Walrus: An Efficient Decentralized Storage Network

The MystenLabs Team
hello@mystenlabs.com

v1.0 RC1 – Sept 13, 2024

1 Introduction

Blockchains support decentralized computation through the State Machine Replication (SMR) paradigm [35]. 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 [28] offer robust resistance to censorship, enhanced reliability and availability during faults, via replication on only a small subset of nodes [45].

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 [43]. These applications benefit from authenticity, traceability, integrity and availability decentralized stores provide out of the box.

- 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 [21].

- 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

[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`

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 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 [28] and Arweave [44] 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 the potential for 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 [22] uses *Reed-Solomon (RS) encoding* [30]. 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, RS encoding has drawbacks. The encoding and decoding processes rely on field operations, polynomial evaluations and interpolations which are computationally expensive. These operations remain practical only if the field size and number of slivers are relatively small, thereby limiting the size of the encoded files and the number of participating storage nodes before the encoding becomes prohibitively costly, thus limiting decentralization. Another 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.

Regardless of the replication protocol, all existing decentralized storage systems face two additional challenges: (1) the need for a continuous stream of challenges to ensure that storage nodes retain the data and do not discard it. This is crucial in an open, decentralized system that offers payments for storage, but it currently limits the system's scalability, as each file requires individual challenges. And, (2) storage nodes require coordination: there is a need to know who is in the system; what blobs have been paid for to store; implement incentives for participation; and manage challenges or mechanisms to mitigate

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

abuse. This is the reason the above examplar systems implement each a custom blockchain to execute transactions, and instantiate a cryptocurrency, in addition to the storage protocol.

Introducing Walrus

We introduce Walrus, a third approach to decentralized blob storage. It combines fast linerarly decodable erasure codes that can scale to 100s of storage nodes to get extremely high resilience at a low storage overhead; and leverages a modern blockchain, Sui [8], for its control plane, from storage node life cycle management, to blob life cycle management, to economics and incentives, doing away with the need for a full custom blockchain protocol.

At the heart of Walrus, lies a new encoding protocol, called Red Stuff that uses a novel two-dimensional (2D) encoding algorithm based on fountain codes [24]. Unlike RS codes, fountain codes rely primarily on XOR or other very fast operations over large data blobs, avoiding complex mathematical operations. This simplicity allows for the encoding of large files in a single pass, resulting in significantly faster processing. The 2D encoding of Red Stuff enables the recovery of lost slivers using bandwidth proportional to the amount of lost data ($O(\frac{|\text{blob}|}{n})$ in our case). Additionally, Red Stuff incorporates authenticated data structures to defend against malicious clients, ensuring that the data stored and retrieved remains consistent.

Walrus operates in epochs, each managed by a committee of storage nodes. All operations within an epoch can be sharded by $blob_{id}$, enabling high scalability. The system facilitates blob writing by encoding data into primary and secondary slivers, generating Merkle commitments, and distributing these slivers across storage nodes. The read process involves collecting and verifying slivers, with both best-effort and incentivized pathways to address potential system failures. To ensure uninterrupted availability to both read and write blobs while handling the naturally occurring churn of a permissionless system, Walrus features an efficient committee reconfiguration protocol.

Another key innovation in Walrus is its approach to storage proofs, which are mechanisms to verify that storage nodes are indeed storing the data they claim to hold. Walrus addresses the scalability challenge associated with these proofs by incentivizing all storage nodes to hold slivers of all stored files. This complete replication enables a novel storage attestation mechanism that challenges the storage node as a whole, rather than each file individually. Consequently, the cost of proving file storage scales logarithmically with the number of stored files, as opposed to the current linear scaling in many existing systems.

Finally, we also introduce an economic model based on staking, with rewards and penalties to align incentives and enforce long-term commitments. The system includes a pricing mechanism for storage resources and write operations, complemented by a token governance model for parameter adjustments.

In summary, we make the following contributions:

- We define the problem of Asynchronous Complete Data-Sharing and propose the Red Stuff the first protocol to solve it efficiently even under Byzantine Faults (Section 3)
- We present Walrus, the first permisionless 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 extend Walrus with an economic model based on staking, with rewards and penalties to align incentives and enforce long-term commitments (Section 5) as well as propose the first asychronous challenge protocol that allows for efficient storage proofs (Section 6.1)

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. WALRUS is a delegated Proof-of-Stake protocol. In the duration of an epoch stakeholders delegate stake to candidate storage nodes. At the end of the epoch $n = 3f + 1$ storage *shards* are assigned proportionally to storage nodes. The set of storage nodes that holds at least one shard is considered the storage committee of the epoch.

We consider an asynchronous network of storage nodes where a malicious adversary can control up to f storage shards, i.e., control any subset of storage nodes such that at most f shards are corrupted. For simplicity in the rest of the paper we assume each shard generates a separate storage node identity such that there are n storage nodes and at most f storage nodes are corrupted.

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

We assume every pair of honest nodes have 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.

Although we provide an analysis on incentives, we do not consider rational nodes with utility functions. This is left for future work.

Erasure codes. As part of WALRUS, we propose Asynchronous Complete Data Storage (ACDS) that uses a linear 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 $\mathrm{Encode}(B, t, n)$ be the encoding algorithm. Its output is n symbols such that any t can be used to reconstruct B with overwhelming probability. This happens by first splitting B into $s \leq t$ symbols of size $O(\frac{|B|}{s})$ which are called *source* symbols. These are then expanded by generating $n - s$ repair symbols for a total of n output symbols. On the decoding side, anyone can call $\mathrm{Decode}(T, t, n)$ where T is a set of at least s correctly encoded symbols, and it returns the blob B. This decoding is probabilistic but as the size of T increases the probability of successfully decoding the blob quickly approaches one. We assume that the difference between the threshold t to decode with overwhelming probability and the number of source symbols s is a small constant, which holds for practical erasure coding schemes (such as RaptorQ [23] which we use in our implementation). For simplicity, we therefore often assume that $s = t$, unless the distinction is relevant.

This is generally safe to assume, since the first t symbols can be easily computed based on the first s symbols and we can just consider these t symbols to be the source symbols, i.e., the only effect of this assumption is a tiny increase in symbol size.

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

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:*

1. *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 blob B), such that B can be recovered from $\mathcal{O}\left(\frac{|B|}{|p|}\right)$ parts.*
2. *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 $\perp$.*
3. *Validity: If an honest writer W successfully writes blob 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 used with changing committees of storage nodes.

3.2 The Red Stuff encoding protocol

In this section, we present the final design of Red Stuff by iterating first through two straw man designs and eliminate their inefficiencies.

3.2.1 1st straw man design: full replication

The simplest protocol uses full replication in the spirit of Filecoin [28] and Arweave [44]. The writer W broadcasts its blob B along with a binding commitment to B (e.g., $H_B = hash(B)$), to all storage nodes and then waits to receive $f + 1$ receipt acknowledgments. These acknowledgments form an availability certificate which guarantees availability because at least one acknowledgement comes from an honest node. The writer W can publish this certificate on the blockchain, which ensures that it is visible to every other honest node, who can then request a Read(B) successfully. This achieves Write Completeness since eventually all honest nodes will hold blob B locally. The rest of the properties also hold trivially. Notice that the reader never reads $\perp$.

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

3.2.2 2nd straw man design: encode and share

To reduce the upfront data dissemination cost, some distributed storage protocols such as Storj [38] and Sia [41] use RS-coding [30]. The writer W divides its blob B into $f+1$ slivers and encodes $2f$ extra repair slivers. Thanks to the encoding properties, any $f+1$ slivers can be used to recover B. Each sliver has a size of $\mathcal{O}(\frac{|B|}{n})$. The writer W then commits to all the slivers using a binding commitment such as a Merkle tree [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.

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

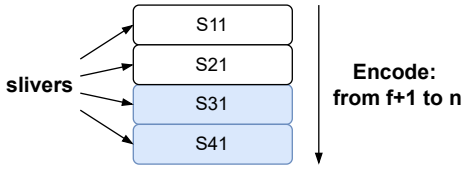


Figure 1: Encoding a Blob in one dimension. First the blob is split into $2f + 1$ systematic slivers and then a further f repair slivers are encoded

A reader continuously requests slivers from the nodes until it receives $f + 1$ valid replies (i.e., replies that are verified against the commitment). The reader is guaranteed to receive them since at least $f + 1$ honest nodes have stored their sliver. The reader then reconstructs blob B from the slivers and then additionally, re-encodes the recovered value and recomputes the commitment [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 $\mathcal{O}(|B|)$ even if the storage cost afterwards is $\mathcal{O}(\frac{|B|}{n})$.

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

3.2.3 Final design: Red Stuff

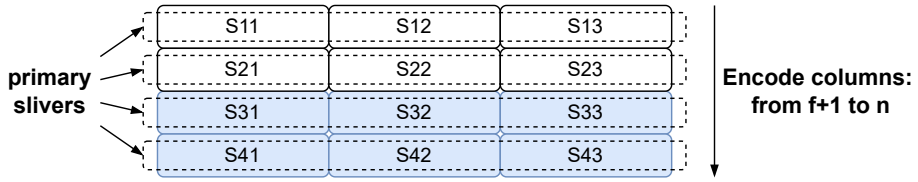
The encoding protocol above achieves the objective of a low overhead factor with very high assurance, but is still not suitable for a long-lasting deployment. The main challenge is that in a long-running large-scale system, storage nodes routinely experience faults, lose their slivers, and have to be 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 like the cost of recovery under churn to be proportional only to the data that needs to be recovered, and scale inversely with n.

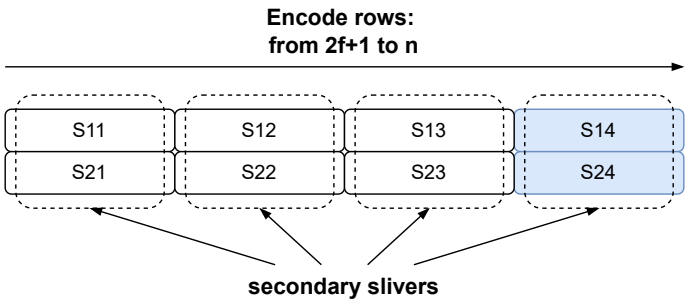
To achieve this, Red Stuff encodes blobs in two dimensions (2D-encoding). The primary dimension is equivalent to the RS-encoding used in prior systems. However, in order to allow efficient recovery of slivers of B we also encode on a secondary dimension. Red Stuff is based on linear erasure coding (see section 2) and the Twin-code framework [29], 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 in the splitting process: the

[5]There might be and extra $\mathcal{O}(\log n)$ costs depending on the commitment scheme used.



(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

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.

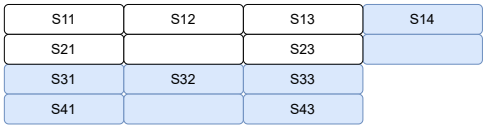
Given this matrix we then generate repair symbols in both dimensions. We take each of the $2f+1$ columns (of size $f+1$) and extend them to n symbols such that there are n rows. We assign each of the rows as the *primary sliver* of a node (Figure 2a). This almost triples the total amount of data we need to send and is very close to what 1D encoding did in the protocol in Section 3.2.2. 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 from $2f+1$ symbols to n symbols (Figure 2b) using our encoding scheme. This creates n columns, which we assign as the *secondary sliver* of each node, respectively.

For each sliver (primary and secondary), W also computes 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 values 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*.

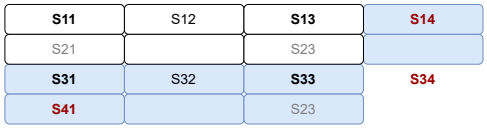
Write protocol The Write protocol of Red Stuff uses the same pattern as the RS-code protocol. The writer W first encodes the blobs and creates a sliver pair for each node. A sliver pair i is the pair of i^{th} primary and secondary slivers. There are $n = 3f+1$ sliver pairs, as many as nodes.

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.

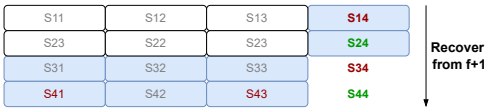
In theoretical asynchronous network models with reliable delivery the above would result in all correct nodes eventually receiving a sliver pair from an honest writer. However, in practical protocols the writer may need to stop re-transmitting. It is safe to stop the re-transmission after $2f+1$ signatures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



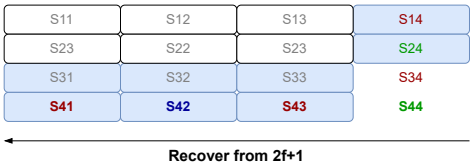
(a) Nodes 1 and 3 collectively hold two rows and two columns



(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

Read Protocol The Read protocol is the same as for RS-codes and nodes only need to use their primary sliver. R first requests the sets of commitments for the blob commitment sought from any node, and checks the returned set corresponds to the blob commitment through the commitment open protocol. Then R requests a read for the blob commitment from all nodes and they respond with the primary sliver they hold (this may happen gradually to save on bandwidth). Each response is checked against the corresponding commitments in the commitment set for the blob. When $f+1$ correct primary slivers are collected R decodes B and then re-encodes, re-computes the blob commitment, it to check it against the blob commitment sought. 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 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 node for the symbols that exist in their row, which should also exist in the (expanded) column of the requesting node (Figure 3b and fig. 3c). This means that eventually all $2f+1$ honest node will have secondary slivers. At that point any node can also recover their primary sliver by asking $2f+1$ honest node for the symbols in their column (Figure 3d) that should also exist in the (expanded) row of the requesting storage node. In each case, the responding node also sends the opening for the requested symbol of the commitment of the source sliver. This allows the receiving node to verify that it received the symbol intended by the writer W, which ensures correct decoding if W was honest.

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

Optimization We further optimize Red Stuff with a few simple ideas. These are inspired by the fact that the first part of the encoding actually holds pieces of the real data, called the *source* symbols. Anyone reading B would then prefer to read the source symbols instead of the recovery symbols as this would save the decoding cost.

In the current design we noticed that some of the source are only held by a single node, for example the top right corner symbol is only held by node 1. This means that if that one node is unavailable there is no direct access to the source data. Additionally the source symbols are always going to the first $2f+1$

nodes. As a result anyone reading would always try to get the slivers from them in order to avoid paying the computation of decoding.

In order to load balance and increase the fault tolerance of the system our implementation has two further optimizations:

1. We permute the indices of where the source symbols go based on the commitment to B which is a Random Oracle. As a result we can get better load balancing over multiple stored files.

2. When performing secondary encoding we reverse the indices (i.e., we assign the first symbol to Node n). This allows for every source symbol to exist in two nodes increasing the fault tolerance of reading without needing to decode.

Finally, we notice that the reads are served first by source symbols and if not available by the primary slivers. As a result storage nodes can keep their secondary encoded slivers in slower, cheaper storage as they will be accessed only in the case of faults in the network.

Red Stuff is an ACDS Appendix B provides proofs that Red Stuff satisfies all properties of a ACDS. Informally, Write Completeness is ensured by the fact that a correct writer will confirm that at least $f+1$ correct nodes received sliver pairs before stopping re-transmissions. And the sliver recovery algorithm can ensure that the remaining honest nodes can efficiently recover their slivers from these, until all honest nodes eventually hold their respective sliver, or can prove that the encoding was incorrect. Validity holds due to the fact that $f+1$ correct nodes always hold correct sliver pairs, and therefore a reader that contacts all nodes will eventually get enough primary 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. In our architecture, we use the Red Stuff encoding/decoding algorithm described in section 3.2.3 instantiated with RaptorQ [23] codes, Merkle trees [27] as set commitments, and the Sui [8] blockchain. Walrus can, however, be generalized to any blockchains and encoding/decoding algorithm that satisfies the minimal requirements described in Section 2.

Refreshing the definitions in Section 2, the actual minimal encoded quantity is not the capacity of one storage node but a storage shard. This allows for heterogeneity in storage capacity between storage nodes who can store one or more storage shards. For simplicity the reader can assume that every storage node is a virtual storage node that has a single shard and a physical storage node can control multiple virtual storage nodes with different virtual identities.

We first describe Walrus flows in a single epoch and then we discuss how we allow for storage node dynamic availability through reconfiguration. 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 $blob_{id}$ by hashing the blob commitment with meta-data such as the length of the file, and the type of the encoding.

Then, the writer (❷) submits a transaction on the blockchain to acquire sufficient space for the blob to be stored during a sequence of epochs, and to *register* the blob. The size of the blob and blob commitment are sent which can be used to re-derive the $blob_{id}$. The blockchain smart contract needs to secure sufficient

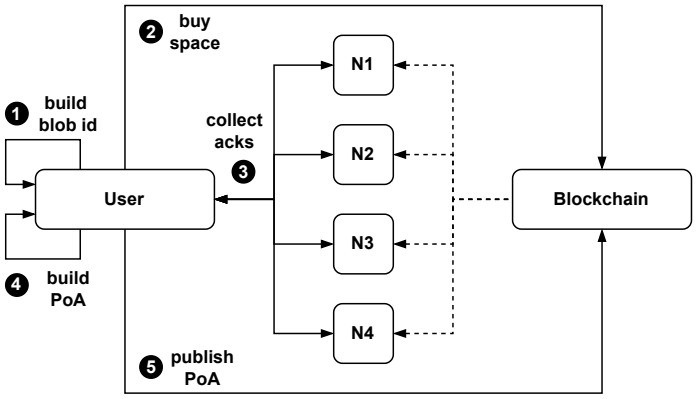


Figure 4: WALRUS write flow. The user generates the blob id of the file they wish to store; acquire storage space through the blockchain; submit the encoded file to WALRUS; collect $2f + 1$ acknowledgements; and submit them as proof of availability to the blockchain.

space to store both the encoded slivers on each node, as well as store all meta-data associated with the commitments for the blob on all nodes. 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 $blob_{id}$, sending them the transaction together with the commitments, and the primary and secondary sliver assigned to the respective storage nodes along with proofs that the slivers are consistent to the published $blob_{id}$. The storage node verifies the commitments and replies with a signed acknowledgment over the $blob_{id}$ once the commitments and sliver pairs are stored.

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

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

4.2 Reading a Blob

In the best-effort read path a reader may ask any of the storage nodes for the commitments and primary sliver (1) for a blob by $blob_{id}$. Once they collect $f + 1$ replies with valid proofs against the $blob_{id}$ (2) they reconstruct the blob. Then (3) the reader re-encodes the blob and re-computes a $blob'_{id}$. If the $blob_{id} = blob'_{id}$ 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.

In Section 6.2, we describe how the user can read a blob under high contention when there is a need for incentivized reads. We also discuss how we foresee many reads being delivered though aggregators and caches without the need to re-construct, at least popular blobs, for every read.

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 6.1).

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 the $blob_{id}$.

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 6.1). To prove inconsistency, the storage node shares the inconsistency proof—consisting of the symbols that it received for recovery and their Merkle proofs—with the other nodes who can verify it by performing a trial recovery themselves. After verifying this fraud proof, the node attest on-chain that the $blob_{id}$ 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

As mentioned in Section 2, Walrus is a Proof-of-Stake protocol, hence it is natural that the set of storage nodes will fluctuate between epoch because some nodes will be unable to be profitable and will stop operating, whereas some nodes will perform better than other attracting more delegated stake. Remember that storage nodes hold multiple storage shards in Walrus, hence fluctuation in stake result in fluctuation the number of shards assigned to each node.

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. Since reconfiguration may take hours, we want to ensure no downtime, and continue to perform reads or writes for blobs during that period.

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 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 shards, 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 properties of an ACDS across multiple 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 eventually 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 Extensions

Here we briefly touch on a few extensions of WALRUS that enable a more complete user experience.

Blob Storage Lifespan Blobs in WALRUS are stored for a number of epochs. A writer can purchase storage for up to 2 years in the future and this is the maximum lifetime of a blob at any time. However, a blob can have its lifetime *extended* by a transaction on Sui using a storage resource of sufficient size and with a later expiration epoch. Since both blobs and storage resources are objects on the Sui blockchain, arbitrary smart contracts can be designed to manage the acquisition of storage resources, and the extension of resource lifetimes. For example to simulate an 'infinite' lifetime, a blob can be encapsulated within a Sui shared object along with some coins; the coins may only be used to purchase periodically storage resources to extend the lifetime of the blob.

Blobs stored on WALRUS can be set as *deletable* by their writer. Such blobs can be deleted via a Sui transaction, and the storage resource reclaimed and reused for its remaining lifetime. Identifying blobs as deletable allows efficient use of storage resources when applications need to update files frequently and reclaim the space of previous versions for reuse or resale.

Blobs that are not deletable on the other hand are guaranteed to remain available for the full period of the underlying storage resource in epochs. Writers may use this property to *prove the blobs availability* to third parties: anyone can convince a third party of the blob's availability, by proving a certified blob event was emitted by Sui with the $blob_{id}$, no deletable flag, and some remaining epochs. The Sui light client protocol allows verifying such events were emitted without requiring a full node to follow the blockchain.

Partial reads As we discussed in Section 2 the first part of the encoded blob is actually the source symbols of the file. Additionally thanks to the 2D encoding these source symbols are the basis of encoding in both dimensions, hence they appear twice in the encoding. As mentioned in Section 3.2.3 we also make sure that these source sumbols exist in two distinct storage nodes.

Having this in mind, another optimization we provide in WALRUS is allowing for faster then standard reads through directly accessing the source symbols. This means that a user can start the read by requesting the slivers that include these source symbols and if all of them are accessible then the user can skip the computationally costly decoding process. Additionally, there can be cases where a specific resource (e.g., an image) is encoded as part of a bigger $blob$, in this case the user can try to access the source location of this resource directly, avoiding the need to download the full blob as well.

Denylist Sometimes storage nodes may have legal or other reasons to not wish to store or serve slivers associated with specific blobs. The design philosophy of WALRUS is that there is no central authority that may censor blobs, due to the wide distribution of slivers over storage nodes. However, the protocol allows individual storage nodes to express preferences when they have a pressing reason to not serve some content, and allows them to both delete all traces of it and not serve any data related to it from their own nodes.

Specifically, each storage node maintain a deny list for itself on the Sui blockchain. It can add a $blob_{id}$ to this deny list at any time, and following this delete all slivers of this blob, refuse new uploads of slivers, and never serve any of its slivers. It also does not need to answer any challenges related to slivers on its deny list.

A blob that is included in the deny list of no more than f nodes is still available, allowing for regional or other local policy preferences of storage nodes to be expressed without censoring blobs globally. If a new storage node takes charge of shards, following reconfiguration, it can reconstruct all slivers of blobs that are available even if the previous storage node had them added in their denylist. However, if a blob is included in more than f nodes' denylists, it is no more guaranteed to be available within WALRUS. At this point, all other storage nodes also add it to their denylists and the blob can no more be read.

Performance and Scaling Walrus available capacity increases with the addition of nodes with new storage and network resources, as well as the increase of resources at each storage node. This is unlike smart contract protocols that see their capacity stay the same, or even decrease, with more nodes due to full replication and overheads.

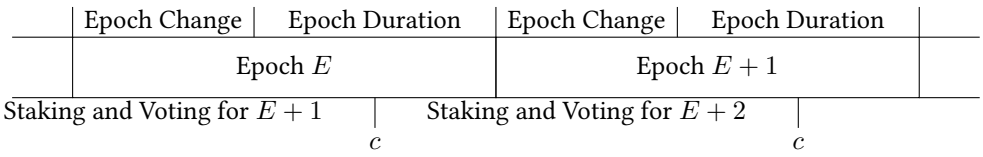
Writes and Reads in Walrus are inherently parallelized across storage nodes. Writing involves encoding a blob and sending slivers to all storage nodes. Uploading slivers happens in parallel across all nodes, making available the aggregate upload bandwidth of the network for writes. Similarly for reads, downloading a blob involves reading $f+1$ slivers from any available storage nodes. This makes available the aggregate download bandwidth of Walrus, and readers may additionally chose storage nodes close to their network location to optimize latency. Unlike writes that have a 4x-5x bandwidth overhead, reads only require downloading data comparable to the size of the blob. This makes large geo-distributed networks formidable in theory: a network of 500 storage nodes with 1 Gbps links, results in an aggregate read capacity of about 500 Gbps, and upload capacity of about 100 Gbps. We expect reads and writers will provision caches and CDNs to support reads for frequent content, that further increases capacity and lowers latency.

A storage node may serve within an epoch one or more shards. As the size of each shard grows, a storage node may scale by using separate machines to serve read and write requests for each shard. However, there may come a point after which even a single shard handles more data than a single machine may store. All WALRUS protocols have been designed such that requests can be sharded by $blob_{id}$ even within each shard. This way different machines may split the space of $blob_{id}$ and handle reads and writes for different blobs separately. This architecture can scale infinitely, and therefore WALRUS has neither an upper capacity limit, nor a throughput limit for reads and writes. Given demand (and payment) for more capacity it can always be expanded by adding more machines with more storage at each storage node.

5 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. Concep-

Figure 5: Timeline of WALRUS Operations



tually, 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 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 5.1 describes the staking mechanism that underpins the security and efficiency of the entire system. Section 5.2 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 5.3 explains how nodes collectively set prices in competitive and fair ways. Finally, Section 5.4 briefly explains the token governance processes.

5.1 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 5.2.

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 5.2, 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.[6]

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 6.1) 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 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] 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.

5.2 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.[7] 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

[7] 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.

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.[8]

Non-Migration Recovery The recovery pathway in Walrus is primarily for shard migrations, but it can also be used for recovery 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 6.1 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.

5.3 Pricing and 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

[8] 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.

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 6.1.

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.[9] 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 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

[9] 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 6.1.

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.

5.4 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 Support for Economic Security

In this Section we briefly present a set of protocols that can be used in tandem with the economic incentives to provide compliance for storage nodes as well as a second layer of security provided by light nodes (Section 6.3).

6.1 Storage Challenges

Walrus requires a challenge protocol to provide correct incentives, and prevent storage nodes from trivially never storing or serving data while getting rewards. 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. We present first the version of the protocol with the fewest assumptions. Nevertheless, we do not plan to implement it as it requires suspend some of the protocol operations while the challenges happen and forces slower reads. Instead in Section 6.1 we present a relaxed version with extra assumption that we plan to implement initially, with the option to upgrade to the more secure version if there is a need.

Fully Asynchronous Challenge Protocol The hardened challenge protocol working in full asynchrony, requires the following protocol tweaks:

- Reads rely on the secondary slivers with a $2f+1$ read threshold. This does not compromise the liveness, since if $f + 1$ honest parties have acknowledged receipt of their slivers, eventually all honest parties recover their primary ($f + 1$ reconstruction threshold) slivers. Then eventually all $2f + 1$ parties also recover the remaining secondary slivers and may serve reads.
- At the beginning of every epoch, the storage nodes setup a random coin with a $2f + 1$ reconstruction threshold. This is possible using any kind of asynchronous DKG [13, 14, 20] or randomness [17, 39].

With these changes in place the challenge protocol proceeds as follows. 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 they broadcast their share of the coin. Since there are $2f + 1$ honest nodes when the coin is generated it also mean that at least $f + 1$ honest nodes will not reply on reads or recover requests. The coin is used to seed a pseudo-random function (PRF) that defines which blobs need to be challenged per storage node. 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 half of the blobs, it has less than a 10^{-30} probability of success on a 100 file challenge. The challenged node sends the common symbols per blob to each of the nodes along with a proof against the commitment of the writer of the blob. These verifiers 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. Finally, when $2f + 1$ certificates are valid, the challenge period ends and the reads and recovery is re-enabled.

During the challenge period, the nodes that witnessed the challenge start message do not reply to read or recovery requests[10]. Since the threshold of the randomness beacon 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.

Relaxations For the challenges protocol we plan to deploy in mainnet we make the following relaxation. First we assume there is a time Δ which is sufficient for all honest nodes to prove their storage

[10]Note that after the shared coin is revealed only the reads to the challenged blobs need to be blocked, making this protocol more practical than blocking all reads.

and submit their certificates on chain (i.e., that the network has some synchrony bound). Secondly, we assume that f is equal to the reconstruction threshold of RaptorQ encoding which is slightly lower than $n/3$.

These assumptions allows us to serve reads as described in Section 4 from the primary sliver with an $f + 1$ reconstruction threshold (with RaptorQ it will be f or $f - 1$ depending on the parameters) as well as simply rate limit the reads during the challenge period such that no honest parties gives out k blobs within Δ. Additionally, we only need to use the randomness just for the first challenged file and then determine the subsequent challenged files per node based on the accessed symbol. Given that this is a subjective challenger-challengee protocol this randomness needs not be global or unbiased, and the challenger can use a local randomness source.

By the end of each epoch, every node reports two pieces of information: all nodes that issued it a challenge, and all nodes that it challenged.[11] For the nodes that it challenged, it also reports a binary indicator on whether it judges the node to have succeeded or failed the challenge.

Nodes that receive 50% or more by stake weight on both sets of reports face no consequences. They are assumed to have performed their duties, both in terms of challenging other nodes and in terms of responding to challenges. The threshold of 50% is set to be tolerant to adversarial actors. Specifically, under the assumption that f nodes are adversarial, nodes who cannot issue or answer challenges correctly should receive at most f positive attestations; and nodes who can issue and answer challenges correctly may not receive more than $2f + 1$ positive attestations.

Nodes that fail to receive 50% on either report are slashed by penalties set by governance. Unlike shard recovery, penalties are not returned to the nodes – as that would otherwise incentivize misreporting, which is free to the nodes beyond reputational costs. Instead, they are burned to enhance economic security. Any other solution, such as redistributing those penalties, placing them into a community treasury program, distributing them in a future epoch, and so on, gives nodes at least a weak incentive to report inaccurately. The best way to dull their incentives is to burn the slashed penalties, as an irreversible action that dilutes and diffuses benefits to misreporting. This aspect of Walrus is strongly inspired by Ethereum's Improvement Proposal 1559, which burns base fees as a security measure to prevent off-chain collusion [32].

6.2 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 [34] 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 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.

[11] For simplicity in accounting, it also issues a "self-challenge" that it trivially passes.

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.

6.3 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 $\mathcal{O}(\frac{|B|}{n^2})$ 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 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 [31], 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 [37], Pastry [33], 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 [42]. Many attacks affect current systems that use them [40]. 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. Instead of relying on replication Walrus uses an erasure code extending a fountain code. While such rate-less codes were invented in the early 2000s alongside peer-to-peer systems, perhaps surprisingly they were never used by a notable system of this era.

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 [28] 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 [44] 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 4x-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. Encoding and decoding files is extreme fast due to the use of RaptorQ fountain codes [36], to allow efficient reads and writes. Furthemore, Walrus does not implement its own separate blockchain to do node management and provide incentives, but uses Sui instead.

Storj [38] represents another decentralized storage solution that leverages encoding to achieve a low replication factor. The system implements a Reed-Solomon based erasure coding scheme with a $29/80$ configuration, wherein a file is encoded into 80 parts, with any 29 sufficient for reconstruction. This approach results in a $2.75x$ replication factor, offering a substantial reduction in storage costs compared to prior systems. However, the use of RS codes imposes limitations on the encoding of very large files. Storj's

64MB segment limit necessitates separate encoding for larger files, increasing computational overhead and potentially reducing fault tolerance. For instance, a 64GB file (comprising 1000 segments) would require 1000 distinct decoding operations, with no guarantee that each segment is reconstructable by the same 29 nodes. A final 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. Additionally, the use of RaptorQ allows for a single encoding of arbitrarily large files mitigating the fragmentation security risks. Red Stuff builds on the Twin-code framework [29], 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 offers proofs of availability with at varying levels of assurance: either a quorum of storage nodes certifies a blob is available, or optimistically after some time one may assume that a plurality of light-clients each have downloaded sufficient data to ensure the file is available on them (see section 6.2). Furthermore, since coding is used to store files – not only distributing code words to light nodes – the overall cost of storage is much lower, and retention periods arbitrary long.

8 Conclusion

We introduce Walrus, a novel approach to decentralized blob storage that leverages fast, linearly decodable 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 a robust economic model based on staking, with rewards and penalties to align incentives and enforce long-term commitments. The system's approach to storage proofs ensures 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, resilient, and economically viable solution for 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 economic model and asynchronous challenge protocol for efficient storage proofs, paving the way for future advancements in decentralized storage technologies.

References

[1] Mustafa Al-Bassam. 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. 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*, pages 279–298. Springer, 2021.

[3] Ross Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, 1996.

[4] Pablo Aragón, Andreas Kaltenbrunner, Antonio Calleja-López, Andrés Pereira, Arnau Monterde, Xabier E Barandiaran, and Vicenç Gómez. 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*, pages 277–287. Springer, 2017.

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

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

[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. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

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

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

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

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, pages 46–66. Springer, 2001.

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

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

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

[15] Roger Dingledine, Michael J Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, pages 67–95. Springer, 2001.

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

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

[18] Martin Kleppmann, Paul Frazee, Jake Gold, Jay Graber, Daniel Holmgren, Devin Ivy, Jeromy Johnson, Bryan Newbold, and Jaz Volpert. 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. Calypso: Private data management for decentralized ledgers. *Proceedings of the VLDB Endowment*, 14(4):586–599, 2021.

[20] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

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

[22] Chuanlei Li, Minghui Xu, Jiahao Zhang, Hechuan Guo, and Xiuzhen Cheng. Sok: Decentralized storage network. *Cryptology ePrint Archive*, 2024.

[23] M Luby, A Shokrollahi, M Watson, T Stockhammer, and L Minder. Rfc 6330: Raptorq forward error correction scheme for object delivery, 2011.

[24] David JC MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

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

[26] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer systems*, pages 53–65. Springer, 2002.

[27] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[28] Yiannis Psaras and David Dias. The interplanetary file system and the filecoin network. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 80–80. IEEE, 2020.

[29] KV Rashmi, Nihar B Shah, and P Vijay Kumar. Enabling node repair in any erasure code for distributed storage. In *2011 IEEE international symposium on information theory proceedings*, pages 1235–1239. IEEE, 2011.

[30] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[31] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE, 2001.

[32] Tim Roughgarden. Transaction fee mechanism design for the ethereum blockchain: An economic analysis of EIP-1559, 2020.

[33] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*, pages 329–350. Springer, 2001.

[34] Paul A. Samuelson. The pure theory of public expenditure. *The Review of Economics and Statistics*, 36(4):387–389, 1954.

[35] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[36] Amin Shokrollahi. Raptor codes. *IEEE transactions on information theory*, 52(6):2551–2567, 2006.

[37] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[38] I Storj Labs. Storj: A decentralized cloud storage network framework, 2018.

[39] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[40] Juan Pablo Timpanaro, Thibault Cholez, Isabelle Chrisment, and Olivier Festor. Bittorrent's mainline dht security assessment. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–5. IEEE, 2011.

[41] David Vorick and Luke Champine. Sia: Simple decentralized storage. *Retrieved May*, 8:2018, 2014.

[42] Dan S Wallach. A survey of peer-to-peer security issues. In *International symposium on software security*, pages 42–57. Springer, 2002.

[43] Karl Werder, Balasubramaniam Ramesh, and Rongen Zhang. Establishing data provenance for responsible artificial intelligence systems. *ACM Transactions on Management Information Systems (TMIS)*, 13(2):1–23, 2022.

[44] Sam Williams, Viktor Diordiiev, Lev Berman, and Ivan Uemlianin. Arweave: A protocol for economically sustainable information permanence. *Arweave Yellow Paper*, 2019.

[45] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. Heading off correlated failures through {Independence-as-a-Service}. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 317–334, 2014.

A Detailed Algorithms

This section supplements Section 4 by providing detailed algorithms for clients (Algorithm 2) and storage nodes operations (Algorithm 3).

In additional to the helper functions specified in Algorithm 1, 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 3.2.3; 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.

B Red Stuff Proofs

This section completes Section 3 by showing that Red Stuff satisfies all the properties of a ACDS (Definition 1).

Algorithm 1 Helper functions

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

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

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

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

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

procedure DecodeBlob({S^(p,*)}_(f+1), M)
    S^p ← ErasureReconstruct({S^(p,*)}_(f+1))          ▷ reconstruct encoded slivers
    E ← SplitIntoMatrix(S^p)                           ▷ size: shard × shard
    S^s ← [E_(*,i) : i ∈ [0, shards]]^⊤
    M' ← MakeMetadata(S^p, S^s)
    if M ≠ M' then return ⊥                            ▷ verify encoding correctness, see Section 4.2
    B ← ErasureDecode(E)                               ▷ matrix: (f + 1) × (2f + 1)
    return B
```

B.1 Write Completeness

We show that Red Stuff satisfies Write Completeness. Informally, if a honest writer writes a blob B to the network, every honest storage nodes eventually holds a primary and secondary correctly encoded sliver of B.

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

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

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

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

Lemma 3 (Primary Sliver Reconstruction from Secondary Slivers)**.** *If a party holds* $(f+1)$ *secondary slivers* $\{S^{(s,*)}\}_{f+1}$*, it can obtain any secondary sliver* $S^{(p,n)}$*.*

Proof. The proofs directly follows from the construction of slivers and the reconstruction property of fountain codes with reconstruction threshold $(f+1)$. The party reconstructs the primary sliver $S^{(p,n)}$ by collecting the n^{th} symbol of each of the $(f+1)$ secondary slivers $\{S_n^{(s,*)}\}_{f+1} \equiv \{S_*^{(p,n)}\}_{f+1}$. Lemma 2 then ensures that the party can obtain the complete primary sliver $S^{(p,n)}$. □

Lemma 4 (Secondary Sliver Reconstruction from Primary Slivers)**.** *If a party holds* $(f+1)$ *primary slivers* $\{S^{(p,*)}\}_{f+1}$*, it can obtain any secondary sliver* $S^{(s,n)}$*.*

Proof. The proof is analogue to Lemma 3. □

Algorithm 2 WALRUS client operations

```
1: nodes                                                          ▷ the committee of storage nodes
2: shards                                                         ▷ see Section 4

   // Store a blob on the network
3: procedure StoreBlob(B, expiry)
4:     // Step 1: Pay and register the blob id on the blockchain
5:     (S^p, S^s) ← EncodeBlob(B)
6:     M ← MakeMetadata(S^p, S^s)
7:     id ← MakeBlobId(M)
8:     size ← ByteSize(B)                                         ▷ size in bytes
9:     ReserveBlob(id, size, expiry)                              ▷ on blockchain
10:
11:    // Step 2: Send the encoded slivers to the storage nodes
12:    R ← { }                                                    ▷ storage requests to send to nodes
13:    for n ∈ nodes do
14:        D^n ← HandledShards(n)                                 ▷ shards handled by node n
15:        S^(p,n) ← [S^p_i : i ∈ D^n]
16:        S^(s,n) ← [S^s_i : i ∈ D^n]
17:        StoreRqst ← (id, M, S^(p,n), S^(s,n))
18:        R ← R ∪ {(n, StoreRqst)}
19:    await_{2f+1} : {c ← Send(n, r) : (n, r) ∈ R}               ▷ wait for 2f + 1 confirmations
20:
21:    // Step 3: Record the proof of storage on the blockchain
22:    StoreCertificate({c}, id)                                  ▷ on blockchain

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

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

Lemma 5 (Message Retrieval)**.** *Let's call M a binding vector commitment to a set of $(2f+1)$ messages $\{m_i\}_{2f+1}$. If a set of $(2f+1)$ storage nodes $\{n_i\}_{2f+1} \subseteq \mathtt{nodes}$ each hold a message m_i, a reader R holding M can eventually output at least $(f+1)$ of these messages. That is, it can output a set $\{n_i\}_{f+1} \subseteq \{n_i\}_{2f+1} \subseteq \mathtt{nodes}$.*

Proof. Let's call $\mathtt{nodes}$ the total set of storage nodes and $\mathtt{nodes}_r \subseteq \mathtt{nodes}$ the set of $(2f+1)$ storage nodes each holding a message m_i. The reader R holding commitment M queries each storage node $n \in \mathtt{nodes}$. Let's assume for the sake of contradiction that the reader R obtains only $f < f+1$ messages matching the commitment M. This implies that the set $\mathtt{nodes}_r$ contains at least $(2f+1) - f = f+1$ dishonest nodes. This implies that the set $\mathtt{nodes}$ contains $(2f+1) + (f+1) = 3f+2 > |\mathtt{nodes}|$ nodes, which is a contradiction. □

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

Proof. To write a blob B, an honest writer W sends at least $(2f+1)$ correctly encoded slivers (parts) to different storage nodes, along with a binding vector commitment M over those slivers. Let $\mathtt{nodes}$ denote the entire set of storage nodes. A node $n \in \mathtt{nodes}$ first queries each storage node $n' \in \mathtt{nodes}$ for their symbols $\{S^{p,n'}_*\}_{f+1}$, uses them to reconstruct the primary sliver $S^{p,n'}$ of node n' (Lemma 1), and drops any primary sliver not matching the commitment M. Lemma 5 ensures that node n obtains at least $(f+1)$ primary slivers $\{S^{(p,*)}\}_{f+1}$ matching the commitment M. Then Lemma 4 ensures that node n can use these $\{S^{(p,*)}\}_{f+1}$ primary slivers to reconstruct its secondary sliver $S^{(s,n)}$. Node n then repeats this process f additional times to obtain a total of $(f+1)$ secondary slivers $\{S^{(s,*)}\}_{f+1}$. Finally, Lemma 3

Algorithm 3 WALRUS store operations

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

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

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

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

   // Recover slivers
34: procedure RecoverSlivers(id)
35:    c ← Client(nodes, shards)                              ▷ build a WALRUS client (Algorithm 2)
36:    B ← c.ReadBlob(id)
37:    D^n ← HandledShards(n)                                 ▷ shards handed by node n
38:    S^(p,n) ← [S_i^p : i ∈ D^n]
39:    S^(s,n) ← [S_i^s : i ∈ D^n]
40:    db_m[id] ← M                                           ▷ persist the metadata
41:    db_b[id] ← (S^(p,n), S^(s,n))                          ▷ persist the slivers
```

ensures that node n uses these $\{S^{(s,*)}\}_{f+1}$ secondary slivers to reconstruct its primary sliver $S^{s,n}$. Since this reasoning applies to any generic node i, it holds for all nodes. This concludes the proof. □

B.2 Read Consistency

We prove that RED STUFF satisfies Read Consistency. Informally, if two honest reader 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. Let's assume by contradiction 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 $\perp$. Let's call nodes the entire set of storage nodes. To read a blob B associated with a commitment M, both R_1 and R_2 query each storage node $n \in$ nodes for their primary sliver $S^{(p,n)}$. Theorem 1 ensures that at least $(2f+1)$ storage nodes hold a primary sliver $S^{(p,n)}$ matching the commitment M and thus Lemma 5 ensures that both readers obtains at least $(f+1)$ primary slivers $\{S^{(p,*)}\}_{f+1}$ matching the commitment M. Given that R_1

successfully reconstructs and outputs B from the slivers $\{S^{(p,*)}\}_{f+1}$, it means that B was correctly encoded; Given that R_2 does not successfully reconstructs B and outputs $\perp$ means that B was not correctly encoded. This is a contradiction. □

B.3 Validity

We prove that RED STUFF satisfies Validity. Informally, if a 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 sends at least $(2f + 1)$ correctly encoded slivers (parts) to a different storage nodes along with a binding vector commitment M over those slivers. Let's note by $\texttt{nodes}$ the entire set of storage nodes. An honest reader queries each storage node $n \in \texttt{nodes}$ for their symbols $\{S_*^{(p,n)}\}_{f+1}$, uses them to reconstruct the primary sliver $S^{p,n'}$ of node n' (Lemma 1), and drops any primary sliver not matching the commitment M. Lemma 5 ensures that node n obtains at least $(f + 1)$ primary slivers $\{S^{(p,*)}\}_{f+1}$ matching the commitment M. Given that B was correctly encoded, the reader can then reconstruct blob B from the these slivers. □