Walrus: An Efficient Decentralized Storage Network

The MystenLabs Team
hello@mystenlabs.com
v2.0 – April 11, 2025

Abstract

Decentralized storage systems face 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 storage-node 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, a two-dimensional erasure coding protocol that achieves high security with only 4.5x replication factor, while enabling self-healing recovery that requires bandwidth proportional to only the lost data ($O(|blob|/n)$ versus $O(|blob|)$ in traditional systems). Crucially, 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.

Walrus also introduces 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.

1 Introduction

Blockchains support decentralized computation through the State Machine Replication (SMR) paradigm [37]. 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 [6] networks emerged to store blobs more efficiently. For example, early networks like IPFS [30] offer robust resistance to censorship, enhanced reliability and availability during faults, via replication on only a small subset of nodes [47].

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 on traditional stores 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 [45]. 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 [22, 29].
- Decentralized storage plays a critical role in ensuring data availability for roll-ups [1], 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).
- Decentralized social network platforms [18] are trying to challenge centralized incumbents. But the nature of social networking requires support for rich media user content, such as long texts, images or videos. Beyond social, collaborative platforms as well as civic participation platforms [4] need a way to store both public interest data and the application data itself in credibly neutral stores such as decentralized stores.
- Finally, the integration of decentralized storage with encryption techniques marks a significant paradigm shift [19]. This approach offers users comprehensive data management aligned with the Confidentiality, Integrity, and Availability (CIA) triad, eliminating the need to rely on cloud 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, our decentralized storage system, 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 stored in a credibly neutral store that provides

[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

Archived copy · cryp2nova.com

high authenticity, integrity, auditability and availability – all this at a reasonable cost and low complexity.

Approaches to Decentralized Storage

Protocols for decentralized storage generally fall into two main categories. The first category includes systems with *full replication*, with Filecoin [30] and Arweave [46] 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 [16], where malicious actors can pretend to store multiple copies of a file, undermining the system's integrity.

The second category of decentralized storage services [23] uses *Reed-Solomon (RS) encoding* [32]. 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.

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 stored and retrieved 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 3)
- 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 4).
- We show how Walrus leverages Red Stuff to implement the first asynchronous challenge protocol (Section 4.6)
- We provide a production-ready implementation of Walrus and deploy a public testnet of Walrus. We then measure its performance and scalability in a real environment (Section 7).

2 Models and Definitions

Walrus relies on the following assumptions.

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.

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

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

Archived copy · cryp2nova.com

Table 1: Comparing Replication Algorithms

	Replication for 10^{-12} Security	Write/Read Cost	Single Shard Recovery Cost	Asychronous Challenges
Replication	25x	$O(n\|blob\|)$	$O(\|blob\|)$	Unsupported
Classic ECC	3x	$O(\|blob\|)$	$O(\|blob\|)$	Unsupported
RedStuff	4.5x	$O(\|blob\|)$	$O(\frac{\|blob\|}{n})$	Supported

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.

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 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 Decode(T, t, n) where T is a set of at least t correctly encoded symbols, and it returns the blob B.

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 [8] and have implemented critical Walrus coordination protocols in the Move smart contract language [7].

3 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.

3.1 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 $O\left(\frac{|B|}{|p|}\right)$ 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.*

We present the ACDS protocols in a context where the storage node set is fixed and static. And in subsequent sections describing its use within Walrus, we discuss how it is adapted to allow for churn into the committees of storage nodes.

3.2 Strawman Design

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

Strawman I: Full Replication. The simplest protocol uses full replication in the spirit of Filecoin [30] and Arweave [46]. 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 $O(n|B|)$ 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 $O(n|B|)$ cost per recovering storage node with a total cost of $O(n^2|B|)$ over the network. Similarly, even a read can be very inefficient in asynchrony, as the reader might need to send $f+1$ requests costing $O(n|B|)$.

Strawman II: Encode & Share. To reduce the upfront data dissemination cost, some distributed storage protocols such as Storj [39]

Archived copy · cryp2nova.com

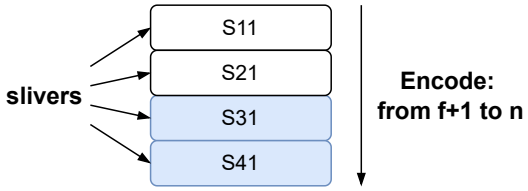


Figure 1: Encoding a Blob in one dimension. First the blob is split into $f+1$ systematic slivers and then a further $2f$ repair slivers are encoded

and Sia [43] use RS-coding [32]. 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 $O(\frac{|B|}{n})$. The writer W then commits to all the slivers using a binding commitment such as a Merkle tree [27] 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.

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 [10, 27]. 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 $O(|B|)$ even if the storage cost afterwards is $O(\frac{|B|}{n})$.

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 $O(|B|)$[5], which is the same cost as reading it. However, complete dispersal still costs $O(n|B|)$, 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 $O(n|B|)$ 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.

[4]Writer W could prove consistency among all slivers, but this is overkill for ACDS.
[5]There may be an extra $O(\log n)$ cost depending on the commitment scheme.

3.3 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 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 2) and the Twin-code framework [31], 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.

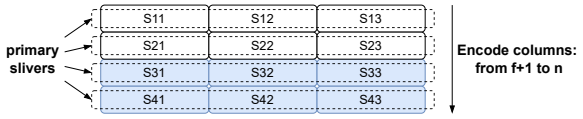
Encoding. Our starting point is the second strawman design that splits the blobs into $f+1$ slivers. Instead of simply encoding repair slivers, we first add one more dimension to the splitting process: the original blob is split into $f+1$ primary slivers (vertical in the figure) into $2f+1$ secondary 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 3.2. In order to provide efficient recovery for each sliver, we also take the initial $[f+1, 2f+1]$ 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.

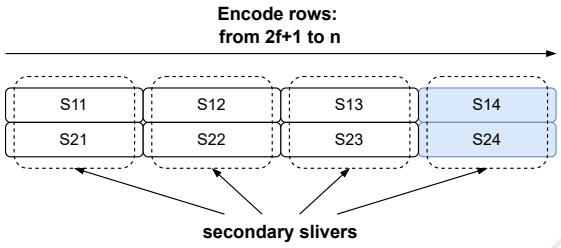
Handling Metadata. For each sliver (primary and secondary), W also computes vector commitments over its symbols. For each primary sliver, the commitment commits to all symbols in the expanded row, and for each secondary sliver, it commits to all symbols in the expanded column. As a last step, the client creates a commitment over the list of these sliver commitments, which serves as a *blob commitment*.

These vector 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. However, these proofs require the opening of the commitments for

Archived copy · cryp2nova.com



(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.

Figure 2: 2D Encoding / Red Stuff

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 they invoke a write or recovery process, 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[6]. This reduces the overhead to a constant per node, i.e., from quadratic to linear system-wide overhead.

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 all of sliver commitments to every node, along with their respective sliver pair. The nodes check their own sliver in the 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 will be available.

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 may need 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.

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

R first collects 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 recovery. 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 their sliver. 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.

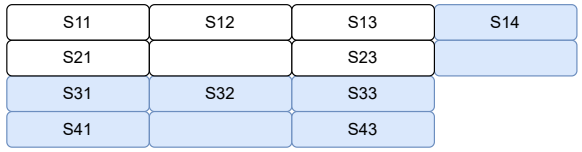
Since the size of a symbol is $O(\frac{|B|}{n^2})$ each, and each storage node will download $O(n)$ total symbols, the cost per node remains at $O(\frac{|B|}{n})$ and the total cost to recover the file is $O(|B|)$ which is equivalent to the cost of a Read and of a Write. As a result by using Red Stuff, the communication complexity of the protocol is (almost[7]) independent of n making the protocol scalable.

Red Stuff is an ACDS. Section 5 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

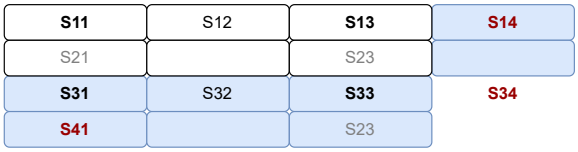
[6] They should also compute a commitment and an opening proof of their sliver.

[7] Depends on the commitment scheme used.

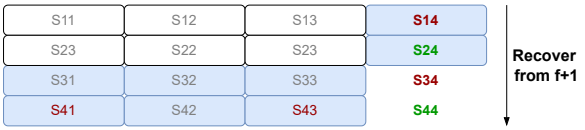
Archived copy · cryp2nova.com



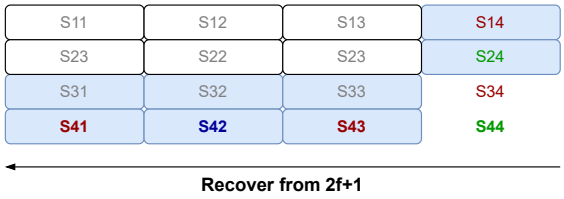
(a) Nodes 1 and 3 collectively hold two rows and two columns



(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)

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

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.

4 The Walrus Decentralized Secure Blob Store

Walrus is the integration of 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. This architecture uses the Red Stuff encoding/decoding algorithm described in section 3.3, Merkle trees [27] as set commitments, and the Sui blockchain [8] . Walrus can, however, be generalized to any blockchains and encoding/decoding algorithm that satisfies the minimal requirements described in Section 2.

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.

4.1 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

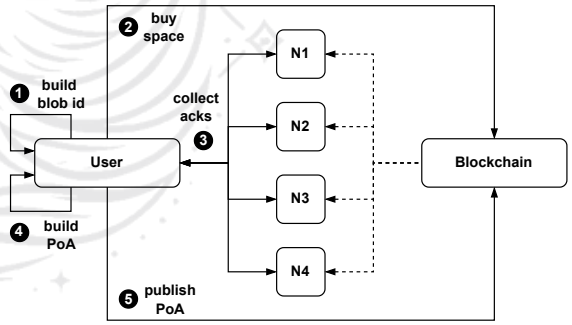


Figure 4: Walrus write flow. The user generates the blob id of the file they wish to store; acquire storage space through the blockchain; submit the encoded file to Walrus; collect $2f+1$ acknowledgements; and submit them as proof of availability to the blockchain.

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 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.

Archived copy · cryp2nova.com

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.

4.2 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.

4.3 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 4.6).

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 sliver.

4.4 Handling Inconsistent Encoding from Malicious Writers

One last case we need to discuss is when the client is malicious and 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 from the symbols that it received. However, in this case it is guaranteed to generate a third party verifiable proof of inconsistency, associated with id_B.

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 excluding it from the challenge protocol (section 4.6). 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 reply with $\perp$ to any request for the inconsistent blob's slivers, along with a pointer to the on-chain evidence for the inconsistency.

4.5 Committee Reconfiguration

Walrus is a decentralized protocol, hence 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 PoA are available, no matter if the set of storage nodes changes. Subject of course to $2f + 1$ nodes being honest in all epochs. Reconfiguration may take hours if a significant amount of data needs to be transferred between nodes. In that period, Walrus must continue to perform reads and writes for blobs to ensure no downtime.

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 main 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 a storage node to transfer them over to the new 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 of epoch e. Fortunately, by using Red Stuff, the bandwidth cost of the faulty case is the same as that of the fault-free case. but it still requires more messages to be sent over the network and more computation to verify proofs and to decode symbols to slivers.

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

Archived copy · cryp2nova.com

point at which both reads and writes are handed over to the new committee. This can unfortunately create challenges when it comes to reading these fresh blobs, as during the handover period it is unclear which nodes store the data. To resolve this, we include 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.

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.

4.6 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. In this section, we first present the simple protocol that is theoretically secure but costly. Then we discuss a relaxation that makes the security probabilistic but reduces the cost of challenging significantly and can be tuned dynamically if reads start to fail.

Fully Asynchronous Challenge Protocol. Close to the end of the epoch, the storage nodes witness a "challenge start" event on-chain, such as a specific block height. At that point, they stop serving read and recovery requests and broadcast an acknowledgment. When $2f+1$ honest nodes have entered the challenge phase, the challenges start.

Every challenged node sends the common symbols per blob to each other along with a proof against the commitment of the writer of the blob. The receiving nodes check the symbols and send a confirmation signature. When the proving storage node collects $2f+1$ signatures, it forms a certificate, which it submits on-chain. When $2f+1$ certificates are valid, the challenge period ends, and the reads and recovery are re-enabled.

During the challenge period, the nodes that witnessed the challenge start message do not respond to read or recovery requests. Since the threshold for starting a challenge is $2f+1$, at least $f+1$ honest will not reply after the challenged files are determined. As a result, even if the adversary has f slivers stored and has slowed down f honest nodes to not see the challenge start message, it can only get $2f$ symbols from their secondary slivers and then $2f$ signatures on its certificate. These are not enough to recover the full secondary sliver and convince the rest of the honest nodes to sign the certificate, and as a result, it will fail the challenge. The proof can be seen in Section 5.4.

Relaxations. Although this protocol is secure, it has the caveat that no reads are served during the challenge period and that a full-blown challenge requires bandwidth equal to the amount stored. To reduce its impact, we plan to trade-off security for allowing most blobs to be readable and not under a challenge.

For the lighweight challenge protocol, we require the storage nodes to setup a random coin with a $2f+1$ reconstruction threshold. This is possible using any kind of asynchronous DKG [13, 14, 20] or randomness generation protocol [17, 40].

The coin is used to seed a pseudo-random function (PRF) that defines which blobs need to be challenged per storage node. Any blob not in the set can be accessible directly again. The number of blobs challenged needs to be sufficiently large compared to the total number of blobs such that storage nodes have a negligible probability of holding all the challenged blobs unless they hold the overwhelming majority of blobs. For example, if a storage node holds 90% (99%) of the blobs, it has less than a 10^{-30} probability of success in a 640 (7000) file challenge.

If we notice that reads fail although challenges pass, it means that we do not challenge enough files. In this case, Walrus will increase the challenges up to the point of reenabling the full challenge protocol. However, for this to happen, it means that the malicious storage nodes have minimal storage savings (less than a constant factor), which is unlikely to have a real impact on their resource cost.

5 Red Stuff and Walrus Proofs

This section completes Section 3 by showing that Red Stuff satisfies all the properties of a ACDS. The casual reader can skip it.

5.1 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 $S^{(p,i)}$.*

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).*

Archived copy · cryp2nova.com

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.

□

5.2 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 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.

□

5.3 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. □

5.4 Asychronous Challenges

We prove that no malicious storage node that does not hold all slivers of blobs will succeed in replying to a challenge. We note that storage nodes of the adversary can collude and hold a single version of their common symbols. However, they would still need to hold $2f+1$ symbols per sliver which is effectively the cost of a full sliver as far as storage is concerned.

Theorem 4 (Secure Challenge Protocol)**.** *No malicious storage node running Walrus that does not hold all slivers of blobs will succeed in replying to a challenge.*

Proof. We assume there exists a storage node that deletes a symbol it is supposed to hold and the respective symbol is held by an honest node. At challenge time it will need to send this symbol to the honest node to get a certificate. To do so, it will need to recover the sliver. For this it will get $f-1$ symbols from the other malicious storage nodes. It can also get f symbols from slow, honest nodes through the read path. 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.

If it fails to reply then it cannot get a certificate as only the $2f-1$ node mentioned above plus its self-signature will be collected, resulting in $2f < 2f+1$ signatures.

□

6 Economics and Incentives

Beyond simple reads and writes, we design robust economic and incentive mechanisms for Walrus to ensure competitive pricing, efficient allocation of resources, and minimal adversarial behavior. Conceptually, Walrus's economic challenges differ from those of a typical blockchain, since Walrus leverages a blockchain as a control plane, and hence inherits the security of the blockchain consensus.

Instead, Walrus faces challenges around enforcing long-term contracts. Walrus nodes and users strike agreements in advance to store data for some duration of time and provide access to that data in the future. But nodes may be tempted to default over time, e.g. resell committed storage at higher prices to other users or withhold later access unless ransoms are paid. Moreover, given that Walrus is decentralized nodes may simply churn from the system over time and leave it to future committees of nodes to fulfill their commitments without even fully internalizing future reputation costs when acting in their own interests, as the well-known "tragedy of the commons" problem. Unlike in traditional markets, where long-term contracts can be enforced through the

Archived copy · cryp2nova.com

legal system, Walrus has to rely on incentives and disincentives to mitigate this.

The primary tool that ensures contracts are honored is staked capital – oriented around the new WAL token – which earns rewards for good behavior and is slashed for bad behavior. This capital can come from the nodes directly, or it can come from users who delegate their capital to well-run nodes. In tandem, the economics behind Walrus continues to assume that up to $f = \lfloor \frac{n}{3} \rfloor$ nodes (or, more accurately, nodes controlling up to $\lfloor \frac{n}{3} \rfloor$ of the shards) are Byzantine, building robustness for arbitrary failures including malicious actions. Finally, this principal tool is paired with other mechanisms to coordinate pricing, access, migrations, and integrity checks in the system.

This section proceeds as follows. Section 6.1 describes the staking mechanism that underpins the security and efficiency of the entire system. Section 6.2 outlines the migration of shards, when new nodes join the system (or increase relative stake) or existing nodes exit the system (or decrease relative stake). Section 6.3 explains how nodes collectively set prices in competitive and fair ways. Finally, Section 6.4 briefly explains the token governance processes.

6.1 Staking

Staking of WAL tokens, by or on behalf of storage nodes, underpins Walrus's security. By earning rewards when storage nodes honor commitments and being slashed when nodes do not. Staking on Walrus has four core components: assignment of stakes and shards, the unstaking process, the accrual of rewards and penalties, and the adjustments needed for self-custodial objects.

Stake and Shard Assignment. Walrus includes an in-built layer of delegated staking, so that users can participate in the network's security regardless of whether they operate storage services directly. Nodes compete with one another to attract stake from users, which in turn governs the assignment of shards to them.

Users make this choice based on the storage node's reputation, the node's own staked capital, the commission rates set, and other factors. Once the epoch is set to change, i.e. point c in Figure 5, the stake is considered locked in its assignment to storage nodes (and any unstaking requires the formal unstaking process described next). Shards are now assigned to nodes for the upcoming epoch in proportion to the nodes' associated stakes. This is described in more detail in Section 6.2.

Walrus does not impose requirements on storage nodes to provide some minimal level of capital, allowing nodes to choose between funding some, all, or none of its own capital. This is deliberate. Delegators will likely consider a nodes' own capital when making delegation decisions, but Walrus need not be prescriptive of the required level.

Walrus does impose one safeguard on commission rates: they must be set by nodes a *full* epoch before the cutoff point. As such, if a node raises commission rates too far, its delegators have sufficient time to unstake before those new rates go into effect.

Unstaking. The unstaking process is similar to the staking process, in that a wallet must register the request before the cutoff point in a given epoch (point c in Figure 5). Once the cutoff point is reached, the departing stake no longer counts towards the node's aggregate stake for allocating shards. However, the departing stake is not released at that point. Instead, the departing stake is held by Walrus through the shard migration process described in Section 6.2, which takes place after the epoch ends; and its value can be slashed if the shard migration procedures involving its node fail. The departing stake is subsequently released. If the node is not involved in shard migration processes, the departing stake is released at the end of the epoch instead. Walrus again imposes no further requirements on the process.[8]

Rewards and Penalties. At the end of each epoch, a node earns rewards or is punished based on its actions throughout the epoch. Nodes that have answered challenges (Section 4.6) correctly and thus have proven data storage, have facilitated writes to the system, or have participated in recovery of shards earn rewards. Nodes that have not answered challenges correctly are penalised. These reward and penalty rates are based on a combination of protocol revenue and parameters tuned by token governance, and they are discussed later. In addition, penalties on nodes around the recovery process for shards are levied earlier, at the end of the migration period. This allows capital that wishes to unstake from the system to be potentially slashed and complete the exit.

Users share in rewards and penalties on a proportionate basis, for the full epochs that their stakes are active. Storage nodes themselves can bring their own capital to earn similar rewards and penalties; and storage nodes separately earn commissions from the rewards that they bring to their delegators. Initially Walrus will treat all stake equivalently, such that there are no junior or senior stake tranches, and all stakes earn rewards and penalties at the same rate. This can be changed in the protocol later, at the community's discretion; or it can be built as custom functionality on top of Walrus by a given node.

Self-Custody in Staking. Like Sui [7], Walrus will implement staking via self-custodied objects. When a wallet stakes funds, those funds are wrapped into an object that the wallet holds. This lowers the vulnerability surface for the Walrus system, and it allows users to build functionality atop those objects. However, this poses an operational problem, as Walrus can slash staking principal but this design means that Walrus does not actually hold custody of the principal.

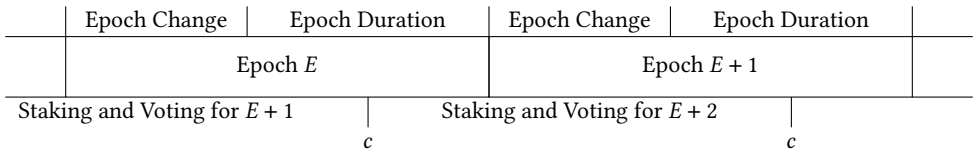
To solve this, Walrus keeps track of any outstanding penalties. When a user wants to reclaim their WAL tokens, they must provide the object to the Walrus smart contracts to have those tokens unwrapped. At the point of unwrapping, those outstanding penalties are assessed on the stake.

This leads to two problems. First, Walrus may face an interim cash flow problem, especially since penalties are sometimes given to other participants to offset the harm imposed on them by the slashed node. Thus, Walrus has two mitigations. The first mitigation is that penalties come out of any rewards being paid to stake, including both by Walrus reclaiming uncollected rewards and garnishing future reward distributions. The second mitigation is that outstanding penalties accrue with interest over time, until

[8] One potential extension for Walrus is to impose an extra epoch delay for a node withdrawing its own capital. This prevents outcomes where the node unstakes its own capital in the last few seconds before an cutoff, and thus withdraws all their capital a full epoch before its delegators, presumably in anticipation of adversarial behavior during that epoch. This mirrors the extra epoch delay for nodes setting commission rates.

Archived copy · cryp2nova.com

Figure 5: Timeline of Walrus Operations



the wallet chooses to settle either by sending tokens directly to Walrus or by unstaking and paying at the point of unwrapping.

Second, objects slashed to zero or near-zero may not be returned by their owners to Walrus, keeping Walrus from claiming those tokens. Thus, Walrus will always redeem a staking object for some baseline amount (e.g. 5%) of its initial principal, regardless of the magnitude of slashing. This small incentive for users should motivate them to return all staked objects to Walrus eventually for unwrapping, and allow Walrus to reclaim the slashed tokens.

6.2 Shard Migration

Shards migrate between nodes as their relative stake rises and falls. This migration is important for Walrus's security, as otherwise Byzantine behavior in a minority of nodes by stake could halt the system. Shard migration has three components: the assignment algorithm, the cooperative pathway of migration, and the recovery pathway.

Assignment Algorithm. Once the cutoff point prior to an epoch's conclusion is reached (i.e. point c in Figure 5), the assignment algorithm proceeds. Staking requests are considered finalized and the associated stake is included; and unstaking requests are similarly considered finalized and the associated stake is excluded (though not released). Shards are assigned on the basis of relative stake across nodes. However, the shard transfer process does not formally begin until the start of the next epoch.

The shard assignment algorithm maintains stability between nodes and shards where possible, and so tries to minimize transfers. Specifically, nodes that gain shards keep all their current shards and only gain extra shards lost by other nodes. In addition, there is some tolerance when losing a shard, such that a node that just earns an extra shard by having some ϵ extra stake will not immediately lose the shard if it loses that ϵ stake in a subsequent epoch.

In future iterations, the shard assignment algorithm can be made more accommodating, e.g. allowing nodes to specify preferences for shards that the assignment algorithm respects, or allowing nodes to trade migration obligations as long as the protocol's overall shard allocation is respected. For instance, nodes that co-locate storage racks might prefer to transfer shards with one another to save on ingress and egress charges (as many storage providers do not charge for intra-facility transfers). Alternatively, if two nodes are migrating one shard each to two other nodes, the four parties can mutually agree to swap obligations for efficiency purposes (perhaps proposed by a third party optimizing transfer costs for Walrus holistically; and perhaps facilitated in turn by side payments between the four parties).

Assignment is done *solely* on the basis of stake disregarding a storage node's storage limit. In other words, a node may be forced to take shards beyond its storage capacity, if its relative stake increases through an influx of stake or an outflow for competitors' stake.[9] This is why the assignment algorithm is run prior to the epoch's conclusion, to allow nodes in this position to have enough time to provision more storage. In addition, nodes can always choose to halt new staking requests, if they wish to lower the risk of provisioning new storage on short notice.

Cooperative Pathway. The shard migration process begins once the epoch ends. It begins with a period of time where nodes are expected to transfer shards. If this is done cooperatively, i.e. the sending node and receiving node coordinate, then no tokens change hands and no further action is taken.

Specifically, for each bilateral relationship of (sender, receiver), the receiver attests to having the shards during the transfer interval. The process is then considered complete, and the receiver is responsible for future challenges to the shards. Once each bilateral transfer a node is involved with finishes, the unstaking requests for that node (if any) are processed.

This mechanism nests a voluntary node exit from the system. The node would mark its entire stake to be withdrawn, cooperate in migrating out shards, and then reclaim all tokens once the receivers attest.

Recovery Pathway. Since Walrus is an open system, it needs to be robust to failures in the shard migration. If, at the end of the transfer interval, the receiving nodes either attest to not having received all shards or do not attest, the transfer is considered failed and the recovery pathway is initialized.

Under the recovery pathway, the sending node's stake is slashed meaningfully, with a penalty amount set by governance. All other nodes participate in recovering the shard (4.3), and the slashed funds are divided amongst them to offset their costs. Importantly, the receiving node's stake is *also* slashed. It is slashed by a relatively small amount, also set by governance, to set incentives correctly on not misreporting a cooperative migration. Once recovery is complete, the receiving node is responsible for challenges to the shards.[10]

Non-Migration Recovery. The recovery pathway in Walrus is primarily for shard migrations, but it can also be used for recovery

[9] An alternate model is to assign shards on the basis of a node's stake and self-declared storage capacity. This model is more robust in terms of individual choices and incentives, but it is more complex operationally. For instance, this model would require Walrus to slash nodes that reduce storage capacity beyond the amount of storage already committed to users, and potentially use those slashed funds to subsidize other nodes who offset the declines.

[10] This mechanism handles nodes withdrawing from Walrus under many adversarial scenarios, but it does so inelegantly for a scenario with a fully unresponsive node. A fully unresponsive node would eventually lose its shards through penalties levied on its stake, but that process would be gradual and take many epochs, constraining Walrus in the meantime. One possible idea for future development is for Walrus to build an emergency migration system that confiscates *all* shards for a node that fails a supermajority of data challenges in several consecutive epochs.

Archived copy · cryp2nova.com

outside of migrations (e.g. a node who suffers physical damage to a hard drive). In this flow, the node missing a shard voluntarily places a request on chain and is slashed by the same penalty amount that the sending node pays during shard migration recovery. A node might do this voluntarily because the alternative is for the node to repeatedly fail the data availability challenges in Section 4.6 before eventually needing to migrate away the shard, which would be more costly. Nodes who participate in the recovery of the missing shard earn these slashed funds, to offset their costs.

In addition, nodes can perform ad-hoc recovery of blobs from one another. These are on a best-effort basis only, and nodes are free to rate limit one another. This pathway handles particular blobs that are missing (especially from recent epochs), but it likely does not scale to entire missing shards.

6.3 Payments for Storage and Writes

Given the distributed nature of Walrus, storage resources and writes must be priced in competitive and collaborative ways. Storage nodes should compete with one another to offer ample storage at low prices, but the system must then jointly represent their submissions as a unified schedule to storage consumers. This section covers the preliminaries of storage resources, how nodes determine the quantity of storage resources, how nodes determine the prices of storage resources and writes, and the payments made to nodes.

Storage Resources. Storage is bought and sold on Walrus as storage resources, represented on the Sui blockchain. These act as reservations for storage space in Walrus, and accordingly have a starting epoch, an ending epoch, and a size. Users register resources to hold specific blobs when ready to store data, and proceed to write the data and establish the Point of Availability.

Prior to being attached to a blob, storage resources can be split across time or across space. Storage resources can also be traded. Finally, storage resources can be disassociated from a deleted blob and reassociated with a new blob. This establishes the foundation for a robust secondary market for storage resources, which allows for economic efficiency in Walrus.

Storage Quantity. Nodes first determine storage for sale by voting on the system-wide shard size. Given the fixed number of shards and the replication factor, this immediately determines the system total size; and so, by extension, the unused storage that is available for sale.

Like staking, voting for storage quantities and prices alike takes place in advance of the epoch. The cutoff point for staking is the same cutoff point for voting (i.e. point c in Figure 5). Storage nodes that do not vote simply have their previous vote populated as their current vote. At that point, all submissions for shard size are ordered in decreasing order and the 66.67th percentile (by stake weight) submission is determined to be the shard size, such that 2/3 of submissions are for larger shard sizes and 1/3 of submissions are for smaller shard sizes.

At this point, unused storage, i.e. the difference between total storage capacity and storage already committed, is calculated. If there is any unused storage, it goes on sale in the upcoming epoch and starts in that epoch. Users specify an ending epoch (between the current epoch and an epoch two years out) and a size when making the purchase, creating a storage resource. The upper bound on the ending epoch is to prevent early generations of nodes from making choices that lock in too many successive generations of nodes. If total storage capacity is *lower* than committed storage, no new storage is available for sale but Walrus nodes are still required to honor the commitments made. They cannot delete storage resources, as they will fail challenges and be slashed, as described in Section 4.6.

Although storage is generally sold as resources that start in the current epoch, Walrus allows one alternate flow to facilitate renewals of existing blobs efficiently. Users with existing blobs can extend the lifespan of those blobs from their current ending epoch to any later epoch (up to two years' out) and pay the posted price for the additional epochs. Renewing parties enjoy a small embedded option, as prices for the next epoch are set midway through the current epoch and when current storage is on sale at current epoch prices. This is minor as long as prices are generally stable in successive epochs. This model allows users to have long-lived or potentially infinite storage lifetimes for their data without lapsing coverage or double-paying for epochs.

Pricing of Storage and Writes. Nodes set the shard size, but also the prices for storage and for writes. Again, over a full epoch in advance and before the cutoff point c in Figure 5, nodes submit prices for storage resources (on a per unit of storage, per epoch basis) and for writes to the system (on a per unit of storage basis). These prices are independently ordered in *ascending* order, and the 66.67th percentile (by stake weight) submissions are selected, such that 2/3 of submissions are for lower prices and 1/3 of submissions are for higher prices.[11] The storage price is taken as is, while the write price is multiplied by a hardcoded factor greater than one.

This hardcoded factor reflects an additional refundable deposit. Walrus formally is responsible for a blob once the Point of Availability is publicly visible on-chain. However, Walrus runs more efficiently if all nodes receive the blob directly from the user, as it saves the overhead needed to recover missing symbols for the nodes missing the data. The refundable deposit on writes incentivizes this. The more node signatures that a user collects on its certificate, the more of that deposit is returned. As such, a user is motivated to upload the data to all nodes and not simply $2f + 1$ of them.

Payments to Nodes. Payments are simple for Walrus at launch. First, users pay the current price for writing data when registering the blob, and those are distributed to nodes at the end of the epoch. Second, users pay for the storage resource at the time of purchase (regardless of when they register the blob). Here, the Walrus smart contracts divide these tokens amongst shard-level buckets for the associated nodes to receive at the end of the epoch.

This simple model of payments for storage resources delivers two core benefits. First, user prices are fixed and prepaid. Users do not need to worry about fluctuations in the price of storage or the WAL token during the lifetime of the contract, and nodes cannot exert predatory pricing on users midway through a contract. Second, contract lengths are fixed and users cannot exit them without forfeiting their payment. Without this, nodes could not lower prices on storage in the future without risking all existing contracts canceling

[11] This resembles the model behind Sui gas fees. As a brief reminder, Sui validators propose reference prices and the 66.67th percentile price is selected. Sui separately allows validators to assess each others' performance subjectively, which is analogous to the data challenges in Walrus outlined in Section 4.6.

Archived copy · cryp2nova.com

and renewing at the lower rate. Thus, the long-term enforcement on both sides delivers stability to the relationship.

This model does have two weaknesses, and so Walrus may add refinements later. First, this is capital-inefficient, as users must prefund the entire contract well in advance of receiving services. Second, nodes often have local currency costs but receive WAL revenues, and so fluctuations in the price of WAL will make it hard for them to compete efficiently. Thus, an alternate model could be one where users and nodes commit to long-term contracts, but with two differences. The first difference is that users only prepay for the last few epochs' of storage rather than the full contract, allowing them to be more capital-efficient but locking enough of a stake to not terminate contracts early. The second difference is that the contract price can only be updated programmatically based on an oracle price (e.g. WAL/USD), to offer more local-currency stability to both parties.

Future Improvements. Walrus currently offers a light incentive to nodes to increase storage capacity or lower prices. Specifically, an increase in the global shard size would prevent nodes without spare capacity from growing, giving a larger share of rewards to well-provisioned nodes; while a decrease in price would push inefficient nodes out of business, giving a larger share of rewards to efficient nodes. However, Walrus could offer much stronger incentives to nodes to expand capacity and drop prices. In particular, Walrus could explicitly reward nodes that vote for shard sizes that are larger than the 66.67th percentile submission and subsequently deliver on their stated preferences. Similarly, Walrus could again reward nodes that vote for prices lower than the consensus value. These potential improvements, if incorporated in the future, would more aggressively keep Walrus at the cutting edge of decentralized storage provision.

6.4 Token Governance

Governance for Walrus adjusts the parameters in the system, and operates through the WAL token. Specifically, nodes collectively determine the level of various penalties, with votes equivalent to their respective WAL stakes. This allows Walrus nodes, who often bear the costs of other nodes' underperformance, to calibrate the appropriate financial repercussions.

In contrast, governance for Walrus does not directly adjust the protocol. Protocol changes are effected by $2f + 1$ storage nodes accepting them at reconfiguration; and thus they are ratified implicitly by staked tokens. Changes to the protocol would likely only follow a robust debate about the security, economic, and business risks of any change. Many L1s have similar dynamics (e.g. Sui Improvement Proposals for Sui).

Governance Flow. Token governance within the context of Walrus is similar to the mechanism by which nodes set prices, differing only on parameter constraints and the method to find consensus. Specifically, it has the following flow:

(1) Until the staking cutoff point for an epoch (i.e. point c in Figure 5), any Walrus node can issue a proposal for the parameter set for the subsequent epoch, regarding costs associated for shard recovery and costs for failing to perform on data challenges.
(2) Once a proposal or set of proposals is live, Walrus nodes can vote for a single proposal (or for the status quo) in that same epoch, with their votes equal to their total stake, including delegated stake.
(3) At the cutoff point c, a proposal that earns over 50% of the votes cast will be implemented for the subsequent epoch, subject to total voting reaching quorum. If no proposal earns 50% of votes cast, the status quo earns 50%, or quorum is not reached, no proposal is implemented and the parameters remain at their current values.
(4) There are no minimum staking requirements to vote or issue proposals.

Governance-Determined Parameters. There are four parameters that the token governance model adjusts through this framework:

(1) The cost associated for shard recovery, for the node sending a shard. This is a per-shard penalty.
(2) The cost associated for shard recovery, for the node receiving a shard. This is a per-shard penalty.
(3) The cost associated for a node failing 50% or more of the data challenges issued. This is a per-shard penalty, and it is multiplied by the number of shards held by a node when assessed.
(4) The cost associated for a node failing to issue 50% of more of the data challenges. This is a per-shard penalty, and it is multiplied by the number of shards held by a node when assessed.

Any proposal that modifies these parameters must satisfy two constraints, in addition to trivial constraints like positivity. First, the shard recovery cost on the sending node must weakly exceed the shard recovery cost on the receiving node. This ensures the transfer onus remains on the sending node. Second, the cost associated with a node failing 50% or more of data challenges must also weakly exceed the shard recovery cost on the receiving node. This ensures that receiving nodes are incentivized to report a failed shard transfer correctly.

6.5 Incentivized Reads

Walrus is designed first and foremost for robust storage of blobs. To access the data, Walrus encourages storage nodes to provide free and rapid read access, but there are no strict requirements and any offering is on a best-effort basis only.

This should work since storage nodes are broadly aligned in making Walrus a successful system. In addition, we foresee other providers, such as caches or content distribution networks (CDNs), offering high-quality read access to data potentially for a fee payed by the reader or even the writer. But strictly speaking, a rational storage node might only hold the data but not serve them in the hopes that other storage nodes will reply (since only $f + 1$ replies are sufficient). This is a classic public goods problem [36] which could devolve into no storage nodes replying to the client. The dynamic will render the system unusable.

There are a few possible solutions to this problem. This paper briefly outlines three: node service models, on-chain bounties, and inclusion of light nodes formally in Walrus.

Node Service Models. One solution is to have users strike paid bilateral contracts with storage nodes to read data. These can take many forms beyond this simple partnership. For instance, a set of nodes could offer direct paid endpoints for users to pay for reads with some service level guarantees. Either readers may pay, or writers of blobs that want to make them available to all (such as in

Archived copy · cryp2nova.com

the case of publishing a website). Alternatively, those nodes could strike enterprise-level deals for parties to access data and resell access onwards. Indeed, this pathway is likely the default way that caches, publishers, and other content providers built atop Walrus will interact with the system, handling day-to-day user accesses for a targeted set of files and turning to Walrus to refresh and update that set over longer horizons.

While this mechanism does not provide a static incentive for nodes to return data, it does provide a dynamic one. Nodes who default on obligations will get bad reputations and will earn fewer future business opportunities. We foresee read infrastructures around serving reads reliably, with very low latency, in a geo-distributed manner, based on caches and content distribution networks (CDN) to develop around Walrus.

The primary advantage of this mechanism is that it does not require any change to Walrus itself. The actual mechanism, without further development, may be somewhat complex for end users (who may not know how to locate their encoded data across shards nor how to negotiate with nodes directly). But intermediaries can abstract much of that away. In doing so, they offer rich long-term business relationships to nodes and so provide stronger incentives for nodes to honor contracts.

On-Chain Bounties. Another solution is for users to post on-chain bounties to access data from Walrus when best-effort reads fail (this can also be done by publishers and caches if they are paid to provide a certain SLA). Specifically, a user that needs data posts a bounty, and either allocates the bounty to replying storage nodes once it receives data or posts a challenge to reclaim the bounty if it does not receive sufficient data. This incentivizes storage nodes to reply and earn bounties; and if a user fraudulently attempts to reclaim the bounty, nodes would overturn the user's challenge by posting the data on-chain and thus provide the data to the user.

This solution again requires no change to Walrus itself. These bounties are intermediated through smart contracts on Sui. The disadvantages are twofold. First, this is potentially a cumbersome way of receiving data, particularly if there are frequent disputes on whether the bounties should be paid. Second, this is complex for end users, who need to post bounties, allocate credit, post challenges, and download data post-challenge.

6.6 Decentralized Security Through Light-Node Sampling

A final way we can address the extreme case that data is unavailable in the system is by enabling and incentivizing a second class of participants, which play the same role as light nodes in Data Availability systems [1]. These light nodes provide a second, more decentralized layer of security. Storage nodes would face certain requirements to interact with these other participants, and could be slashed for failing to make data available to them.

This is naturally more complex in terms of ensuring security guarantees are met, but it is more robust. In addition, it offers a pathway for the community to be involved in the operation of Walrus without having to run full storage nodes. Finally, this could also be combined with the on-chain bounties solution.

Protocol. This is another place were our 2D encoding shines. Since the size of each symbol is only $O(\frac{|B|}{n^2})$ light nodes simply store randomly sampled symbols of files they consider important and expect bounties to be posted in case of unavailability. They can also post blames in a rate limited fashion if storage nodes do not send the symbols requested, similar to how on-chain bounties work.

The protocol is very simple; from the time the data is written, we allow light nodes to sample symbols from the storage nodes directly by performing best-effort reads, or download blobs through caches and re-encode them. Then, when an on-chain bounty is posted, the first light node to send a missing symbol to the client with a signature will be included in a resolution transaction (we can use the chain if we want fairness otherwise the client can simply reward itself as a Sybil). The resolution transaction is simply the addresses of all the light nodes that helped (with the number of symbols they provided) in order to recover the data. The reward is then split proportionally between the validators who helped with the incentivized read and the light nodes. The reward per symbols should be proportional to the reward per sliver such that the storage nodes are not incentivized to generate Sybil light nodes to earn more rewards.

In the future we will design even more efficient protocols based on off-chain channels to facilitate payments to light nodes that meaningfully contribute to shard recovery after reconfiguration.

7 Evaluation

We implement a *production-ready* networked multi-core Walrus storage node in Rust. All networking uses HTTPS through axum [35], it uses fastcrypto [21] for cryptography, rocksdb [28] for storage, and reed-solomon-simd [24] for erasure coding. We opt to connect our implementation to Sui [41] as an example of fast blockchain. We release the codebase as open-source[12].

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.

7.1 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.

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 6 details the shard distribution

[12] Link omitted for blind review.

Archived copy · cryp2nova.com

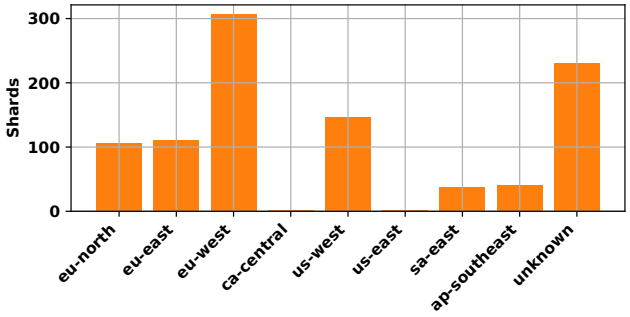


Figure 6: Geo-distribution of shards.

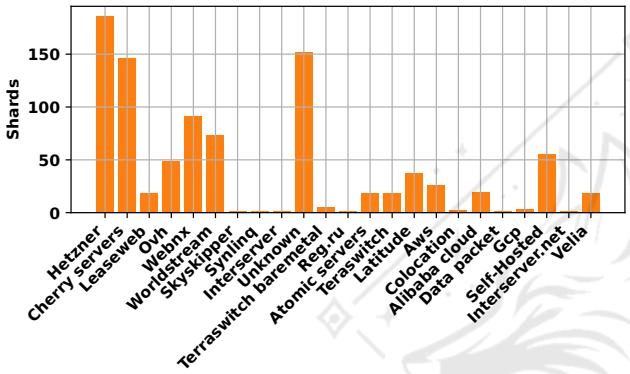


Figure 7: Distribution of shards by hosting providers.

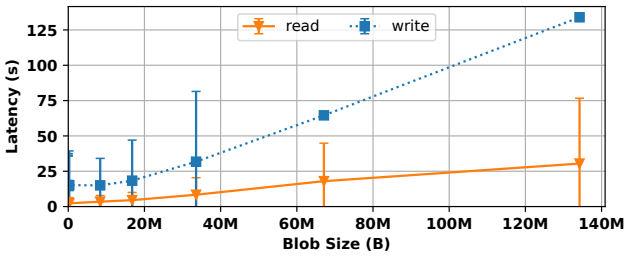


Figure 8: Latency for different blob sizes.

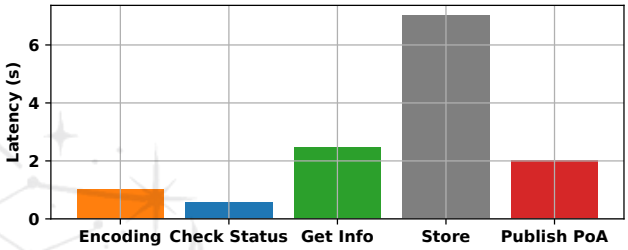


Figure 9: Latency breakdown for small blobs (1KB).

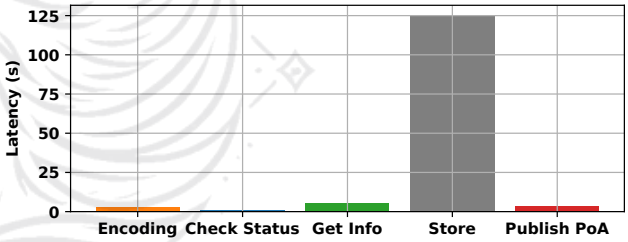


Figure 10: Latency breakdown for large blobs (130MB).

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 7 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).

7.2 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 client runs in US East (N. Virginia), the other in Canada Central.

Walrus Latency. Figure 8 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 9 and Figure 10 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 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.

Single Client Throughput. Figure 11 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

Archived copy · cryp2nova.com

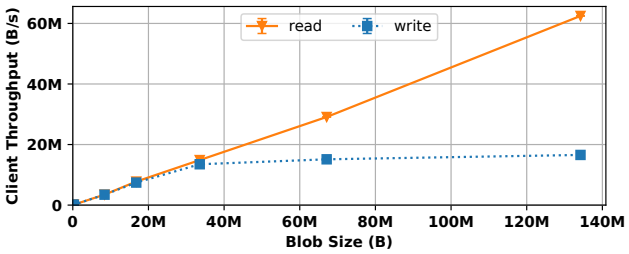


Figure 11: Single client throughput for different blob sizes.

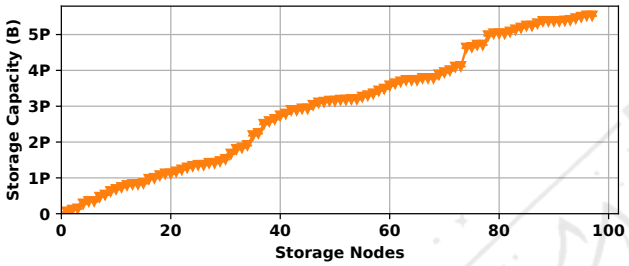


Figure 12: Storage capacity versus committee size.

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.

7.3 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). As described in Section 7.1, 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 12 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.

8 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 [3] in 1996, to ensure documents cannot be suppressed. Within the commercial and open source communities systems like Napster [9], Gnutella [33], and Free Haven [15] and early Freenet [11] 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 [38], Pastry [34], Kademlia [26], 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 attacks: Sybil attacks [16] were named and identified within the context of these systems first; and they are hard to defend against routing attacks [44]. Many attacks affect current systems that use them [42]. Bittorrent [12] 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 [8] 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 [5] 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 [30] 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 [46] 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. Furthermore, Walrus does not implement its own separate blockchain to do node management and provide incentives, but uses Sui instead.

Storj [39] 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

Archived copy · cryp2nova.com

replication factor, offering a substantial reduction in storage costs compared to prior systems. However, a key limitation of Storj lies in 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 [31], which uses two linear encodings of data to enhance the efficiency of sliver recovery. However, unlike the Twin-code framework [25], Red Stuff encodes data across differently sized dimensions and integrates authenticated data structures, achieving Completeness (as defined in Section 2) 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 [2], 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.

9 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.

References

[1] Mustafa Al-Bassam. 2019. Lazyledger: A distributed data availability ledger with client-side smart contracts. *arXiv preprint arXiv:1905.09274* (2019).

[2] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. 2021. 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, 279–298.

[3] Ross Anderson. 1996. The Eternity Service. In *Proceedings of Pragocrypt '96*.

[4] Pablo Aragón, Andreas Kaltenbrunner, Antonio Calleja-López, Andrés Pereira, Arnau Monterde, Xabier E Barandiaran, and Vicenç Gómez. 2017. Deliberative platform design: The case study of the online discussions in Decidim Barcelona. In *Social Informatics: 9th International Conference, SocInfo 2017, Oxford, UK, September 13-15, 2017, Proceedings, Part II 9*. Springer, 277–287.

[5] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[6] Nazanin Zahed Benisi, Mehdi Aminian, and Bahman Javadi. 2020. Blockchain-based decentralized storage networks: A survey. *Journal of Network and Computer Applications* 162 (2020), 102656.

[7] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc* (2019), 1.

[8] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. 2023. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042* (2023).

[9] Bengt Carlsson and Rune Gustavsson. 2001. The rise and fall of napster-an evolutionary approach. In *International Computer Science Conference on Active Media Technology*. Springer, 347–354.

[10] Dario Catalano and Dario Fiore. 2013. 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, 55–72.

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. 2001. 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, 46–66.

[12] Bram Cohen. 2003. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, Vol. 6. Berkeley, CA, USA, 68–72.

[13] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. 2023. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5359–5376.

[14] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. 2022. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2518–2534.

[15] Roger Dingledine, Michael J Freedman, and David Molnar. 2001. 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, 67–95.

[16] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[17] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. 2021. Aggregatable distributed key generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 147–176.

[18] Martin Kleppmann, Paul Frazee, Jake Gold, Jay Graber, Daniel Holmgren, Devin Ivy, Jeromy Johnson, Bryan Newbold, and Jaz Volpert. 2024. Bluesky and the AT protocol: Usable decentralized social media. *arXiv preprint arXiv:2402.03239* (2024).

[19] Eleftherios Kokoris Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Svetolik Jovanovic, Ewa Syta, and Bryan Alexander Ford. 2021. Calypso: Private data management for decentralized ledgers. *Proceedings of the VLDB Endowment* 14, 4 (2021), 586–599.

[20] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. 2020. 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*. 1751–1767.

[21] Mysten Labs. 2025. Fastcrypto. =https://github.com/MystenLabs/fastcrypto.

[22] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software* 39, 2 (2021), 62–70.

[23] Chuanlei Li, Minghui Xu, Jiahao Zhang, Hechuan Guo, and Xiuzhen Cheng. 2024. SoK: Decentralized Storage Network. *Cryptology ePrint Archive* (2024).

[24] malair. 2025. Reed-Solomon SIMD. =https://github.com/AndersTrier/reed-solomon-simd.

[25] Ninoslav Marina, Aneta Velkoska, Natasha Paunkoska, and Ljupcho Baleski. 2015. Security in twin-code framework. In *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, 247–252.

[26] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer*

Archived copy · cryp2nova.com

systems. Springer, 53–65.
[27] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.
[28] Metar. 2025. Rocksdb. =https://rocksdb.org.
[29] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. {CHAINIAC}: Proactive {Software-Update} transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*. 1271–1287.
[30] Yiannis Psaras and David Dias. 2020. 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, 80–80.
[31] KV Rashmi, Nihar B Shah, and P Vijay Kumar. 2011. Enabling node repair in any erasure code for distributed storage. In *2011 IEEE international symposium on information theory proceedings*. IEEE, 1235–1239.
[32] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
[33] Matei Ripeanu. 2001. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*. IEEE, 99–100.
[34] Antony Rowstron and Peter Druschel. 2001. 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, 329–350.
[35] The Tokio rs team. 2025. Axum. =https://github.com/tokio-rs/axum.
[36] Paul A. Samuelson. 1954. The Pure Theory of Public Expenditure. *The Review of Economics and Statistics* 36, 4 (1954), 387–389. http://www.jstor.org/stable/1925895
[37] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.
[38] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking* 11, 1 (2003), 17–32.
[39] I Storj Labs. 2018. Storj: A decentralized cloud storage network framework.
[40] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. 2017. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee, 444–460.
[41] The Sui team. 2025. Build Beyond. =https://sui.io.
[42] Juan Pablo Timpanaro, Thibault Cholez, Isabelle Chrisment, and Olivier Festor. 2011. Bittorrent's mainline dht security assessment. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*. IEEE, 1–5.
[43] David Vorick and Luke Champine. 2014. Sia: Simple decentralized storage. *Retrieved May* 8 (2014), 2018.
[44] Dan S Wallach. 2002. A survey of peer-to-peer security issues. In *International symposium on software security*. Springer, 42–57.
[45] Karl Werder, Balasubramaniam Ramesh, and Rongen Zhang. 2022. Establishing data provenance for responsible artificial intelligence systems. *ACM Transactions on Management Information Systems (TMIS)* 13, 2 (2022), 1–23.
[46] Sam Williams, Viktor Diordiiev, Lev Berman, and Ivan Uemlianin. 2019. Arweave: A protocol for economically sustainable information permanence. *Arweave Yellow Paper* (2019).
[47] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. 2014. Heading Off Correlated Failures through {Independence-as-a-Service}. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 317–334.

A Detailed Algorithms

This section supplements Section 4 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: ByteSize(B) to compute the size of a blob B in bytes; MerkleTree(v) to compute a merkle tree over a vector input v; Hash($\cdot$) to compute a cryptographic hash; ErasureEncode(B), ErasureReconstruct($\cdot$), and 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 3.3; HandledShards(n) to get the shards handled by a node n; and 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: ReserveBlob($\cdot$) to reserve a blob id on the blockchain; StoreCertificate($\cdot$) to store a proof of storage on the blockchain; IsRegistered(id) to check if a blob id id is registered on the blockchain; and ReadCertificate(id) to read a proof of storage of blob id id from the blockchain.

Table 2 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 2: Main notations

Algorithm 1 Walrus client operations

```
nodes                                                  ▷ the committee of storage nodes
shards                                                 ▷ see Section 4

// 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
```


Archived copy · cryp2nova.com

Algorithm 2 Helper functions

nodes ▷ the committee of storage nodes
shards ▷ see Section 4

procedure EncodeBlob(B)
  $E \leftarrow$ ErasureEncode(B) ▷ expand size: $[(f+1) \times (2f+1)] \rightarrow [\text{shards} \times \text{shards}]$
  $S^p \leftarrow [E_{(i,*)} : i \in [0, \text{shards}]]$ ▷ encoded primary slivers: $[\text{shards} \times 1]$
  $S^s \leftarrow [E_{(*,i)} : i \in [0, \text{shards}]]^\top$ ▷ encoded secondary slivers: $[1 \times \text{shards}]$
  return (S^p, S^s)

procedure MakeMetadata(S^p, S^s)
  $M^p \leftarrow [\text{Hash}(s) : s \in S^p]$ ▷ length: $2f+1$
  $M^s \leftarrow [\text{Hash}(s) : s \in S^s]$ ▷ length: $f+1$
  $M \leftarrow (M^p, M^s)$
  return M

procedure MakeBlobId(M)
  $(M^p, M^s) \leftarrow M$
  $id \leftarrow (\text{MerkleTree}(M^p), \text{MerkleTree}(M^s))$
  return id

procedure VerifySliver($S^{(*,n)}, M$)
  $(M^p, M^s) \leftarrow M$
  return $(\text{Hash}(s) = M^p_n : \forall s \in S^{(p,n)}) \vee (\text{Hash}(s) = M^s_n : \forall s \in S^{(s,n)})$

procedure DecodeBlob($\{S^{(p,*)}\}_{f+1}, M$)
  $S^p \leftarrow$ ErasureReconstruct($\{S^{(p,*)}\}_{f+1}$) ▷ reconstruct encoded slivers
  $E \leftarrow$ SplitIntoMatrix(S^p) ▷ size: shard × shard
  $S^s \leftarrow [E_{(*,i)} : i \in [0, \text{shards}]]^\top$
  $M' \leftarrow$ MakeMetadata(S^p, S^s)
  if $M \neq M'$ **then return** $\perp$ ▷ verify encoding correctness, see Section 4.2
  $B \leftarrow$ ErasureDecode(E) ▷ matrix: $(f+1) \times (2f+1)$
  return B

Algorithm 3 Walrus store operations

n ▷ the identifier of the storage node
nodes ▷ the committee of storage nodes
shards ▷ see Section 4
db_m ▷ perists the metadata
db_b ▷ perists the slivers

// Store slivers
procedure StoreSlivers(StoreRqst)
  $(id, M, S^{(p,n)}, S^{(s,n)}) \leftarrow$ StoreRqst

  // Check 1: Ensure the node is responsible for the shards
  $D^n \leftarrow$ HandledShards(n)
  if $\exists s_i \in S^p \cup S^s$ s.t. $i \notin D^n$ **then return** $\perp$

  // Check 2: Verify the blob id is registered on chain
  if $\neg$IsRegistered(id) **then return** $\perp$ ▷ read blockchain

  // Check 3: Verify the metadata is correctly formed
  if $\neg$VerifySliver($S^{(p,n)}, M$) **then return** $\perp$
  if $\neg$VerifySliver($S^{(s,n)}, M$) **then return** $\perp$
  $id' \leftarrow$ MakeBlobId(M)
  if $id \neq id'$ **then return** $\perp$

  $\text{db}_m[id] \leftarrow M$ ▷ persist the metadata
  $\text{db}_b[id] \leftarrow (S^{(p,n)}, S^{(s,n)})$ ▷ persist the slivers
  Send(ack) ▷ reply with an acknowledgment

// Server metadata
procedure ServeMetadata(MetadataRqst)
  $id \leftarrow$ MetadataRqst
  return $\text{db}_m[id]$ ▷ return the metadata or $\perp$ if not found
  Reply(ack)

// Server slivers
procedure ServeSlivers(SliversRqst)
  $id \leftarrow$ SliversRqst
  if $\neg$ReadCertificate(id) **then return** $\perp$ ▷ proof of storage on the blockchain
  $(S^{(p,n)}, S^{(s,n)}) \leftarrow \text{db}_b[id]$ ▷ return the slivers or $\perp$ if not found
  Reply($S^{(s,n)}$)

// Recover slivers
procedure RecoverSlivers(id)
  $c \leftarrow$ Client(nodes, shards) ▷ build a Walrus client (Algorithm 1)
  $B \leftarrow c$.ReadBlob(id)
  $D^{\text{n}} \leftarrow$ HandledShards(n) ▷ shards handed by node n
  $S^{(p,\text{n})} \leftarrow [S^p_i : i \in D^{\text{n}}]$
  $S^{(s,\text{n})} \leftarrow [S^s_i : i \in D^{\text{n}}]$
  $\text{db}_m[id] \leftarrow M$ ▷ persist the metadata
  $\text{db}_b[id] \leftarrow (S^{(p,\text{n})}, S^{(s,\text{n})})$ ▷ persist the slivers

Cryp2Nova

Archived copy · cryp2nova.com