proof is not a necessity to ensure consistent reads. However, agreeing on the inconsistency allows nodes to delete this blobs' data and exclude it from the challenge protocol (section IV-F). To prove inconsistency, the storage node shares the inconsistency proof—consisting of the symbols that it received for recovery and their inclusion proofs—with the other nodes, who can verify it by performing a trial recovery themselves. After verifying this fraud proof, the node attests on-chain that id_B is invalid. After observing a quorum of $f+1$ such attestations, all nodes will subsequently respond with $\perp$ to any request for the inconsistent blob's slivers, along with a pointer to the on-chain evidence for the inconsistency.

E. Committee Reconfiguration

Walrus is a decentralized protocol, therefore it is natural that the set of storage nodes will fluctuate between epochs. When a new committee replaces the current committee between epochs, reconfiguration takes place. The goal of the reconfiguration protocol is to preserve the invariant that all blobs past the Point of Availability (PoA) are available, no matter if the set of storage nodes changes. Additionally, Walrus must continue to perform reads and writes for blobs to ensure no downtime even if the reconfiguration process takes hours.[9]

[9]Unlike blockchain protocol where reconfiguration simply transfers responsibility of validation, in Walrus it transfers responsibility for data storage in the order of PBs.

Core Design: At a high-level the reconfiguration protocol of Walrus is similar to the reconfiguration protocols of blockchain systems, since Walrus also operates in quorums of storage nodes. However, the reconfiguration of Walrus has its own challenges because the migration of state is orders of magnitude more expensive than classic blockchain systems. The most important challenge is the race between writing blobs for epoch e and transferring slivers from outgoing storage nodes to incoming storage nodes during the reconfiguration event between e and $e+1$. More specifically, if the amount of data written in epoch e is greater than the ability of the departing storage node to transfer them over to the incoming storage node, then the epoch will never finish. This problem is exacerbated when some of the outgoing storage nodes of e are unavailable, as this means that the incoming storage nodes need to recover the slivers from the committee.

To resolve this problem without shutting off the write path, we take a different approach by requiring writes to be directed to the committee of $e + 1$ the moment the reconfiguration starts, while still directing reads to the old committee, instead of having a single point at which both reads and writes are handed over to the new committee. This creates challenges when it comes to reading these fresh blobs, as during the handover period it is unclear which nodes store the data. We handle this by including in the $metada$ of every $blob$ the epoch in which it was first written. If the epoch is $e + 1$ then the client is asked to direct reads to the new committee; otherwise, it can direct reads to the old committee. This happens only during handover period (when both committees need to be live and secure).

Once a member of the new committee has bootstrapped their part of the state, i.e., they have gotten all slivers for their shard, they signal that they are ready to take over. When $2f+1$ members of the new committee have signaled this, the reconfiguration process finishes and all reads are redirected to the storage nodes of the new committee.

The importance of Red Stuff for Reconfiguration:: As we mentioned above, a significant challenge during epoch change is when some nodes are faulty. In prior work, a single faulty node would require bandwidth equal to the size of the file to be transferred over the network of storage nodes. This makes epoch changes prohibitively expensive and is the reason no prior decentralized system has one. The key enabler to Walrus handling this gracefully is our Red Stuff algorithm, as it allows for the bandwidth cost of the faulty case to be the same as that of the fault-free case.

Security arguments:: In a nutshell, reconfiguration ensures all ACDS properties across epochs. The key invariant is: the reconfiguration algorithm ensures that if a blob is to be available across epochs, in each epoch $f + 1$ correct storage nodes (potentially different ones) hold slivers. This is the purpose of the explicit signaling that unlocks the epoch change by $2f+1$ nodes. Therefore, eventually all other honest storage nodes can recover their sliver pairs, and in all cases, $f + 1$ honest nodes in the next epoch are able to recover

correct sliver pairs as a condition to move epochs.

F. Storage Challenges

Walrus uses a challenge protocol to prevent cheating nodes that trivially never store or serve data from receiving rewards and to incentivize honest nodes. To the best of our knowledge, we present here the first storage proof protocol to make no assumptions about network synchrony. It leverages the completeness property of Red Stuff and the ability to reconstruct blobs with $2f + 1$ threshold.

Fully Asynchronous Challenge Protocol: For the lightweight challenge protocol, we require the storage nodes to set up a random coin with a $2f + 1$ reconstruction threshold. This is possible using any kind of asynchronous DKG [22], [23], [24] or randomness generation protocol [25], [26].

Close to the end of the epoch, the storage nodes witness a "challenge start" event on-chain, such as a specific block height. This event is automatically generated by the blockchain/smart contract. At that point, the honest storage nodes witnessing the event stop serving read and recovery requests and post an acknowledgment on-chain, revealing a randomness share as part of the acknowledgment. When $2f + 1$ nodes have posted their acknowledgment a random coin is revealed, which is used to seed a pseudo-random function. The output is used to select a subset of the stored blobs per storage node that are to be challenged. The subsets need to be disjoint for the challenge to work. Any non-challenged blob can now be read and recovered again. When $2f + 1$ nodes have entered the challenge phase and their acknowledgment is public, the challenges begin.

Every node broadcasts to all other nodes their primary sliver for the challenged blobs. Only nodes that actually hold the blob data will be able to produce this. The receiving nodes check that the slivers match the commitment of the blob and send a confirmation signature.

Each storage node finishes $2f + 1$ pairwise interactions of being challenger and challengee, leading to collecting $2f + 1$ signatures that form a certificate of storage (CoS). Then the storage node submits the CoS on-chain to prove their honesty. The challenge period ends when $2f+1$ storage nodes have submitted their CoS at which point the reads and recovery are re-enabled.

Since the threshold for starting a challenge is $2f + 1$, at least $f + 1$ honest will not reply to the adversary trying to recover files in order to reply to the challenge. As a result, even if the adversary has slowed down f honest nodes to not see the challenge start message, it can only get f symbols from their secondary slivers. These are not enough to recover their primary sliver even if all other malicious node did not delete their data and can also give $f - 1$ extra symbols as $f - 1 + f = 2f - 1$ which is still not enough.

The proof for the asychronous challenge protocols can be seen in Section VI-D.

V. Detailed Algorithms

This section supplements Section IV by providing detailed algorithms for clients (Algorithm 1) and storage nodes operations (Algorithm 3).

In additional to the helper functions specified in Algorithm 2, these algorithms also leverages the following (intuitive) functions: $\textsc{ByteSize}(B)$ to compute the size of a blob B in bytes; $\textsc{MerkleTree}(v)$ to compute a merkle tree over a vector input v; $\textsc{Hash}(\cdot)$ to compute a cryptographic hash; $\textsc{ErasureEncode}(B)$, $\textsc{ErasureReconstruct}(\cdot)$, and $\textsc{ErasureDecode}(\cdot)$, to respective erasure encode a blob B, reconstruct a blocb from enough erasure coded parts, and erasure decode a blob as described in Section III-C; $\textsc{HandledShards}(n)$ to get the shards handled by a node n; and $\textsc{SplitIntoMatrix}(\cdot)$ to reshape a matrix into the specified size.

Furthermore, the client and storage nodes use the following functions to interact with the blockchain: $\textsc{ReserveBlob}(\cdot)$ to reserve a blob id on the blockchain; $\textsc{StoreCertificate}(\cdot)$ to store a proof of storage on the blockchain; $\textsc{IsRegistered}(id)$ to check if a blob id id is registered on the blockchain; and $\textsc{ReadCertificate}(id)$ to read a proof of storage of blob id id from the blockchain.

Table II summarizes the main notations used in the algorithms. Subscripts of matrices and vectors denote access to a specific index.

$E_{(i,j)}$	Symbol at position (i, j) of an encoded blob
S^p	The set of primary slivers
S^s	The set of secondary slivers
$S^{(p,n)}$	The primary sliver held by storage node n
$S^{(s,n)}$	The secondary sliver held by storage node n
$\{S^{(p,*)}\}_{f+1}$	Any set of $f + 1$ primary slivers
M^p	Metadata associated with the primary slivers
M^s	Metadata associated with the secondary slivers
D^n	The set of shards handled by node n

TABLE II: Main notations

VI. Red Stuff and Walrus Proofs

This section completes Section III by showing that Red Stuff satisfies all the properties of a ACDS. The casual reader can skip it.

A. Write Completeness

We show that Red Stuff satisfies Write Completeness. Informally, if an honest writer writes a blob B to the network, every honest storage node eventually holds a primary and secondary correctly encoded sliver of B. For this part we assume the writer is honest and provides a correct vector commitment M.

Lemma 1 (Primary Sliver Reconstruction)**.** *If a party holds a set of* $(2f + 1)$ *symbols* $\{E(i, *)\}_{2f+1}$*from a primary sliver* $S^{(p,i)}$*, it can obtain the complete primary sliver* $S^{(p,i)}$*.*

Algorithm 1 WALRUS client operations

```
nodes                                                  ▷ the committee of storage nodes
shards                                                 ▷ see Section IV

// Store a blob on the network
procedure StoreBlob(B, expiry)
    // Step 1: Pay and register the blob id on the blockchain
    (S^p, S^s) ← EncodeBlob(B)
    M ← MakeMetadata(S^p, S^s)
    id ← MakeBlobId(M)
    size ← ByteSize(B)                                 ▷ size in bytes
    ReserveBlob(id, size, expiry)                      ▷ on blockchain

    // Step 2: Send the encoded slivers to the storage nodes
    R ← { }                                            ▷ storage requests to send to nodes
    for n ∈ nodes do
        D^n ← HandledShards(n)                         ▷ shards handed by node n
        S^(p,n) ← [S^p_i : i ∈ D^n]
        S^(s,n) ← [S^s_i : i ∈ D^n]
        StoreRqst ← (id, M, S^(p,n), S^(s,n))
        R ← R ∪ {(n, StoreRqst)}
    await_{2f+1} : {c ← Send(n, r) : (n, r) ∈ R}       ▷ wait for 2f + 1 confirmations

    // Step 3: Record the proof of storage on the blockchain
    StoreCertificate({c}, id)                          ▷ on blockchain

// Read metadata from the network
procedure RetrieveMetadata(id)
    MetadataRqst ← (id)
    D ←$ {0, shards}^n                                 ▷ request all shards
    N ← {n ∈ nodes s.t. ∃s ∈ D ∩ HandledShards(n)}
    await_{2f+1} : {M ← Send(n, MetadataRqst) : n ∈ N} ▷ wait for 2f + 1 responses
    if ∃M ∈ {M} s.t. MakeBlobId(M) = id then return M
    return ⊥

// Read a blob from the network
procedure ReadBlob(id)
    M ← RetrieveMetadata(id)
    SliversRqst ← (id)
    await_{2f+1} : {S^(s,n) ← Send(n, SliversRqsts) s.t. n ∈ nodes : VerifySliver(S^(s,n), M)}
    B ← DecodeBlob({S^(s,*)}_{2f+1}, M)
    return B
```

Algorithm 2 Helper functions

```
nodes                                                  ▷ the committee of storage nodes
shards                                                 ▷ see Section IV

procedure EncodeBlob(B)
    E ← ErasureEncode(B)                               ▷ expand size: [(f + 1) × (2f + 1)] → [shards × shards]
    S^p ← [E_(i,*) : i ∈ [0, shards]]                  ▷ encoded primary slivers: [shards × 1]
    S^s ← [E_(*,i) : i ∈ [0, shards]]^⊤                ▷ encoded secondary slivers: [1 × shards]
    return (S^p, S^s)

procedure MakeMetadata(S^p, S^s)
    M^p ← [Hash(s) : s ∈ S^p]                          ▷ length: 2f + 1
    M^s ← [Hash(s) : s ∈ S^s]                          ▷ length: f + 1
    M ← (M^p, M^s)
    return M

procedure MakeBlobId(M)
    (M^p, M^s) ← M
    id ← (MerkleTree(M^p), MerkleTree(M^s))
    return id

procedure VerifySliver(S^(*,n), M)
    (M^p, M^s) ← M
    return (Hash(s) = M^p_n : ∀s ∈ S^(p,n)) ∨ (Hash(s) = M^s_n : ∀s ∈ S^(s,n))

procedure DecodeBlob({S^(p,*)}_{f+1}, M)
    S^p ← ErasureReconstruct({S^(p,*)}_{f+1})          ▷ reconstruct encoded slivers
    E ← SplitIntoMatrix(S^p)                           ▷ size: shard × shard
    S^s ← [E_(*,i) : i ∈ [0, shards]]^⊤
    M′ ← MakeMetadata(S^p, S^s)
    if M ≠ M′ then return ⊥ ▷ verify encoding correctness, see Section IV-B
    B ← ErasureDecode(E)                               ▷ matrix: (f + 1) × (2f + 1)
    return B
```

Proof. The proofs directly follows from the reconstruction property of erasure codes with reconstruction threshold $(2f + 1)$. □

Lemma 2 (Secondary Sliver Reconstruction)**.** *If a party holds a set of $(f + 1)$ symbols $\{E(*, i)\}_{f+1}$ from a secondary sliver $S^{(s,i)}$, it can obtain the complete secondary sliver $S^{(s,i)}$.*

Proof. The proofs directly follows from the reconstruction property of erasure codes with reconstruction threshold $(f + 1)$. □

Theorem 1. *RED STUFF satisfies Write Completeness (Definition 1).*

Proof. To write a blob B, an honest writer W sends at least $(2f + 1)$ correctly encoded slivers (parts) to different storage nodes, along with a binding vector commitment M over those slivers. For these nodes the property holds by definition. Now let's assume a node j that is not in the initial $2f + 1$ recipients. The node will ask every node i for their shared symbols in its primary (i.e., $E(j, i)$) and secondary (i.e., $E(i, j)$) sliver. Given the binding vector commitment M node i can either send the true symbols or not reply. Given that at least $2f + 1$ nodes acknowledged M then j will get $f+1$ correct symbols for its primary sliver $\{E(j, *)\}_{f+1}$ and $f + 1$ correct symbols for its secondary sliver $\{E(*, j)\}_{f+1}$. From Lemma 2 this means that j will reconstruct its full secondary sliver $S^{(s,j)}$.

Since this reasoning applies to any generic node i, it holds for all nodes. As a result, eventually all $2f + 1$ honest nodes will reconstruct their secondary slivers $S^{(s,*)}$. Every time a node reconstructs their secondary sliver, they also reply to node j with the shared symbol which is part of the primary sliver of j(i.e., $E(j, *)$) . As a result, eventually j will go from $\{E(j, *)\}_{f+1}$ to $\{E(j, *)\}_{2f+1}$ This allows node j to apply Lemma 1 and reconstruct its primary sliver $S^{(p,j)}$.

Since this reasoning applies to any generic node i, it holds for all nodes and concludes the proof that all honest nodes will eventually hold both their primary and secondary sliver. □

B. Read Consistency

We prove that RED STUFF satisfies Read Consistency. Informally, if two honest readers read a blob B written to the network, they either both eventually obtain B or both eventually fail and obtain $\perp$.

Theorem 2. *RED STUFF satisfies Read Consistency (Definition 1).*

Proof. Notice that the encoding scheme is deterministic and the last step of reading is to re-run the encoding and

Algorithm 3 WALRUS store operations

```
1: n                                              ▷ the identifier of the storage node
2: nodes                                          ▷ the committee of storage nodes
3: shards                                         ▷ see Section IV
4: db_m                                           ▷ perists the metadata
5: db_b                                           ▷ perists the slivers

   // Store slivers
6: procedure StoreSlivers(StoreRqst)
7:     (id, M, S^(p,n), S^(s,n)) ← StoreRqst
8:
9:     // Check 1: Ensure the node is responsible for the shards
10:    D^n ← HandledShards(n)
11:    if ∃ s_i ∈ S^p ∪ S^s s.t. i ∉ D^n then return ⊥
12:
13:    // Check 2: Verify the blob id is registered on chain
14:    if ¬IsRegistered(id) then return ⊥
                                                  ▷ read blockchain
15:
16:    // Check 3: Verify the metadata is correctly formed
17:    if ¬VerifySliver(S^(p,n), M) then return ⊥
18:    if ¬VerifySliver(S^(s,n), M) then return ⊥
19:    id' ← MakeBlobId(M)
20:    if id ≠ id' then return ⊥
21:
22:    db_m[id] ← M                               ▷ persist the metadata
23:    db_b[id] ← (S^(p,n), S^(s,n))              ▷ persist the slivers
24:    Send(ack)                                  ▷ reply with an acknowledgment

   // Server metadata
25: procedure ServeMetadata(MetadataRqst)
26:    id ← MetadataRqst
27:    return db_m[id]                            ▷ return the metadata or ⊥ if not found
28:    Reply(ack)

   // Server slivers
29: procedure ServeSlivers(SliversRqst)
30:    id ← SliversRqst
31:    if ¬ReadCertificate(id) then return ⊥      ▷ proof of storage on the
   blockchain
32:    (S^(p,n), S^(s,n)) ← db_b[id]              ▷ return the slivers or ⊥ if not found
33:    Reply(S^(s,n))

   // Recover slivers
34: procedure RecoverSlivers(id)
35:    c ← Client(nodes, shards)                  ▷ build a WALRUS client (Algorithm 1)
36:    B ← c.ReadBlob(id)
37:    D^n ← HandledShards(n)                     ▷ shards handed by node n
38:    S^(p,n) ← [S_i^p : i ∈ D^n]
39:    S^(s,n) ← [S_i^s : i ∈ D^n]
40:    db_m[id] ← M                               ▷ persist the metadata
41:    db_b[id] ← (S^(p,n), S^(s,n))              ▷ persist the slivers
```

reconstruct M. As a result, a reader that accepts the read as correct needs to output B.

The challenge with Read Consistency is if the writer can convince different readers that collect different slivers to output B and $\perp$. Let's assume that two honest readers R_1 and R_2 read a blob B from the network and R_1 eventually obtains B while R_2 eventually fails and obtains $\perp$.

There are two scenarios for R_2 to output $\perp$:

1) R_2 gets $2f+1$ replies matching M and tries to reconstruct. During reconstruction, the commitment does not much M
2) Some node failed to reconstruct their secondary sliver. By the algorithm this nodes will hold a proof of inconsistency, which it will send to R_2

In either scenario R_1 during their reconstruction should have also detected the inconsistency and output $\perp$ otherwise the binding property of the vector commitment does not hold. Hence a contradiction.

□

C. Validity

We prove that RED STUFF satisfies Validity. Informally, if an honest writer writes a correctly encoded blob B to the network, every honest reader eventually obtains B.

Theorem 3 (Validity)**.** *RED STUFF satisfies Validity (Definition 1).*

Proof. To write a blob B, an honest writer W construct n correct encoded slivers (parts) along with a binding vector commitment M over those slivers. Since the writer is honest from Theorem 1 all (at least $2f+1$) honest storage nodes will hold their respective slivers. Let's note by nodes the entire set of storage nodes. An honest reader queries each storage node $n \in$ nodes for their secondary sliver, verifies them against M and when it holds $2f+1$ uses them to reconstruct the B. Since all honest storage nodes will eventually reply to the reader and W was honest, the reader will eventually obtain B. □

D. Asychronous Challenges

We prove that an adversary that drops its slivers cannot pass a storage challenge even if colluding with the other $f-1$ malicious storage nodes. Later we discuss probabilities of passing a challenge if storing a subset of the slivers and how we can tune it.

Theorem 4 (Secure Challenge Protocol)**.** *No malicious storage node running WALRUS that deletes its storage will succeed in a challenge.*

Proof. We assume there exists a storage node j that deletes all slivers it is supposed to hold.

At challenge time of some random blob B it will need to produce its primary sliver. To do this it needs to find $2f+1$ symbols of this sliver. For this it can ask for $f-1$ symbols from the other malicious storage nodes who hopefully did not also delete them. Contorolling the network it could also slow down honest nodes to not see the challenge start message posted on-chain and request a read for the sliver.

However, for the challenge to start and the randomness to be revealed there needs to be $2f+1$ acknowledgements. From them f can be the malicious nodes but there is at least $f+1$ honest nodes who have seen the challenge start message and will not reply to read/recovery requests. Hence it can only collect another f symbols.

However $f+f-1 = 2f-1$. From lemma 1 the node need $2f+1$ to reconstruct the primary sliver and get this symbol. Hence it will fail to reply. As a result no honest node will sign the Certificate of Storage and the storage node will fail the challenge.

□

Reducing the Cost: Notice that no honest node will sign off, not even the ones slowed down. Additionally, the adversary can only recover symbols from the same slowed down nodes

from all blobs challenged. These two observations can reduce the cost of challenging significantly. Namely:

1) The set of verifying nodes could be randomly assigned to have k nodes such that at least $k/2+1$ sign. As long as less than $k/2+1$ are malicious and at least one is honest whp the proof above would hold
2) If n is large, we do not need to challenge the full sliver, but only a random subset of symbols of each sliver. Given that the adversary can only slow down f honest nodes, the chances that all slivers from all blobs are held by the same f nodes gets diminishingly small quickly.

Adversary Trade-offs: The proofs are done for an adversary that deletes all blobs. However the adversary could decide to hold a percentage of blobs, saving some constant factor in its storage costs. For this we can tune the number of blobs to make it unlikely. For example, if a storage node holds 90% of the blobs, it has less than a 10^{-30} probability of success in a 640 file challenge.

We believe this is unlikely to happen as it would mean that the malicious storage nodes have minimal storage savings (less than a constant factor) and probably no real reduction in their resource costs. However, even if it does happen Walrus is still secure as long as $2f+1$ honest storage nodes store their blobs and only slowed down during recovery and epoch-change procedures.

VII. Evaluation

We implement a *production-ready* networked multi-core Walrus storage node in Rust. All networking uses HTTPS through axum [27], it uses fastcrypto [28] for cryptography, rocksdb [29] for storage, and reed-solomon-simd [30] for erasure coding. We opt to connect our implementation to Sui [31] as an example of fast blockchain. We release the codebase as open-source[10].

We evaluate Walrus's performance and scalability on the *real, publicly available*, testnet. This is the most realistic evaluation setting, exposing the system to real-world conditions, real users, and infrastructure outside our control. We observe the Walrus testnet over a period of 60 days, ending the 22nd of March.

Our evaluation aims at demonstrating the following claims:

1) **C1 (low latency):** Walrus achieves low latency, bounded by network delay.
2) **C2 (throughput):** Walrus clients achieve high read and write throughput.
3) **C3 (scalability):** Walrus's total capacity scales with the number of storage nodes.

A. Experimental Setup

The Walrus testbed is decentralized, comprising 105 independently operated storage nodes and 1,000 shards. All reported measurements are based on data voluntarily shared by node operators.

[10]https://github.com/mystenLabs/walrus

Shards are allocated based on each operator's stake, reflecting the mainnet deployment model. Satisfying the $f+1$ quorum requires collaboration from at least 19 nodes; the $2f+1$ quorum requires 38 nodes. No operator controls more than 18 shards. Nodes span at least 17 countries, including Lithuania, USA, France, Canada, Netherlands, Thailand, Ireland, Russia, and others. Eleven operators did not disclose their location. Figure 5 details the shard distribution by region. The "eu-west" region aggregates shards from at least five countries. Roughly 220 shards are labeled "unknown" due to missing regional data. Figure 6 shows shard distribution by hosting providers. "Self-Hosted" nodes run on-premises, while "Unknown" indicates missing provider information.

Most nodes run Ubuntu (22.04 or 24.04) with at least 16 CPU cores, 128 GB RAM, and 1 Gbps bandwidth. Hardware varies across Intel and AMD CPUs and HDD, SSD, and NVMe storage. Node storage ranges from 15 to 400 TB (median 56.9 TB, P90 69.98 TB).

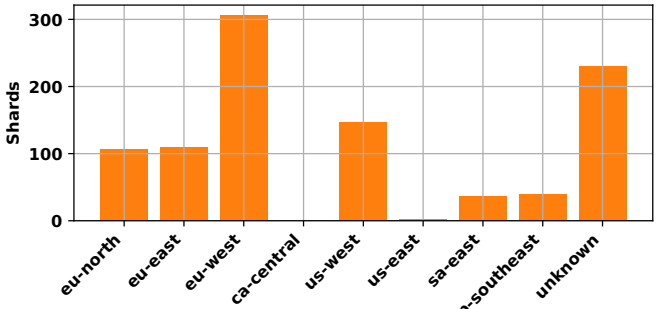


Fig. 5: Geo-distribution of shards.

B. System Performance

We evaluate performance from the client's perspective, deploying two clients on AWS `m5d.8xlarge` instances (10 Gbps bandwidth, 32 vCPUs, 128 GB RAM, Ubuntu 22.04). One in US East (N. Virginia), the other in Canada Central.

Walrus Latency: Figure 7 illustrated the end-to-end latency experienced by the client. We start measuring before the client encodes the blob and finish when it observes a proof-of-availability confirmation on the blockchain. Each point represents the p50 over 5 minutes of runs; error bars indicate p90.

The graph shows that read latency remains low, even for large blobs. For small blobs (less than 20 MB), the latency stays below 15 seconds. For large blobs (130 MB), the latency increases to around 30 seconds.

Write latency is consistently higher than read latency. For small blobs (less than 20 MB), write latency remains relatively flat and stays under 25 seconds. This overhead is primarily due to the blockchain interaction and the need to upload metadata to all storage nodes, rather than the blob size itself. For large blobs (greater than 40 MB), latency grows linearly with the blob size as network transfer becomes the dominant cost. Figure 8 and Figure 9 illustrate this behavior by breaking down the latency for small blobs (1 KB) and large blobs (130 MB), respectively. Each write operation consists of five

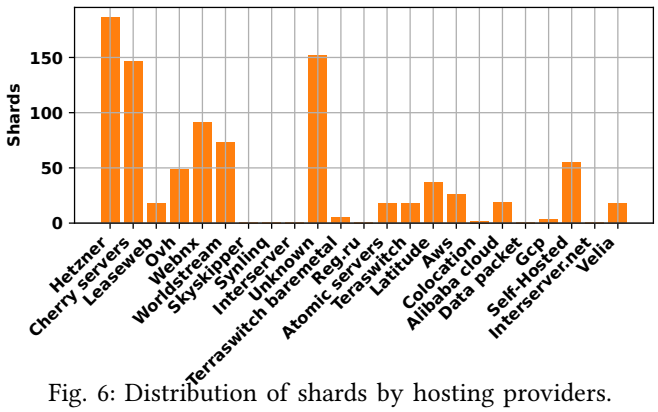


Fig. 6: Distribution of shards by hosting providers.

key steps: *encoding* (time to erasure-code the blob), *check status* (time to check the blob's current state), *get info* (time to fetch blob status and reserve space), *store* (time to upload slivers to storage nodes), and *publish PoA* (time to commit the proof of availability to the blockchain). For small blobs, the fixed overhead from metadata handling and blockchain publication dominates, adding roughly 6 seconds—about 50% of the total write latency. For large blobs, the storage phase dominates due to network transfer, while metadata operations and blockchain interaction remain relatively constant.

These results validate our claim **C1**: Walrus achieves low latency and is bounded by network delays.

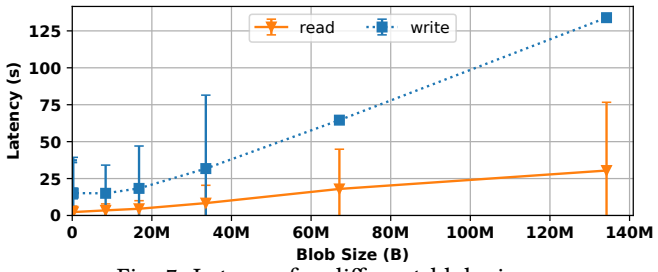


Fig. 7: Latency for different blob sizes.

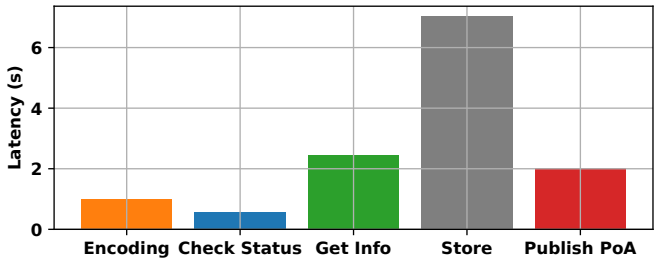


Fig. 8: Latency breakdown for small blobs (1KB).

Single Client Throughput: Figure 10 illustrates the throughput that can be achieved by a single client in bytes per second. As expected, read throughput scales linearly with blob size as it is mostly network interactions. Write throughput plateaus around 18 MB/s because of the need to interact with the blockchain and the storage nodes multiple times. This does not mean that a user cannot upload faster, as Sui supports a much higher throughput in transactions per second, but that a single blob cannot be uploaded faster. For much larger blobs, a user can deploy multiple clients, each uploading a chunk of data in parallel, effectively creating a fan-out pattern. These results validate **C2**: Walrus enables clients to read and write at high throughput.

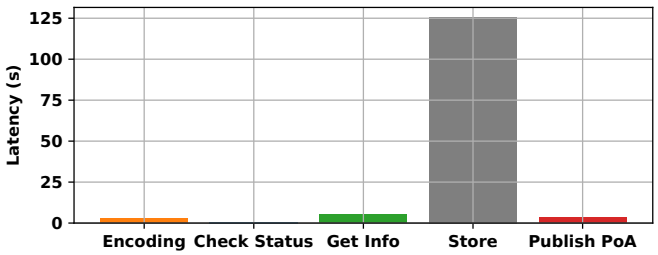


Fig. 9: Latency breakdown for large blobs (130MB).

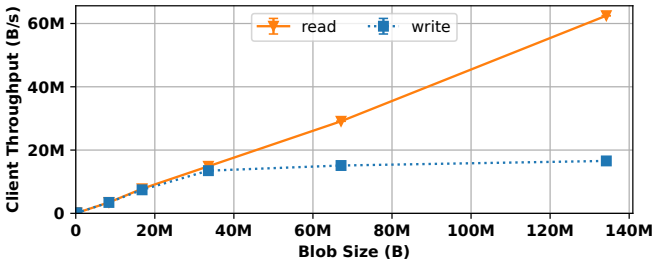


Fig. 10: Single client throughput for different blob sizes.

C. Scalability

Over 60 days, Walrus stores a median of 1.18 TB of slivers (P90 1.08 TB) and 221.5 GB of blob metadata (P90 46.34 GB). Each storage node contributes between 15 and 400 TB of capacity. Yet, the system as a whole can store over 5 PB—a key feature of Walrus. Figure 11 illustrates how Walrus's total storage capacity scales with the committee size. This result supports our final claim **C5**: the system's capacity grows proportionally with the number of storage nodes.

VIII. Related Work

Censorship resistant storage and blob data dissemination motivated much of the early peer-to-peer movement and the need for decentralization. Within academia Anderson proposed the Eternity service [32] in 1996, to ensure documents cannot be suppressed. Within the commercial and open source communities systems like Napster [33], Gnutella [34], and Free Haven [35] and early Freenet [36] used nodes in an unstructured topology to offer storage, routing and distribution largely of media files. These systems operated on the basis of centralized or flood fill algorithms for lookup and search; and full replication of files, often on node used to route responses. These provide best effort security and poor performance.

Later research, in the early 2000s, proposed structured peer-to-peer topologies in the form of distributed hash tables (DHT), such as Chord [37], Pastry [38], Kademlia [39], largely to improve lookup performance, as well as reduce the replication factor for each file. DHTs remarkably do not require consensus or full state machine replication to operate. However, have been shown to be susceptible to a number of

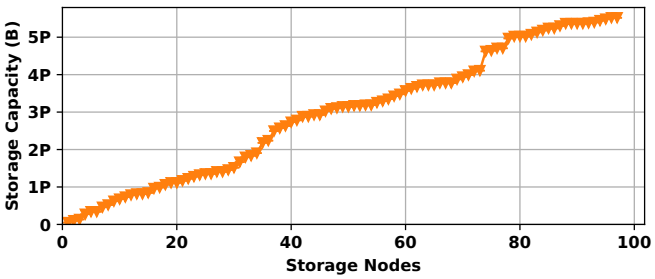


Fig. 11: Storage capacity versus committee size.

attacks: Sybil attacks [11] were named and identified within the context of these systems first; and they are hard to defend against routing attacks [40]. Many attacks affect current systems that use them [41]. Bittorrent [42] eventually came to dominate the file dissemination application space, in part due to its simplicity and built-in incentives. It initially used a full replication strategy for storage and centralized trackers for node coordination. It later added decentralized trackers based on Kademlia.

In contrast to these early system Walrus maintains a full and consistent list of all nodes through using the Sui [16] blockchain, as well as their latest meta-data. It assumes these are infrastructure grade nodes and will not suffer great churn, but rather operate to get incentives and payments, and come in and out of the system based on a reconfiguration protocol.

In the blockchain era, IPFS [43] provides a decentralized store for files, and is extensively being used by blockchain systems and decentralized apps for their storage needs. It provides content addressable storage for blocks, and uses a distributed hash table (DHT) to maintain a link between file replicas and nodes that store them. Publishers of files need to pin files to storage nodes, to ensure files remain available, usually against some payment. The underlying storage uses full replication on a few nodes for each file.

Filecoin [3] extends IPFS, using a longest chain blockchain and a cryptocurrency (FIL) used to incentivize storage nodes to maintain file replicas. Publishers acquire storage contracts with a few nodes, and payments are made in the cryptocurrency. Filecoin mitigates the risk that these nodes delete the replicas by requiring storage nodes to hold differently encoded copies of the file, and performing challenges against each other for the encoded files. These copies are encoded in such a way that it is slow to reproduce them from the original copy, to avoid relay attacks. As a result, if the user wants to access the original file, it needs to wait a long time for the decoding of a copy, unless some storage node has a hot copy. Since, there is no in-built incentive for storing hot copies, this service usually costs extra.

Arweave [10] mitigates slow reads through a Proof-of-Access algorithm that incentives storage nodes to have as many files as possible locally to maximise rewards. This is implemented in conjunction with a full replication strategy, and results in replication levels almost equal to classic state machine replication. Additionally, the system only allows file to be stored 'for ever', through a mechanisms of pre-payment - which lacks the flexibility to control lifetime and deletion, and is capital inefficient since payment is upfront.

In contrast to Filecoin and Arweave, Walrus uses erasure coding to maintain a very low overhead of 4.5x while ensuring data survives up to 2/3 of any shards being lost, and continues to operate by allowing writes even if up to 1/3 of shards are unresponsive. Furthemore, Walrus does not implement its own separate blockchain to do node management and provide incentives, but uses Sui instead.

Storj [17] represents another decentralized storage solution that leverages encoding to achieve a low replication factor. The system implements a Reed-Solomon based erasure coding scheme with a $29/80$ configuration, wherein a file is encoded into 80 parts, with any 29 sufficient for reconstruction. This approach results in a $2.75x$ replication factor, offering a substantial reduction in storage costs compared to prior systems. However, a key limitation is its inability to efficiently heal lost parts. The system relies on users to reconstruct the full file and subsequently re-encode it to facilitate the recovery of lost parts. In contrast Walrus's use of Red Stuff incorporates an efficient reconstruction mechanism which is critical for the efficient healing of the erasure coding scheme, especially due to churn which is naturally occuring in a permissionless system. Red Stuff builds on the Twin-code framework [21], which uses two linear encodings of data to enhance the efficiency of sliver recovery. However, unlike the Twin-code framework [44], Red Stuff encodes data across differently sized dimensions and integrates authenticated data structures, achieving Completeness (as defined in Section II) and ensuring Byzantine Fault Tolerance.

Modern blockchains provide some storage, but it is prohibitively expensive to store larger blobs due to the costs of full replication across all validators, as well as potentially long retention times to allow verifiability. Within the Ethereum eco-system specifically, the current scaling strategy around L2s involves posting blobs of transactions on the main chain, representing bundles of transactions to be executed, and verified either via zero-knowledge or fraud proofs. Specialised networks, such as Celestia based on availability sampling [45], have emerged to fulfill this need off the main Ethereum chain. In Celestia, two dimensional Reed-Solomon codes are used to encode blobs, and code words distributed to light nodes to support 'trustless' availability. However, all blobs are fully replicated across the validators of the system, for a limited time period of about month. Walrus instead offers proofs of availability with arbitrarily long retention periods and a reduced cost of storage per node which allows the system to scale inpexpensively.

The most closely related work to ours is Semi-AVID [15] which has also been explored as an alternative to provide Data Availability for rollups. It is similar to the Strawman II design meaning that although it can achieve the critical property of verifiable data storage it cannot achieve write completeness unless the full data is reconstructed. This makes it prohibitively expensive for epoch-change and only suitable for either fully permissioned systems with no churn or short-

lived data storage. This is the main challenge with AVID [14], which is optimized to provide verifiability of data storage, disallowing the output of $\perp$. This is an overkill of our use case, as a malicious writer can simply encode garbage data instead of a failed encoding. Hence, the machinery to detect and reject failed encodings that AVID provides is unnecessarily constraining and expensive. Finally, AVID protocols are not designed with storage challenges in mind, which separates them from Walrus which uses the completeness property to be able to support incentives.

IX. Conclusion

We introduce Walrus, a novel approach to decentralized blob storage that leverages fast erasure codes and a modern blockchain technology. By utilizing the Red Stuff encoding algorithm and the Sui blockchain, Walrus achieves high resilience and low storage overhead while ensuring efficient data management and scalability. Our system operates in epochs, with all operations sharded by $blob_{id}$, enabling it to handle large volumes of data effectively. The innovative two-dimensional BFT encoding protocol of Red Stuff allows for efficient data recovery, load balancing, and dynamic availability of storage nodes, addressing key challenges faced by existing decentralized storage systems.

Furthermore, Walrus introduces storage proofs that ensure data availability without relying on network synchrony assumptions, and its committee reconfiguration protocol guarantees uninterrupted data availability during network evolution. By combining these features, Walrus offers a scalable, and resilient decentralized storage, providing high authenticity, integrity, auditability, and availability at a reasonable cost. Our contributions include defining the problem of Asynchronous Complete Data-Sharing, presenting the Red Stuff protocol, and proposing an asynchronous challenge protocol for efficient storage proofs, paving the way for future advancements in decentralized storage technologies.

Acknowledgments

We would like to express our gratitude to Dmitry Perelman, Sadhan Sood, Zue Wu, He Liu, and Pei Deng for their invaluable contributions in bringing Walrus to production. We also extend our sincere appreciation to Damir Shamanaev for his assistance in constructing the smart contracts that connect Walrus with the Sui blockchain. Lastly, we would like to extend a special thank you to Joachim Neu for identifying a serious vulnerability in our previous Testnet implementation. This vulnerability was related to its utilization of RaptorQ for erasure coding and lead us to replace it with RS Codes.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[2] N. Z. Benisi, M. Aminian, and B. Javadi, "Blockchain-based decentralized storage networks: A survey," *Journal of Network and Computer Applications*, vol. 162, p. 102656, 2020.

[3] Y. Psaras and D. Dias, "The interplanetary file system and the filecoin network," in *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2020, pp. 80–80.

[4] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford, "Heading off correlated failures through Independence-as-a-Service," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 317–334.

[5] K. Werder, B. Ramesh, and R. Zhang, "Establishing data provenance for responsible artificial intelligence systems," *ACM Transactions on Management Information Systems (TMIS)*, vol. 13, no. 2, pp. 1–23, 2022.

[6] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.

[7] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1271–1287.

[8] M. Al-Bassam, "Lazyledger: A distributed data availability ledger with client-side smart contracts," *arXiv preprint arXiv:1905.09274*, 2019.

[9] E. Kokoris Kogias, E. C. Alp, L. Gasser, P. S. Jovanovic, E. Syta, and B. A. Ford, "Calypso: Private data management for decentralized ledgers," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 586–599, 2021.

[10] S. Williams, V. Diordiiev, L. Berman, and I. Uemlianin, "Arweave: A protocol for economically sustainable information permanence," *Arweave Yellow Paper*, 2019.

[11] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.

[12] C. Li, M. Xu, J. Zhang, H. Guo, and X. Cheng, "Sok: Decentralized storage network," *Cryptology ePrint Archive*, 2024.

[13] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[14] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 2005, pp. 191–201.

[15] K. Nazirkhanova, J. Neu, and D. Tse, "Information dispersal with provable retrievability for rollups," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 180–197.

[16] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino *et al.*, "Sui lutris: A blockchain combining broadcast and consensus," *arXiv preprint arXiv:2310.18042*, 2023.

[17] I. Storj Labs, "Storj: A decentralized cloud storage network framework," 2018. [Online]. Available: https://storj.io

[18] D. Vorick and L. Champine, "Sia: Simple decentralized storage," *Retrieved May*, vol. 8, p. 2018, 2014.

[19] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[20] D. Catalano and D. Fiore, "Vector commitments and their applications," in *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*. Springer, 2013, pp. 55–72.

[21] K. Rashmi, N. B. Shah, and P. V. Kumar, "Enabling node repair in any erasure code for distributed storage," in *2011 IEEE international symposium on information theory proceedings*. IEEE, 2011, pp. 1235–1239.

[22] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1751–1767.

[23] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2518–2534.

[24] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5359–5376.

[25] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Aggregatable distributed key generation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 147–176.

[26] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 444–460.
[27] T. T. rs team, "Axum," 2025. [Online]. Available: https://github.com/tokio-rs/axum
[28] M. Labs, "Fastcrypto," 2025. [Online]. Available: https://github.com/MystenLabs/fastcrypto
[29] Metar, "Rocksdb," 2025. [Online]. Available: https://rocksdb.org
[30] malair, "Reed-solomon simd," 2025. [Online]. Available: https://github.com/AndersTrier/reed-solomon-simd
[31] T. S. team, "Build beyond," 2025. [Online]. Available: https://sui.io
[32] R. Anderson, "The eternity service," in *Proceedings of Pragocrypt '96*, 1996.
[33] B. Carlsson and R. Gustavsson, "The rise and fall of napster-an evolutionary approach," in *International Computer Science Conference on Active Media Technology*. Springer, 2001, pp. 347–354.
[34] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," in *Proceedings first international conference on peer-to-peer computing*. IEEE, 2001, pp. 99–100.
[35] R. Dingledine, M. J. Freedman, and D. Molnar, "The free haven project: Distributed anonymous storage service," in *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Springer, 2001, pp. 67–95.
[36] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Springer, 2001, pp. 46–66.
[37] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on networking*, vol. 11, no. 1, pp. 17–32, 2003.
[38] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer, 2001, pp. 329–350.
[39] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 53–65.
[40] D. S. Wallach, "A survey of peer-to-peer security issues," in *International symposium on software security*. Springer, 2002, pp. 42–57.
[41] J. P. Timpanaro, T. Cholez, I. Chrisment, and O. Festor, "Bittorrent's mainline dht security assessment," in *2011 4th IFIP International Conference on New Technologies, Mobility and Security*. IEEE, 2011, pp. 1–5.
[42] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6. Berkeley, CA, USA, 2003, pp. 68–72.
[43] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.
[44] N. Marina, A. Velkoska, N. Paunkoska, and L. Baleski, "Security in twin-code framework," in *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, 2015, pp. 247–252.
[45] M. Al-Bassam, A. Sonnino, V. Buterin, and I. Khoffi, "Fraud and data availability proofs: Detecting invalid blocks in light clients," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 2021, pp. 279–298.