Sunfish: Reading Ledgers with Sparse Nodes

Giulia Scaffino
TU Wien & Common Prefix
giulia.scaffino@gmail.com

Philipp Slowak
TU Wien
philipp.slowak@gmail.com

Karl Wüst
Mysten Labs
karl@mystenlabs.com

Deepak Maram
Mysten Labs
deepak@mystenlabs.com

Alberto Sonnino
Mysten Labs & UCL
alberto@mystenlabs.com

Lefteris Kokoris-Kogias
Mysten Labs
lefteris@mystenlabs.com

Abstract

Users who wish to interact with blockchains typically engage with only a small number of decentralized applications (dApps) whose state they need to monitor. However, securely and trustlessly tracking the state of even a single dApp currently requires running a full client, which independently downloads and verifies the entire blockchain, re-executes all transactions, and reconstructs the global state. Operating a full client contrasts sharply with the traditional client–server paradigm, where clients retrieve only the data they need, and becomes increasingly difficult to sustain as blockchains' throughput increases. Light clients do not offer a viable alternative: while they are more resource-efficient thanks to their use of succinct proofs, they can verify only limited information about the ledger and its state and rely on additional trust assumptions for this. As a result, a gap emerges in the blockchain client design space: enabling secure and verifiable monitoring of dApp state, isolating the workload of a given dApp from the workload of the entire chain.

To bridge this gap, we introduce a *sparse client*, a new type of blockchain client that only downloads the transactions that modify the state of a specific dApp and only computes and stores the dApp state, isolating the dApp's workload from the one of the entire chain. We also present Sunfish, a secure sparse client protocol available in two variants: one off-the-shelf compatible with Ethereum Virtual Machine (EVM)-based blockchains and one virtually compatible with any chain. We also introduce an *event client*, a special case of sparse clients that only tracks a particular stream of events emitted by a dApp. We benchmark sparse and event clients against a full client by implementing prototypes for Ethereum. Our results show that our sparse and event clients respectively save 66% and 85% operating cost when compared to a full node.

1 Introduction

Blockchains gain their security through a consensus protocol running among a large and changing group of mutually distrusting participants, known as validators. Over time, validators receive transactions from clients and return to them a sequence of confirmed transactions arranged into blocks called a ledger. Blockchains typically offer two types of clients: full clients and light clients.

Full and Light Clients. A full client downloads all the transactions in the ledger, computes the associated state, and locally stores the full history of transactions and the current state. It independently verifies transactions and blocks, it computes state changes, it listens to events, and it maintains audit proofs. A full client allows interaction with the blockchain in a private, unrestricted, and censorship-resistant manner, without relying on trusted third-parties. It also provides the data access layer by exposing RPC interfaces to wallets, applications, and light clients. In contrast, a light client [4, 7, 11, 20, 24, 29] downloads and verifies only a compact summary of each block, known as the block header. The sequence of headers provides commitments to both the ledger and the global state of the blockchain, enabling the light client to verify transaction inclusion or read a specific state value (e.g., the balance of an account).

Full and light clients offer different trade-offs between security and efficiency: full clients provide strong security guarantees but are resource-intensive, whereas light clients are cheap to operate but provide weaker security without additional trust assumptions. As blockchains scale to higher throughput and larger gas limits, the bandwidth, computational, and storage requirements for operating a full client grow substantially, making it increasingly expensive to host and operate. Despite providing the data access layer of the ecosystem, blockchain protocols typically offer no direct incentives to operate full clients, resulting in a limited number of publicly accessible full clients. Light clients, while more efficient, rely on additional trust assumptions (i.e., existential honesty of provers) and are limited to basic checks (they can verify inclusion of a transaction but they cannot make sure they have received *all* transactions modifying the state of a contract). As such, they are not interchangeable with full clients: they do not validate transactions, verify state transitions, nor listen to events. They are also vulnerable under adverse conditions such as forks or reorganizations, where they may accept invalid state transitions.

The Need for a New Client Type. While casual users are often satisfied with light clients to check balances or confirm payments, dApp developers and active participants (e.g., bridge operators, sequencers and watchers, market makers, DAO token holders) require stronger security guarantees. For accounting, compliance, and auditing purposes, they must independently track application-relevant state transitions, process emitted events, and maintain a verifiable record of transactions and historical asset ownership. This leaves them no choice but to run full clients and process all on-chain data, not just that of their own dApp, imposing unnecessary costs and becoming increasingly unsustainable as throughput grows. This comes in stark contrast to the Web2 client–server paradigm, where clients fetch only the data they need without processing the entire underlying dataset and hinders Web3 adoption. The high operating costs and lack of incentives lead dApp users and (especially early-stage) developers to rely on a small number of RPC providers to access dApp data. This dependency effectively centralizes access,

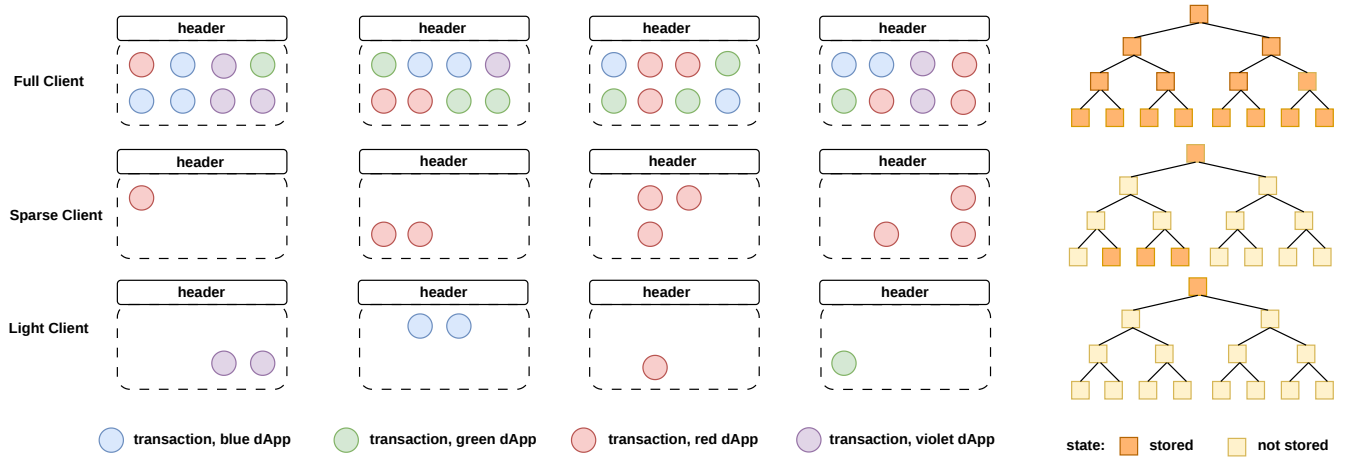


Figure 1: Comparison of full, sparse, and light clients in terms of the block and transaction data they download and the state they store. A full client downloads all blocks and transactions and recomputes and stores the entire blockchain state. A sparse client downloads all block headers and a specific, *complete* subsequence of transactions (shown in red), storing the corresponding portion of the state together with a digest of the entire state (included in the headers). A light client downloads block headers and any set of transactions, without being able to verify whether the set is complete; a light client also has access to a digest of the global state, if it is part of the headers.

allowing these providers to act as gatekeepers that may restrict service or censor users. Moreover, routing queries through centralized providers may expose sensitive metadata such as IP addresses and query patterns.

Taken together, we conclude that neither full nor light clients adequately satisfy the needs of high-security actors who require verifiable access to application-specific data at costs that scale proportionately with the subset of data they actually need. This reveals a fundamental gap in the blockchain client design space.

1.1 Our Contribution

In this work we introduce a new type of blockchain client: the *sparse client*. A sparse client downloads, executes, and stores only a specific subsequence of transactions, thereby locally maintaining the corresponding partial state of the blockchain. Specifically, a sparse client downloads all transactions that modify the state of a dApp contract, executes them to compute the state of the dApp, and, if the execution succeeds, it updates its internal state accordingly. Essentially, a sparse client mirrors the tasks and properties of a full client, but only with respect to a specific, well-defined subset of the blockchain state. Figure 1 compares full, sparse, and light clients in terms of block and transaction data they download, and state they store. Importantly, a sparse client isolates the state and traffic of a specific dApp contract from the state and traffic of the entire blockchain, optimizing its operating costs. At the same time, because a sparse client locally executes all dApp-specific transactions, it does not accept invalid application-level state transitions. Even if the underlying blockchain is compromised, the client maintains a consistent and valid view of the dApp state, while detecting missing updates or invalid transitions. This property is particularly important for dApp developers, as it allows to preserve a correct application state and, if needed, migrate it to a different platform.

Designing a sparse client protocol requires simultaneously minimizing the amount of data the client downloads while preserving strong correctness and verifiability guarantees. In particular, without downloading the entire ledger or state, the client must obtain a verifiably correct subsequence of transactions of the ledger (the ones that modify the state of the dApp). This requires succinctly proving that every transaction in the subsequence appears in the ledger in the same relative order and that no relevant transaction has been omitted; this is a core distinguishing attribute between sparse clients and light clients. In addition, the client must be able to locally execute the transactions in order to compute the corresponding part of the global state. This requires authenticating the parts of the state accessed during transaction execution that lie outside the portion of the state monitored by the client. This is not trivial, as many blockchains do not currently allow to directly authenticate transaction read sets, especially in execution environments where state access is determined dynamically.

After formalizing the interface and security model of a sparse client (Section 2), we present Sunfish, a secure sparse client protocol (Section 3). Sunfish introduces new mechanisms that produce succinct proofs, enabling a client to verify that a sequence of transactions forms a correct order-preserving subsequence of the ledger. These mechanisms also ensure correct execution while remaining compatible with a wide range of existing blockchains. We present two variants of Sunfish: one that is off-the-shelf compatible with existing EVM-based blockchains and leverages their data structures to construct efficient proofs, and another with a chain-agnostic design that enables sparse clients on a broader class of blockchain systems. We also introduce the *event client*, a special case of a sparse client that only downloads and verifies a particular subset of chain metadata: a stream of events emitted by a contract. We then prove that Sunfish is secure (Section 4).

We benchmark sparse and event clients against a full client by implementing Sunfish prototypes for Ethereum and evaluating their gas overhead, resource consumption, and operational costs (Section 5). Our results show that sparse and event clients substantially reduce resource requirements compared to a full client. For representative Ethereum applications such as USDT transfers, CryptoKitties auctions, and Uniswap swaps, the event client reduces inbound bandwidth by 95%, both clients achieve more than four orders-of-magnitude storage savings, and the sparse client reduces transaction execution by 92%. These improvements translate into operating cost reductions of up to 85% compared to running a full Ethereum node. These benefits lead the Sui blockchain to integrate an event client into its protocol [3]. Finally, we compare sparse clients with existing client designs and with validators in sharding-based protocols, showing that sparse clients occupy a previously unexplored point in the design space (Section 6).

We highlight that sparse clients benefit a wide range of crucial blockchain actors—including bridge operators, rollup sequencers and watchers, DEX market makers, DeFi risk monitoring services, DAO token holders—that need to monitor and verify the state of a dApp. Sparse clients can also help optimize blockchain infrastructure by acting as read caches or replicas that serve light clients and maintain custom indexes for on-chain data. By separating read-heavy workloads from write operations, they can reduce load on full clients and be dynamically deployed in hotspots—such as highly active dApps or specific geographical regions—to increase redundancy and read bandwidth where demand is greatest.

2 Model and Preliminaries

We consider a setting with n *validators*, each of which equipped with a cryptographic identity (typically, a public/secret key pair for digital signatures) known to all other validators (public key infrastructure, PKI). We also consider a set of *clients*, whose size is unknown to the validators. Over time, the validators receive transactions from the clients and, over time, they return a ledger of transactions to the clients. Clients send transactions to the validators through a *network* and, in return, receive a ledger of transactions. Validators also send each other messages through the network to reach agreement on an ordering of the input transactions into an output ledger. The validators' output ledgers are *confirmed*.

Time proceeds in discrete communication *rounds*. Throughout the entire protocol execution, the validators have *synchronized clocks*: all validators always know which is the current round. In every round, clients possibly send transactions to the validators, and receive the output ledger of the validators. Also in every round, each validator receives messages from the network and, possibly, transactions from the clients. In every round, each validator updates its internal state, produces messages to send to other validators and clients via the network, and outputs a ledger of confirmed transactions to the clients.

An *adversary* aims to disrupt the system by corrupting validators, clients, and delaying network messages. We consider an adversary that corrupts a fixed set of at most f validators at the start of the protocol execution, before any randomness is drawn (static corruption). At any round, the adversary can also corrupt an unbounded number of clients. The adversary learns the internal state of corrupted validators and clients and causes them to deviate arbitrarily from the protocol for the entire execution (permanent Byzantine faults). Corrupted validators and clients, also known as adversarial, run any algorithm chosen by the adversary in time polynomial in the size of the inputs (computationally bounded adversary).

Validators can exchange messages between each other via a fully-connected network of point-to-point links. The network is *partially synchronous* [18]: there exists an upper-bound message delay Δ that is known to all validators and a global stabilization time $\mathsf{GST} < \infty$ that is adaptively chosen by the adversary. Before GST, the adversary can delay messages arbitrarily (asynchronous network). After GST, the adversary can delay messages by at most Δ (synchronous network). During synchronous periods, when a validator instructs the network to send a message to another validator, the message is enqueued in the recipient's pending message queue together with a countdown initialized to Δ; the countdown decreases by 1 with each round. The adversary can also decrease the countdown of every message at will. Once the countdown hits 0, the message is delivered to the recipient at the beginning of the next round. This means that every message sent by an honest validator or client by time t is delivered to all honest validators and clients by time $\max(t, \mathsf{GST}) + \Delta$.

State Machine Replication Interface. The validators run a Byzantine fault-tolerant consensus protocol of the state machine replication (SMR) flavour to reach agreement on their output ledgers, i.e., a common ordering of the input transactions received from clients. The protocol proceeds in *slots*, with each slot being associated with a leader validator and a proposal window, during which the leader proposes a payload of transactions to be appended to the output ledger that they broadcast to the network. Slots typically consists of three phases: a proposal phase, and two voting phases known as notarization and confirmation.

Definition 1 (State Machine Replication Interface). *A SMR protocol* Π *is run by* n *validators and an unbounded number of clients. In every round, validators possibly receive transactions from the clients and they return to the clients a confirmed sequence of transactions called ledger* $\mathcal{L}$.

We consider executions of the SMR protocol in which, regardless of the strategy of the adversary, all correct validators and clients output a ledger $\mathcal{L}$ that is safe and live. We write $\mathcal{L} \preceq \tilde{\mathcal{L}}$ to denote that $\mathcal{L}$ is a prefix of $\tilde{\mathcal{L}}$.

Definition 2 (SMR Security). *A SMR protocol is* secure with resilience τ *in partially synchronous networks iff, in every partially-synchronous execution with up to* $f \leq \tau$ *adversarial validators, except with probability* $\mathrm{negl}(\kappa)$ *where* κ *is the security parameter, the following properties hold:*

Safety: *For every two honest clients* i, j *and for every two rounds* r, r', *let* $\mathcal{L}_i^r$ *be the ledger output by* i *at* r *and* $\mathcal{L}_j^{r'}$ *the ledger output by* j *at* r'. *Then, the two ledgers are consistent, i.e.,* $\mathcal{L}_i^r \preceq \mathcal{L}_j^{r'}$, *or vice versa.*

Liveness: *For every round* r_0 *and every transaction* tx, *if every honest validator receives* tx *by* r_0, *then there exists a round* $r_1 \geq r_0$ *such that for every round* $r_2 \geq r_1$ *and for every honest client* i, $\mathsf{tx} \in \mathcal{L}_i^{r_2}$.

At every round r, validators update their internal state S^{r-1} according to a state transition function δ that, on input the state S^{r-1} and a sequence of transactions, it returns the new state S^r. Consider an initial state S^0. The state S^r of a validator is associated to the ledger $\mathcal{L}^r = [\mathsf{tx}_1, \ldots, \mathsf{tx}_m]$ and computed as $S^r := \delta(\ldots\delta(S^0, \mathsf{tx}_1)\ldots, \mathsf{tx}_m)$; hereafter, we write $S^r := \delta(S^0, \mathcal{L}^r)$ to denote successive application of all transactions in $\mathcal{L}^r$ given an initial state S^0. Without loss of generality, we model the state S as a key-value store, i.e., a collection of (k, v), with $k \in K$ and $v \in V$, where K and V are sets of valid keys and valid values, respectively. In particular, a key k is a unique identifier (e.g., address of an account or contract, or the hash of an UTXO) used to reference a specific value, whereas a value 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^{r-1} and writes to an output state S^r by consuming state elements in S^{r-1} and generating new ones in S^r. We refer to the set of state keys (and their corresponding values) that are accessed during execution as the *read set*, denoted by $\mathcal{R}(\mathsf{tx})$, and the set of *state keys* (and their corresponding values) that are modified as the *write set*, denoted as $\mathcal{W}(\mathsf{tx})$.[1]

Sparse Client Model. A sparse client downloads a *subset of transactions* of the ledger and updates its internal state by applying a (modified) state transition function to those transactions and its current local state. The state of a sparse client is a subset of the global state of the ledger. Let us define the sparse client formally.

We recall that the state S associated with a ledger is a key value store (k, v) such that each key is associated with exactly one value. We introduce a *state predicate* $\mathcal{X}_s : K \to \{1, 0\}$ that, at every round r, identifies a subset $\hat{S}^r$ of the elements of the state S^r, i.e., $\hat{S}^r \subseteq S^r$. We call $\hat{S}^r$ the *sparse state* identified by $\mathcal{X}_s$ at round r.

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

Let X be the universe of all possible state predicates, and L be the universe of all possible ledgers. We introduce a filtering function $\varphi : X \times L \to L$ that, on input a state predicate $\mathcal{X}_s \in X$ and a ledger $\mathcal{L} \in L$, it returns a *sparse ledger* $\hat{\mathcal{L}}$. At every round r, the sparse ledger $\hat{\mathcal{L}}^r$ is the order-preserving subsequence of transactions from $\mathcal{L}^r$ whose write sets include state elements identified by $\mathcal{X}_s$.

Definition 4 (Sparse ledger $\hat{\mathcal{L}}$). *$\hat{\mathcal{L}}^r = \varphi(\mathcal{X}_s, \mathcal{L}^r) = [\mathsf{tx} | \mathsf{tx} \in \mathcal{L}^r \wedge \exists k \in \mathcal{W}(\mathsf{tx}) : \mathcal{X}_s(k)]$ is the sparse ledger identified by $\mathcal{X}_s$ at round r.*[2]

We observe that for any two state predicates $\mathcal{X}_s$ and $\tilde{\mathcal{X}}_s$, we have that $\varphi(\mathcal{X}_s, \varphi(\tilde{\mathcal{X}}_s, \mathcal{L})) = \varphi(\tilde{\mathcal{X}}_s, \varphi(\mathcal{X}_s, \mathcal{L}))$. We now define the sparse client protocol interface and security.

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

[1] A more precise formulation of the read and write set is $\mathcal{R}(S^r, \mathsf{tx})$ and $\mathcal{W}(S^r, \mathsf{tx})$ because, depending on the state S^r of the ledger at round r, the read and write sets may have different keys or different values. For simplicity, however, we do not make explicit the dependency to the state.

[2] A stronger version of this definition could be: $\hat{\mathcal{L}}^r = \varphi(\mathcal{X}_s, \mathcal{L}^r) = [\mathsf{tx} | \mathsf{tx} \in \mathcal{L}^r \wedge (\exists k \in \mathcal{R}(\mathsf{tx}) : \mathcal{X}_s(k) \vee \exists k \in \mathcal{W}(\mathsf{tx}) : \mathcal{X}_s(k))]$ is the sparse ledger identified by $\mathcal{X}_s$ at round r.

Provers can be validators, full nodes, or other sparse nodes parameterized by the same predicate $\mathcal{X}_s$.

Definition 6 (Sparse Client Protocol Security). *A sparse client protocol Π parameterized by $\mathcal{X}_s$ is* secure *in partially synchronous networks iff, in every partially-synchronous execution with up to $f \leq \tau$ adversarial validators, except with probability* $\mathrm{negl}(\kappa)$*, where κ is the security parameter, the following properties hold:*

Safety: *For every two honest sparse clients i, j and for every two rounds r, r', let $\hat{\mathcal{L}}_i^r$ be the ledger output by i at r and $\hat{\mathcal{L}}_j^{r'}$ the ledger output by j at r'. Then, the two ledgers are consistent, i.e., $\hat{\mathcal{L}}_i^r \preceq \hat{\mathcal{L}}_j^{r'}$, or vice versa.*

Liveness: *For every round r_0 and every transaction* tx*, if $\exists k \in \mathcal{W}(\mathsf{tx}) : \mathcal{X}_s(k)$ and if every honest validator receives* tx *by r_0, then there exists a $r_1 \geq r_0$ such that for every $r_2 \geq r_1$ and for every honest sparse client i,* $\mathsf{tx} \in \hat{\mathcal{L}}_i^{r_2}$.

We recall that at every round r, given state transition function δ, a ledger $\mathcal{L}^r$ and an initial state S^0 uniquely determine the internal state S^r of a validator. Similarly, we want that at every round r, given a modified state transition function $\hat{\delta}$, a ledger $\hat{\mathcal{L}}^r$ and an initial state $\hat{S}^0$ must uniquely determine the internal sparse state $\hat{S}^r$ of a sparse client. By definition, we want $\hat{S}^r = \{(k, v) | (k, v) \in S^r \wedge \mathcal{X}_s(k)\}$.

If we assign a sparse client the same execution semantics of a validator two issues arise. First, a validator evaluates the state transition function over the global state, whereas a sparse client, by construction, only has a partial state. Second, naively replacing the global state with the sparse state—i.e., evaluating $\delta(\hat{S}^{r-1}, \mathsf{tx})$ instead of $\delta(S^{r-1}, \mathsf{tx})$—leads execution to fail when the transaction tx reads from state elements (e.g., gas objects, contracts) that are not in $\hat{S}^{r-1}$. To overcome these issues, we define the execution semantics of a sparse client through a modified state transition function $\hat{\delta}$ derived from the full transition function δ. The function $\hat{\delta}$ takes as input a partial view of the global state together with a transaction tx (domain restriction). It executes tx exactly as δ would, but it is designed to return only the portion of the resulting state that satisfies the client's predicate $\mathcal{X}_s(k)$ (range restriction). If an element was deleted, $\hat{\delta}$ outputs $(k, \perp)$. With this transition rule, the sparse client updates its local state in round r by first augmenting its current sparse state with the transaction's read set and then applying $\hat{\delta}$, i.e., $\hat{S}^r = \hat{\delta}(\hat{S}^{r-1} \cup \mathcal{R}(\mathsf{tx}), \mathsf{tx})$.

3 Sunfish: A Sparse Client Protocol

Our goal is to design a sparse client protocol with the interface defined in Definition 5 and the security guarantees in Definition 6. Crucially, we want our client to operate efficiently without downloading the entire ledger.

In Section 3.1, we describe the Sunfish protocol. In Section 3.2 we discuss how to complement the client with execution. Finally, in Section 3.3, we introduce an *event client*, i.e., a special case of sparse client that only downloads and verifies a specific stream of events emitted by a contract.

3.1 Sunfish Protocol

Sunfish Overview. Our sparse client protocol, called Sunfish, follows the traditional structure of existing blockchain clients. We

Sparse Client Initiation

(1) The client C connects to a set of provers and sends a state predicate $\mathcal{X}_s$ to each prover.

Prover Response Construction

Let $\mathcal{L}$ be the confirmed ledger consisting of $m > 0$ transactions. Upon receiving $\mathcal{X}_s$ from C, each prover does the following:

(2) Computes the sparse ledger $\hat{\mathcal{L}} := \varphi(\mathcal{X}_s, \mathcal{L})$.
(3) It generates a proof π proving that: (a) for all transactions $\mathsf{tx} \in \hat{\mathcal{L}}$, $\mathsf{tx} \in \mathcal{L}$, and (b) for all transactions $\mathsf{tx} \in \mathcal{L}$ s.t. $\exists k \in \mathcal{W}(\mathsf{tx}) : \mathcal{X}_s(k)$, $\mathsf{tx} \in \hat{\mathcal{L}}$.
(4) For all transactions $\mathsf{tx} \in \hat{\mathcal{L}}$, it retrieves the read set $\mathcal{R}(\mathsf{tx})$. We denote this as $\mathcal{R} = [\mathcal{R}(\mathsf{tx}) | \mathsf{tx} \in \hat{\mathcal{L}}]$.
(5) It generates a proof π_{ex} proving that $\mathcal{R}$ is correct.
(6) Sends $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\mathsf{ex}})$ to C.

Sparse Client Response Verification and Transaction Execution

Upon receiving $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\mathsf{ex}})$ from a prover, the client:

(7) If $\hat{\mathcal{L}}$ does not extend the longest sparse ledger seen so far, abort.
(8) Computes $\hat{\mathcal{L}}' := \varphi(\mathcal{X}_s, \hat{\mathcal{L}})$. If $\hat{\mathcal{L}}' \neq \hat{\mathcal{L}}$, abort.
(9) Verifies the proof π. If the proof is not valid, abort.
(10) Verifies $\mathcal{R}$ against π_{ex}. If verification fails, abort.
(11) Executes $\hat{S} = \hat{\delta}(\ldots\hat{\delta}(\hat{\delta}(S^0 \cup \mathcal{R}(\mathsf{tx}_1), \mathsf{tx}_1) \cup \mathcal{R}(\mathsf{tx}_2), \mathsf{tx}_2) \cup \cdots \cup \mathcal{R}(\mathsf{tx}_m), \mathsf{tx}_m)$. If execution fails, abort.
(12) Updates its internal state to $\hat{S}$, and outputs $\hat{\mathcal{L}}$.

Figure 2: Pseudocode for the Sunfish protocol.

present the pseudocode of Sunfish in Figure 2. Sunfish is run between a client and a set of provers, that can be validators, full nodes, or other sparse nodes.[3] To ensure the protocol is live, at least one of the provers must be non-eclipsed, i.e., maintain at least one communication channel with an honest peer, and honest, i.e., follow the correct protocol execution.

The protocol is parameterized by a predicate $\mathcal{X}_s$ and, when the client first connects to the provers, it is only aware of $\mathcal{X}_s$, of the genesis state $\hat{S}^0$, and the identities of the validators at genesis. The client submits $\mathcal{X}_s$ as a query to the provers (Figure 2, step (1)), which respond with the sparse ledger $\hat{\mathcal{L}}$ identified by $\mathcal{X}_s$, and a succinct cryptographic proof π that allows to efficiently verify that $\hat{\mathcal{L}}$ is constructed as per Definition 4 (Figure 2, steps (2) and (3)). To additionally allow the client to execute the transactions in $\hat{\mathcal{L}}$, the provers return $\mathcal{R} = [\mathcal{R}(\mathsf{tx}) | \mathsf{tx} \in \hat{\mathcal{L}}]$, i.e., the sequence of read sets of the transactions in $\hat{\mathcal{L}}$, along with a proof π_{ex} attesting to the correctness of $\mathcal{R}$ (Figure 2, steps (4) and (5)). Upon receiving potentially different $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\mathsf{ex}})$ from the provers, the client independently verifies the proofs and accepts the tuple $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\mathsf{ex}})$ accompanied by valid proofs (Figure 2, steps (7), (8), (9) and (10)). Finally, the client executes the transactions in the sparse ledger and updates its internal state $\hat{S}$ (Figure 2, steps (11) and (12)). This concludes the bootstrapping phase of the protocol.

After bootstrapping, a maintenance phase can start: at every round, the provers possibly send to the client new transactions txs that extend $\hat{\mathcal{L}}$, accompanied by the corresponding read sets and proofs, i.e., $(\mathsf{txs}, \pi', \mathcal{R}', \pi'_{\mathsf{ex}})$. The client verifies the newly received $(\mathsf{txs}, \pi', \mathcal{R}', \pi'_{\mathsf{ex}})$ and, if the proofs are valid and the execution $\hat{S}' = \hat{\delta}(\hat{S} \cup \mathcal{R}', \mathsf{txs})$ succeeds, it appends the new transactions to $\hat{\mathcal{L}}$ and updates its internal sparse state to $\hat{S}'$.

From here onwards, we only focus on the bootstrapping phase of the protocol: once the client is synchronized, each subsequent update can be viewed as a new bootstrapping step starting from the previously synchronized state.

Full and Light Clients: Queries, Proofs, and Digests. While the steps of client protocols remain roughly the same across full, light, and sparse clients (client queries provers, provers generate proofs, client verifies proofs and terminates), what usually varies between clients is the type of query the client makes and the type of response it receives. In turn, the logic run by the prover (that generate the proof) and the client (that verifies it) also changes. Full and light clients only support very specific types of queries: a full node asks for the entire ledger and, upon receiving it, it executes its transactions to compute the associated state. A light client, on the other hand, asks whether a specific transaction is included in the ledger or, alternatively, which is the value (e.g., balance) associated with a particular state element (e.g., an account)—in other words, a light client asks for a commitment to the ledger or to the state, accompanied by a succinct opening proof. To help clients answer these queries, validators augment their ledger with additional metadata: leaders make proposals, called *blocks*, that consist of a payload of transactions and an associated metadata structure, called *header*. Validators confirm blocks. A header typically includes a digest[4] of the transactions in the payload, a digest of the state of the ledger after executing the payload, and the hash of the previous block header. The digest of the transactions, usually in the form of a Merkle root, helps full nodes verify the integrity of the payload received; when combined with a Merkle proof, the transaction digest also helps light clients to efficiently verify transaction inclusion without having to download the entire payload. Similarly, the digest of the state helps full nodes verify the correctness of the entire state computed from executing all transactions, but, when coupled with a proof, also helps light clients to efficiently and reliably read single elements of the state, without having to execute the entire ledger. Finally, cryptographically hash-chaining blocks together ensures that the last confirmed block commits to the entire sequence of blocks, giving clients guarantees about the integrity of the chain (and, therefore, of the corresponding payloads).

Sparse Client: Query, Proofs, and Digests. A sparse client, on the other hand, asks for the sparse ledger $\hat{\mathcal{L}}$ identified by $\mathcal{X}_s$, together with a cryptographic proof π certifying that $\hat{\mathcal{L}}$ is exactly the order-preserving subsequence of transactions obtained by filtering the confirmed ledger $\mathcal{L}$ according to $\mathcal{X}_s$. The proof π must guarantee that $\hat{\mathcal{L}}$ is constructed as per Definition 4. This means that: (a) all transactions in the sparse ledger $\hat{\mathcal{L}}$ are included in the confirmed ledger $\mathcal{L}$ in the same relative order, and (b) the sparse ledger $\hat{\mathcal{L}}$ is complete, i.e., all transactions surviving the filtering $\varphi(\mathcal{X}_s, \mathcal{L})$ appear in $\hat{\mathcal{L}}$ (Figure 2, step (3)). The condition (a) requires proving that each transaction appears in a confirmed block and that the relative order between transactions is preserved. This can be achieved in three steps: proving that each transaction is included in

[3] If the sparse client connects to validators or other sparse nodes (parameterized by the same predicate), its synchronization time is much faster, as it is not bottlenecked by the one of full nodes.

[4] A digest is a fixed-length, compact representation (or fingerprint) of the data.

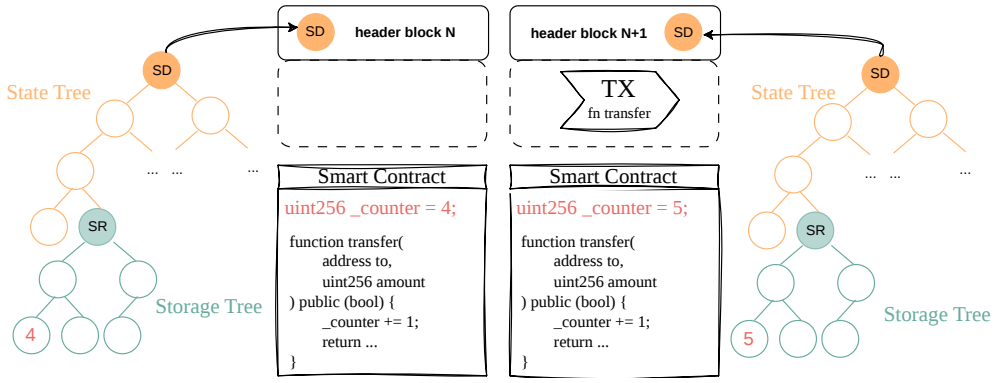


Figure 3: EVM-based variant. A contract stores a counter (red) in its storage tree. A transaction in block N+1 calls the transfer function of a contract and increments the counter by 1, i.e., from 4 to 5. The counter's value affects the storage root (SR), which is a leaf of the state tree. In turn, the counter value affects the state digest (SD) in block headers.

a block, proving that the relative order of transactions in $\hat{\mathcal{L}}$ matches their order in the sequence of blocks, and proving that the block is confirmed. In this work, we do not discuss how to prove that a block is confirmed: solutions to this have been extensively studied in the context of light clients [1, 4, 7, 11, 15, 20, 29, 31] and the proposed mechanisms can be integrated into Sunfish out-of-the-box.

The condition (b) requires proving to the client that the sparse ledger is complete, i.e., that no relevant transaction has been withheld by a malicious prover. Verifying completeness allows the client to rely on provers only for the liveness of the protocol, not for its safety. This is important in real-world deployments, as it allows the client to connect to a single prover, as availability of provers may be scarce, and avoids that the resource consumption of the client grows with the number of provers. Should the prover fail to respond or provide old blocks (blocks with an old timestamp), the client can switch to a different prover. To prove completeness, we cannot naively use existing digests: their opening proofs only allow the client to verify completeness by downloading the entire ledger. Instead, we want the provers to build a proof that allows the client to verify completeness by only receiving the sparse ledger accompanied by a succinct proof. To prove (a) and (b), we propose two different approaches. The first, which we call the *EVM-based Sunfish variant*, leverages the digests and structures of EVM-based chains to extract useful information. The second, which we call the *chain-agnostic Sunfish variant*, introduces a brand new digest to the block headers and it is virtually compatible with any blockchain.

EVM-Based Variant: Counter-Based Proof. EVM-based chains, pioneered by Ethereum, are the ones hosting the greatest variety of decentralized applications (dApps) and, therefore, having the most interesting and hot sparse states. These chains have an account-based model, a quasi-Turing-complete execution and, in the header of their blocks, include a digest of the block's payload, and a digest of the state after the payload execution. EVM chains feature two types of accounts: *externally owned accounts* (EOAs), controlled by a user and its secret key, and *contract accounts*, associated with executable code and persistent storage. Contract accounts are not controlled by a private key: their behavior is entirely determined by the deployed code, whose functions can be invoked through transactions or inter-contract calls. In this discussion, we focus on a sparse client whose predicate $\mathcal{X}_s$ identifies the state of a contract account, as this is the most interesting use-case: the state of a contract typically holds large amounts of assets and its correctness may need to be independently audited and verified by third parties such as external observers or regulators. It is also more difficult to track, as many independent users can collectively modify it. For EOAs, often a light client suffices, as the owner has direct knowledge of outgoing and incoming transactions and the associated state; if one wants to run a sparse client for an EOA, completeness can be ensured by authenticating the value of the nonce (field of the EOA account that gets incremented by 1 upon every successful outgoing transaction) and the balance of the account, and making sure that after execution of the transactions in $\hat{\mathcal{L}}$, the resulting nonce and balance match the ones authenticated against the header.

To prove that each transaction appears in a confirmed block (condition (a), first step), we can use the standard Merkle inclusion proof used by light clients. This proof has logarithmic size in the number of transactions in the block. Because a transaction Merkle proof certifies the membership and position of a transaction in a block and blocks are ordered by height, this proof also proves the relative order between transactions (condition (a), second step).

Let us now discuss how to prove completeness (condition (b)). The state of a contract account includes a field called *storage root*, which is a digest that commits to the contract's persistent storage. As shown in Figure 3, the value of the storage root influences the state digest in the headers; therefore, we can use it to verify the completeness of the set of transactions that modify the state of the contract. However, we cannot use this digest naively: for instance, the client cannot simply execute the received transactions in $\hat{\mathcal{L}}$ and check whether the resulting state of the contract authenticates against the state digest in the header. This is because there may be transactions that, within the same block, have mutually cancelling effects on the contract state, such that intermediate state modifications occur but the final state reverts to its original value, thereby leaving no net change in the contract storage (e.g., a transaction update a variable from 2 to 5, and a subsequent one updates back it from 5 to 2). To wisely use the storage root, we require the contract to maintain a persistent, monotonically increasing integer counter that keeps track of the number of its successful state-modifying function calls;[5] this is depicted in red in Figure 3.

[5]To check whether a state-modifying function call was successful, transaction receipts (or equivalent execution status data) for the included transactions are needed. Transaction receipt inform whether a transaction succeeded or was reverted. Only successful transactions contribute to the counter

Algorithm 1 EVM-based variant. Sparse client algorithm.

```
ctr ← 0
L̂ ← [ ]
Ŝ ← Ŝ^0
X_s ⇢ P                                          ▷ Send predicate to prover
(L̂′, π, R, π_ex) ⇠ P                             ▷ Receive data from prover
if L̂′ ≠ φ(X_s, L̂′) then                          ▷ Verify filtering
    abort
(π^incl, π^ctr) ← π          ▷ Decompose into inclusion and counter proofs
for tx ∈ L̂′ do                                   ▷ Check transaction inclusion
    if ¬VerifyInclusion(tx, π^incl_tx) then
        abort
if ¬VerifyExecutionMetadata(R, π_ex) then
    abort
(ok, Ŝ′, ctr) ← Execute(Ŝ, L̂′)
if ¬ok then
    abort
ctr′ ← ReadAndAuthenticateCtr(π^ctr)
if ctr ≠ ctr′ then                ▷ Compare counters (completeness check)
    abort
L̂ ← L̂′                                           ▷ Update local sparse ledger
Ŝ ← Ŝ′                                           ▷ Update local sparse state
return L̂                                         ▷ Output sparse ledger
```

The contract design must ensure that if a transaction does not update the contract's state, the value of the counter remains the same; if a transaction updates the state of the contract, the value of the counter is incremented by at leat 1. In particular, a single transaction may trigger multiple counted calls (e.g., multiple invocations of state-modifying functions) and increase the counter by more than one. This counter is initialized at 0 when the contract is deployed and it is incremented by 1 upon every successful counted call. The counter's value is stored as part of the contract's persistent state and it affects the value of the storage root. In turn, as illustrated in Figure 3, it affects the value of the state digest included in the block headers. Therefore, the Sunfish completeness proof consists of the value of the counter together with a proof that authenticates it against the state digest in the block header. The client can verify completeness by executing the transactions in $\hat{\mathcal{L}}$, computing the resulting counter value, and checking whether it matches the one authenticated against the header. We showcase the algorithms run by the Sunfish client and prover in Algorithm 1 and Algorithm 2, respectively.

While this approach keeps the completeness proof logarithmic in the size of the entire state, it requires contracts to implement a counter that increments by 1 at every invocation of a state-modifying function. This requires contracts to be implemented ad-hoc to support this interaction counter mechanism. We note that this is a small one-time cost for contract developers that wants to enjoy the benefits of verifiably tracking the state of the contract while isolating the workload of the dApp from the workload of the entire chain. We also observe that updating the counter introduces a small gas overhead for end-users on each transaction. This overhead could be mitigated, for example, by contract developers subsidizing the additional gas cost or by deploying the contract on a layer-2 network with lower fees. We leave the exploration of these options to future work.

Algorithm 2 EVM-based variant. Prover algorithm.

```
S ← S^0
π^incl ← [ ], R ← [ ]
X_s ⇠ C                                          ▷ Receive predicate from client
L ← GetLedger()                                  ▷ Get confirmed ledger from network
(R′, π_ex) ← ExecuteWithTrace(S, L)              ▷ Authenticate execution metadata
L̂ ← φ(X_s, L)                                    ▷ Filter ledger by predicate
for tx ∈ L̂ do
    π^incl_tx ← GenInclusionProof(tx)            ▷ Generate inclusion proof
    π^incl ← π^incl ‖ π^incl_tx
    R_tx ← GetReadSet(R′, tx)                    ▷ Get read set from execution trace
    R ← R ‖ R_tx
header ← GetLastConfirmedHeader(L)
π^ctr ← AuthenticateCounter(header)              ▷ Counter proof
π ← (π^incl, π^ctr)
(L̂, π, R, π_ex) ⇢ C                              ▷ Send data to the client
```

Algorithm 3 Chain-agnostic variant. Sparse client algorithm.

```
h ← 0
L̂ ← [ ]
Ŝ ← Ŝ^0
X_s ⇢ P                                          ▷ Send predicate to prover
(L̂′, π, R, π_ex) ⇠ P                             ▷ Receive data from prover
if L̂′ ≠ φ(X_s, L̂′) then                          ▷ Verify filtering
    abort
for tx ∈ L̂′ do
    h ← H(h, tx)                                 ▷ Compute hash chain head
h* ← ReadAndAuthenticateHead(π)
if h ≠ h* then                                   ▷ Compare hash chain heads
    abort
if ¬VerifyExecutionMetadata(R, π_ex) then
    abort
(ok, Ŝ′) ← Execute(Ŝ^0, L̂′)
if ¬ok then
    abort
L̂ ← L̂′                                           ▷ Update local sparse ledger
Ŝ ← Ŝ′                                           ▷ Update local sparse state
return L̂                                         ▷ Output sparse ledger
```

Chain-Agnostic Variant: Hash Chain-Based Proof. The EVM-based variant is not compatible with many types of chains: for performance reasons, validators of high throughput chains do not include in the block headers a state digest that allows for efficient opening proofs (e.g., Solana, Sui). This is often a deliberate design choice to optimise performance, as committing to the full chain state at each block or checkpoint introduces additional latency. To unlock completeness proofs for these chains, we introduce a chain-agnostic variant that is designed to be virtually compatible with any blockchain—even those that have limited scripting capabilities such as Bitcoin.[6] As we will see in the following paragraphs, this variant requires validators to perform a very cheap extra task.[7]

[6] Our model is tailored to BFT-based protocols. With appropriate modifications—e.g., synchronous network, probabilistic confirmation rule—it can be generalized to longest-chain protocols as well.

[7] This is not unprecedented: in modern blockchains, validators have taken on responsibilities beyond block proposal and voting (e.g., serving as storage nodes for data availability, issuing slashing requests, and participating in randomness generation).

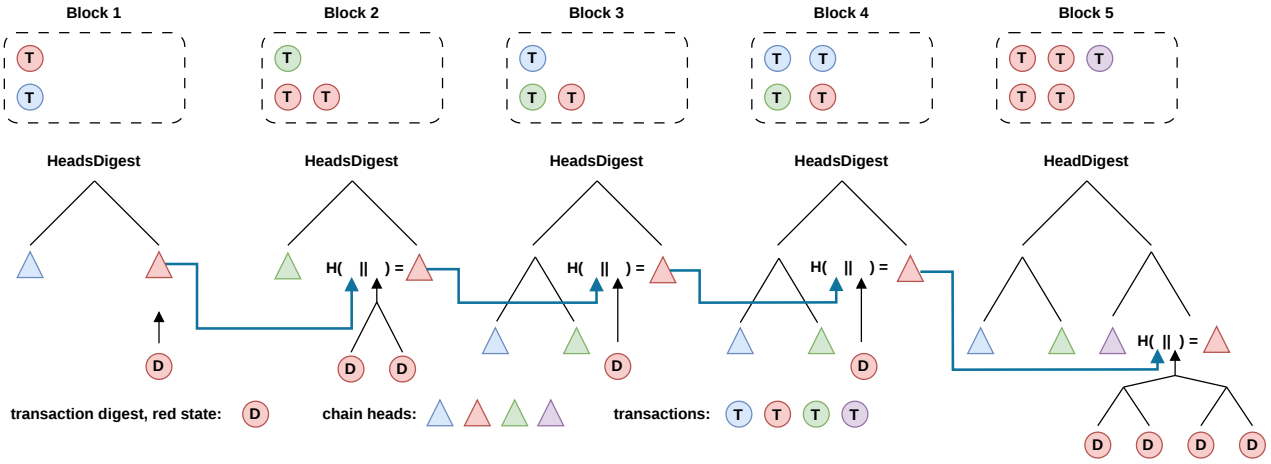


Figure 4: Chain-agnostic variant. Consider the blue, green, red, and violet substates. We expand the data structure for the red sparse state; the others are analogous. When a red transaction appears in a block, validators build a tree out of the red transactions in the block. The root of the tree is an element of the red hash chain, whose head is an entry of an overlay tree of chain heads. Periodically, validators include in the overlay tree the chain heads for the blue, green, and violet sparse states (block 5).

Algorithm 4 Chain-agnostic variant. Prover algorithm.

$S \leftarrow S^0$
$\mathcal{R} \leftarrow [\,]$
$h \leftarrow 0$
$\mathcal{X}_s \dashleftarrow C$ ▷ Receive predicate from client
$\mathcal{L} \leftarrow \textsc{GetLedger}()$ ▷ Retrieve confirmed ledger from network
$\hat{\mathcal{L}} \leftarrow \varphi(\mathcal{X}_s, \mathcal{L})$ ▷ Filter ledger by predicate
$(\mathcal{R}', \pi_{\text{ex}}) \leftarrow \textsc{ExecuteWithTrace}(S, \mathcal{L})$ ▷ Authenticate execution metadata
for tx $\in \hat{\mathcal{L}}$ **do**
  $\mathcal{R}_{\text{tx}} \leftarrow \textsc{GetReadSet}(\mathcal{R}', \text{tx})$ ▷ From execution trace
  $\mathcal{R} \leftarrow \mathcal{R} \| \mathcal{R}_{\text{tx}}$
  $h \leftarrow H(h, \text{tx})$ ▷ Compute hash chain head
header $\leftarrow \textsc{GetLastConfirmedHeader}(\mathcal{L})$
$\pi \leftarrow \textsc{GenHeadProof}(\text{header}, \mathcal{X}_s, h)$ ▷ Prove head inclusion in overlay tree digest
$(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\text{ex}}) \dashrightarrow C$ ▷ Send data to the client

This variant requires adding to block headers a new digest that allows to verify the sparse ledger completeness (condition (b)) and, simultaneously, the transaction inclusion and ordering (condition (a), first and second step). As depicted in Figure 4, for each block, validators build a Merkle tree with the transactions that modify a given sparse state (red sparse state in Figure 4). The roots of these trees are then linked via a hash chain to produce a chain head for that sparse state. Since a blockchain may support multiple sparse clients simultaneously, each associated with a different $\mathcal{X}_s$, validators construct an overlay tree whose leaves are tuples $(\text{id}\hat{\text{S}}, \text{head})$, where $\text{id}\hat{\text{S}}$ denotes the identifier of the sparse state and head the corresponding chain head. Only sparse states that have state-modifying transactions in the block appear in this overlay tree. Finally, the Merkle root of the overlay tree is included in the block header. Provers can now provide the client with a proof of inclusion of the head in the head digest. The client can compute the chain head locally from $\hat{\mathcal{L}}$, and accepts $\hat{\mathcal{L}}$ only if the value of the head matches the one authenticated against the head digest. The algorithms run by the client and prover are shown in Algorithm 3 and Algorithm 4, respectively.

To enable efficient bootstrapping of the client, validators can periodically include (e.g., once a day, or every 500 blocks) in the overlay tree a leaf for each supported sparse state, even if no state-modifying transactions are included in the block—in Figure 4, block 5 includes the blue and green sparse states even though no blue or green transactions are included in the block. In this way, a sparse client can periodically authenticate the chain head of its sparse state from a recent block header: this can be particularly useful in the case the client monitors a sparse state that does not have a lot of traffic. This approach can also easily support non-inclusion proofs, so that a client can verify for its sparse state the absence of state-modifying transactions in a block. To enable non-inclusion proofs, validators can build the overlay tree so that the leaves are lexicographically sorted by $\text{id}\hat{\text{S}}$. These proofs are particularly useful during the maintenance phase, when the client processes incremental updates: rather than inferring the absence of state-modifying transactions from the lack of an update, the client can verify it via a non-inclusion proof. The completeness proof size is logarithmic in the number of sparse states modified in the last finalized block.

While validators could theoretically support arbitrary many sparse states, in practice, they may restrict the granularity of supported sparse states (e.g., via subscriptions or only considering hot states) to avoid unbounded growth.

Comparing the Two Variants. Table 1 summarizes the asymptotic size of data downloaded by the Sunfish client for each variant. By $|\hat{\mathcal{L}}|$ we denote the number of transactions in the sparse

Table 1: Asymptotic size of data downloaded by the Sunfish client for each variant. Here M is the average number of transactions per block, Q is the average number of sparse states modified in a block, $|S|$ is the size of the global state, and $\psi|S|$ is the average size of state read by transactions.

Component	EVM-based	Chain-agnostic
Sparse ledger	$O(\lvert\hat{\mathcal{L}}\rvert)$	$O(\lvert\hat{\mathcal{L}}\rvert)$
Read sets and proof π_{ex}	$O(\psi\lvert S\rvert)$	$O(\psi\lvert S\rvert)$
Inclusion proofs	$O(\lvert\hat{\mathcal{L}}\rvert \log M)$	–
Completeness proof	$O(\log \lvert S\rvert)$	$O(\log Q)$

ledger, by $|\hat{S}|$ the number of state elements in the sparse state, and by $\psi|S|$ the average size of state read by transactions. Both variants download the sparse ledger and the read sets with their proof π_{ex}, each of size $O(|\hat{\mathcal{L}}|)$ and $O(\psi|S|)$ respectively. The EVM-based variant requires inclusion proofs of size $O(|\hat{\mathcal{L}}| \log M)$, one per transaction (with M the average number of transactions per block), while the chain-agnostic variant needs none, as the hash chain commits to the transaction sequence and proves inclusion implicitly. For completeness, the EVM-based variant uses a single counter authenticated against the state digest, yielding $O(\log |S|)$; the chain-agnostic variant proves inclusion of the chain head in the overlay tree, yielding $O(\log Q)$ (with Q being the number of sparse states modified in a block). In total, the EVM-based client downloads $O(|\hat{\mathcal{L}}| + \psi|S| + \log |S| + |\hat{\mathcal{L}}| \log M)$ and the chain-agnostic one $O(|\hat{\mathcal{L}}| + \psi|S| + \log Q)$.

We observe that the EVM-based variant requires downloading more data, but it is also out-of-the-box compatible with most EVM-based chains. In contrast, the chain-agnostic variant demands that validators add a new digest to block headers and maintain the overlay tree of hash chain heads, which may require a protocol upgrade.

3.2 Sunfish Execution

We equip the Sunfish client with the ability to execute the transactions of the sparse ledger $\hat{\mathcal{L}}$, so that it can locally compute the corresponding sparse state $\hat{S}$ and, if the computation succeeds, update its internal state.

Requiring the client to execute the transactions of $\hat{\mathcal{L}}$ gives a fundamental advantage: increased security. In fact, by locally executing the transactions, the sparse client is always guaranteed to *accept valid state transitions for the partial state it monitors, even in the presence of a temporary violation of the chain's adversarial threshold.* This guarantee follows from the fact that the client detects and rejects invalid transactions that attempt at modifying its partial state (e.g., transactions that create coins out of thin air), it detects and rejects reorganizations of its partial state, and, in the presence of forks, it always maintains a consistent partial state—this property is sometimes called *fork consistency* [14, 23]. During a temporary violation of the chain's adversarial threshold, this makes the security of the sparse client a middle ground between the security of a full node, which only accepts valid state transitions for the entire state, and the security of a light client, which can be fooled into accepting invalid state transitions. This new flavor of security offered by the sparse client, which only accepts valid state transitions for its partial state, is particularly meaningful for dApp developers and operators who are exclusively interested in a valid snapshot of the dApp state that they can migrate to a different chain should the chain be compromised.

We highlight that because the sparse client is solely interested in a specific partial state, its security does not degrade with the portion of external state accessed during execution. For typical DeFi applications such as Uniswap or Aave, the portion of the read set that lies outside the protocols' contracts is limited to well-known ERC-20 token contracts and price oracles. These dependencies are often relatively stable over time and one can extend the sparse client to also monitor the state of those external contracts.

Authenticating the Transaction Read Set. We recall from Section 2 that the sparse client state transition is expressed as $\hat{S}^r = \hat{\delta}(\hat{S}^{r-1} \cup \mathcal{R}(\text{tx}), \text{tx})$. In general, execution requires the client to obtain from provers the read set of transactions that access portions of the state external to the current sparse state. To prevent the client from accepting invalid transactions, the client must be able to verify that the provided read set faithfully reflects the external state accessed during execution. Authenticating a transaction read set is straightforward in UTXO-based systems, where transactions include commitments to their inputs. In account-based systems, such as EVM-based chains, the set of accessed state is determined dynamically during execution. This is not a problem: these chains include state commitments in block headers and authenticating the read set can be done indirectly by verifying that the post-execution state matches the committed state root. Therefore, explicit read-set authentication becomes primarily relevant only for chains that do not expose state commitments in their headers. We note that the next Ethereum upgrade will introduce block-level access lists (EIP-7928 [32]) that requires the leader to authenticate the read set of all transactions in a block by committing, at the block level, to all accounts and storage keys accessed during execution. Once this commitment is included in the block header, sparse clients can directly authenticate the transactions' read sets.

3.3 Event Client

While a sparse client requires authenticated access to a subsequence of transactions to verifiably reconstruct a subset of the state of the ledger, an event client is concerned only with *events* emitted during execution, i.e., a *subset of the log metadata.* An event client can be formalized as a simpler variant of a sparse client. Intuitively, given as input to the client a predicate over a stream of log metadata, the event client outputs the corresponding stream of events. The client security of Definition 6 can be adapted accordingly.

The protocol of an event client needs to be carefully designed to allow authentication of events: because events are not part of the ledger, they have no inclusion proofs, nor they need to be executed. While the chain-agnostic variant of Sunfish can be used out-of-the-box by the event client by replacing transactions with events, the counter mechanism discussed in the EVM-based variant of Sunfish requires modifications. An EVM-based event client can still use the storage root of a contract, but in a different way: instead of using an interaction counter, the contract can store the

head of the hash chain of events emitted by the contract. For instance, let $(e_1, e_2, \ldots, e_k)$ be the sequence of events emitted by the contract. The head h_k of the hash chain is the result of computing $h_i = H(h_{i-1} \,||\, e_i)$ for $i = 1, 2, \ldots, k$, with h_0 being some fixed initialization constant (e.g., a zero-value). By construction, h_k commits to the entire sequence $(e_1, e_2, \ldots, e_k)$. By having the head persistently stored in the contract state, the client can verify the integrity and completeness of the stream of events by locally hash-chaining the sequence of events received from provers and verifying that the resulting head matches the one committed in the state digest included in the block header.

The primary advantages of an event client are efficiency and modularity. Because it processes only events, it significantly reduces bandwidth, computational, and storage requirements compared to a sparse client. Moreover, event clients are particularly useful for applications whose functionality depends primarily on observing specific contract signals, rather than reconstructing full contract state. For example, bridges and decentralized exchanges often rely on events such as "lock", "mint", or "swap" to trigger off-chain or cross-chain actions, or update databases. In this case, processing the entire ledger or storing the state is unnecessary; what matters is reliably detecting all relevant events in the chosen stream.

Inspired by this work, the Sui blockchain is integrating the chain-agnostic variant of our Sunfish event client into its protocol [3].

4 Analysis

We prove security of the Sunfish protocol. Let $\mathcal{L}$ be the output of a secure SMR protocol as per in Definition 1 and Definition 2.

Lemma 1. *For any predicate $\mathcal{X}_s$ identifying the state of a contract account in an EVM-based chain, a Sunfish sparse client that implements Algorithm 1 is guaranteed to output a sparse ledger $\hat{\mathcal{L}}$ that is a prefix of $\varphi(\mathcal{X}_s, \mathcal{L})$.*

Proof of Lemma 1 is deferred to Section A.

Lemma 2. *For any predicate $\mathcal{X}_s$ supported by the system, a Sunfish sparse client that implements Algorithm 3 is guaranteed to output a ledger $\hat{\mathcal{L}}$ that is a prefix of $\varphi(\mathcal{X}_s, \mathcal{L})$.*

Proof. Consider a chain that implements the hash chain-based mechanism described in Section 3. Also consider a sparse client parameterized by $\mathcal{X}_s$ that has received from a prover a tuple $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\text{ex}})$. The proof π authenticates the value of the head of the hash chain that commits to the sequence of transactions in $\hat{\mathcal{L}}$ against the head digest in the header of a recent confirmed block. Because the block and its header are confirmed by the validators via consensus, and because consensus is a secure SMR protocol, the head digest in the header is correct: i.e., it commits to the value of the head. By construction, the authenticated head in the header is the hash-chain commitment to the filtered sequence $\varphi(\mathcal{X}_s, \mathcal{L})$, where $\mathcal{L}$ is the confirmed ledger pinned by that header. The client locally recomputes the hash chain from the transactions in $\hat{\mathcal{L}}$ (Algorithm 3, line 9) and compares the resulting chain head with the value authenticated against the header (Algorithm 3, lines 10–12). Because of the collision and second-preimage resistance of the hash function, except with negligible probability, the two chain heads match only if the underlying sequences match, i.e., $\hat{\mathcal{L}} = \varphi(\mathcal{X}_s, \mathcal{L}')$. Because the client only outputs $\hat{\mathcal{L}}$ if the locally computed value of the chain head matches the one authenticated against the head digest in the header, it is ensured to output a sparse ledger $\hat{\mathcal{L}}$ that is a prefix of $\varphi(\mathcal{X}_s, \mathcal{L}')$. □

Theorem 1. *The Sunfish protocol for sparse clients is secure as per Definition 6.*

Proof. **Safety:** Let i, j be two honest sparse clients and let $\hat{\mathcal{L}}_i^r$ be the ledger output by i at round r and $\hat{\mathcal{L}}_j^{r'}$ the ledger output by j at round r'. We need to prove that for every i, j parameterized by $\mathcal{X}_s$ and for every r, r', the two ledgers are consistent, i.e., $\hat{\mathcal{L}}_i^r \preceq \hat{\mathcal{L}}_j^{r'}$, or vice versa.

Suppose, by contradiction, that there exist two Sunfish clients i, j parameterized by $\mathcal{X}_s$ and reading from the same chain, that, for some rounds r, r' output two inconsistent sparse ledgers: i.e., $\hat{\mathcal{L}}_i^r \not\preceq \hat{\mathcal{L}}_j^{r'}$ or $\hat{\mathcal{L}}_j^{r'} \not\preceq \hat{\mathcal{L}}_i^r$. Without loss of generality, suppose $\hat{\mathcal{L}}_i^r \not\preceq \hat{\mathcal{L}}_j^{r'}$; the case $\hat{\mathcal{L}}_j^{r'} \not\preceq \hat{\mathcal{L}}_i^r$ proceeds analogously. For $\hat{\mathcal{L}}_i^r \not\preceq \hat{\mathcal{L}}_j^{r'}$ to happen, one of the following must be true: (a) the client i has output a sparse ledger $\hat{\mathcal{L}}_i^r$ that is not constructed as per Definition 4, (b) the client j has output a sparse ledger $\hat{\mathcal{L}}_j^{r'}$ that is not constructed as per Definition 4. Both cases contradict either Lemma 1 or Lemma 2. This concludes the proof.

Liveness: We need to prove that for every round r_0 and every transaction tx, if $\exists k \in \mathcal{W}(\text{tx}) : \mathcal{X}_s(k)$ and if every honest validator receives tx by r_0, then there exists a $r_1 \geq r_0$ such that for every $r_2 \geq r_1$ and for every honest sparse client i parameterized by $\mathcal{X}_s$, $\text{tx} \in \hat{\mathcal{L}}_i^{r_2}$.

By assumption, the Sunfish client connects to at least one honest, non-eclipsed prover. Because the network is partially synchronous, the honest prover eventually receives $\mathcal{X}_s$ from the client (Algorithm 2, line 3 and Algorithm 4, line 4). Let a prover be a validator or full node that, by running the SMR client interface, maintains an output confirmed ledger $\mathcal{L}$ consistent with honest clients' ledgers. Because of the liveness of the SMR protocol (Definition 2), we know that for every round r_0 and every transaction tx such that $\exists k \in \mathcal{W}(\text{tx}) : \mathcal{X}_s(k)$, there exists a $r_1 \geq r_0$ such that for every $r_2 \geq r_1$ and for every honest prover i, $\text{tx} \in \mathcal{L}_i^{r_2}$. Therefore, by r_2, the honest prover obtains the confirmed ledger $\mathcal{L}^{r_2}$ that includes tx (Algorithm 2, line 4 and Algorithm 4, line 5). Then, the prover locally generates $\pi, \mathcal{R}, \pi_{\text{ex}}$ and sends $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\text{ex}})$ to the client (Algorithm 2, line 15 and Algorithm 4, line 14). Because the prover is honest and the network is partially synchronous, the client receives $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\text{ex}})$ from the honest prover (Algorithm 1, line 5 and Algorithm 3, line 5) by $r_3 \geq r_2$. Because the prover is honest, the client successfully verifies π and π_{ex} (Algorithm 1, lines 9–19 and Algorithm 3, lines 13–14), successfully executes the transactions (Algorithm 1, line 14 and Algorithm 3, line 15), updates its internal state (Algorithm 1, line 21 and Algorithm 3, line 19), and outputs $\hat{\mathcal{L}}^{r_3}$ (Algorithm 1, line 22 and Algorithm 3, line 20). Therefore, for every $r_3 \geq r_2$, tx appears in the sparse ledger $\hat{\mathcal{L}}_i^{r_3}$ output by the client. □

5 Evaluation

This section evaluates Sunfish sparse and event clients for EVM chains, for which we measure the gas cost incurred by the users,

the reduced resource consumption of the clients, and evaluate the reduced operational costs. The evaluation of the *chain-agnostic* design on Sui is summarized at the end of this section and reported separately in Section B.

Implementation. We implemented a proof-of-concept of both a sparse and event client to demonstrate the feasibility of the Sunfish protocol. The implementation targets Ethereum and relies on Solidity smart contracts to support sparse monitoring through two primitives: an interaction counter for state-modifying transactions and a hash-chain mechanism that commits to emitted events. The prototype client is implemented in Go and interfaces with Ethereum through the Geth libraries and the JSON-RPC API. The complete proof-of-concept implementation is publicly and anonymously available here as open source to support reproducibility and further experimentation.

Quantifying Gas Overhead. The Sunfish sparse and event clients for EVM-based chains require smart contracts to store two additional values in persistent storage: an interaction counter and the head of a hash chain, respectively. As a result, users interacting with these contracts incur slightly higher gas costs, while node operators benefit from reduced costs when running sparse or event clients. Given the importance of keeping transaction fees low, we quantify this overhead.

In Ethereum, updating the interaction counter requires one storage read and one storage write for each state-modifying function call. This introduces an overhead of approximately 5,154 gas per transaction. On the other hand, updating the chain head requires reading the current head, computing the new head, and writing it to storage. While the cost of computing the new head depends on the size of the event data, storage operations dominate the gas overhead. Table 2 reports the relative gas overhead introduced by the counter (sparse client) and chain head (event client) mechanisms across representative Ethereum applications. Overall, the additional cost remains moderate, ranging between 4% and 16% depending on the application and monitoring scope.

We compute concrete numbers for three different popular applications on Ethereum. For a simple ERC-20 transfer (USDT), both mechanisms introduce an overhead of about 11–12%, reflecting the relatively low baseline gas cost of token transfers. For auctioning a CryptoKitties token, the overhead is smaller: approximately 7% for the interaction counter and 4% for the chain head, as the more complex transaction logic amortizes the fixed storage costs. For DEX token swaps, we consider Uniswap V2, which exhibit the widest range, with an overhead between 8–16% for the interaction counter and 5–13% for the chain head. The upper bound corresponds to monitoring the multiple contracts involved in the swap, while the lower bound applies when only the core Uniswap contracts are monitored. These results indicate that the relative impact decreases as transaction complexity increases and that storage operations remain the dominant source of additional gas consumption.

Resource Consumption. We benchmark Sunfish sparse and event clients against a full node (no pruning) to quantify savings in terms of bandwidth, storage, and computational resources.

To model an intense yet realistic workload, we analyze Ethereum Mainnet activity and identify Uniswap V2 as the dominant decentralized exchange. Over a one-week period (September 14–21, 2025),

Table 2: Relative gas overhead introduced by the interaction counter (sparse client) and chain head (event client) mechanisms across representative Ethereum dApps.

Application	Relative Increase	
	Counter	Chain Head
USDT (ERC-20 Transfer)	≈ 11%	≈ 12%
CryptoKitties (Auction Creation)	≈ 7%	≈ 4%
Uniswap V2 (Token Swap)	≈ 8-16%	≈ 5-13%

it accounted for more than 50% of all DEX transactions, while a one-year analysis (September 1, 2024–September 1, 2025) shows that it represents approximately 7.8% of total Mainnet activity—making it a conservative worst-case target for contract-specific monitoring. Other applications typically generate substantially lower load. To reproduce this workload, we deploy a simplified Uniswap V2-style contract on Sepolia testnet, consisting of two ERC-20 token contracts and an exchange contract. We scale transaction activity proportionally to match mainnet traffic volume and execute 3,190 transactions within one hour, achieving 103% of the target rate. All experiments run on a single Ubuntu 24.04.1 VM. We deploy a fully synchronized full node using Geth (`v1.16.2`) with Prysm (`v6.0.4`) and evaluate both sparse and event clients, monitoring the exchange and token contracts in each case.

We measure the client's inbound bandwidth, storage growth, and transaction execution counts during the benchmark interval, capturing the dominant operational costs. Our results are shown in Figure 5: the event client reduces inbound bandwidth by 95.2% compared to a full node, as it downloads only block headers, relevant event logs, and a proof authenticating the hash chain value. Both client configurations achieve more than a four-order-of-magnitude reduction in storage compared to a full node. Moreover, the sparse client executes only transactions relevant to the monitored contracts, resulting in a 92% reduction in computational workload; for the event client, there is no execution. We highlight that our sparse client prototype consumes 150% of the bandwidth of a full node: since the Ethereum JSON-RPC API provides no mechanism to selectively download transactions of interest, our client must download entire blocks along with the necessary proofs, resulting in bandwidth consumption exceeding that of a full node.

Operational Costs. We compare the USD costs of running the Sunfish sparse and event clients against a standard Ethereum full node using current Amazon Web Service (AWS) pricing. The results are shown in Table 3. A full node, provisioned according to Geth's Mainnet recommendations (quad-core CPU, 16 GB RAM, and up to 2 TB storage), incurs an estimated monthly cost of USD 243, driven primarily by storage and bandwidth requirements. The selective data retrieval of the Sunfish event client dramatically reduces resource consumption. A `t4g.large` instance (2 vCPUs, 8 GB RAM) suffices, with inbound bandwidth reduced to roughly 1 GB per month (5% of the one of a full node) and storage limited to approximately 24 GB after one month. The resulting monthly cost is a USD 37, a 85% reduction relative to a full node. We provision the Sunfish sparse client with a `t4g.xlarge` EC2 instance to support transaction filtering and selective execution. Due to the lack

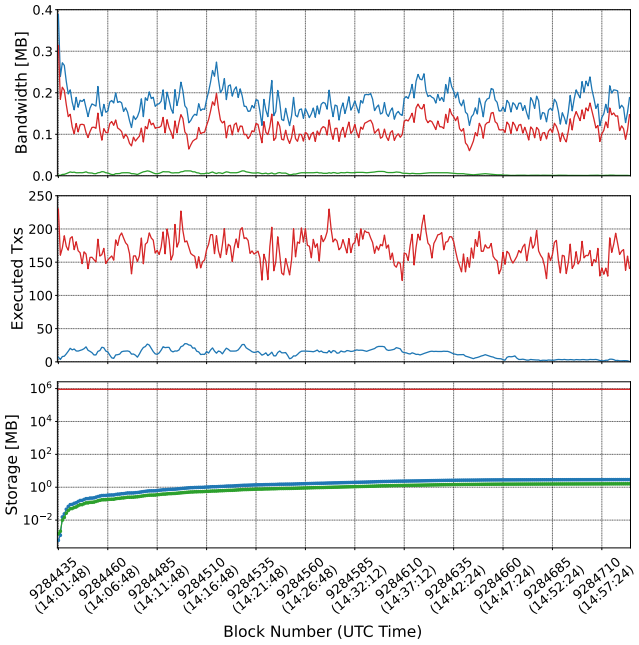


Figure 5: Bandwidth (top), computation (middle), and storage (bottom) comparison between the Sunfish sparse client (—), Sunfish event client (—), and a full node (no pruning, —).

Table 3: Estimated monthly operating costs of Ethereum client configurations on AWS, including EC2 compute, EBS storage, and inbound network traffic. Percentage savings are reported relative to the full node baseline.

Client	EC2	EBS	Inbound data	Costs	Saving
Full	t4g.xlarge	2 TB	19 GB	USD 243	0%
Sparse	t4g.xlarge	160 GB	28.5 GB	USD 83	66%
Event	t4g.large	24 GB	1 GB	USD 37	85%

of appropriate API in Ethereum, the client needs to download the entire blocks, causing bandwidth to increase: 150% of the one of a full node. Even so, the estimated monthly cost is USD 83, a 66% reduction compared to the USD 243 baseline for a full node.

Evaluation on the Sui Blockchain. We also evaluate the Sunfish *chain-agnostic* design on Sui, a high-throughput blockchain; the full analysis is reported in Section B. On a full day of Sui mainnet, we estimate the bandwidth required by a sparse node to track a single application. For a high-traffic dApp such as the Wormhole bridge, our Sui measurements indicate that a Sunfish client downloads only a small fraction of full node data: $\sim 3\%$ in the event client configuration and up to $\sim 7\%$ in the sparse client mode. For a low-traffic target such as an address-based wallet, the required bandwidth is essentially negligible, $\sim 10^{-7}\%$ of the full node bandwidth.

6 Related Work

Full and Light Clients. Sunfish bridges the gap between full nodes [2, 16, 27] and light nodes [1, 4, 7, 11, 15, 20, 29, 31]. Unlike full nodes, Sunfish does not require to download and 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 execution. Some light client designs [22, 33] consider completeness as an important property, however, they achieve it by relying on trusted execution environments [26] and do not consider execution. Other light clients [6] 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 [6] in scope and functioning: it operates stand-alone by reading the chain and maintaining a valid substate of the global state of the ledger.

Partially-Stateless Clients. A client similar to a sparse node is mentioned in an independent blog post by Vitalik Buterin [13], co-founder of Ethereum, under the name of *partially-stateless client*. The post does not provide a formalization, nor a design, but it highlights interest in this direction and points toward the relevance and need for this new type of client.

Light Clients for Lazy Blockchains. Light clients that help secure the chain have become popular in the context of lazy ledgers [5, 28]. 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.

Sharding. Finally, in sharding [8, 21] 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 Zeta Avarikioti, George Danezis, Joachim Neu, and Dionysis Zindros 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] 2023. Mina Docs. https://docs.minaprotocol.com/about-mina.

[2] 2024. Sui Full Node Transaction Signatures are not verified. https://github.com/MystenLabs/sui/blob/7bc276d534c6c758ac2cfefe96431c2b1318ca01/crates/sui-sdk/src/apis.rs#L1128

[3] 2026. Sui Authenticated Events. https://github.com/MystenLabs/sui/blob/392e6448cdcbdc3ad2f46bfd01cb0d7dea187882/docs/content/guides/developer/accessing-data/authenticated-events.mdx

[4] 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

[5] Mustafa Al-Bassam. 2019. LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts. arXiv:1905.09274 [cs.CR] https://arxiv.org/abs/1905.09274

[6] 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.

[7] 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. https://eprint.iacr.org/2024/692

[8] 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.

[9] 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).

[10] 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. arXiv:2310.18042 https://arxiv.org/abs/2310.18042

[11] 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).

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

[13] Vitalik Buterin. 2025. A local-node-favoring delta to the scaling roadmap. https://ethresear.ch/t/a-local-node-favoring-delta-to-the-scaling-roadmap/22368

[14] 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

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

[16] 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.

[17] DefiLlama. 2026. Uniswap Protocol Metrics. https://defillama.com/protocol/uniswap. Accessed: 2026-04-27.

[18] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.

[19] Etherscan. 2026. Etherscan: Transactions. https://etherscan.io/txs. Accessed: 2026-04-27.

[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] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.

[22] 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.

[23] 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

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

[25] Wave Wallet on Sui. 2026. https://waveonsui.com/.

[26] 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).

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

[28] Christos Stefo, Zhuolun Xiang, and Lefteris Kokoris-Kogias. 2023. Executing and proving over dirty ledgers. In *International Conference on Financial Cryptography and Data Security*. Springer, 3–20.

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

[30] The Block. 2026. 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. Accessed: 2026-04-27.

[31] 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. https://celo.org/papers/plumo

[32] Toni Wahrstaetter, Dankrad Feist, Francesco D'Amato, Jochem Brouwer, and Ignacio Hagopian. 2025. EIP-7928: Block-Level Access Lists. https://eips.ethereum.org/EIPS/eip-7928#block-structure-modification

[33] 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, February 18–22, 2019*. Springer, 179–198.

A Deferred Proofs

Proof of Lemma 1. Consider a smart contract on an EVM-based chain that implements the counter mechanism described in Section 3. Let $\mathcal{X}_s$ identify a contract account; therefore, the sparse ledger includes the transactions that successfully modify the contract state. Consider a sparse client parameterized by $\mathcal{X}_s$ that has received from a prover the tuple $(\hat{\mathcal{L}}, \pi, \mathcal{R}, \pi_{\text{ex}})$ (Algorithm 1, line 5). The proof π consists of two parts: a counter-based proof and a transaction inclusion proof for all transactions in $\hat{\mathcal{L}}$ (Algorithm 1, line 8). The client verifies that all transactions in $\hat{\mathcal{L}}$ are included in the ledger $\mathcal{L}$ (Algorithm 1, lines 9–11). This means that the client certifies that all transactions in $\hat{\mathcal{L}}$ are included in confirmed blocks of $\mathcal{L}$ by checking Merkle proofs of inclusion that verify against the transaction digest in the block headers. Because the transaction digests in all headers have been computed by the validators via consensus, and because consensus is a secure SMR protocol, the digests truthfully report which transactions have been included in the ledger. Because the transactions forming the leaves of the transaction Merkle tree are, by construction, appearing in the same order of the transactions in the block, this proof also certifies that the relative positions of transactions in $\hat{\mathcal{L}}$ is preserved. It follows that all transactions in $\hat{\mathcal{L}}$ appear in $\mathcal{L}$ without any tampering: their integrity is guaranteed and adversarially injected transactions are not accepted.

Because the client authenticates the execution metadata against the state digest in the header (Algorithm 1, lines 12–13) and because it executes the transactions in $\hat{\mathcal{L}}$ (Algorithm 1, lines 14–16), it locally computes the value of the counter. If the execution is successful, the client authenticates the value of the counter against the state digest in the header of a recently confirmed block (Algorithm 1, lines 17–19). Because the blocks and their headers are confirmed by the validators via consensus, and because consensus is a secure SMR protocol, the state digests in all headers correctly reports the value of the counter as it appears in the contract's persistent state after executing $\mathcal{L}$.

Let $\mathcal{T}$ be the set of transactions in $\mathcal{L}$ identified by $\mathcal{X}_s$. By the counter mechanism in Section 3, each transaction in $\mathcal{T}$ triggers at least one state-modifying function call and therefore increases the counter by at least 1, whereas transactions outside $\mathcal{T}$ do not change the value of the counter. Suppose, for contradiction, that the prover withholds some transaction $\text{tx}^\star \in \mathcal{T}$ from $\hat{\mathcal{L}}$. Then the counter

value locally computed by the client when executing $\hat{\mathcal{L}}$ is strictly smaller than the value reported by consensus in the persistent state of the contract after executing $\mathcal{L}$. Hence, the locally computed counter cannot match the authenticated counter in the header, and the client aborts at line 19. Therefore, any adversarially withheld transaction causes a mismatch with the counter in the header and cannot be accepted.

Because the client verifies that the locally computed value of the counter matches the one authenticated against the header and because the client has verified the integrity, inclusion, and relative ordering of transactions, it is ensured to output a sparse ledger $\hat{\mathcal{L}}$ identified by $\mathcal{X}_s$ that is a prefix of $\varphi(\mathcal{X}_s, \mathcal{L})$, as per Definition 4.

□

B Evaluation of Chain-Agnostic Sunfish on Sui

This section evaluates the Sunfish *chain-agnostic* design on the Sui blockchain; the evaluation of our *EVM-based instantiation* (sparse and event clients) is reported separately in Section 5.

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 [19], while Uniswap 85,210 transactions across multiple chains [17]. Without considering transaction dependencies, a sparse node running Sunfish in the hash-chain configuration and monitoring Uniswap would consume $\ll$ 5% of full nodes resources, as only 20% Uniswap activity occurs on Ethereum [30]. We then evaluate Sunfish on Sui to demonstrate the practicality of the chain-agnostic design on a high-performance blockchain.

The Sui Blockchain. Sui [9] 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* [10] 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 [12] and the Wave wallet [25]. 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).

Table 4: 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	$\lvert\pi_c\rvert$	$\lvert\pi_{hc}\rvert$
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}%)

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