Sunfish: Reading Ledgers with Sparse Nodes

Giulia Scaffino
TU Wien, Common Prefix, CDL-BOT

Karl Wüst
Mysten Labs

Deepak Maram
Mysten Labs

Alberto Sonnino
Mysten Labs,
University College of London

Lefteris Kokoris-Kogias
Mysten Labs

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 and, in turn, the computational and storage resources, 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 [5, 8, 20, 32], with the Bitcoin Simplified Payment Verification client [27] 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

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

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

with respect to its local current sparse state; (ii) *fork consistency* [15, 23, 25], 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.

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 [19, 33], 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, still 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 6) 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 7).

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 [16, 21, 22]. 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.

A ledger protocol is *secure* if it fulfills the following properties:

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

Definition 2 (Ledger Safety). *For any rounds 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.*

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 the *read set* and the *write set*, respectively. This is to clearly distinguish it from the input and output of a transaction, which, in some chains like Ethereum [2], is the whole state of the ledger. We let $\mathcal{R}(\mathsf{tx})$ and $\mathcal{W}(\mathsf{tx})$ denote the read and the write set of a transaction tx, respectively.[1]

Adversarial Model. We let f denote the resilience of a ledger protocol against a Byzantine adversary. For synchronous protocols, $f < n/2$, while for asynchronous and partially synchronous protocols $f < n/3$. In this work, we consider an adversary whose

[1] A more precise formulation of the read and write set is $\mathcal{R}(S, \mathsf{tx})$ and $\mathcal{W}(S, \mathsf{tx})$ because, depending on the current state S of the ledger, the state elements in the read and write sets may be assigned to different keys.

corruption level can vary over time, possibly violating the resilience threshold of the ledger.

Prover-Verifier Model. A sparse node protocol $\Pi(\mathcal{P}, V)$ is a protocol between a sparse node, acting as verifier V, and a non-empty set $\mathcal{P}$ of provers. We envision a sparse node protocol to be a *streaming and stateful* protocol, i.e., it maintains a connection to the provers and it persists its state (received transactions, current sparse state) over time. We present, however, also non-streaming operating modes. We assume V is honest and adheres to the correct protocol execution.

We will first present a *backward compatible* protocol for sparse nodes in which provers are full nodes, at least one of which is honest (existential honesty assumption), while all others can be adversarial and execute any probabilistic polynomial-time algorithm. Then, we will present an *optimized* protocol for sparse nodes that operates under a different model: the sparse node connects 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. We assume synchronous communication between the sparse node and the provers.

3 THE SPARSE NODE

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.

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. In this work, we will only consider valid state predicates. 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 reason about the inputs of $\hat{\delta}$. As seen in Section 2, the standard transition function δ of the ledger takes as inputs the global state and a transaction: $S^{i+1} = \delta(S^i, \mathsf{tx})$. A sparse node, however, has no knowledge of the global state: it only knows of the sparse state $\hat{S}$. If we let $\hat{\delta}$ take as input $\hat{S}$ and a transaction, i.e., $\hat{\delta}(\hat{S}^i, \mathsf{tx})$, then $\hat{\delta}$ might not properly execute: the transaction might read from state elements (e.g., gas objects, contract bytecode) not in the sparse state. We solve this by letting $\hat{\delta}$ take $\hat{S}$ and $(\mathcal{R}(\mathsf{tx}), \mathsf{tx})$ as inputs. This means that whenever the sparse node receives a transaction accessing its sparse state, it *must also receive the transaction's read set* in order to execute it.

This has two implications: First, if a transaction specifies an external state as dependency, e.g., it calls a smart contract external to the node's sparse state, then this dependency affects the sparse node's resource consumption (we discuss this thoroughly in Section 3.5). Second, the sparse node must be able to verify the correctness of the read set provided by the provers, otherwise it may be fooled into accepting an invalid read set. 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.[2] In this way, even if a transaction reads from state elements external to the sparse state, a sparse node can verify against the commitment the correctness of the values v received from untrusted provers; then, 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, without loss of generality, assume that the commitment is to the read set of the transaction.

Before defining the sparse state transition $\hat{\delta}$, we introduce the function $\tilde{\delta}$ to be both a *domain restriction* and a *range restriction* of the function δ. This means that $\tilde{\delta}$, on input a *subset* of the global state and a transaction, modifies the state elements in the subset in the exact same way of δ; then, $\tilde{\delta}$ outputs a *subset* of the output elements, i.e., only the ones for which $\mathcal{X}_s(k) = 1$.

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 $\hat{\delta}$ is a deterministic function that 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 $\tilde{\delta}(\hat{\mathcal{R}}^r, \mathsf{tx})$ 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 $\hat{S}^{r+1} = \tilde{\delta}(\hat{\mathcal{R}}^r, \mathsf{tx})$.*

If any of the checks fail or the execution of $\tilde{\delta}$ fails, $\hat{\delta}$ outputs error.

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

Informally, $\hat{\delta}$ checks that all elements in the transaction's read set are valid, i.e., they are in $\hat{S}^r$, and that all values in the read set of tx are in the commitment. Then, on input $\hat{\mathcal{R}}^r$ and tx, and by assuming valid the read set elements external to $\hat{\mathcal{R}}^r$, $\hat{\delta}$ behaves as the transition function δ of the ledger. Finally, $\hat{\delta}$ outputs the result of δ pruned by all the elements for which $\mathcal{X}_s(k) = 0$.

We highlight that $\hat{\delta}$ gives weaker validity compared to δ. This is because the 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, while the *sparse execution* enabled by $\hat{\delta}$ *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: $\hat{S}^2 = \hat{\delta}(\hat{\delta}(\hat{S}^0, [\mathsf{tx}_1, \mathsf{tx}_2]))$. A ledger for which sparse validity holds is termed *sparsely valid ledger* or a *valid sparse ledger*.

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.*

In this work, we will only consider valid transaction predicates, i.e., transaction predicates derived from valid state predicates. We define a *sparse ledger* $\hat{\mathcal{L}}$ as the sequence of transactions in $\mathcal{L}$ s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$. Therefore, a sparse ledger is always defined with respect to a transaction predicate $\mathcal{X}_t$ or, equivalently, by a state predicate $\mathcal{X}_s$.

Definition 8 (Sparse Ledger $\hat{\mathcal{L}}$). *Let $\mathcal{L}$ be a ledger and $\mathcal{X}_t$ a transaction predicate of $\mathcal{L}$. 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 note that a full node is a sparse node for which, at any round r, $\hat{S}^r = S^r$ and $\hat{\mathcal{L}}^r = \mathcal{L}^r$. We further observe that, by definition, $\hat{\mathcal{L}}^r$ is complete with respect to $\mathcal{L}^r$.

Observation 1 (Completeness). *By definition, a sparse ledger $\hat{\mathcal{L}}$ defined by a predicate $\mathcal{X}_t$ is complete with respect to $\mathcal{L}$. This means that, 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 relax the notion of completeness of a sparse ledger by introducing the *prefix completeness* property. This will help us defining the security of a sparse node protocol in terms of safety.

Definition 9 (Prefix Completeness). *A sparse ledger $\hat{\mathcal{L}}$ identified by a predicate $\mathcal{X}_t$ is prefix complete with respect to a ledger $\mathcal{L}$ if, for a round r and a round $r' \leq r$, $\hat{\mathcal{L}}^r := [\mathsf{tx} \in \mathcal{L}^{r'} | \mathcal{X}_t(\mathsf{tx})]$.*

We now show another interesting property of a sparse ledger: *sparse validity*. Let $\hat{\mathcal{L}}^r \setminus \hat{\mathcal{L}}^{r-1}$ denote the sequence of transactions in $\hat{\mathcal{L}}^r$ but not in $\hat{\mathcal{L}}^{r-1}$, and $\hat{S}^{r-1}$ be the sparse state defined by $\hat{\mathcal{L}}^{r-1}$.

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

Finally, we define a *sparse mempool* $\hat{\mathcal{M}}$. We note that the mempool is always defined in the view of a node, and honest nodes may never share the same view of it.

Definition 11 (Sparse Mempool). *At any round r, the sparse mempool $\hat{\mathcal{M}}^r$ of a sparse node is the sequence of transactions s.t. $\mathcal{X}_t(\mathsf{tx}) = 1$ that the sparse node has received and validated, but they have not yet been included in a valid block of the ledger.*

In the rest of this work, we will only consider sparse nodes interested in maintaining a sparse ledger and a sparse state; for some applications and for some ledgers, however, users or protocol operators might benefit from maintaining a sparse mempool as well to have early insights on the queue of transactions waiting to be included into the ledger.

The above definitions are general and apply to any ledger, as well as to different flavours of sparse states and sparse ledgers. For instance, a sparse state could identify the coins owned by a particular *address*, the balance of a set of addresses, the state of a specific *contract* or the events emitted by a contract (this particular case will be discussed in Section 3.3). The flavours of sparse states that can be defined on a ledger depend on the variety of state predicates that the ledger supports.

3.2 Sparse Node Security

A sparse node fulfilling the following definition is said *secure*.

Definition 12 (Sparse Node Security). *Let $\mathcal{L}$ be a ledger. Let $\mathcal{X}_s$ be a state predicate for $\mathcal{L}$ and $\mathcal{X}_t$ the transaction predicate derived from $\mathcal{X}_s$. Consider an adversary that controls less than f nodes, with f being the ledger's adversarial resilience. A sparse node protocol $\Pi(\mathcal{X}_s; \mathcal{P}, V)$ is* secure *if, at any round r, V outputs $(\hat{\mathcal{L}}^r, \hat{S}^r)$ such that the following properties hold:*

Safety: *$\hat{\mathcal{L}}^r$ is identified by $\mathcal{X}_t$, $\hat{\mathcal{L}}^r$ is sparsely valid and prefix complete with respect to $\mathcal{L}^r$, and $\hat{S}^r = \hat{\delta}(\hat{S}^{r-1}, \hat{\mathcal{L}}^r \setminus \hat{\mathcal{L}}^{r-1})$.*

Eventual Liveness: *$\hat{\mathcal{L}}^r$ is identified by $\mathcal{X}_t$ and it will eventually be complete with respect to $\mathcal{L}^r$.*

Consider now an adversary that controls more than f nodes. Then, a sparse node protocol $\Pi(\mathcal{X}_s; \mathcal{P}, V)$ is secure *if, at any round r, V outputs $(\hat{\mathcal{L}}^r, \hat{S}^r)$ such that the following property holds:*

Weak Safety: *$\hat{\mathcal{L}}^r$ is identified by $\mathcal{X}_t$ and it is sparsely valid.*

We observe that when the ledger is safe and live, a secure sparse node protocol achieves both safety and eventual liveness. Should the adversary control more nodes than the adversarial resilience of the ledger, any security notion of the ledger is compromised and *any notion of completeness is meaningless*. However, in this case, a sparse node still achieves the weak safety property, i.e., it outputs a sparse ledger that is sparsely valid.

Lemma 3.1 (Fork Consistency). *For any rounds r and $r' \leq r$, any two secure sparse nodes i, j reading from the same fork of a ledger $\mathcal{L}^r$ output $(\hat{\mathcal{L}}_i^r, \hat{S}_i^r)$ and $(\hat{\mathcal{L}}_j^r, \hat{S}_j^r)$ s.t. $\hat{\mathcal{L}}_i^{r'} \preceq \hat{\mathcal{L}}_j^r$ and s.t. $\hat{S}_j^r = \hat{\delta}(\hat{S}_i^{r'}, \hat{\mathcal{L}}_j^r \setminus \hat{\mathcal{L}}_i^{r'})$.*

Proof. Towards contradiction, suppose that there exists a round r such that sparse nodes i, j reading from the same fork of $\mathcal{L}^r$ 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 prefix complete with respect to $\mathcal{L}^r$ or not sparsely valid, contradicting the nodes' security. □

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 [15, 25, 26], and they could be used for sparse nodes as well. We will explore this as future work.

3.3 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* and *prefix completeness* are now properties of the sparse state: for instance, $\hat{S}^r$ is complete w.r.t. $\mathcal{L}^r_{\log}$ if, at any round r, $\nexists (k, v)$ s.t. $\mathcal{X}_s(k) = 1 \wedge v \notin \hat{S}^r \wedge v \in \mathcal{L}^r_{\log}$.

It follows that an event node has weaker security than a sparse node, as there is no notion of sparse validity. 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 6, besides sparse nodes, we evaluate event nodes, showing their efficiency.

3.4 Operating Modes

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

A *header node* is always online and it reads *all block headers* irrespective of whether a relevant transaction is in the block (they are similar to SPV light nodes). 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.

A *continuous* sparse node only receives the (scattered) headers of those blocks that include transactions relevant for them. Such a node is always online so that it can be immediately notified when a relevant transaction gets appended to the ledger. Assuming it connects to at least one honest node, the ledger of a continuous sparse node is complete at all times.[3] This is the *primary operating mode* we consider in this work.

An *intermittent* sparse node alternates between wake and sleep periods, either with some periodicity or at random. Assuming it connects to at least one honest node, the ledger of an intermittent node is prefix complete at all times and complete only when awake.

[3]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.

	Resources
Full Node	$O(\lvert\mathcal{L}\rvert + \lvert S\rvert)$
Sparse Node	$O(\lambda\rho\lvert\mathcal{L}\rvert + \eta\lvert\hat{\mathcal{L}}\rvert + \psi\lvert\hat{S}\rvert)$
Light Node	$O(\lambda\lvert\mathcal{L}\rvert)$

Table 2: Resources consumed by full, sparse, and light nodes.

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.

3.5 Sparse Node Resources

We now discuss the resources consumed by a sparse node in terms of bandwidth, computation, and storage. Table 2 compares the resources consumed by full, sparse, and light nodes.

Similarly to light nodes and full nodes, sparse nodes need to receive regular updates about consensus parameters, e.g., validators in the current committee. This requires expending $O(\lambda|\mathcal{L}|)$ resources where λ captures the rate of committee changes. For most of the existing PoS chains, these changes occur rarely, e.g., once a day, so the overhead is extremely small. For sparse nodes that sync very rarely or only once, committee change updates can be further compressed using Succinct Zero-Knowledge Proofs [1, 12, 34].

Sparse nodes that only download the headers of those blocks that include relevant transactions, might receive only a subset of the blocks of the ledger. The node needs to verify that the received scattered blocks belong to the same (fork of the) chain. This could be trivially done by, e.g., providing the sparse node with the headers of all the blocks in the chain, but more efficient techniques exist under the name of *proofs of ancestry*. These add within block headers a commitment to a data structure, e.g., a Merkle Mountain Range, a vector commitment or a skip list, that allows to navigate the chain backward with a logarithmic or constant overhead in the age of the last block received by the node. Let us consider ancestry proof openings to require $O(\rho|\mathcal{L}|)$ resources.

A sparse node requires bandwidth, computational power, and storage for downloading, validating, and storing transactions as well as storing the sparse state. Assume a constant upper bound on the computation associated with a transaction, these resources are *proportional to the size of* $\hat{S}$, to the *number of transactions* in $\hat{\mathcal{L}}$, but also to the *degree of dependency* these transactions have on some state external to the sparse state [6]. A sparse node that follows a substate might have to download parts of an external substate in order to execute transactions (recall, this external substate is passed to the sparse node in the read set of a transaction). For instance, think of a sparse node tracking a flash loan contract: transactions that touch this contract often interact with multiple other contracts (e.g., DEXs). The resource consumption introduced by the dependencies may vary across different transactions, but also depending on the specific sparse state the node monitors. In some cases, when a sparse state has a high degree of dependency with respect to another substate, a clever choice could be to include the external dependency into the sparse state. To capture the overhead on the sparse node introduced by external dependencies, we use the multiplying factor ψ next to $|\hat{S}|$. To verify transaction

inclusion in a block, if the sparse node reads from ledgers that use 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.

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

We observe that $O(\lambda\rho|\mathcal{L}|+\eta|\hat{\mathcal{L}}|+\psi|\hat{S}|)$ is dominated by different terms: if the sparse state is touched by many, frequent transactions, then $\eta|\hat{\mathcal{L}}| + \psi|\hat{S}|$ dominates. If the sparse state is rarely used, then the impact of $\lambda\rho|\mathcal{L}|$ increases.

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., $O(\eta|\mathcal{L}| + \psi|\hat{S}|)$, 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, among others, efficient reads for clients. From now on, we consider a blockchain that has commitments that the sparse node can use for efficient reads.

4.1 Sunfish Prover-Verifier Protocol

Algorithm Notation. For the algorithms, we assume a BFT-based Proof-of-Stake ledger (no forks), and we use $m \dashrightarrow A$ to indicate that message m is sent to party A and $m \dashleftarrow A$ to indicate that message m is received from party A. In the algorithms, we denote with B a block header, with σ the validators' signatures that assess validity of a block, and with π_i, π_a, and π_c the inclusion, ancestry, and completeness proofs, respectively. We let Σ_s be the set of valid state predicates supported by the ledger. Let C be the local chain of the prover.

Backward Compatible Protocol. We now describe the backward compatible version of our Sunfish protocol. Algorithm 1 presents the pseudocode run by the sparse node for the backward compatible Sunfish, while Algorithm 3 showcases the protocol run by the provers with the exception that line 18 is removed, and the data structure $\mathcal{D}$ is without π_c.

Consider a sparse node V that when wakes up has only knowledge of the sparse state and of the sparse state transition function at genesis (we denote genesis as $\mathcal{G}$). The node connects to a set $\mathcal{P}$ of full nodes (provers), with at least one of them being honest. The node selects a predicate χ_s that is supported by the ledger and it sends it to the provers (Algorithm 1, lines 2-5). Upon receiving χ_s, the provers extract χ_t from χ_s and they send to the sparse node the following (Algorithm 1, line 12):

(1) The block headers necessary to extract the historical consensus parameters of the chain. For BFT protocols, these are, e.g., signed end-of-epoch block headers [5] that include the handover message in which the current validator set appoints the next one;
(2) All transactions such that $\chi_t(\mathsf{tx}) = 1$, along with the headers of the blocks they are included in, and the inclusion proofs;
(3) The data necessary to verify that the block headers in (2), however sporadic they are, belong to the same (fork of the) chain.

Upon receiving this data, the sparse node checks the validity of all received block headers, the correct inclusion of transactions s.t. $\chi_t(\mathsf{tx}) = 1$ in a valid block header, and that the transactions execute correctly according to the sparse state transition function (Algorithm 1, line 13). The sparse node then rejects all the responses from provers that do not satisfy the checks above, and it adds to its local sparse ledger the one that follows the fork choice rule of the underlying chain *and* that contains the largest number of relevant transactions (Algorithm 1, line 18). Finally, it updates it local sparse state (Algorithm 1, line 19). As the ledger makes progress, the node receives from the provers new blocks with relevant transactions; upon receiving these, it updates its sparse ledger and sparse state. We observe that thanks to the existential honesty assumption, this protocol achieves liveness with parameter Δ, with Δ being the network delay; this is stronger than the eventual liveness defined in Definition 12.

Optimized Protocol. Although the protocol we have just described realizes a secure sparse nodes, there are still a few improvements that would allow to design an optimized and verifiably secure sparse node. First, we note that in the design above the sparse node cannot verify completeness: it can only get completeness by relying on the fact that, among the provers, there is an honest node that will not withhold any relevant transaction (existential honesty). Furthermore, even though connecting to a set of full nodes is a standard assumption for light clients, this results in the following: (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 [3]; (iii) finally, as the resource requirements for operating a full node increase proportionally to the 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*. To prevent a malicious validator from withholding transactions and remain undetected, we equip Sunfish with a mechanism to *verify completeness*. For this, we introduce an authenticated data structure that enables completeness checks, and we require validators to add a commitment to such data structure into the block header (similar to what already exists for light clients, that use transaction Merkle roots stored in block headers). Should the validator-prover withhold transactions, the sparse node can now detect the misbehavior and connect to a different prover. We will show that this optimized protocol is *safe and eventually live* when the ledger is safe and live, as the node will eventually hit an honest validator-prover. Algorithm 2 presents the pseudocode run by the sparse node for the optimized Sunfish, while Algorithm 3

Algorithm 1 The algorithm ran by the backward compatible Sunfish client V operating in *continuous mode*.

```
1: function VERIFIER_G(Σ_s, G, Ŝ)
2:     X_s ← Σ_s, Ŝ ← G, L̂ ← [ ], Ŝ ← [ ], boolean ← ⊥, receivedData ← [ ], validData ← [ ]
3:     for P ∈ P do
4:         X_s --> P
5:     end for
6:     while True do
7:         for P ∈ P do
8:             D <-- P                                   ▷ Implicit timeout when receiving.
9:             ReceivedData[P ] = D
10:        end for
11:        for P ∈ P do
12:            (B, σ, txs, π_a, π_i) = ReceivedData[P ]
13:            boolean = IsBValid(B, σ) ∧ IsAncestryValid(B, π_a) ∧ AreTxsIncluded(B, txs, π_i) ∧ AreTxsRelevant(txs, X_s) ∧ AreTxsSpValid(txs, Ŝ)
14:            if boolean then
15:                validData[P] = receivedData[P]
16:            end if
17:        end for
18:        L̂ = AppendRelevantTxs(arg max_{P∈P} size(validData[P].txs))
19:        Ŝ = ComputeState(Ŝ, L̂)
20:    end while
21: end function
```

showcases the protocol run by the provers in the same setting. Note that in Algorithm 2 , line 10, the sparse node now verifies a completeness proof, while in Algorithm 3, line 18, the prover now generates a completeness proof by using the new commitment in the block headers.

4.2 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 $\mathsf{id}\hat{\mathsf{S}}$; in case of the substate of a dApp, e.g., the identifier could be the hash of the application logic.

As shown in Figure 1, Sunfish-C requires validators building a per-block Merkle tree (or Merkle Mountain Range (MMR) [29]) 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 φ along with the height h of the block, yields 0. We require φ to be a deterministic, predictable, and periodic function: e.g., $\varphi(\mathsf{id}\hat{\mathsf{S}}, \mathsf{h}) := (\mathsf{id}\hat{\mathsf{S}} + \mathsf{h})\%N$ 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 $\varphi = 0$. Finally, by reading the counters for two

Algorithm 2 The algorithm ran by the optimized Sunfish client V operating in *continuous mode*.

```
1:  function VERIFIER_G(Σ_s, G, δ̂, t)
2:      X_s ← Σ_s, Ŝ ← G, L̂ ← [ ], bool ← ⊥, timeout ← t
3:      for P ∈ P do
4:          X_s ⇢ P
5:          while True do
6:              if ¬ IsReceived(D ⇠ P, timeout) then
7:                  break
8:              end if
9:              (B, σ, txs, π_a, π_i, π_c) = D
10:             bool = IsBValid(B, σ) ∧ IsAncestryValid(B, π_a) ∧ IsTxSetComplete(txs, B, π_c) ∧ AreTxsIncluded(B, txs, π_i) ∧ AreTxsSpValid(txs, δ̂)
11:             if bool ∧ AreTxsRelevant(txs, X_s) then
12:                 L̂ = AppendRelevantTxs(txs)
13:                 Ŝ = ComputeState(Ŝ, L̂)
14:             else
15:                 break
16:             end if
17:         end while
18:     end for
19: end function
```

Algorithm 3 The algorithm ran by a Sunfish prover P. For the backward compatible protocol, line 18 is removed as well as π_a in the $\mathcal{D}$ structure.

```
1:  function CONSTRUCTOR( )
2:      X_t ← ⊥
3:  end function

4:  function ONBOOTSTRAP( )                          ▷ Upon establishing a new connection.
5:      X_s ⇠ V
6:      X_t = DERIVETXPREDICATE(X_s)
7:  end function

8:  function ONNEWBLOCK(X_t)                         ▷ Upon seeing a new valid block.
9:      txs ← [ ], π_i ← [ ], π_c ← [ ], π_a ← [ ]
10:     for tx ∈ C[−1] do                            ▷ C[−1] is the last block of P's local chain.
11:         if X_t(tx) then
12:             txs.append(tx)
13:             π_i.append(GENINCLPROOF(tx, C [-1].header))
14:         end if
15:     end for
16:     if txs ≠ ∅ then
17:         π_a.append(GENANCESTRYPROOF(C[−1].header))
18:         π_c = GENCOMPLETENESSPROOF(tx, C[−1].header)   ▷ This line is only run in the optimized version of the protocol.
19:     end if
20:     D ← (B, σ, txs, π_a, π_i, π_c)               ▷ π_c is only included in the optimized version of the protocol.
21:     D ⇢ V
22: end function
```

adjacent blocks with $\varphi = 0$, 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 MMRs in a different manner. The data structure used by Sunfish-HC is depicted in Figure 2 and it 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

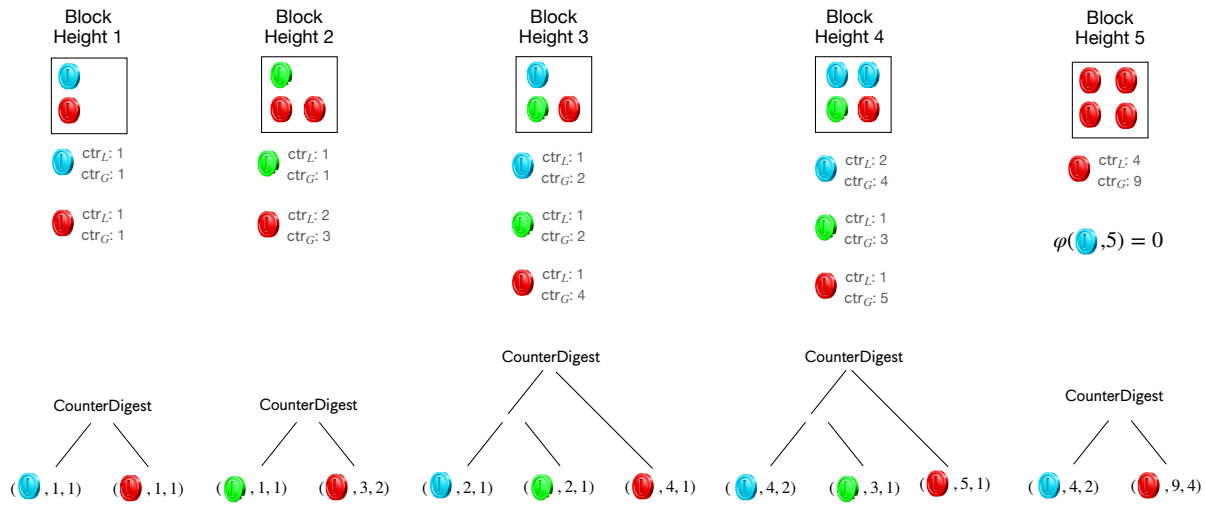

Figure 1: Data structure for Sunfish-C. For each block, validators create an MMR whose leaves are tuples $(\mathsf{id}\hat{\mathsf{S}}, \mathsf{ctr}_G, \mathsf{ctr}_L)$ lexicographically sorted by $\mathsf{id}\hat{\mathsf{S}}$ (in the picture, the $\mathsf{id}\hat{\mathsf{S}}$ is represented by a colored coin). The MMR of a block has one leaf for each $\mathsf{id}\hat{\mathsf{S}}$ with $\mathsf{ctr}_L \neq 0$ and one leaf for each $\mathsf{id}\hat{\mathsf{S}}$ s.t. $\varphi(\mathsf{id}\hat{\mathsf{S}}, \mathsf{h}) = 0$. For instance, notice the block with height 5 on the right: its MMR includes a leaf for the red coin, because 4 transactions relevant for that coin appear in the block; however, it also includes a leaf for the blue coin, because height 5 yields the right periodicity for the blue coin. The counter digests are included in the respective block headers.

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 φ. 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 $O(|\hat{\mathcal{L}}|)$ transactions, reads the counters in $O(|\hat{\mathcal{L}}| \log Q)$, and checks transaction inclusion in $O(|\hat{\mathcal{L}}| \log M)$. The proof size is $O(\eta|\hat{\mathcal{L}}|)$, with $\eta = \log M + \log Q$. In Sunfish-HC, the sparse node receives $O(|\hat{\mathcal{L}}|)$ transactions and verifies the chain head inclusion in $O(\eta|\hat{\mathcal{L}}|)$ 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 $O(1)$ updates). In Sunfish-HC, validators store and update 1 chain head per sparse state (64 bytes with $O(1)$ updates).

4.3 Analysis

We adopt a property-based approach to prove security and efficiency of Sunfish, both in its backward compatible and in its optimized version. In particular, assuming existential honesty, we prove that the backward compatible Sunfish protocol is *safe and live* according to Definition 12 when the underlying ledger is safe and live; instead, when the underlying ledger is *not* safe and live, the protocol still achieves *weak safety* (Definition 12). Then, we show that the same properties hold for the optimized Sunfish protocol by assuming the validator-prover is trusted for liveness.

Theorem 1 (Sunfish Safety). *In the presence of an adversary that controls less than f nodes, Sunfish achieves* safety *as defined in Definition 12.*

Proof. *Backward compatible Sunfish:* Suppose, towards contradiction, that a sparse node running the backward compatible version of Sunfish does not achieve safety. This means that, under the existential honesty assumption, the $(\hat{\mathcal{L}}^r, \hat{S}^r)$ output by the node at any round r is either not sparsely valid, not prefix complete, or not identified by the correct predicate $\mathcal{X}_t$. The sparse node cannot accept a sparsely invalid state as, by design, it is equipped with a sparse state transition function and it sparsely executes transactions (Algorithm 1, lines 2 and 13). For the sparse node to accept a $\hat{\mathcal{L}}^r$ that is not prefix complete, it means that the node has not taken the largest set of valid transactions s.t. $\mathcal{X}_t = 1$ it has received, or that the node has not received the complete set: the first case cannot happen by design (Algorithm 1, line 18), whereas the second case cannot happen because of the existential honesty assumption. Finally, $\hat{\mathcal{L}}^r$ cannot be identified by an incorrect predicate $\mathcal{X}_t$, because the node has knowledge of the predicate and, before accepting any transaction tx, it checks that $\mathcal{X}_t = 1$ (Algorithm 1, lines 2 and 13). This concludes the contradiction.

Optimized Sunfish: Suppose, towards contradiction, that a sparse node running the optimized version of Sunfish does not achieve safety. This means that the $(\hat{\mathcal{L}}^r, \hat{S}^r)$ output by the node at any round r is either not sparsely valid, not prefix complete, or not identified by the correct predicate $\mathcal{X}_t$. The sparse node cannot

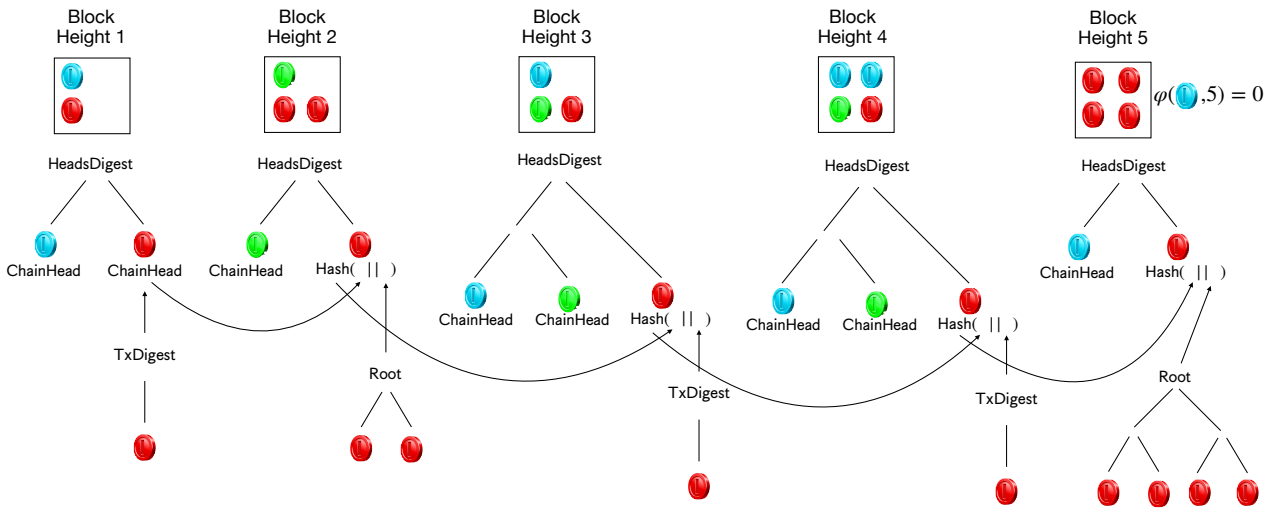

Figure 2: Data structure for Sunfish-HC. Let the blue, green, and red coins represent transactions of the blue, green, and red sparse state, respectively. Let us consider the red sparse state: whenever a new red transaction appears in a block, validators generate a lexicographically sorted MMR with the red transactions in the block. The MMR roots of different blocks are used to generate a hash chain, whose chain head is included in an MMR of head digests. The MMR head digest is then included into the block headers. In the picture we depict the MMR for the red transactions only, but the same occurs for any other sparse state (blue, green). Notably, the block on the left with height 5 also include the chain head of the blue sparse state, since the blue sparse state id$\hat{S}$ and height 5 giving the right periodicity.

accept a sparsely invalid state as, by design, it is equipped with a sparse state transition function and it sparsely executes transactions (Algorithm 2, lines 2 and 10). For the sparse node to accept a $\hat{\mathcal{L}}^r$ that is not prefix complete, it means that the node appended to its local $\hat{\mathcal{L}}$ a set of transactions that is not complete. Given the adversarial threshold ensures the data structures are correctly generated, this is not possible because of the properties of hash chains in Sunfish-HC and because of the inclusion of global counters in Sunfish-C. Finally, $\hat{\mathcal{L}}^r$ cannot be identified by an incorrect predicate $\mathcal{X}_t$, because the node has knowledge of the predicate and, before accepting any transaction tx, it checks that $\mathcal{X}_t = 1$ (Algorithm 2, lines 2 and 11). This concludes the contradiction. □

Theorem 2 (Sunfish Eventual Liveness). *In the presence of an adversary that controls less than f nodes, Sunfish achieves* eventual liveness *as defined in Definition 12.*

Proof. *Backward compatible Sunfish:* The backward compatible version of Sunfish achieves eventual liveness because of the existential honesty and synchronous network assumptions: the sparse node connects to at least one honest node, who is live and responsive whose message is received within a Δ delay.

Optimized Sunfish: The optimized version of Sunfish achieves eventual liveness because of the assumption the adversary controls less than f nodes; this means, should the node receive no answer from the connected validator within a timeout (Algorithm 2, line 6) it connects to a different validator until it will connect to an honest one. □

Theorem 3 (Sunfish Weak Safety). *In the presence of an adversary that controls more than f nodes, Sunfish achieves* weak safety, *as defined in Definition 12.*

Proof. *Backward compatible Sunfish:* The backward compatible version of Sunfish achieves weak safety in the presence of an adversary with more than f node because it will never output a sparse ledger that includes transactions that are sparsely invalid (Algorithm 1, line 13).

Optimized Sunfish: The optimized version of Sunfish achieves weak safety in the presence of an adversary with more than f node because it will never output a sparse ledger that includes transactions that are sparsely invalid (Algorithm 2, line 10). □

Theorem 4 (Sunfish Security). *Sunfish is secure according to Definition 12.*

Proof. This trivially follows from Theorem 1, Theorem 2 and Theorem 3. □

Theorem 5 (Sunfish Resources). *The bandwidth, computational, and storage resources consumed by Sunfish are $O(\lambda\rho|\mathcal{L}| + \eta|\hat{\mathcal{L}}| + \psi|\hat{S}|)$.*

Proof. *Backward compatible Sunfish:* The resource analysis for the backward compatible protocol is similar to the one of the optimized version (see below): the only difference is that instead of downloading and verifying completeness proofs, this protocol downloads and verifies an amount of data proportional to the number of provers. This is the case for any client (most light clients) that relies on the existential honesty assumption.

Optimized Sunfish: Bandwidth: Downloading ancestry proofs requires $O(|\mathcal{L}| \log N)$ in the case of MMR-based ancestry proofs, so $\rho = \log N$. Downloading transactions requires $O(|\hat{\mathcal{L}}|)$. Downloading inclusion proofs requires $O(|\hat{\mathcal{L}}| \log M)$ with M being the average number of transactions in a block. Downloading completeness proofs requires $O(|\hat{\mathcal{L}}| \log Q)$, with Q being the average number of updated sparse states in a block. Therefore, for Sunfish-C we have $O(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log M + \log Q$, while for Sunfish-HC we have $O(\eta|\hat{\mathcal{L}}|)$ 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 $O(|\hat{\mathcal{L}}| \log M)$ and $O(|\hat{\mathcal{L}}| \log Q)$, respectively. The computational complexity of execution is $O(|\hat{\mathcal{L}}|)$, being β a constant. Therefore, this makes for a total computation of $O(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log M + \log Q$ for Sunfish-C and $O(\eta|\hat{\mathcal{L}}|)$ with $\eta = \log Q$ for Sunfish-HC. *Storage*: the sparse node stores $\hat{\mathcal{L}}$ as well as $\hat{S}$, yielding $O(|\hat{\mathcal{L}}| + |\hat{S}|)$ storage complexity.

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

□

5 APPLICATIONS AND DISCUSSION

Sparse Node Applications. 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* [33] and *remote staking* projects [19], *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 [26]. 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.

Predicate Composability. To express more complex sparse states and sparse ledgers, one might want to use Sunfish to compose predicates using logical operators. Combining two or more predicates with the OR ($\vee$) logical operator is supported by default by both Sunfish-C and Sunfish-HC, as the corresponding $\hat{S}$ and $\hat{\mathcal{L}}$ result from checking each predicate individually and then merging the obtained states and ledgers together. Others logical operators such as AND ($\wedge$), XOR ($\oplus$), NAND ($\uparrow$), and NOR ($\downarrow$) are not supported. For instance, let us consider $\mathcal{X}_t(\mathsf{tx})_1 \wedge \mathcal{X}_t(\mathsf{tx})_2$: evaluating $\mathcal{X}_t(\mathsf{tx})_1$ and $\mathcal{X}_t(\mathsf{tx})_2$ individually and merging the resulting states and ledgers, yields a much larger set $\hat{S}$ and a much longer $\hat{\mathcal{L}}$ than the desired ones. This is because they would not only include state elements and transactions resulting from both predicates being $\top$ at the same time, but also the ones resulting from a single predicate being $\top$. States and ledgers resulting from applying these operators, can be obtained only if the ledger natively supports them as valid predicates in Σ_s.

Finally, we highlight that a sparse node can simultaneously operate over different ledgers, each one with its own set of valid predicates, e.g., Σ_{s1} for $\mathcal{L}_1$ and Σ_{s2} for $\mathcal{L}_2$. For instance, a client can be a sparse node for the contracts of a bridge deployed on two different ledgers. We further explore predicate composability in future work.

Completeness as a Service. As described in Section 4, the augmented Sunfish achieves verifiable completeness by introducing extra workload for validators. Indeed, they have to include an extra commitment to block headers. While this overhead is similar to what is in place for light clients (block headers include a transaction Merkle root), the completeness data structure can be introduced in a more nuanced way. For instance, developers that want to run a sparse node to monitor the state of their dApp, could pay a small fee to validators, incentivizing them to include their sparse state of interest into the commitment. With an ad-hoc business model, validators could offer a *subscription-based service*: this would optimize and better identify the sparse states to include in the data structure, rather than including by default all the possible ones supported by the ledger. We leave as future work the design of such an incentive mechanism.

6 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 [10] 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* [11] 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 [14] and the Wave wallet [28]. We consider sparse states identified by different predicates:

package-based, event-based, and address-based. We consider: (i) the Wormhole bridge, via package: $\mathcal{X}_t(\text{tx}) = 1$ if tx touches a Wormhole package. (ii) The Wormhole bridge, via events: $\mathcal{X}_t(\text{tx}) = 1$ if tx emits Wormhole events. Here, the sparse node only receives events, not transactions. (iii) The Wave wallet, via address: $\mathcal{X}_t(\text{tx}) = 1$ 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. In this analysis we omit the term $\lambda\rho|\mathcal{L}|$, as we consider popular applications for which its weight is very small compared to $\eta|\hat{\mathcal{L}}| + \psi|\hat{S}|$. 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, $C = \min(R, 4.12)$); (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	$\lvert\pi_c\rvert$	$\lvert\pi_{hc}\rvert$
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).

7 RELATED WORK

Sunfish positions itself as a middle ground between full nodes [4, 18, 31] and light nodes [1, 5, 8, 13, 17, 20, 32, 34]. Unlike full nodes, Sunfish does not require to download and re-execute a complete copy of the ledger: it only downloads and re-executes a subset thereof. Sunfish ensures notions of validity, completeness, and consistency that light clients do not provide because of their minimalist design and the lack of transaction re-execution. Some light client designs [24, 35] consider completeness as an important property, however, they achieve it by relying on trusted execution environments [30] and do not consider re-execution. Other light clients [7] have been designed to help securing the chain: these include data availability and validity checks, and require to deploy multiple client instances. In this way, each client verifies a small *random* subset of the chain and, all together, they ensure validity of the whole chain. A sparse node is different from [7] in scope and functioning: it operates stand-alone by reading the chain and consistently verifying a specific, well-defined subset of the chain.

Light clients that help securing the chain have become popular in the context of lazy ledgers [6]. Lazy ledgers decouple consensus from transaction verification and execution to increase throughput. Validity of these chains is defined at the client level, and nodes that need to validate a specific application do not need to validate transactions pertaining to external applications. In this sense, a sparse node and an application-specific client of a lazy ledger share some similarities. However, application-specific clients of lazy ledgers need to download the *entire* dirty ledger. These clients get completeness by forgoing communication efficiency. In our case, completeness is much harder to get because we limit the amount of data that is downloaded by the sparse node. Moreover, for the first time, we formalize and generalize the idea of a sparse node, sparse ledger, and sparse state, defining the security and efficiency properties, even under an adversarial majority.

Finally, sharding [9] is a scalability solution in which the consensus nodes of a system are divided to work in groups, with each group running the consensus of a shard, i.e., one of many parallel blockchains. In a sharding protocol, a subset of nodes run the consensus of the shard over a *subset* of the transactions and a *subset* of the state of the entire system. A sparse node is different from a node of a shard: a sparse node does not participate in the consensus protocol. A blockchain that enables sparse reads does not need to shard its state, its transactions, or its consensus nodes.

ACKNOWLEDGMENTS

This work was supported by Mysten Labs and conducted during Giulia Scaffino's internship with the company. We thank George Danezis, Zeta Avarikioti, and Dionysis Zindros for fruitful discussions and feedback. We thank the Mysten Labs Data Science team for providing necessary data to conduct our evaluation. The support by the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT) is gratefully acknowledged.

REFERENCES

[1] 2023. Mina Docs. (2023). https://docs.minaprotocol.com/about-mina.
[2] 2024. Ethereum Yellowpaper. (2024). https://ethereum.github.io/yellowpaper/paper.pdf
[3] 2024. Nodewatch. (2024). https://nodewatch.io/
[4] 2024. Sui Full Node Transaction Signatures are not verified. (2024). https://github.com/MystenLabs/sui/blob/7bc276d534c6c758ac2cfefe96431c2b1318ca01/crates/sui-sdk/src/apis.rs#L1128
[5] Shresth Agrawal, Joachim Neu, Ertem Nusret Tas, and Dionysis Zindros. 2023. Proofs of Proof-Of-Stake with Sublinear Complexity. In *5th Conference on Advances in Financial Technologies (AFT 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2023.14
[6] Mustafa Al-Bassam. 2019. LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts. (2019). arXiv:cs.CR/1905.09274 https://arxiv.org/abs/1905.09274

[7] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. 2021. Fraud and Data Availability Proofs: Detecting Invalid Blocks in Light Clients. In *Financial Cryptography and Data Security*, Nikita Borisov and Claudia Diaz (Eds.). Springer Berlin Heidelberg.

[8] Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. 2024. Blink: An Optimal Proof of Proof-of-Work. Financial Cryptography and Data security 2025. (2024). https://eprint.iacr.org/2024/692

[9] Zeta Avarikioti, Antoine Desjardins, Lefteris Kokoris-Kogias, and Roger Wattenhofer. 2023. Divide & Scale: Formalization and Roadmap to Robust Sharding. In *Structural Information and Communication Complexity*. Springer Nature Switzerland.

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

[11] 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. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. (2024). arXiv:2310.18042 https://arxiv.org/abs/2310.18042

[12] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Coda: Decentralized Cryptocurrency at Scale. (2020). https://eprint.iacr.org/2020/352.pdf.

[13] Sean Braithwaite, Ethan Buchman, Ismail Khoffi, Igor Konnov, Zarko Milosevic, Romain Ruetschi, and Josef Widder. 2020. A tendermint light client. *arXiv preprint arXiv:2010.07031* (2020).

[14] Wormhole Bridge. 2024. (2024). https://docs.sui.io/concepts/tokenomics/sui-bridging.

[15] Christian Cachin, Abhi Shelat, and Alexander Shraer. 2007. Efficient fork-linearizable access to untrusted shared memory *(PODC '07)*. Association for Computing Machinery. https://doi.org/10.1145/1281100.1281121

[16] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. (2002). https://doi.org/10.1145/571637.571640

[17] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. 2022. Sok: Blockchain light clients. In *International Conference on Financial Cryptography and Data Security*. Springer, 615–641.

[18] Anamika Chauhan, Om Prakash Malviya, Madhav Verma, and Tejinder Singh Mor. 2018. Blockchain and scalability. In *2018 IEEE international conference on software quality, reliability and security companion (QRS-C)*. IEEE, 122–128.

[19] Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. 2024. Remote Staking with Economic Safety. (2024). https://arxiv.org/abs/2408.01896

[20] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. 2020. Non-interactive Proofs of Proof-of-Work. In *Financial Cryptography and Data Security*, Joseph Bonneau and Nadia Heninger (Eds.). Springer International Publishing.

[21] Leslie Lamport. 1978. The Implementation of Reliable Distributed Multiprocess Systems. (1978). https://doi.org/10.1016/0376-5075(78)90045-4

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* (1982). https://doi.org/10.1145/357172.357176

[23] Jinyuan Li and David Maziéres. 2007. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. USENIX Association.

[24] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin lightweight client privacy using trusted execution. In *28th USENIX Security Symposium (USENIX Security 19)*. 783–800.

[25] David Mazières and Dennis Shasha. 2002. Building secure file systems out of byzantine storage. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/571825.571840

[26] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 383–398. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[27] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. (2009). http://bitcoin.org/bitcoin.pdf.

[28] Wave Wallet on Sui. (????). https://waveonsui.com/.

[29] Merkle Mountain Ranges. 2024. (2024). https://docs.grin.mw/wiki/chain-state/merkle-mountain-range/.

[30] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. 2022. Sok: Hardware-supported trusted execution environments. *arXiv preprint arXiv:2205.12742* (2022).

[31] Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. 2023. Executing and Proving Over Dirty Ledgers. In *Financial Cryptography and Data Security 2023*.

[32] Ertem Nusret Tas, David Tse, Lei Yang, and Dionysis Zindros. 2024. Light Clients for Lazy Blockchains. In *Financial Cryptography and Data Security 2024 (FC24)*.

[33] EigenLayer Team. 2024. EigenLayer: The Restaking Collective. (2024). https://shorturl.at/sl9tE

[34] Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. 2023. Plumo: An Ultralight Blockchain Client. (2023). https://celo.org/papers/plumo.

[35] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. 2019. 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*. Springer, 179–198.