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. Currently, dApp developers run full nodes, which download and re-execute every transaction to track the global state of the chain. This becomes prohibitively expensive for high-throughput chains due to increased bandwidth, computational, and storage requirements. Alternatively, light nodes only verify the inclusion of a set of transactions and have no guarantees on whether the set includes *all* relevant transactions. In extreme cases with dishonest majority, light nodes will also forfeit safety and accept 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 confirm their validity. A sparse node requires resources roughly proportional to the number of transactions in the substate and to the size of the substate itself, retaining a meaningful validity notion even under adversarial majorities. Further, we present Sunfish, a secure sparse-node protocol compatible with most existing blockchains. Our analysis and evaluation show that Sunfish reduces the computational and storage resources by several orders of magnitude when compared to a full node.

1 Introduction

Modern blockchains, such as Sui, Aptos, and Solana, scale up to thousands of transactions per second, with Ethereum targeting a comparable throughput in its roadmap. This increased capacity improves the user experience and allows for onboarding millions of new users and dApps. However, 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. This is no longer possible, as their bandwidth, computational, and storage costs are 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 dangerous, as it fully relies on the honesty of the full node operator and negates the trust benefits of using a decentralized blockchain.

An alternative approach to operating full nodes is to run light nodes [1–4], such as clients using the Bitcoin Simplified Payment Verification protocol [5]. Unfortunately, light nodes are insufficient to verifiably track the state of an application: they only verify the transaction inclusion and, without additional trust assumptions, they have no guarantees over whether the set of transactions received is complete, e.g., if it includes *all* transactions 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: by not re-executing transactions, 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, for the first time, a new type of blockchain node that sits between light and full nodes: the *sparse node*. 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 account. A seemingly similar type of client is mentioned in an *independent* post by Vitalik Buterin [6] under the name of *partially-stateless client*. While the post does not provide a formalization, nor a design, it highlights interest in this direction and points toward the relevance of such clients. We define sparse nodes formally through a *predicate*, which, when applied to the global state of the chain, identifies a subset thereof called the *sparse state*. We introduce a filtering function φ which, given a predicate and a ledger, outputs an order-preserving subsequence of the ledger—the *sparse ledger*—consisting of *all* transactions in the global ledger that read from or write to the sparse state associated with the predicate. Finally, we define the security of a sparse node protocol in terms of safety and liveness of its sparse output ledger. By re-executing the sparse ledger, the sparse node ensures that its local sparse state is always valid, even when the ledger is compromised by adversarial majorities.

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 relevant to their dApp, thus increasing the robustness of applications, especially for 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 [7, 8], 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. Therefore, the advantages of using sparse nodes equally benefit high- and low-throughput blockchains.

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 (Section 3). Then, we present Sunfish (Section 4), the first secure protocol for sparse nodes. We describe two instances of Sunfish that offer different trade-offs: Sunfish-C uses counters and minimizes validator overhead, Sunfish-HC uses trees and hash chains and minimizes the overhead on the client side (small proofs). *The Sunfish design can be adopted out-of-the-box by validators and clients of most existing blockchains, as it is independent from any specific transaction model or scripting capabilities.* Afterward, we showcase the required resources for both Sunfish-C and Sunfish-HC (Section 7) 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 dapps, 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 full and light nodes, and also with orthogonal concepts such as clients for lazy ledgers and sharding (Section 8).

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 [9–11]. 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 output a consistent state, even in the presence of a subset of adversarial nodes. On input a transaction tx from the environment, correct nodes move 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. 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 Safety) *For every two honest nodes i, j and for every two times t, t', if $\mathcal{L}_i^t$ is the log output by i at t and $\mathcal{L}_j^{t'}$ is the log output by j at t', then the two logs are consistent, i.e., $\mathcal{L}_i^t \preceq \mathcal{L}_j^{t'}$, or vice versa.*

Definition 2 (Ledger Liveness) *For every t_0 and every transaction* tx*, if every honest node i receives* tx *by time t_0, then there exists a $t_1 \geq t_0$ such that for every $t_2 \geq t_1$ and for every honest node i,* $\mathsf{tx} \in \mathcal{L}_i^{t_2}$.

Let K and V be sets of valid keys and valid values, respectively. Without loss of generality, we model the *state of a node as a key-value store*, i.e., a collection of (k, v), with $k \in K$ and $v \in V$. In particular, k is a unique identifier (e.g., address of an account or contract, or the hash of a UTXO) used to reference a specific value, whereas v is the data (e.g., account balance, contract state, or the UTXO itself) 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 [12], 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.[4] For efficiency of Sunfish (and not for its security), we assume the ledger includes intermediate state commitments as in [13], or transactions include commitments to the read set as in Bitcoin and Solana [14]. In Ethereum, block-level access lists are under discussion in the EIP-7928 [15] and may be included in the future: this would help verify transaction read sets.

Client Model. A client protocol is a protocol between a client, acting as verifier V, and a non-empty set $\mathcal{P}$ of prover nodes. We assume V is honest, i.e., it adheres to the correct protocol execution, and that it connects to at least one honest, non-eclipsed prover.

Cryptographic Assumptions. We assume collision resistant and preimage resistant hash functions.

Network Assumptions. We consider protocols whose execution proceed in discrete rounds $r = \{0, 1, 2, \dots\}$. We assume the communication between the client and the provers is synchronous, i.e., a message sent by one honest node at the end of round r is received by all honest nodes within $r + \Delta$, with Δ being a known upper bound on the network delay. We observe that synchronous communication is already needed for client security (identify the correct chain) on blockchains such as Bitcoin and Ethereum. In these cases, our work does not introduce any strict network assumption for the security of the client. In blockchains that run partially synchronous consensus protocols, our work naturally extends to the partially synchronous model.

[4] 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 read and write sets may have different keys or different values.

3 Sparse Client: Formalization and Security

Our goal is to define a node that only downloads and validates a specific subset of transactions of the ledger and, through re-execution of these transactions, maintains a partial state of the ledger. To this end, we must first formalize substates and subsequences of the ledger by introducing predicates and filtering operations. Consider a ledger $\mathcal{L}$. The state S of the ledger is a key value store (k, v) s.t. each key is associated with exactly one value: $\forall(k, v) \in S : (k, v') \in S \rightarrow v = v'$. We let $\mathcal{X}_s(k)$ be a *state predicate*, i.e., a function $K \rightarrow \{1, 0\}$ that identifies a subset $\hat{S} \subseteq S$ of state elements we call *sparse state.*

Definition 3 (Sparse State $\hat{S}$) $\hat{S} := \{(k, v) | (k, v) \in S \wedge \mathcal{X}_s(k)\}$ *is the sparse state identified by* $\mathcal{X}_s$.

Although, in principle, any sparse state can be defined by specifying an appropriate state predicate — leading to a potentially unbounded number of such substates — in this work we restrict our attention to practically meaningful cases, such as the substate of a particular dApp or the state of a selected set of accounts (see Section 5). Since the state of the ledger changes any time a new transaction is appended to it, at every new append $\mathcal{X}_s$ must be evaluated on all updated and added elements.

Let X be the set of all possible state predicates, and L be the set of all possible ledgers. We now define a filtering function $\varphi : X \times L \rightarrow L$ that, on input a state predicate $\mathcal{X}_s \in X$ and a ledger $\mathcal{L} \in L$, it returns a *sparse ledger* $\hat{\mathcal{L}}$. The sparse ledger $\hat{\mathcal{L}}$ is the order-preserving subsequence of transactions from $\mathcal{L}$ whose read or write sets include state elements that satisfy the predicate $\mathcal{X}_s$.

Definition 4 (Sparse Ledger $\hat{\mathcal{L}}$) *Let $\mathcal{L}$ be a ledger and $\mathcal{X}_s$ a state predicate. Then,* $\hat{\mathcal{L}} = \varphi(\mathcal{X}_s, \mathcal{L}) = [\mathsf{tx} | \mathsf{tx} \in \mathcal{L} \wedge (\exists k \in \mathcal{R}(\mathsf{tx}) : \mathcal{X}_s(k) \vee \exists k \in \mathcal{W}(\mathsf{tx}) : \mathcal{X}_s(k))]$ *is a sparse ledger.*

We now define the sparse client protocol interface and its security.

Definition 5 (Sparse Client Protocol Interface) *A sparse client protocol is an interactive protocol between a client and a set of provers. On input a state predicate $\mathcal{X}_s$ at the client, the protocol outputs a sparse ledger $\hat{\mathcal{L}}$ at the client.*

We now define what it means for a sparse client of a ledger $\mathcal{L}$ to be *secure.* Let $\mathcal{L}^r_{\bigcap}$ be the intersection of the view of all honest parties at the end of round r, whereas let $\mathcal{L}^r_{\bigcup}$ be the blocktree of the union of the ledgers of all honest nodes – we recall, honest nodes have different views of the ledger because of network delays. Informally, a sparse client protocol is *safe* if, for every round r, it outputs a sparse ledger $\hat{\mathcal{L}}^r$ that is a prefix of the ledgers in all honest nodes's views filtered by the predicate $\mathcal{X}_s$, and is *live* if it outputs a sparse ledger $\hat{\mathcal{L}}^r$ that extends the shortest among the honest nodes' stable ledgers filtered by the predicate $\mathcal{X}_s$.

Definition 6 (Sparse Client Protocol Security) *Let κ and v be protocol security parameters. A sparse client protocol parameterized by $\mathcal{X}_s$ is* secure *in a synchronous network iff, in every synchronous execution with at least one honest, non-eclipsed prover, except with probability* negl(κ)*, the client outputs a sparse ledger $\hat{\mathcal{L}}$ such that, for every round r and $r' \geq r + v$, the following properties hold:*

Safety: *$\hat{\mathcal{L}}^r$ is a prefix of $\varphi(\mathcal{X}_s, \mathcal{L}_{\bigcup}^{r'})$.*

Liveness: *$\varphi(\mathcal{X}_s, \mathcal{L}_{\bigcap}^{r})$ is a prefix of $\hat{\mathcal{L}}^r$.*

As we will see in Section 6, the safety parameter v is the liveness parameter of the ledger, i.e., the time it takes for transactions to be confirmed in the ledger. We note that a full node is a sparse node for which, at any round r, $\hat{S}^r = S^r$ and $\hat{\mathcal{L}}^r = \mathcal{L}^r$. To briefly summarize, so far, we have formalized a sparse state as a substate of the global state of the system and a sparse ledger as a subsequence of the global ledger that preserves the relative order of transactions. We have also defined a sparse client protocol interface and stated the security of a sparse client protocol, specifying the safety and liveness properties that the sparse ledger output by the client satisfies. We now characterize the execution semantics of the client. As seen in Section 2, the transition function δ of the ledger takes as inputs the global state S and a transaction tx: $S^{i+1} = \delta(S^i, \mathsf{tx})$. Therefore, the sequence of transactions uniquely determines the global state. Unfortunately, we cannot let a sparse client use the same execution semantics: If we allow δ to take as input a sparse state and a transaction, i.e., $\delta(\hat{S}^i, \mathsf{tx})$, execution may fail because the transaction could read from state elements (e.g., gas objects, contracts, etc) that lie outside the sparse state. In other words, a sparse ledger does not uniquely determine the sparse state: to uniquely define the sparse state one needs the sparse ledger as well as the read sets of all transactions in it. Therefore, we define a *new state transition function* $\hat{\delta}$ that takes as input the sparse state together with a transaction and its read set: $\hat{S}^{i+1} = \hat{\delta}(\hat{S}^i \cup \mathcal{R}(\mathsf{tx}), \mathsf{tx})$. Formally, we define $\hat{\delta}$ as both a *domain restriction* and a *range restriction* of δ: Given as input a *subset* of the global state and a transaction, $\hat{\delta}$ updates the state elements in exactly the same manner as δ, but returns only the *subset* of output elements satisfying $\mathcal{X}_s(k) = 1$. If an element was deleted, $\hat{\delta}$ outputs $(k, \perp)$. We recall that, in the standard setting, ledger validity ensures that all transactions are valid with respect to the global state. For sparse ledgers, *validity* instead requires that each transaction is valid with respect to the client's sparse state. We discuss which notion of validity one can achieve with sparse nodes and how we do it in Sunfish in Section 4 and Section 6.

It is important to note that while the ledger is produced by the consensus protocol and, thus, is available and verifiable by clients, the read set of a transaction is not directly accessible, nor often easily verifiable. For instance, Ethereum has 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, values of contract variables) are not specified nor committed in the transaction itself. Ethereum blocks include a commitment to the state tree, providing a per-block validity check of transaction read sets. However, this only suffices to verify the read set of the first transaction in the block, not those of subsequent transactions. On the other hand, Bitcoin transactions commit to the hash of the transaction containing the read element, yielding a somewhat convoluted yet effective per-transaction validity check of the read set. In this work, we assume the correctness of a transaction read set can be easily verified: in EVM-based chains, this is achieved by adding intermediate state roots within block headers [13]. Alternatively, transactions could include a commitment to the values of their read set, as in Bitcoin.

4 Sunfish: A Protocol for Sparse Nodes

Before introducing Sunfish, we first show how block headers can be augmented with commitments to multiple sparse ledgers, allowing sparse clients to verify the correctness and completeness of the sparse ledger received. A block is valid only if its commitment is valid. This simple extension makes *Sunfish out-of-the-box compatible with most existing chains*, such as Bitcoin, Ethereum, Sui, Solana, Aptos, and the EVM-based chains. We discuss which sparse states validators could include in the commitment in Section 5. Below, we present two authenticated data structures, Sunfish-C and Sunfish-HC, each with different trade-offs: Sunfish-C minimizes validator overhead, while Sunfish-HC reduces the cost for sparse nodes. Due to space constraints, trade-offs are discussed in Appendix B. The different operating modes for a sparse client are discussed in Appendix C, e.g, *header mode*, *continuous mode*, *intermittent mode*, and *on-demand mode*: the data structures below enable them all.

4.1 Committing to Sparse Ledgers: Sunfish-C

One way to verify that a sparse ledger update includes *all* transactions newly appended to $\varphi(\hat{S}, \mathcal{L})$ is to include, in each block, a commitment to the number of transactions to be appended to the sparse ledger, combined with the transactions inclusion proofs. Since transaction inclusion proofs are already widely used, we focus our discussion on the counter mechanism. We will refer to the combination of counter and inclusion proofs as *commitment*, even if this is a slight abuse of terminology, as it is not a cryptographic commitment.

Naive Attempts. 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 touching the sparse state is added to the ledger. One option would be for validators to build a Merkle tree with all counters ctr_G and include it in every block header; unfortunately, this comes with the impractical cost of having validators maintain a massive tree and update 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 touching the sparse state is added to the block.

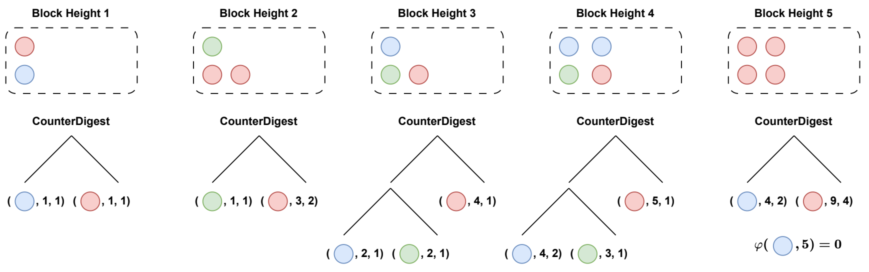

Fig. 1: Sunfish-C. In each block, validators build a Merkle tree with leaves $(\mathsf{id}\hat{\mathsf{S}}, \mathsf{ctr}_G, \mathsf{ctr}_L)$, sorted lexicographically by $\mathsf{id}\hat{\mathsf{S}}$ (colored coin), and include the digest in the block header. The tree includes one leaf for every $\mathsf{id}\hat{\mathsf{S}}$ with $\mathsf{ctr}_L \neq 0$ and for every $\mathsf{id}\hat{\mathsf{S}}$ satisfying $\xi(\mathsf{id}\hat{\mathsf{S}}, \mathsf{h}) = 0$. E.g., for block height 5 the tree contains a leaf for the red transactions and a leaf the blue coin's periodicity.

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.

Sunfish-C. 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 (see Appendix C). 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 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 ξ along with the height h of the block, yields 0. We require ξ to be a deterministic, predictable, and periodic function: e.g., $\xi(\mathsf{id}\hat{\mathsf{S}}, \mathsf{h}) := (\mathsf{id}\hat{\mathsf{S}} + \mathsf{h})\%N$ 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 $\xi = 0$. Finally, by reading the counters for two adjacent blocks with $\xi = 0$, sparse nodes can read chunks of the chain with constant cost.

4.2 Committing to Sparse Ledgers: Sunfish-HC

With the Sunfish-C data structure, operating a sparse node assumes a light client protocol exists for the blockchain. In fact, besides verifying the correct number of transactions have been received, the sparse node would also need to (efficiently) check that the received transactions are the ones included on-chain. In many chains, this can be achieved by verifying transaction inclusion with a Merkle proof; however, some high-throughput chains, such as Sui, do not include in block headers an efficiently verifiable commitment to prove transaction inclusion. Therefore, we propose adding to block header an alternative commitment to a data structure that allows to verify at the same time completeness and inclusion of transactions–this time, a cryptographic commitment in the usual sense of the term. Sunfish-HC asks 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. Towards efficiency and parallelizability, we combine hash chains and Merkle trees as depicted in Figure 2.

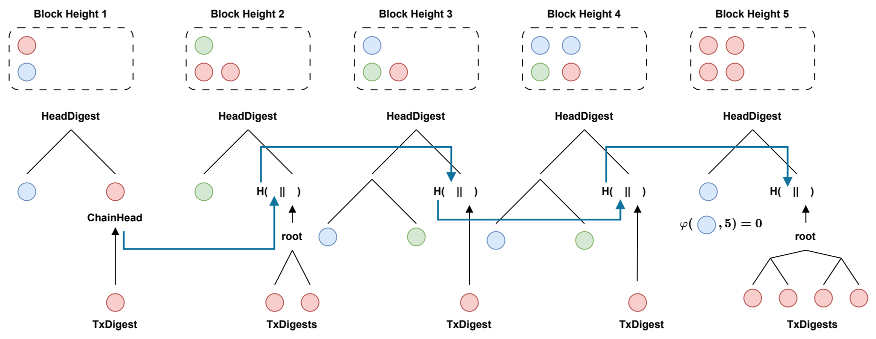

Fig. 2: Sunfish-HC. Blue, green, and red coins denote transactions of different sparse states. When a red transaction appears in a block, validators build a lexicographically ordered Merkle tree of all red transactions in that block. The roots across blocks form a hash chain, whose head is added to a Merkle tree of digests, and this digest is included in the block header. The same applies to other sparse states. Notably, the block at height 5 also includes the chain head for the blue sparse state, as required by its periodicity.

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 ξ. Finally, validators include the Merkle root of the overlay tree in the block header.

4.3 Sunfish: A Sparse Client Protocol

We now present the Sunfish protocol. In Appendix A.1, we showcase the algorithms run by the verifier (Algorithm 1) and the prover (Algorithm 2).

Consider a sparse client that wishes to monitor the state of a DeFi contract on Ethereum. We use Ethereum as an example because it hosts the largest number of dApps, though a similar logic applies to other chains such as Bitcoin, Sui, or Solana. Recalling Section 3 and accounting for Ethereum's account-based model, we represent a contract by using its address as the key of the state element and its storage as the value. Let $\mathcal{X}_s$ be the state predicate that outputs 1 on input the contract address. We also recall that the client is interested in all transactions that read from or write to this contract, from its deployment to the present. Crucially, the client must be protected against an adversary that attempts to inject invalid transactions into its sparse ledger or censor valid ones, fooling it into computing an incorrect contract state. The Sunfish client connects to a set of full nodes acting as provers, at least one of which must be honest and non-eclipsed, and sends $\mathcal{X}_s$ to all provers. Each prover then returns:

- **Tip of the chain:** Block headers identifying the current tip of the chain. The way this is achieved is out of the scope of this work, as it can be inherited verbatim from existing light client protocols. We refer to Appendix A.1 for more details.
- **Relevant transactions:** All transactions in $\hat{\mathcal{L}} = \varphi(\mathcal{X}_s, \mathcal{L})$, together with: their read sets and their proof of correctness (recall, we assume blocks include intermediate state roots [13] or we assume transactions include a commitment to the read set as in Bitcoin), the headers of the blocks they are included in, and if Sunfish-C is used, also their inclusion proof.
- **Sparse ledger commitment opening:** The opening of the Sunfish-C or Sunfish-HC commitment to the node's sparse ledger. This commitment verifies completeness (Sunfish-C, Sunfish-HC) and integrity (Sunfish-HC) of the transaction set.
- **Ancestry proofs:** Data to ensure that block headers containing relevant transactions belong to the same (fork of the) chain. This is particularly important for the Sunfish client in the bootstrap phase where, upon identifying

the tip of the chain, it starts receiving older, non-sequential blocks that include relevant transactions.Ancestry proofs [16] are enabled by using Merkle Mountain Ranges, vector commitments, or skip lists –alternatively, all block headers can be sent, as in SPV clients. The prover provides the required commitments and openings. We refer to Appendix A.1 for more details.

Upon receiving this data, the client reconstructs the validator set history, verifies all proofs, and, if successful, computes the contract's current state by locally applying $\hat{\delta}$ to the sparse ledger transactions. Blocks containing transactions invalid with respect to the client's sparse state are rejected. As the chain grows, provers continue supplying new relevant transactions and associated proofs. After verification, the client extends its sparse ledger and updates its sparse state.

5 Discussion

Optimizations. Instead of connecting to multiple full nodes, a sparse client could connect to a single validator, only trusted for liveness. Should the validator not respond or provide incorrect data, the sparse node can detect the misbehavior and connect to a different one. This avoids that the resource consumption of the node grows with the number of provers. It also allows sparse clients to synchronize much faster, not bottlenecked by the sync time of full nodes.

Sparse State Policy. As discussed in Section 4, Sunfish requires validators to add an extra commitment to block headers. In principle, validators could support any sparse state, including logical operations over multiple sparse states, but the space of admissible sparse states could grow uncontrollably, overloading validators and slowing down the chain. To mitigate this, validators can adopt a *sparse state policy* that limits how fine-grained the supported sparse states may be. For example, the service could be offered only by subscription, or restricted to high-traffic dApps. We are currently exploring protocols that shift this overhead away from validators and move the commitment to the execution layer. These approaches, however, require a quasi–Turing-complete scripting language and impose additional gas costs on users. We highlight that requiring validators to perform additional tasks is not without precedent. In recent years, validator nodes have evolved into increasingly resource-intensive machines (Solana, Sui) and have assumed responsibilities beyond block proposal and voting. Examples include serving as storage nodes for data availability, participating in randomness generation or participating as custodians for bridging through re-staking.

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. 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 (e.g., popular contracts, or geographic locations with high network load) to take load off of full nodes, and communicate with other sparse nodes in a transparency network to detect forks* [17].

Event Client. A typical way to read blockchains is to listen to events emitted by smart contracts, which are stored in the transaction's logs as part of the transaction metadata. 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. Therefore, one could define an *event client* that given an *event predicate* $\mathcal{X}_e$, outputs *all* events in a specific stream (e.g., lock and/or mint events of a bridge). On one hand, an event client is very similar to a light client that instead of checking transaction inclusion it checks event inclusion. On the other hand, while light clients can use transaction Merkle roots to verify transaction inclusion, event clients need to resort to additional data structures to verify event inclusion. By having validators to commit to all the events in an event stream –an approach similar to Sunfish–event clients could offer completeness guarantees, i.e., be sure to receive *all* events in the stream of choice. We explore this in future work.

6 Analysis

Sunfish Security. We prove the theorem below in Appendix A.2.

Theorem 1 (Sunfish Security) *Sunfish is secure as per Definition 6, with* v *being the liveness parameter of the ledger.*

We now discuss which *validity* property fulfills the sparse ledger output by Sunfish. We recall that in the standard setting, ledger validity ensures that all transactions are valid with respect to the global state. For sparse ledgers, validity instead requires that each transaction is valid with respect to the client's sparse state. Concretely, the Sunfish client verifies that every state element in a transaction's read set such that $\mathcal{X}_s(k) = 1$ belongs to its local sparse state; otherwise, the block containing that transaction is rejected. When the Sunfish client reads from a secure ledger containing only valid transactions, the validity of its sparse ledger follows directly from the validity of the global ledger. If, however, the chain is compromised and an adversarial majority injects invalid transactions into the global ledger, the Sunfish client only accepts transactions that represent valid state transitions for its sparse state, even if they correspond to invalid transitions with respect to the global state of the chain (for example, a Sunfish client monitoring an ERC-20 contract will accept as valid a block that correctly updates ERC-20 balances, even if the same block also contains an invalid transaction—such as one that illegitimately assigns ownership of an ERC-721 token to an attacker). Therefore, the *validity property of the sparse ledger output by Sunfish is weaker* than the validity property of the ledger output by full nodes, but still meaningful for dApp developers and project operators who are exclusively interested in a valid snapshot of the dApp state to migrate to a different chain once security is broken for the current chain.

Finally, we note that sparse nodes could use synchronous gossip techniques [17–19] to efficiently detect forks. In these protocols, nodes periodically exchange their view of the state with a randomly selected subset of other nodes in the network, ensuring that any divergence is detected within a bounded time under network synchrony. We will further explore this as future work.

Sunfish Resources. Like light and full nodes, a Sunfish sparse node must regularly update consensus parameters (e.g., the current validator set or PoW difficulty). This costs $\mathcal{O}(\lambda|\mathcal{L}|)$, where λ captures the rate of committee changes. In most PoS chains, committees change infrequently (e.g., once per day), so the overhead is negligible, i.e., $\lambda \ll 1$. For sparse nodes that synchronize rarely or only once, these updates can be further compressed using succinct zero-knowledge proofs [20–22]. The Sunfish client downloads only the headers of blocks containing relevant transactions, i.e., a subset of the ledger. To ensure these scattered blocks belong to the same chain, we rely on *proofs of ancestry* [16] and we denote the resources for ancestry proofs as $\mathcal{O}(\rho|\mathcal{L}|)$, with $\rho \ll 1$. The bandwidth, computation, and storage costs of a Sunfish client are proportional to the size of its $\hat{S}$, the number of transactions in its $\hat{\mathcal{L}}$, and the degree of external state dependencies [23] passed through transaction read sets. For example, a sparse node monitoring a flash loan contract must often handle interactions with external contracts such as DEXs. Incorporating external dependencies directly into the tracked state may be more efficient for frequently accessed read sets. To account for sparse node access to the read set and its correctness, we introduce a factor $\psi > 0$ multiplying $|\hat{S}|$, with $\psi\hat{S} \leq S$. Finally, verifying transaction inclusion in ledgers with non-constant opening sizes (e.g., Merkle trees) adds a factor $\eta > 0$ to $|\hat{\mathcal{L}}|$, with $\eta\hat{\mathcal{L}} \leq \mathcal{L}$. We prove the theorem below in Appendix A.2.

Theorem 2 (Sunfish Resources) *The bandwidth, computational, and storage resources consumed by Sunfish are* $\mathcal{O}(\lambda\rho|\mathcal{L}| + \eta|\hat{\mathcal{L}}| + \psi|\hat{S}|)$.

We observe that the cost $\mathcal{O}(\lambda\rho|\mathcal{L}| + \eta|\hat{\mathcal{L}}| + \psi|\hat{S}|)$ is dominated by different terms depending on the workload. If the sparse state is frequently accessed, then $\eta|\hat{\mathcal{L}}| + \psi|\hat{S}|$ dominates. If the sparse state is rarely updated, then $\lambda\rho|\mathcal{L}|$ becomes more significant. This is in contrast with full nodes, which consume $\mathcal{O}(|\mathcal{L}| + |S|)$ resources, and light nodes, which consume $\mathcal{O}(\lambda\rho|\mathcal{L}|)$ resources.

7 Evaluation

Since Sunfish performs simple operations (DB lookups, integer arithmetic, hashing), we expect minimal computational impact on validators for a controlled number of sparse states tracked. We highlight that benchmarking sparse node resources against the ones of full and light nodes is challenging, as sparse nodes resources greatly vary depending on the size and the transaction traffic of the sparse state. On the contrary, Resources do not vary much for full nodes (validators are incentivized to completely fill blocks) and light nodes (signature verification of rarely-changing validators). Hot, interdependent Ethereum substates (DEXes,bridges) may incur higher overheads (2-5% the one of full nodes), whereas colder substates (wallets), $\ll 0.1\%$. For instance, consider Uniswap, one of the most popular dApps on Ethereum: in the last 24h, Ethereum has roughly processed 1,673,545 transactions [24], while Uniswap 85,210 transactions across multiple chains [25]. Without considering transaction dependencies, a sparse node running Sunfish-HC and monitoring Uniswap would consume $\ll 5\%$ of full nodes resources, as only 20% Uniswap activity occurs on Ethereum [26]. We evaluate Sunfish on Sui to show its practicality for high-performance blockchains.

The Sui Blockchain. Sui [27] is a 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* [28] that contain a sequence of transactions and form a hash-chain. Sui checkpoints contain a *summary* (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 commitment to sparse states.

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 [29] and the Wave wallet [30]. We consider sparse states identified by different predicates: package-based, event-based, and address-based. We consider: (i) the Wormhole bridge, via package (if tx touches the Wormhole package). (ii) The Wormhole bridge, via events (if tx emits Wormhole events). Here, the sparse node only receives events, not transactions. (iii) The Wave wallet, via address (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, $C = \mathsf{min}(R, 4.12)$); (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 1 we show the actual stream rate R, ob-

Table 1: 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).

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

tained 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).

8 Related Work

Sunfish bridges the gapbetween full nodes [31–33] and light nodes [4, 34, 35, 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-executes a subset thereof. Sunfish ensures sparse validity, completeness, and fork consistency properties that light clients do not provide because of their minimalist design and the lack of transaction re-execution. Some light client designs [36, 37] consider completeness as an important property, however, they achieve it by relying on trusted execution environments [38] and do not consider re-execution. Other light clients [13] have been designed to help secure the chain: these include data availability and validity checks, and require deploying 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 [13] in scope and functioning: it operates stand-alone by reading the chain and maintaining a valid substate of the global state of the ledger.

Light clients that help secure the chain have become popular in the context of lazy ledgers [23, 39]. Lazy ledgers decouple consensus from transaction verification and execution to increase throughput. The 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*, forgoing communication efficiency. Instead, a sparse nodes achieves the same properties by taking a much harder approach that limits the amount of data that is downloaded.

Finally, in sharding [40, 41] the consensus nodes 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 the Mysten Labs Data Science team for providing necessary data to conduct our evaluation. We thank George Danezis, Joachim Neu, Philipp Slowak, Dionysis Zindros, and Zeta Avarikioti for fruitful discussions and feedback. 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. Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. Blink: An optimal proof of proof-of-work. Financial Cryptography and Data security 2025, 2024.
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. Vitalik Buterin. A local-node-favoring delta to the scaling roadmap, 2025.
7. EigenLayer Team. Eigenlayer: The restaking collective, 2024.
8. 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.
9. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.
10. Leslie Lamport. The implementation of reliable distributed multiprocess systems. 1978.
11. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. 2002.

12. Ethereum yellowpaper, 2024.
13. Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud and data availability proofs: Detecting invalid blocks in light clients. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2021.
14. Solana: Transactions and instructions, 2025.
15. Toni Wahrstaetter, Dankrad Feist, Francesco D'Amato, Jochem Brouwer, and Ignacio Hagopian. Eip-7928: Block-level access lists, 2025.
16. Pericle Perazzo and Riccardo Xefraj. Smartfly: Fork-free super-light ethereum classic clients for internet of things. *IEEE Internet of Things Journal*, 11(9):15348–15358, 2024.
17. 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.
18. David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. New York, NY, USA, 2002. Association for Computing Machinery.
19. Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. PODC '07. Association for Computing Machinery, 2007.
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.
23. Mustafa Al-Bassam. LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts, 2019.
24. Etherscan. https://etherscan.io/txs.
25. Defillama. https://defillama.com/protocol/uniswap.
26. Unichain surpasses ethereum as top chain for uniswap v4 by transaction volume. https://www.theblock.co/post/353769/unichain-surpasses-ethereum-as-top-chain-for-uniswap-v4-by-transaction-volume.
27. 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.
28. 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.
29. Wormhole Bridge, 2024. https://docs.sui.io/concepts/tokenomics/sui-bridging.
30. Wave Wallet on Sui. https://waveonsui.com/.
31. Sui full node transaction signatures are not verified, 2024.
32. 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.
33. Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. Executing and proving over dirty ledgers. In *Financial Cryptography and Data Security 2023*, 2023.

34. 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.
35. 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.
36. 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.
37. 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, February 18–22, 2019*, pages 179–198. Springer, 2019.
38. Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. Sok: Hardware-supported trusted execution environments. *arXiv preprint arXiv:2205.12742*, 2022.
39. Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. Executing and proving over dirty ledgers. In *International Conference on Financial Cryptography and Data Security*, pages 3–20. Springer, 2023.
40. Zeta Avarikioti, Antoine Desjardins, Lefteris Kokoris-Kogias, and Roger Wattenhofer. Divide & scale: Formalization and roadmap to robust sharding. In *Structural Information and Communication Complexity*. Springer Nature Switzerland, 2023.
41. Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*, pages 583–598. IEEE, 2018.

A Algorithms and Proofs

A.1 Algorithms

We showcase the algorithms for the Sunfish protocol: Algorithm 1 for the client and Algorithm 2 for the prover. 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, and with π_i, π_a, π_c, and π_{rs} the inclusion, ancestry, sparse ledger, and read set proofs, respectively. We now intuitively describe the behavior of the functions used in the algorithms.

IsBlockValid: This function verifies that the block header received from the prover is valid, i.e., that it is indeed an ancestor of the chain's tip. As discussed in Section 4, the sparse node relies on a secure light client protocol to identify the tip of the chain. The details of the light client protocol are beyond the scope of this work, as they can be inherited directly from state-of-the-art constructions. For example, PoW light client protocols such as [1] enable identification of the chain tip without downloading the full history of block headers, while PoS light client protocols such as [2] allow efficient discovery of the current validator set and, consequently, of the tip of the chain. Since the sparse client is only concerned with block headers that contain transactions relevant to its sparse state, it must additionally verify ancestry proofs ensuring that these headers are indeed ancestors of the identified tip. This is particularly important in the bootstrap phase where the Sunfish client, after identifying the tip of the chain, starts receiving all the older blocks that include relevant transactions.

isSparseLedgerCommitmentVerified: This function verifies that the transactions received from the prover verify against the sparse state commitment included in the block header. For Sunfish-C, this means that the client counts how many transactions it had received from the prover and compares it with the value of the counter included in the block header. Additionally, it verifies that such transactions are included in the ledger. For Sunfish-HC, this means that the transactions received from the prover verify against the hash chain head included in the block header. In other words, the client locally recomputes the hash chain from the sequence of transactions received from the prover and compares the obtained chain head with the one included in the block header.

isValidSparseStateUpdate: This function is responsible for executing the transactions that touch the sparse state of the client, checking the correctness oftheir read sets and assessing their validity with respect to the current sparse state. The execution takes place via $\hat{\delta}$, which is a domain and range restriction of the global transition function δ.

UpdateSparseLedger: The function updates the sparse ledger with the transactions received —provided that all the checks in IsBlockValid, isSparseLedgerCommitmentVerified, and isValidSparseStateUpdate pass.

We note that, upon updating the sparse ledger, the sparse client updates its sparse state (Algorithm 1, line 18).

A.2 Proofs

Theorem 3 (Sunfish Security) *Sunfish is secure as per Definition 6.*

Proof. **Safety:** To prove safety of Sunfish, we need to prove that, for any r, the sparse ledger $\hat{\mathcal{L}}^r$ output by the client is a prefix of $\hat{\mathcal{L}}^r \preceq \varphi(\mathcal{X}_s, \mathcal{L}_{\bigcup}^{r'})$, with $r' \geq r + v$. Towards this, consider the sparse ledger $\hat{\mathcal{L}}^r$ output by the client in Algorithm 1, line 17. At each round r, the client tries to extend $\hat{\mathcal{L}}^r$ with the transactions received by the provers (Algorithm 1, line 10). Because the sparse client runs a secure light client protocol, it is able to correctly identify the stable tip of the chain. Because the client verifies that the received batch of transactions belong to a block that is an ancestor of the stable tip of the chain (Algorithm 1, line 12), at round r, it only attempts to extend $\hat{\mathcal{L}}^r$ with transactions that are final, i.e., that are at least v rounds old–with v being the transaction confirmation time of the ledger. Because the block header the client accepts is final, and because we consider secure blockchains operated by an honest (super)majority of consensus nodes, the commitments to the sparse ledger that are included in block headers are correct. The client extends its local sparse ledger (Algorithm 1, line 17) only if the transactions received correctly verify against the commitment (Algorithm 1, line 13). Therefore, at any round r, the sparse ledger $\hat{\mathcal{L}}^r \preceq \varphi(\mathcal{X}_s, \mathcal{L}_{\bigcup}^{r'})$. To complete the proof, we now show that the

Algorithm 1 The algorithm executed by the Sunfish sparse client V. If Sunfish-HC is used, the transaction inclusion proof π_i is not needed. We omit from the algorithm the logic used by the client to verify the canonical chain, as it varies from chain to chain and it can be inherited verbatim from existing light client protocols.

```
1:  function VERIFIER($\mathcal{X}_s$, $\mathcal{P}$, $\hat{\mathcal{G}}$)
2:      $\hat{S} \leftarrow \hat{\mathcal{G}}$, $\hat{\mathcal{L}} \leftarrow []$, $\mathsf{bool} \leftarrow \perp$
3:      for $P \in \mathcal{P}$ do
4:          $\mathcal{X}_s \dashrightarrow P$
5:          run VERIFY($\hat{S}$, $\mathcal{X}_s$, $\hat{\mathcal{L}}$)
6:      end for
7:  end function
8:  function VERIFY($\hat{S}$, $\mathcal{X}_s$, $\hat{\mathcal{L}}$)
9:      while True do
10:         $\mathcal{D} \dashleftarrow P$
11:         $(\mathsf{header}, \mathsf{txs}, \mathcal{R}(\mathsf{txs}), \pi_a, \pi_c, \pi_i, \pi_{\mathsf{rs}}) = \mathcal{D}$
12:         $\mathsf{bool} = $ isBlockValid(header, $\pi_a$) $\wedge$
13:                 isSparseLedgerCommitmentVerified(txs, header, $\mathcal{X}_s$, $\pi_c$, $\pi_i$)
14:         if bool then
15:             (bool, $\hat{S}_{\mathsf{temp}}$) = isValidSparseStateUpdate($\mathcal{X}_s$, $\hat{S}$, txs, $\mathcal{R}$(txs), $\pi_{\mathsf{rs}}$)
16:             if bool then
17:                 $\hat{\mathcal{L}}$ = UpdateSparseLedger($\hat{\mathcal{L}}$, txs)
18:                 $\hat{S} = \hat{S}_{\mathsf{temp}}$
19:             end if
20:         end if
21:     end while
22: end function
```

Sunfish-C and Sunfish-HC commitments to the sparse ledger uniquely identify a sparse ledger.

Consider a state predicate $\mathcal{X}_s$ and a blockchain that uses Sunfish-C: We show that combining a commitment to the number of transactions in a block touching $\mathcal{X}_s$ with a commitment to the transactions included in that block, uniquely identifies the sparse ledger $\varphi(\mathcal{X}_s, \mathcal{L}_{\bigcup}^{r'})$. The per-block commitment described in Section 4.1 commits to a counter whose value corresponds to the number of transactions in the block that touch the sparse state $\mathcal{X}_s$. It also commits to the hash of the transactions included in the block. Because of the collision resistance and preimage resistance properties of the hash function, an adversary cannot find different preimages to the same counter and inclusion commitments, unless with negligible probability in κ. Because we consider secure blockchains, we know that the commitments are correct and, therefore, the two commitments together ensure completeness and uniqueness of the set of transactions touching $\mathcal{X}_s$. Therefore, the Sunfish-C commitments to the sparse ledger uniquely identifies a sparse ledger.

Algorithm 2 The algorithm executed by a Sunfish prover P. We omit from the algorithm the logic used by the prover to identify the canonical chain and communicate it to the client, as it varies from chain to chain and it can be inherited verbatim from existing light client protocols.

```
1: function EXECUTE($1^\kappa$)
2:     $\mathcal{X}_s \dashleftarrow V$
3:     $\mathcal{C} \leftarrow$ GETCHAIN()
4:     BOOTSTRAP($\mathcal{C}$)                          ▷ Bootstrapping the sparse node.
5:     while True do
6:         block = WAITFORNEWFINALBLOCK($\mathcal{C}$)   ▷ Update the sparse node at every new block.
7:         PROCESSBLOCK(block)
8:     end while
9: end function

10: function BOOTSTRAP($\mathcal{C}$)
11:     for height in $(0, |\mathcal{C}|)$ do
12:         block = GETBLOCK($\mathcal{C}$,height)
13:         PROCESSBLOCK(block)
14:     end for
15: end function

16: function PROCESSBLOCK(block)
17:     txs $\leftarrow$ [], $\pi_c \leftarrow$ [], $\pi_a \leftarrow$ [], $\pi_i \leftarrow$ [], $\pi_{\mathsf{rs}} \leftarrow$ []
18:     for tx in $\varphi(\mathcal{X}_s$, block.transactions) do
19:         txs.append(tx)
20:         $\pi_i$.append(GENINCLPROOF(tx, block))      ▷ Inclusion proof not needed if Sunfish-HC is used.
21:     end for
22:     if txs is not $\varnothing$ then
23:         $\pi_a \leftarrow$ GENANCESTRYPROOF($\mathcal{C}$, block.header)
24:         $\pi_c \leftarrow$ GENSPARSECOMMITMENTPROOF(txs, block.header)
25:         ($\mathcal{R}$(txs), $\pi_{\mathsf{rs}}$) $\leftarrow$ GENREADSETPROOF(txs, block.header)
26:         (block.header, txs, $\mathcal{R}$(txs), $\pi_a, \pi_i, \pi_c, \pi_{\mathsf{rs}}$) $\dashrightarrow V$   ▷ Inclusion proof not sent if Sunfish-HC is used.
27:     end if
28: end function
```

Consider now a state predicate $\mathcal{X}_s$ and a blockchain that uses Sunfish-HC: We show that committing to the head of the hash chain generated by hashing the transactions that touch $\mathcal{X}_s$, uniquely identifies the sparse ledger $\varphi(\mathcal{X}_s, \mathcal{L}_{\bigcup}^{r'})$. Because we consider secure blockchains, we know that the per-block commitment described in Section 4.2 is correct. Because of the collision resistance and preimage resistance of the hash function, an adversary cannot find a different sequence of transactions whose hash yields to the same commitment, unless with negligible probability in κ. Therefore, the Sunfish-HC commitment to the sparse ledger uniquely identifies a sparse ledger.

Liveness: To prove liveness of Sunfish, we need to prove that $\varphi(\mathcal{X}_s, \mathcal{L}^r_{\bigcap}) \preceq \hat{\mathcal{L}}^r$. In other words, we need to prove that the sparse ledger output by the client at any round r extends the shortest among the honest nodes' stable ledgers filtered by the predicate $\mathcal{X}_s$. Because we assume that the client connects to at least one honest prover, the client is guaranteed to receive updates for its sparse ledger from that prover. Because we assume the prover is non-eclipsed, at any round r, the ledger of the honest prover extends the ledger in the view of at least one honest consensus node. The honest prover runs the ProcessBlock function at each new final block that it sees (Algorithm 2, line 7): Therefore, it generates the Sunfish necessary proofs (Algorithm 2, line 19, 20, 23, 24, 25), and it sends them to the client (Algorithm 2, line 26). Because the communication is synchronous, the client is guaranteed to receive the update from the honest, non-eclipsed prover (Algorithm 2, line 26) within Δ time. Because the time between ledger updates is larger than the network delay for security reasons, the sparse ledger $\hat{\mathcal{L}}^r$ output by the client always extends $\varphi(\mathcal{X}_s, \mathcal{L}^r_{\bigcap}) \preceq \hat{\mathcal{L}}^r$.

□

Theorem 2 (Sunfish Resources) *The bandwidth, computational, and storage resources consumed by Sunfish are* $\mathcal{O}(\lambda\rho|\mathcal{L}| + \eta|\hat{\mathcal{L}}| + \psi|\hat{S}|)$.

Proof. Bandwidth: Merkle tree-based ancestry proofs require the node to download $\mathcal{O}(|\mathcal{L}| \log N)$ of data, so $\rho = \log N$. Downloading transactions requires $\mathcal{O}(|\hat{\mathcal{L}}|)$ and downloading their read sets along with their correctness proofs requires $\mathcal{O}(\psi|\hat{S}|)$. 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*: We consider a constant upper bound to the computation associated to a transaction. The computation required to verify transaction inclusion, check the sparse ledger commitment, and verify the transaction read set is $\mathcal{O}(|\hat{\mathcal{L}}| \log M)$, $\mathcal{O}(|\hat{\mathcal{L}}| \log Q)$, and $\mathcal{O}(|\psi\hat{S}|)$, respectively. The computational complexity of execution is $\mathcal{O}(|\hat{\mathcal{L}}|)$. 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 $\mathcal{O}(\lambda\rho|\mathcal{L}| + \eta|\hat{\mathcal{L}}| + \psi|\hat{S}|)$, with $\eta = \log M + \log Q$ for Sunfish-C and $\eta = \log Q$ for Sunfish-HC. □

B Sunfish-C vs Sunfish-HC

We note that Sunfish-C requires the blockchain to have, in each block header, a commitment to a Merkle tree over the transaction of the block, so to verify transaction inclusion. This is not the case when using the data structure of Sunfish-HC, as hash chains entirely commit to transactions, allowing to verify completeness and inclusion at the same time. Sunfish-C is minimizes the validators' overhead, while Sunfish-HC minimizes the resources of the sparse node. We now compare the two data structures in terms of proof size and validators' storage and computation. Let Q be the average number of sparse states having transactions in a block, and M the average number of transactions per block. We omit from the client proof size the overhead due to the read set proof, as it is the same for both data structures.

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).

C Sparse Node Operating Modes

Sparse nodes can have various operating modes.

Header Mode. A *header node* is always online and reads *all block headers*, irrespective of whether a relevant transaction is in the block (similar to SPV light nodes). This mode offers the benefit that liveness failures are detectable early on (assuming blocks are produced at a known rate and the network is synchronous), although at the cost of increased resource consumption.

Continuous Mode. 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 an honest prover, the ledger of a continuous sparse node is complete at all times. 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. This is the *primary operating mode* considered in the main body of this work.

Intermittent Mode. 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.

On-Demand Mode. An *on-demand* sparse node 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.