Sunfish: Reading Ledgers with Sparse Nodes

Giulia Scaffino[1], Karl Wüst[2], Deepak Maram[2], Alberto Sonnino[2,3], and Lefteris Kokoris-Kogias[2]

[1] TU Wien & Common Prefix & CDL-BOT
[2] Mysten Labs
[3] University College of London (UCL)

Abstract. The increased throughput offered by modern blockchains, such as Sui, Aptos, and Solana, enables processing thousands of transactions per second, but it also introduces higher costs for decentralized application (dApp) developers who need to track and verify changes in the state of their application. This is true because dApp developers run full nodes, which download and re-execute every transaction to track the global state of the chain. However, this becomes prohibitively expensive for high-throughput chains due to high bandwidth, computational, and storage requirements. A common alternative is to use light nodes. However, light nodes only verify the inclusion of a set of transactions and have no guarantees that the set is complete, i.e., that includes *all* relevant transactions. Under a dishonest majority, light nodes can also be tricked into accepting invalid transactions.

To bridge the gap between full and light nodes, we propose and formalize a new type of blockchain node: the *sparse node*. A sparse node tracks only a subset of the blockchain's state: it verifies that the received set of transactions touching the substate is complete, and re-executes those transactions to assess their validity. A sparse node retains important security properties even under adversarial majorities, and requires an amount of resources proportional to the number of transactions in the substate and to the size of the substate itself.

We further present Sunfish, an instantiation of a sparse node protocol. Our analysis and evaluation show that Sunfish reduces the bandwidth consumption of real blockchain applications by several orders of magnitude when compared to a full node.

1 Introduction

In recent years, the landscape of blockchain technology has rapidly evolved thanks to the important advancements in the area of consensus and layer-2 solutions, but also to the variety of decentralized applications (dApps) that run on-chain and, often, cross-chain. Modern blockchains, such as Sui, Aptos, and Solana, scale up to thousands of transactions per second, while earlier-generation chains, such as Bitcoin and Ethereum, only handle a throughput in the single or double digits. This increased capacity enhances the user experience and allows for onboarding hundreds of thousands of new users and dApps. Still, it also introduces a new, important challenge: dApp developers who want to verifiably and

trustlessly track the state of their application face higher costs. Traditionally, dApp developers run full nodes to listen to events, follow changes in the state of their application, and keep audit proofs. Full nodes receive and re-execute all blockchain transactions, and their bandwidth, computational, and storage costs become prohibitively expensive for high-throughput blockchains. As a result, developers resort to querying third-party full node operators and accepting their responses blindly. This behavior is questionable, as it fully relies on the honesty of the full node operator and negates the trust benefits a decentralized blockchain provides. As blockchain throughput increases with further advances and usage, this problem is only getting worse: fewer full nodes will be operated by independent entities as, at the moment, there are no incentives for providing this costly infrastructure.

An alternative approach to operating full nodes is to run light nodes [1,2,3,4], with the Bitcoin Simplified Payment Verification client [5] being an example. Unfortunately, light nodes are insufficient to verifiably track the state of an application: they only verify the inclusion of a set of desired transactions but have no guarantees over whether this set is complete, e.g., if it includes *all* transactions reading from or writing to the state of a particular dApp. This can lead to stale and, over time, potentially inconsistent results if the light node connects to a full node that withholds data, either inadvertently or maliciously.

A light node is also problematic in case the security of a blockchain is compromised: since it only verifies transaction inclusion, a light node can be tricked into accepting invalid transactions. In contrast, a full node re-executes all transactions and, therefore, always maintains a valid local state, regardless of the number of adversarial validators. The lack of validity guarantees for light nodes is troublesome, as users and dApp operators mainly care about the security of their dApps and less about the security of the blockchain as a whole: If the underlying chain is compromised, e.g., there are forks, dApp operators that run their own full nodes can choose one of the forks, be sure that there are no validity violations or lost updates, and migrate the dApp state to another blockchain.

In this paper, we introduce *sparse nodes*, a new type of blockchain node that sits between light and full nodes. Sparse nodes follow a subset of the blockchain state by retrieving, verifying inclusion of, and re-executing *only* the set of transactions that read from or write to, e.g., the state of a specific dApp or a user. We define sparse nodes formally through a *predicate*, which, when applied to the global state, identifies a subset thereof called the *sparse state*. Sparse nodes *verify completeness and validity of the set of received transactions*, guaranteeing that their local sparse state is always valid. Table 1 compares the security properties of sparse nodes with those of full and light nodes. These properties hold even under adversarial majorities, and they are the following: (i) *sparse validity*, which means that the node will only accept transactions that are valid with respect to its local current sparse state; (ii) *fork consistency* [6,7,8], which means that two sparse nodes with the same predicate and reading from the same fork will output the same sparse state. Finally, (iii) *verifiable completeness*, which

means that a sparse node can verify if it received all the transactions that touch its sparse state.

Table 1: Properties of full, sparse, and light nodes under adversarial majorities.

	Validity	Fork Consistency	Completeness
Full Node	Yes	Yes	Per Fork
Sparse Node	Sparse	Yes	Per Fork
Light Node	No	No	No

The cost of running a sparse node is roughly proportional to the number of transactions touching its sparse state, thus isolating the cost of running a sparse node from the external workload of other dApps. This makes it feasible again for dApp developers to download, verify, execute, and store transactions on the dApp sparse state, thus increasing the robustness of applications in high-throughput blockchains. Sparse nodes can be run by users or operators that wish to monitor the state of an application and listen to the events: notable examples are *bridge* operators, *rollup* sequencers and watchers, *payment channel* users and watchtowers, *re-staking* and *remote staking* collectives [9,10], user *wallets*, *DAO* token holders, and many more. Sparse nodes can additionally function as read caches or replicas, facilitating the separation of read and write operations during scaling. This enables the dynamic deployment of sparse nodes to increase redundancy and read bandwidth for popular dApps, reducing the need for more full nodes and thereby saving network bandwidth and disk space. While sparse nodes offer more marked benefits when deployed for high-throughput blockchains, their deployment on *low-throughput chains* such as Ethereum, results in lower bandwidth, computational, and storage consumption when compared to full nodes.

Contributions. After presenting the model and assumptions (Section 2), we introduce and formalize, for the first time, the concept of a sparse node, and we define the security guarantees it provides under both honest and dishonest majorities (Section 3). We focus our analysis on quorum-based blockchains that are secure in non synchronous networks, as this is the setting in which most blockchains operate. Our formalization is, nevertheless, easily extendable to other settings, e.g., Nakamoto-style consensus chains.

Then, we present Sunfish (Section 4), the first secure protocol for sparse nodes. We describe two instances of Sunfish that differ in the choice of data structures to offer different trade-offs: Sunfish-C uses counters and minimizes validator overhead, Sunfish-HC uses trees and hash chains and optimizes the reads (proof size). Afterward, we showcase the required resources for both Sunfish-C and Sunfish-HC (Section 5) based on real-world usage data of two dApps: a blockchain bridge and a wallet user. We estimate bandwidth reductions of 10x and 10^8x for the bridge and wallet, respectively, when compared to running a full node (improvement is inversely proportional to how frequently the app interacts with the chain). Finally, we compare sparse nodes with related work and conclude with a discussion of the impact of our work along with new research directions (Section 6).

2 Preliminaries and Models

Notation. The curly bracket notation $\{\cdot\}$ refers to sets, whereas the square bracket notation $[\cdot]$ refers to ordered sequences. The symbols $A \preceq B$ and $A \prec E$ indicate that A is a prefix of B and a strict prefix of E. The notation $|D|$ denotes the size of the sequence if D is a sequence, or the size of a set if D is a set.

Ledger Model. We model a ledger $\mathcal{L}$ as the output of a Byzantine fault tolerant state machine replication (BFT-SMR) protocol [11,12,13]. State machines are deterministic machines that, at all times, store the state of the system and, upon receiving a set of inputs, they output a new, updated state by evaluating the inputs over a *state transition function* δ. A state transition is *valid* if δ executes without errors. In a network of mutually distrusting nodes, each running a replica of the same state machine, a BFT-SMR protocol ensures that all correct nodes maintain a consistent state, even in the presence of a subset of adversarial nodes. Consider a BFT-SMR protocol with $n = 3f + 1$ nodes of which f are controlled by the adversary. Upon receiving on input a new transaction tx from the environment, a correct node moves from state S^i to $S^{i+1} = \delta(S^i, \mathsf{tx})$ only if $\delta(S^i, \mathsf{tx})$ is a valid state transition and if a *quorum* of at least $2f + 1$ nodes have acknowledged the transition. Consider an empty ledger $\mathcal{L}^0$ with genesis state S^0. To ascertain the i-th state S^i of a ledger $\mathcal{L}^i = [\mathsf{tx}_1, \ldots, \mathsf{tx}_i]$, with $i > 0$, transactions are applied as follows: $S^i := \delta(\ldots\delta(\delta(S^0, \mathsf{tx}_1), \mathsf{tx}_2)\ldots, \mathsf{tx}_i)$. As shorthand notation, we use $S^i := \delta(S^0, \mathcal{L}^i)$ to denote successive application of all transactions $\mathsf{tx} \in \mathcal{L}^i$ given an initial state S^0. The definitions in this paper also apply to longest chain protocols [14,15,16] by considering the stable ledger.

Let K and V be sets of valid keys and valid values, respectively. We model the *state of a node as a key-value store*, i.e., a collection of (k, v), with $k \in K$ and $v \in V$. k is a unique identifier (e.g., account or contract address) used to reference a specific value in the store, whereas v is the data (e.g., account balance or contract state) associated with a particular key. A transaction reads from an input state S^i and writes to an output state S^{i+1} by consuming some state elements (k, v) in S^i and generating new ones. We refer to the values that are read and written by a transaction as *read set* and *write set*, respectively. This is to clearly distinguish it from the input and output of a transaction, which, in some chains like Ethereum [17], is the whole state of the ledger.

Adversarial Model. Since sparse nodes retain interesting security properties under dishonest majority, we follow [18] and consider an adversary whose corruption level varies over time. The adversarial resilience for synchronous protocols is $f < n/2$, while for asynchronous and partially synchronous protocols is $f < n/3$. We focus on asynchronous and partially synchronous protocols. When $f < n/3$, a ledger fulfills the following properties:

Definition 1 (Ledger Validity). *A ledger $\mathcal{L}$ is valid if, for any round r, $S^{r+1} = \delta(S^0, \mathcal{L}^{r+1})$ executes errorless.*

Definition 2 (Ledger Safety). *At any round s, $r \leq s$, any two correct nodes i, j output a ledger $\mathcal{L}$ such that $\mathcal{L}^i_r \preceq \mathcal{L}^j_s$.*

Definition 3 (Ledger Liveness). *Any valid transaction that is provided to a correct node will eventually be included in the ledger.*

In a round with $n/3 \leq f \leq 2n/3$, the adversary can break liveness by preventing correct nodes from creating a *quorum*, and safety by creating quorums over conflicting states (*forks*) of the ledger. However, at least one correct node must have participated in the quorum and each fork remains valid.

For $f > 2n/3$, the adversary can single-handedly violate validity, safety and liveness of the ledger by including invalid state transitions, creating forks, or stop appending transactions to the ledger.

Prover-Verifier Model. A sparse node protocol is an interactive protocol between a sparse node, acting as verifier V, and a non-empty set $\mathcal{P}$ of provers. We assume V is honest and adheres to the correct protocol execution. Provers can be adversarial and execute any probabilistic polynomial-time algorithm.

Light nodes operate under the assumption that they connect to multiple full node-provers, with at least one of them being honest (existential honesty assumption). In Sunfish, we adopt a different model, motivated in Section 4.1: we let our sparse node *connect to a single validator-prover, only trusted for liveness.*

Cryptographic Assumptions. We assume collision resistant hash functions.

Network Assumptions. We consider protocols whose execution proceed in discrete rounds $r = \{0, 1, 2, \dots\}$. We inherit the classic network assumption of a full node, i.e., the sparse node can receive transactions by either connecting to full nodes or validators, or by joining the gossip network.

3 Sparse Nodes

A full node downloads, validates, and re-executes all transactions, maintaining a complete copy of the ledger and storing the entire state. A *sparse node*, instead, downloads, validates, and re-executes only a specific set of transactions, maintaining a partial copy of the ledger, named *sparse ledger*, and storing a subset of the global state, named *sparse state*. Transactions in the sparse ledger share a common property: for instance, they all read from or write to the state of the same contract or of the same address. The sparse ledger output by a sparse node must be both *complete* and *sparsely valid*, which means that it must include all transactions in the ledger with the desired property and these transactions must correctly execute when applied to the node's sparse state. Otherwise, the sparse node outputs an error.

3.1 Definitions

Consider a ledger $\mathcal{L}$. At any round r, the state S^r of $\mathcal{L}^r$ is the set of (k, v) s.t. $\forall (k, v) \in S^r : k \in K, v \in V$.

Definition 4 (State Predicate $\mathcal{X}_s$). *A state predicate $\mathcal{X}_s$ is a function $\mathcal{X}_s(k) : K \to \{1, 0\}$.*

A state predicate is valid for a ledger $\mathcal{L}$, if it is supported by the ledger protocol. A *sparse state* $\hat{S} \subseteq S$ is the subset of the state elements $(k, v) \in S$ s.t. $\mathcal{X}_s(k) = 1$.

Definition 5 (Sparse State $\hat{S}$). *At any round r, $\hat{S}^r := \{(k, v) | (k, v) \in S^r \wedge \mathcal{X}_s(k)\}$ is the sparse state identified by $\mathcal{X}_s$.*

Since S^r changes any time a new transaction is appended to the ledger, at every new append $\mathcal{X}_s$ must be evaluated on all updated and added elements $(k, v) \in S^r$.

We now define the *sparse state transition* $\hat{\delta}$, the function that allows to move from $\hat{S}^i$ to $\hat{S}^{i+1}$. Let us look at the inputs to $\hat{\delta}$. The standard transition function δ of the ledger takes as inputs the global state and a transaction; a sparse node, however, does not have the global state, but just the sparse state. If we let $\hat{\delta}$ take as input the sparse state and a transaction, $\hat{\delta}$ cannot properly execute, as the transaction might read from state elements (e.g., gas objects, contract bytecode) not in the sparse state. Let $\mathcal{R}(\mathsf{tx}), \mathcal{W}(\mathsf{tx})$ be the read and write sets of a transaction tx, respectively. Now, we solve this by letting $\hat{\delta}$ take $\hat{S}$ and $(\mathcal{R}(\mathsf{tx}), \mathsf{tx})$ as inputs.

Some ledgers have transactions that directly include a commitment to the values in their read set, e.g., in Bitcoin is the hash of the transaction holding the unspent output. Other ledgers, e.g., Ethereum, have transactions that specify the keys of the state elements they read from (e.g., account addresses or contract storage slots), but the actual values associated with those keys (e.g., account balances, contract storage values) are not included nor committed in the transaction itself. This way of interacting with the global state is known as *access-by-key*. To define $\hat{\delta}$ for access-by-key chains, we therefore need to assume that any transaction includes a commitment to the values in its read set or, alternatively, being $\hat{\delta}$ deterministic, a commitment to the values in its write set.[4] In this way, even if a transaction reads from state elements external to the sparse state, a sparse node can verify the values v provided by an untrusted prover against the commitment and, after assessing their correctness, it can apply $\hat{\delta}$ over the values. In the following, to ease notation, we will omit the data and metadata necessary to open the commitment, and, w.l.o.g., assume that the commitment is to the read set of the transaction.

Definition 6 (Sparse State Transition $\hat{\delta}$ (aka. Sparse Execution)). *At any round r, on input $\hat{S}^r$ and $(\mathcal{R}(\mathsf{tx}), \mathsf{tx})$, a sparse state transition function $\hat{\delta}$ outputs a sparse state $\hat{S}^{r+1} = \hat{\delta}(\hat{S}^r, (\mathcal{R}(\mathsf{tx}), \mathsf{tx}))$ by executing the following steps:*

1. *It computes $\hat{\mathcal{R}}^r = \hat{S}^r \cup \mathcal{R}(\mathsf{tx})$.*
2. *It checks that $\forall (k, v) \in \hat{\mathcal{R}}^r$ s.t. $\mathcal{X}_s(k) = 1$, $(k, v) \in \hat{S}^r$.*
3. *It checks that $\forall (k, v) \in \mathcal{R}(\mathsf{tx})$, (k, v) is in the commitment in tx.*
4. *It executes $\delta(\hat{\mathcal{R}}^r, \mathsf{tx})$ assuming that $\forall (k, v) \in \hat{\mathcal{R}}^r$ s.t. $\mathcal{X}_s(k) = 0$, $(k, v) \in S^r$.*
5. *It outputs $\hat{S}^{r+1}$ computed by taking the result of $\delta(\hat{\mathcal{R}}^r, \mathsf{tx})$ and removing all (k, v) s.t. $\mathcal{X}_s(k) = 0$.*

[4] In many blockchains, e.g. Ethereum, such commitments are provided in blocks through a commitment to a state tree.

If any of the checks fail, or if δ fails, output error.

Informally, $\hat{\delta}$ checks that all elements in $\mathcal{R}$ (tx) s.t. $\mathcal{X}_s(k) = 1$ are valid and that all values in the read set of tx are in the commitment. Then, on input $\hat{\mathcal{R}}^r$ and tx, $\hat{\delta}$ executes the transition function δ of the ledger assuming that all the state elements in $\hat{\mathcal{R}}^r$ s.t. $\mathcal{X}_s(k) = 0$ are valid. Finally, $\hat{\delta}$ outputs the result of δ pruned by all the elements for which $\mathcal{X}_s(k) = 0$.

We highlight that while the standard execution of δ fails if any state element in $\hat{\mathcal{R}}^r$ s.t. $\mathcal{X}_s(k) = 0$ is not in the global state S^r of the ledger, $\hat{\delta}$ *assumes* that all elements in $\hat{\mathcal{R}}^r$ s.t. $\mathcal{X}_s(k) = 0$ are in S^r and therefore it does not fail. The definition above can be generalized to a sequence of transactions by applying $\hat{\delta}$ sequentially: $\hat{S}^2 = \hat{\delta}(\hat{\delta}(\hat{S}^0, (\mathcal{R}(\mathsf{tx}_1), \mathsf{tx}_1)), (\mathcal{R}(\mathsf{tx}_2), \mathsf{tx}_2))$. As a shorthand notation for $\hat{\delta}$, we consider the read set as part of the transactions: $\hat{S}^2 = \hat{\delta}(\hat{\delta}(\hat{S}^0, [\mathsf{tx}_1, \mathsf{tx}_2]))$.

From a state predicate $\mathcal{X}_s$ we now derive a *transaction predicate* $\mathcal{X}_t$ which, on input a transaction tx, it outputs 1 if at least one of the elements in $\mathcal{R}(\mathsf{tx})$ or $\mathcal{W}(\mathsf{tx})$ yields $\mathcal{X}_s(k) = 1$.

Definition 7 (Transaction Predicate $\mathcal{X}_t$). *Let $\mathcal{X}_s$ be a state predicate. On input a transaction tx, a transaction predicate $\mathcal{X}_t$ outputs 1 if $\exists (k, v) \in (\mathcal{R} \cup \mathcal{W})(\mathsf{tx})$ s.t. $\mathcal{X}_s(k) = 1$. Else, it outputs 0.*

We define a *sparse ledger* as the sequence of transactions in $\mathcal{L}$ s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$.

Definition 8 (Sparse Ledger $\hat{\mathcal{L}}$). *Let $\mathcal{L}$ be a ledger. At any round r, $\hat{\mathcal{L}}^r := [\mathsf{tx} \in \mathcal{L}^r | \mathcal{X}_t(\mathsf{tx})]$ is the sparse ledger identified by $\mathcal{X}_t$.*

We observe that a full node is a sparse node for which, at any round r, $\hat{S}^r = S^r$ and $\hat{\mathcal{L}}^r = \mathcal{L}^r$. By definition, $\hat{\mathcal{L}}$ is complete with respect to $\mathcal{L}$.

Definition 9 (Completeness). *Consider a ledger $\mathcal{L}$ and a transaction predicate $\mathcal{X}_t$. A sparse ledger $\hat{\mathcal{L}}$ is complete with respect to $\mathcal{L}$ if, at any round r, it does not exist a transaction tx s.t. $\mathcal{X}_t(\mathsf{tx}) \wedge \mathsf{tx} \notin \hat{\mathcal{L}}^r \wedge \mathsf{tx} \in \mathcal{L}^r$.*

We now state another interesting property of a sparse ledger, namely *sparse validity*. Let $\hat{\mathcal{L}}^r \setminus \hat{\mathcal{L}}^{r-1}$ denote the transactions in $\hat{\mathcal{L}}^r$ but not in $\hat{\mathcal{L}}^{r-1}$.

Definition 10 (Sparse Validity). *A sparse ledger $\hat{\mathcal{L}}^r$ is valid if, for any round r, $\hat{\delta}(\hat{S}^{r-1}, \hat{\mathcal{L}}^r \setminus \hat{\mathcal{L}}^{r-1})$ executes errorless.*

A ledger for which sparse validity holds is termed *sparsely valid ledger* or a *valid sparse ledger*. We note that sparse validity follows from sparse execution and it is weaker than validity.

The above definitions are general and apply to different flavours of sparse states. For instance, a sparse state can only include the state elements identifying the coins owned by a particular *address*, the balance maintained by a set of addresses, the state of a specific *contract* or, as discussed in Section 3.4, the events emitted by a contract. The flavours of sparse states that can be defined depend on the variety of state predicates the ledger supports; this, in turn, depends on the availability of authenticated data structures and commitments for addresses, accounts, or contracts.

3.2 Sparse Node Security

We define sparse node security under the adversarial model defined in Section 2. Consider an adversary $f \leq 2n/3$.

Definition 11 (Sparse Node Security, $f \leq 2n/3$). *When $f \leq 2n/3$, a sparse node protocol $\Pi(\mathcal{P}, V)$ is* secure *if, at any round r, V outputs $(\hat{\mathcal{L}}^r, \hat{S}^r)$ such that $\hat{\mathcal{L}}^r$ is complete with respect to $\mathcal{L}^r$ and sparsely valid with respect to its previous outputs (if any), or it outputs an error.*

A sparse node fulfilling Definition 11 is *secure*. One can define a *custom error handling*: for instance, if an update to $\hat{\mathcal{L}}$ leads to an incomplete sparse ledger, the node can be instructed to connect to a different set $\mathcal{P}$ of provers and continue. Similarly, if an update to $\hat{\mathcal{L}}$ leads to an sparsely invalid $\hat{\mathcal{L}}$, the node can be instructed to terminate, deeming the security of $\mathcal{L}$ compromised. We note that, in the absence of a byzantine quorum, a sparse node outputs a *valid* $\hat{\mathcal{L}}$.

In Appendix A, we prove that a secure sparse nodes i fulfill the *future self-consistency* (defined in Lemma 2) and the *fork consistency* properties.

Lemma 1 (Fork Consistency, $f \leq 2n/3$). *For any rounds r and $r' \leq r$, any two secure sparse nodes i, j reading from the same fork of $\mathcal{L}$ output sparse ledgers s.t. $\hat{\mathcal{L}}_i^{r'} \preceq \hat{\mathcal{L}}_j^r$ and sparse states s.t. $\hat{S}_j^r = \hat{\delta}(\hat{S}_i^{r'}, \hat{\mathcal{L}}_j^r \setminus \hat{\mathcal{L}}_i^{r'})$, if none of them outputs an error.*

As the name suggests, fork consistency is a per-fork property. In this paper, we do not discuss fork detection, but synchronous gossip techniques are shown to solve this problem [19,7,8].

Consider now an adversary $f > 2n/3$. In this setting, any security notion on $\mathcal{L}$ is compromised and any notion of completeness is meaningless. However, a sparse node still outputs a sparse ledger that is sparsely valid (and self-consistent).

Definition 12 (Sparse Node Security, $f > 2n/3$). *When $f > 2n/3$, a sparse node protocol $\Pi(\mathcal{P}, V)$ is* secure *if, at any round r, V outputs $(\hat{\mathcal{L}}^r, \hat{S}^r)$ such that $\hat{\mathcal{L}}^r$ is sparsely valid with respect to its previous outputs (if any), or it outputs an error.*

3.3 Continuous, Intermittent, and On-Demand Sparse Nodes

Sparse nodes can have various operating modes that differ in the extent of completeness. Sparse validity is unconditional in all modes.

A *continuous* sparse node is always online and is immediately notified when a relevant transaction ($\mathcal{X}_t(\mathsf{tx}) = 1$) gets appended to the ledger. Assuming that at least one node it is connected to is live, a continuous sparse node is complete at all times.[5] This is the *primary operating mode* we consider in this work.

[5] We can further categorize based on how quickly a sparse node is notified, e.g., as soon as a transaction gets added to a block or as soon as it gets finalized (which is consensus-specific and may be earlier). We leave this exploration for future work.

A *header node* is also always online but reads *all block headers* (similarly to SPV nodes) irrespective of whether a relevant transaction is in the block. This mode offers the benefit that liveness failures are immediately detectable (assuming blocks are produced at a known rate), although at the cost of increased resource consumption.

An *intermittent* sparse node alternates between wake and sleep periods, either with some periodicity or at random. The sparse ledger of an intermittent node is only complete up to the most recent wake period, a property we call *prefix completeness*. Such a node only exhibits completeness when awake.

An *on-demand* sparse node is a node that wakes up, stays awake for the time it takes to get the data, and then falls asleep forever. This node is only interested in a single snapshot of a complete and valid sparse ledger and its state.

3.4 Event-Based Sparse Node

A typical way to read blockchains is to listen to events emitted by transactions that call a smart contract. Blockchains exhibit more structure than a ledger, and events are stored as part of the transaction's metadata, called transaction logs. Importantly, they are not stored in the state of the chain. Events inform about changes in the state of a contract or about calls to specific functions. At present, most applications developers or operators run full nodes to listen to specific logs generated during execution.

In the spectrum between full and light nodes, a special type of sparse node is an *event node*, i.e., a node that only reads specific events. An event node is more lightweight than a sparse node: it has no sparse ledger and no sparse execution, as events cannot be executed. Its sparse state is an *append-only* key-value store whose elements (k, v) are the events of interest emitted. *Completeness* is now a property of the sparse state: $\hat{S}^r$ is complete w.r.t. $\mathcal{L}^r$ if, at any round r, $\nexists (k, v)$ s.t. $\mathcal{X}_s(k) = 1 \wedge v \notin \hat{S}^r \wedge v \in \mathcal{L}^r$.

It follows that an event node has weaker security than a sparse node, as there is no notion of sparse validity. An event node either outputs a *complete sparse state* or outputs an error. In the remainder of the paper, we will focus on sparse nodes as defined in Section 3.1 and Section 3.2, as they are more complex. In Section 5, besides sparse nodes, we evaluate event nodes, showing their efficiency.

3.5 Sparse Node Resources

A sparse node requires bandwidth, computational power, and storage for downloading, validating, and storing transactions as well as storing the sparse state. These resources are *proportional to the number of transactions in* $\hat{\mathcal{L}}$ *and to the size of* $\hat{S}$. In ledgers using commitments with non-constant-sized openings (e.g., Merkle trees), a *small* multiplying factor η appears next to $|\hat{\mathcal{L}}|$, proportional to the opening sizes.

Table 2: Normalized resources for a full, sparse, and light node.

	Norm. Res.
Full Node	$\mathcal{O}(\lvert\mathcal{L}\rvert + \lvert S\rvert)$
Sparse Node	$\mathcal{O}(\eta\lvert\hat{\mathcal{L}}\rvert + \lvert\hat{S}\rvert)$
Light Node	$\mathcal{O}(1)$

On top of this, like light and full nodes, sparse nodes also need to receive regular updates about consensus parameters, e.g., validators in the current committee. This requires expending $\mathcal{O}(\lambda|\mathcal{L}|)$ resources where λ captures the rate of committee changes. For most existing PoS chains, these changes occur rarely, e.g., once a day, so λ is extremely small. Therefore, in this work, we omit it from resource usage for all types of nodes.[6]

Table 2 compares the resources consumed by full, sparse, and light nodes excluding the resources expended to verify committee changes. We assume a constant upper bound on the computation associated with a transaction.

Definition 13 (Sparse Node Resources). *The bandwidth, computational, and storage resources consumed by a sparse node are* $\mathcal{O}(\eta|\hat{\mathcal{L}}| + |\hat{S}|)$.

Note that the above refers to both intermittent, continuous, and on-demand sparse nodes. The complexity for header nodes is in-between the one of a sparse and of a full node, i.e., $\mathcal{O}(\eta|\mathcal{L}| + |\hat{S}|)$, where $|\mathcal{L}|$ appears because all headers are read.

4 Sunfish: A Protocol for Sparse Nodes

While a ledger outputs a partial order of transactions, a blockchain forces transactions in a total order to have more structure and enable, e.g., efficient reads. To describe Sunfish, we leverage a blockchain that is enhanced with commitments to allow for efficient reads.

4.1 Design Choices

Consider a sparse node that wants to receive information about all historical transactions that read from or write to a sparse state defined by a state predicate $\mathcal{X}_s$. Currently, a sparse node would connect to a large set $\mathcal{P}$ of full nodes (provers), so that at least one of them can be assumed honest and, therefore, be sure it will receive *all* relevant transactions. The sparse node will then wait until a timeout to get the responses from as many provers as possible and adopt the response carrying the largest number of transactions, without the ability to verify completeness other than by trusting the existential honesty assumption.

Besides the non-verifiability of completeness, connecting to multiple full nodes comes with additional disadvantages: (i) the communication and computation of the sparse node is proportional to the number of provers; (ii) the time it takes the sparse node to synchronize with the ledger is bottlenecked by the synchronization time of full nodes - interestingly, even a low throughput chain like Ethereum has roughly 1/3 of its full nodes constantly out-of-sync [23]; (iii) finally, as the resource requirements for operating a full node increase proportionally to the

[6] For sparse nodes that sync very rarely or only once, committee change updates can be further compressed using Succinct Zero-Knowledge Proofs [20,21,22].

size and/or the throughput of the ledger, the initial existential honesty assumption becomes less realistic as the number of public full nodes reduces, especially for high-throughput blockchains.

If we require the sparse node to connect to validators, under existential honesty, the node would need to connect to at least $f + 1$ validators. Instead, in Sunfish, we want to minimize bandwidth requirements for the sparse node and reduce the communication load on validators, thus our sparse node *connects to a single validator, only trusted for liveness*. In case this validator does not respond or stops responding, the sparse node can connect to a different one, until it finds a validator that is live. To prevent a malicious validator from withholding transactions and remain undetected, we equip the sparse node with a mechanism to verify completeness.

4.2 Sunfish Data Structures for Verifying Completeness

To verify completeness, in Sunfish, we present two distinct authenticated data structures (counters and hash chains) that achieve different trade-offs.

Sunfish-C. A sparse node can verify completeness by having knowledge of the total number of transactions in the ledger that touch its sparse state.

Consider validators maintaining a *global counter* ctr_G *for any sparse state whose predicate is supported by the ledger*. The global counter is *initialized at 0 at genesis* and incremented by 1 every time a new transaction tx s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ is added to the ledger. One option would be to have validators building a Merkle tree with all the counters ctr_G, and include the Merkle roots in every block header; unfortunately, this comes with the unpractical cost of having validators maintaining a massive tree and updating it at every block. Alternatively, validators could maintain a *local counter* ctr_L *for any sparse state whose predicate is supported by the ledger*, with ctr_L *initialized at 0 at every new block* and incremented by 1 every time a transaction tx s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ is added to the block. For each new block, validators construct a Merkle tree with the non-zero local counters for the block and commit this tree within the block header. Since the number of transactions in a block is rather small, it is feasible for validators to handle these trees; however, to know the total number of transactions in the sparse state, a sparse node needs to download and check all block headers (it needs to necessarily be a header node).

While the first approach commits to the global state of counters ctr_G, the second approach commits to the local, per-block state of counters ctr_L. Towards our final data structure, we get the best of both worlds by combining global and local counters, but without committing to the global state of counters. Instead, we periodically and deterministically include in the local per-block tree a subset of global counters, to ease bootstrapping and securely enable other operating modes (continuous, intermittent, on-demand). Let each sparse state $\hat{S}$ supported by the chain have a unique identifier $\mathsf{id}\hat{\mathsf{S}}$.

Sunfish-C requires validators building a per-block Merkle tree as follows: (i) the leaves of the tree are tuples $(\mathsf{id}\hat{\mathsf{S}}, \mathsf{ctr}_G, \mathsf{ctr}_L)$ lexicographically sorted by $\mathsf{id}\hat{\mathsf{S}}$,

(ii) the tree has one leaf for each sparse states with $\mathsf{ctr}_L \neq 0$, and (iii) the tree has one leaf for each sparse states whose $\mathsf{id}\hat{\mathsf{S}}$, given on input to a function ψ along with the height h of the block, yields 0. We require ψ to be a deterministic, predictable, and periodic function: e.g., $\psi(\mathsf{id}\hat{\mathsf{S}}, \mathsf{h}) := (\mathsf{id}\hat{\mathsf{S}} + \mathsf{h})\%N$ for a period N. The root of the tree is then included in the block header. Thus, block headers commit to the counters updated in the block and, periodically, to a subset of global counters as well.

With this data structure, we get several advantages. A sparse node can verify if its sparse state with identifier $\mathsf{id}\hat{\mathsf{S}}$ has a leaf in the tree of a block (inclusion proof) and, if this is the case, it checks completeness by reading the correspondent counters. A sparse node can also verify if its sparse state lacks a leaf in the tree because the tree is lexicographically sorted. A *non-inclusion proof* consists of two inclusion proofs for the leaves lexicographically preceding and following the $\mathsf{id}\hat{\mathsf{S}}$ of the sparse state, and it is verified by checking adjacency and validity of the two proofs. With this data structure, a sparse node can only download the block headers relevant for its sparse state, while periodically having completeness guarantees (no relevant block header was skipped) by verifying that the number of transactions received match the value of ctr_G committed in the last block for which $\psi = 0$. Finally, by reading the counters for two adjacent blocks with $\psi = 0$, sparse nodes can read chunks of the chain with constant cost.

Sunfish-HC. An alternative approach to using counters, is to ask validators to generate, per sparse state, a hash chain of transactions and include the chain head in every block header. A sparse node can be certain to have a complete set of transactions by locally computing the hash chain and compare the obtained chain head with the one in the block header. Given the possibly high number of sparse states, to optimize space, validators could arrange the chain heads of all sparse states supported by the ledger in a tree and include the root in every block header. However, this has two drawbacks: validators maintaining a massive tree and updating it at every block, and requiring strict sequentiality in processing transactions of each sparse state.

Towards efficiency and parallelizability, we combine hash chains and trees in a different manner. The data structure used by Sunfish-HC is constructed as follows: for each sparse state and each block, validators build a Merkle tree with the transactions in the block that touch the sparse state. Then, per sparse state, they generate a hash chain with the roots of these trees spread across different blocks. Finally, per each block, validators construct an overlay Merkle tree including the chain heads of the sparse states that got transactions in the block; the leaves of this tree are of the form $(\mathsf{id}\hat{\mathsf{S}}, \mathsf{head})$, with head being the chain head for the sparse state $\mathsf{id}\hat{\mathsf{S}}$. To enable the same features of Sunfish-C, i.e., non-inclusion proofs, efficient bootstrapping and reads, the overlay tree is lexicographically sorted by $\mathsf{id}\hat{\mathsf{S}}$ and further includes a leaf for a sparse state with periodicity given by ψ. Finally, validators include the Merkle root of the overlay tree in the block header.

Comparison. Let Q be the average number of sparse states having transactions in a block, and M the average number of transactions per block.

Proof Size: In Sunfish-C, the sparse node receives $\mathcal{O}(|\hat{\mathcal{L}}|)$ transactions, reads the counters in $\mathcal{O}(|\hat{\mathcal{L}}| \log Q)$, and checks transaction inclusion in $\mathcal{O}(|\hat{\mathcal{L}}| \log M)$. The proof size is $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$, with $\eta = \log M + \log Q$. In Sunfish-HC, the sparse node receives $\mathcal{O}(|\hat{\mathcal{L}}|)$ transactions and verifies the chain head inclusion in $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log Q$. The proof size for Sunfish-HC is smaller because the hash chain already guarantees transaction inclusion.

Validators' storage and compute: In Sunfish-C, validators store and update 1 counter per sparse state (8 bytes with $\mathcal{O}(1)$ updates). In Sunfish-HC, validators store and update 1 chain head per sparse state (64 bytes with $\mathcal{O}(1)$ updates).

4.3 Sunfish Prover-Verifier Protocol

Consider a sparse node V that when wakes up is only aware of the genesis state. The node selects a predicate $\mathcal{X}_s$ supported by the ledger, it connects to 1 validator-prover P and sends $\mathcal{X}_s$ to P, asking for all historical transactions that read from or write to the sparse state defined by $\mathcal{X}_s$. Upon receiving $\mathcal{X}_s$, P extracts $\mathcal{X}_t$ from $\mathcal{X}_s$ and starts sending to V all transactions in $\mathcal{L}$ s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$, along with the completeness and inclusion proofs. Specifically, P sends to V:

1. The block headers necessary to extract the historical consensus parameters of the chain. For BFT protocols, these are end-of-epoch block headers [2];
2. all transactions such that $\mathcal{X}_t(\mathsf{tx}) = 1$, along with the headers of the blocks they are included in, and the inclusion proofs;
3. The header of the latest block B_c that includes the completeness indicator for $\mathcal{X}_s$ (ctr_G or chain head) and the completeness proof;
4. The header of any block following B_c that includes transactions s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$, the transactions themselves, and the inclusion and completeness proofs;
5. *Optional (header node):* The header of any block descending from B_c that do not include transactions s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$, along with the non-inclusion proof.

Upon receiving this data from P, V checks: (i) the validity of the history of consensus parameters, (ii) the validity of all received block headers, (iii) the correct inclusion of transactions s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ in a valid block header, and (iv) the completeness of the set of received transactions. If any check fails, V outputs error. Else, V *sparsely executes* the transactions in the chain: If any error is output by $\hat{\delta}$, the node outputs error. Otherwise, it stores all the transactions (optionally, e.g., to serve reads to light nodes, it also stores the received block headers and proofs) and maintains the state resulting from the sparse execution.

4.4 Analysis

In Appendix A we prove the following theorems:

Theorem 1 (Sunfish Security, $f(r) \leq 2n/3$). *Sunfish is a secure protocol for sparse nodes as per Definition 11.*

Theorem 2 (Sunfish Security, $f > 2n/3$). *Sunfish is a secure protocol for sparse nodes as per Definition 12.*

Theorem 3 (Sunfish Resources). *The bandwidth, computational, and storage resources consumed by Sunfish are* $\mathcal{O}(\eta|\hat{\mathcal{L}}| + |\hat{S}|)$.

5 Evaluation

Since Sunfish performs simple operations (DB lookups, integer arithmetic, hashing), we expect minimal computational impact on validators. We evaluate Sunfish on the Sui blockchain to show that it is practical even for high-performance blockchains, although it can be integrated in most chains.

The Sui Blockchain. Sui [24] is a recent decentralized, permissionless smart-contract platform designed for high-throughput and low-latency asset management. Sui uses the Move programming language to define assets as objects. The basic unit of storage in Sui is the object, addressable on-chain by a unique ID. A smart contract is also an object ("package"), and it manipulates objects on the Sui network. To support on-chain activity monitoring, the Sui network emits events. Sui validators produce certified *checkpoints* [25] that contain a sequence of transactions and form a hash-chain, similar to traditional blockchains. Each Sui checkpoint contains a *summary*, i.e., equivalent to a block header, containing the various digests: We assume each summary includes the Merkle root of all the transactions in the checkpoint and their execution results ("effects"), as well as the Merkle root for checking completeness.

Integrating Sunfish into Sui. We consider a few applications currently running on the Sui blockchain. The state of Sui can be viewed as a key-value store with object IDs as keys and the digests as values. We compare the data consumed by a full and a sparse node for the Wormhole bridge [26] and the Wave wallet [27]. We consider sparse states identified by different predicates: package-based, event-based, and address-based. We consider: (i) the Wormhole bridge, via package: $\mathcal{X}_t(\mathsf{tx}) = 1$ if tx touches a Wormhole package. (ii) The Wormhole bridge, via events: $\mathcal{X}_t(\mathsf{tx}) = 1$ if tx emits Wormhole events. Here, the sparse node only receives events, not transactions. (iii) The Wave wallet, via address: $\mathcal{X}_t(\mathsf{tx}) = 1$ if tx sends coins to or receives coins from the address of a Wave wallet user.

Data collection. We have collected real-world data from the Sui blockchain measuring past traffic patterns of the aforementioned applications. Specifically, we looked at a day's worth of data corresponding to epoch 507 (August 31st, 2024). On that day, Sui had 356279 checkpoints, i.e., an average of 4.12 checkpoints per second. We then measured the following data: (1) Number of dapp-specific transactions or events emitted per second (R); (2) Number of checkpoints with at least one dapp-specific transaction or event emitted per second ($C \leq R$ and $C \leq 4.12$; worst-case estimate, $C = \mathsf{min}(R, 4.12)$); (3) Avg. transaction effect size $e = 1044.74$ B and avg. event size $v = 106.48$ B; (4) Avg. number of transactions per checkpoint ($T = 9.35$) and unique streams touched per checkpoint (S, which we approximate and set to $S = T$); (5) Avg size of summary $\alpha = 1457.40$ B/s and full checkpoint $\beta = 213.49$ KB/s. In Table 3 we show the actual stream rate R, obtained from a blockchain analytics software. We approximate other values by sampling 1000 checkpoints (out of 356279) and calculating the mean.

Results. We compare the proof sizes. The average size of a transaction inclusion proof is $|\pi_{tx}| = e + 32 \cdot \log(T) = 1172.74$ B, and the average size of a stream inclusion proof is $|\pi_s| = 32 \cdot \log(S) = 128$ Bytes.

If the blockchain implements Sunfish-C, a sparse node only needs to download $\pi_c = R|\pi_{tx}| + C(\alpha + |\pi_s|)$ B/s. With Sunfish-HC, we have $\pi_{hc}^{tx} = Re + C(\alpha + |\pi_s|)$ B/s. With event-nodes and Sunfish-HC, the proof sizes are smaller at $\pi_{hc}^{event} = Rv + C(\alpha + |\pi_s|)$ B/s (because transactions are not downloaded by event nodes).

App [type]	R	$\|\pi_c\|$	$\|\pi_{hc}\|$
Wormhole bridge [package]	8.55	16.56 KB/s (7.75%)	15.46 KB/s (7.24%)
Wormhole bridge [event]	8.55	16.56 KB/s (7.75%)	7.44 KB/s (3.4%)
Wave wallet user [address]	$2 \cdot 10^{-5}$	0.05 B/s (10^{-7}%)	0.05 B/s (10^{-7}%)

Table 3: Rate of traffic (R) generated by different dapps on 31st August, 2024. Last two columns show the amount of data a sparse node needs to download if a blockchain enables Sunfish commitments along with the percentage improvement over a full node (213.49 KB/s).

6 Related Work, Discussion, and Future Work

Sunfish has no close related work. It positions itself as a middle ground between full nodes [28,29,30] and light nodes [4,31,32,1,3,2,22,21]. Unlike full nodes, Sunfish does not require to download and re-execute a complete copy of the ledger: it only downloads and re-execute a subset thereof. Sunfish ensures (sparse) validity, completeness, and fork consistency (Section 3), properties that light clients do not provide because of their minimalist design and the lack of transaction re-execution. Some light client designs [33,34] consider completeness as an important property, however, they achieve it by relying on trusted execution environments [35] and do not consider re-execution.

Users can choose which node type fits their desiderata and use case best. Prior to our work, if they have high-security requirements (e.g., exchange), running a full node is the go-to option; if they run an application over a resource-constrained environment (e.g., a wallet on a phone) and favor efficiency over security, light nodes are instead the best fit. However, after a blockchain enables support for sparse nodes, operators, developers, or users that want strong security guarantees while retaining practical costs can now choose to run a sparse node. Examples are *bridge* operators, *DAO* token holders, *re-staking* [9] and *remote staking* projects [10], *on-chain gaming platforms*, sequencers and watchers of *rollups*, and users and watchtowers of *state and payment channels*.

Sparse nodes can also help optimize the blockchain infrastructure: they can serve reads to light nodes, maintain custom indexes for on-chain data, take care of hot spots to take load off of full nodes, and communicate with other sparse nodes in a transparency network to detect forks [19]. Finally, we conjecture that full nodes could be fully replaced by a fleet of sparse nodes whose combined sparse states cover the whole state of the ledger. We leave this as future work.

Acknowledgments

This work was supported by Mysten Labs and conducted during Giulia Scaffino's internship with the company. We thank the Mysten Labs Data Science team for providing necessary data to conduct our evaluation. The support by the Christian Doppler Research Association is gratefully acknowledged.

References

1. Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. Blink: An optimal proof of proof-of-work. Cryptology ePrint Archive, Paper 2024/692, 2024. `https://eprint.iacr.org/2024/692`.
2. Shresth Agrawal, Joachim Neu, Ertem Nusret Tas, and Dionysis Zindros. Proofs of Proof-Of-Stake with Sublinear Complexity. In *5th Conference on Advances in Financial Technologies (AFT 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
3. Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*. Springer International Publishing, 2020.
4. Ertem Nusret Tas, David Tse, Lei Yang, and Dionysis Zindros. Light clients for lazy blockchains. In *Financial Cryptography and Data Security 2024 (FC24)*, 2024.
5. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. `http://bitcoin.org/bitcoin.pdf`.
6. Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. USENIX Association, 2007.
7. David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. New York, NY, USA, 2002. Association for Computing Machinery.
8. Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. PODC '07. Association for Computing Machinery, 2007.
9. EigenLayer Team. Eigenlayer: The restaking collective. https://shorturl.at/sl9tE.
10. Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. Remote staking with economic safety, 2024.
11. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.
12. Leslie Lamport. The implementation of reliable distributed multiprocess systems. 1978.
13. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. 2002.
14. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*. Springer International Publishing, 2017.
15. Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Journal of the ACM (to appear)*, 2024.
16. Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. Berlin, Heidelberg, 2019. Springer-Verlag.

17. Ethereum yellowpaper, 2024.
18. Srivatsan Sridhar, Dionysis Zindros, and David Tse. Better safe than sorry: Recovering after adversarial majority. Cryptology ePrint Archive, Paper 2023/1556, 2023.
19. Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, Washington, D.C., August 2015. USENIX Association.
20. Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale, 2020. https://eprint.iacr.org/2020/352.pdf.
21. Mina docs, 2023. `https://docs.minaprotocol.com/about-mina`.
22. Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. Plumo: An Ultralight Blockchain Client, 2023. https://celo.org/papers/plumo.
23. Nodewatch, 2024.
24. Same 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.
25. Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. Sui lutris: A blockchain combining broadcast and consensus, 2024.
26. Wormhole Bridge. https://docs.sui.io/concepts/tokenomics/sui-bridging.
27. Wave Wallet on Sui. https://waveonsui.com/.
28. Sui full node transaction signatures are not verified, 2024.
29. Anamika Chauhan, Om Prakash Malviya, Madhav Verma, and Tejinder Singh Mor. Blockchain and scalability. In *2018 IEEE international conference on software quality, reliability and security companion (QRS-C)*, pages 122–128. IEEE, 2018.
30. Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. Executing and proving over dirty ledgers. In *Financial Cryptography and Data Security 2023*, 2023.
31. Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. Sok: Blockchain light clients. In *International Conference on Financial Cryptography and Data Security*, pages 615–641. Springer, 2022.
32. Sean Braithwaite, Ethan Buchman, Ismail Khoffi, Igor Konnov, Zarko Milosevic, Romain Ruetschi, and Josef Widder. A tendermint light client. *arXiv preprint arXiv:2010.07031*, 2020.
33. Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin lightweight client privacy using trusted execution. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 783–800, 2019.
34. Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. Zlite: Lightweight clients for shielded zcash transactions using trusted execution. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pages 179–198. Springer, 2019.
35. Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. Sok: Hardware-supported trusted execution environments. *arXiv preprint arXiv:2205.12742*, 2022.

A Proofs

Lemma 1 (Fork Consistency, $f \leq 2n/3$). *For any rounds r and $r' \leq r$, any two secure sparse nodes i, j reading from the same fork of $\mathcal{L}$ output sparse ledgers s.t. $\hat{\mathcal{L}}_i^{r'} \preceq \hat{\mathcal{L}}_j^{r}$ and sparse states s.t. $\hat{S}_j^r = \hat{\delta}(\hat{S}_i^{r'}, \hat{\mathcal{L}}_j^r \setminus \hat{\mathcal{L}}_i^{r'})$, if none of them outputs an error.*

Proof. Towards contradiction, suppose that there exists a round r such that sparse nodes i, j reading from the same fork of $\mathcal{L}$ output $\hat{\mathcal{L}}_i^{r'} \not\preceq \hat{\mathcal{L}}_j^r$ and $\hat{S}_j^r \neq \hat{\delta}(\hat{S}_i^{r'}, \hat{\mathcal{L}}_j^r \setminus \hat{\mathcal{L}}_i^{r'})$. This happens only if $\hat{\mathcal{L}}_i^{r'}$ or $\hat{\mathcal{L}}_j^r$ are not complete with respect to $\mathcal{L}$ or not sparsely valid, contradicting the fact that the nodes are secure. □

Lemma 2 (Future Self-Consistency). *For any correct sparse node i and any round r, $\hat{\mathcal{L}}_i^{r' \leq r} \preceq \hat{\mathcal{L}}_i^r$.*

Proof. This trivially follows from sparse validity (execution). □

Theorem 4 (Sunfish Security, $f \leq 2n/3$). *Sunfish is a secure protocol for sparse nodes as per Definition 11.*

Proof. We recall that $f \leq 2n/3$ guarantees that at least one correct consensus node contributed to the quorum. To prove Sunfish security against a $f \leq 2n/3$ adversary, we prove that, at any new block with a quorum, the sparse node either outputs a complete and sparsely valid ledger, or it outputs error. For both Sunfish-C and Sunfish-HC, sparse validity is trivially fulfilled by applying $\hat{\delta}$ (sparse execution) to the set of received transactions.

Sunfish-C: At each new block, the sparse node receives a completeness proof that opens the commitment to the values of local and global counters for sparse states. If the counters for the desired sparse state are included in the commitment, the sparse node knows the number of transactions tx s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ included in the block (ctr_L) and in the whole ledger (ctr_G). If the commitment includes no entry for the counters of the sparse state, the sparse node knows there are no transactions tx s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ in the block. Therefore, upon receiving a set of transactions from the prover, the sparse node can verify completeness. If it does not hold, it outputs error; else, it outputs a complete ledger.

Sunfish-HC: The security of Sunfish-HC follows from the security of Sunfish-C, with the only difference that counter values are replaced by hash chain heads. We recall that the head of the hash chain for a sparse state is a tuple that includes the roots of the Merkle trees for the sparse state in the second-to-last and in the last block. Two hash chains have the same head if and only if they have been generated from the same sequences of transactions and blocks. Therefore, if any transaction tx s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ has been withheld by the prover, the sparse node outputs an error as its locally computed chain head differs from the one in the commitment. Else, the sparse node outputs a complete ledger. □

Theorem 5 (Sunfish Security, $f > 2n/3$). *For $f > 2n/3$, at any round r, Sunfish achieves sparse node protocol security as per Definition 12.*

Proof. For $f > 2n/3$ it is sufficient to prove that, at any new block, the sparse node either outputs a sparsely valid ledger, or it outputs error.

For both *Sunfish-C* and *Sunfish-HC*, sparse validity is trivially fulfilled by the sparse execution of the received transactions. Either the sparse execution succeeds, or the sparse node outputs an error. □

Theorem 3 (Sunfish Resources). *The bandwidth, computational, and storage resources consumed by Sunfish are $\mathcal{O}(\eta|\hat{\mathcal{L}}| + |\hat{S}|)$, as per Definition 13.*

Proof. Bandwidth: a Sunfish sparse node needs to download all transactions touching its sparse state, as well as the inclusion and completeness proofs. Downloading transactions requires $\mathcal{O}(|\hat{\mathcal{L}}|)$. Downloading inclusion proofs requires $\mathcal{O}(|\hat{\mathcal{L}}| \log M)$ with M being the average number of transactions in a block. Downloading completeness proofs requires $\mathcal{O}(|\hat{\mathcal{L}}| \log Q)$, with Q being the average number of updated sparse states in a block. Therefore, for Sunfish-C we have $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log M + \log Q$, while for Sunfish-HC we have $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log Q$. *Computation*: a Sunfish sparse node needs to verify transaction inclusion, completeness, and it needs to execute all transactions and compute the sparse state. We consider an upper bound β to the computation associated to a transaction. The computation required to verify transaction inclusion and completeness is $\mathcal{O}(|\hat{\mathcal{L}}| \log M)$ and $\mathcal{O}(|\hat{\mathcal{L}}| \log Q)$, respectively. The computational complexity of execution is $\mathcal{O}(|\hat{\mathcal{L}}|)$, being β a constant. Therefore, this makes for a total computation of $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log M + \log Q$ for Sunfish-C and $\mathcal{O}(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log Q$ for Sunfish-HC. *Storage*: the sparse node stores $\hat{\mathcal{L}}$ as well as $\hat{S}$, yielding $\mathcal{O}(|\hat{\mathcal{L}}| + |\hat{S}|)$ storage complexity.

It follows that the resources consumed by Sunfish are $\mathcal{O}(\eta|\hat{\mathcal{L}}| + |\hat{S}|)$, with $\eta = \log M + \log Q$ for Sunfish-C and $\eta = \log Q$ for Sunfish-HC.

□