Guppy: Efficient Light Clients via Recursive Zero-Knowledge Proofs

George Danezis[1,2], Deepak Maram[1], Arnab Roy[1], Alberto Sonnino[1,2], and Karl Wüst[1]

[1] Mysten Labs
[2] University College London

Abstract. Traditional light clients rely on validators committing to the *entire blockchain state* at every block via a state commitment such as a Merkle tree, allowing clients to verify facts using short proofs. However, maintaining large and ever-growing state trees imposes a significant burden on validators and lies on the critical path of block production. As a result, many modern high-throughput chains avoid this approach altogether. This work asks whether efficient inclusion proofs can be supported *without requiring validators to maintain full state commitments*. We present Guppy, a protocol that achieves this by having validators commit to just the *state updates*. An off-chain, untrusted service, secured by recursive Zero-Knowledge Proofs (ZKPs), then maintains a verifiable Merkle tree over the full state. This design keeps validator overhead negligible and does not increase the asymptotic complexity of block construction. Our design rests on two key technical ideas. First, a hash-chain commitment moves validator signature verification out of the ZK circuit, keeping the proving circuit efficient. Second, we design a parallel recursive proving pipeline that leverages cheap recursion in modern ZKPs to ensure latency grows only logarithmically with throughput. Our Plonky2-based implementation demonstrates that Guppy can maintain a Merkle tree of size 2^{30} while processing thousands of updates per second, adding only 2-4 s of latency.

1 Introduction

Light clients let resource-constrained devices, such as wallets on phones, IoT devices, or smart contracts on other chains, verify facts about the chain with short cryptographic proofs instead of downloading and executing the entire ledger. The common design has validators commit to the entire state, of size M, with a state commitment such as a Merkle tree of depth $\log M$, as Ethereum [12] does. Maintaining that commitment is costly because each update touches multiple database entries, the state only grows over time, and the computation sits on the critical path to finality. High-throughput chains [8,53] process thousands of transactions per second and therefore do not maintain a full-state commitment at all. Their light clients are left without a way to verify state.

Sunfish [38] shows that a weaker query is cheap to serve: the state of a value at the checkpoint where it was last modified (we call each block a checkpoint). What

applications need is completeness, the state of any object at any checkpoint, for example a wallet's current balance or a dapp's package state. This raises the question: *can a blockchain support efficient light clients with completeness without requiring validators to commit to the entire state?* Efficient means that proof sizes remain logarithmic in the system parameters.

Guppy answers it by having validators commit only to the state updates of each checkpoint, not to the state. The difference is large: Sui holds roughly 2^{30} objects but updates at most about 10k of them per second (about 0.001%). An untrusted off-chain ZK service, akin to a ZK co-processor [5,19,28], maintains a Merkle tree over the entire state and, after every checkpoint, proves with a recursive zero-knowledge proof [6,7] that the new root is the correct successor of the previous one under the checkpoint's updates. The straightforward recursive statement verifies the previous proof, verifies the checkpoint under the validator keys, and applies the updates to the tree. Plonky2 [43] makes such recursion practical, at about 6k gates and 0.3 s to prove. A client asking for the balance of an address at height h receives a Merkle path and the proof, even if the address last changed years ago. Because committee key rotations are infrequent (about daily on Ethereum and Sui), clients can be expected to know the validator keys.

Instantiating this design on a high-throughput chain faces three obstacles: validator signature schemes such as BLS are not ZK-friendly, extracting the relevant fields (say balances) may require decoding the entire checkpoint, and the proof must complete before the next checkpoint arrives because the previous proof is one of its inputs, which is hard at thousands of updates per second.

Guppy-A (Section 3.2) removes the first two obstacles with one new header field. Suppose checkpoint h sets the balances of addresses `0x123` and `0x234` to 30 and 100, and let U_h be the list of such updates. Each validator computes the hash-chain head $d_h = H(U_h, d_{h-1})$ with a ZK-friendly hash (Poseidon), in practice over fixed-size batches, and puts d_h in the header of checkpoint h. The circuit no longer verifies checkpoint validity; it exposes the chain head as a public input, and the client checks that the head appears in the signed header (a few hundred bytes) and verifies the validator signatures. Checking the head at height h attests to every earlier checkpoint because heads are chained. The third obstacle remains: every proof must finish before the next checkpoint arrives, $t^A \leq t_{\mathsf{bundle}}$ (middle row of Figure 1); our Plonky2 benchmarks (Section 4) cap Guppy-A at about 290 updates/s for a tree of 2^{30} streams, beyond which latency grows without bound.

Guppy-B (Section 3.3) starts from the observation that the sequential dependency is inherent to incrementally verifiable computation (IVC) [48] but can be confined to a small recursive step. It splits the statement: non-recursive base proofs, each covering a sub-bundle of updates, are produced in parallel on many machines (the boxes labeled S_{par} in the bottom row of Figure 1), an aggregation tree combines them, and a small cyclic proof (S_{seq}) verifies the previous cyclic proof and the aggregated proof, once per checkpoint. Only the cyclic proof, whose cost is constant (0.63 s), must fit within the checkpoint interval, so

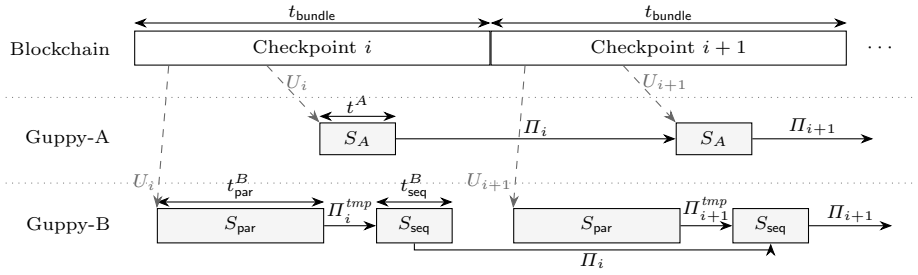

Fig. 1. The Guppy protocols. The top row shows a blockchain producing update bundles. The middle row depicts Guppy-A, which takes t^A to prove a bundle. The bottom row depicts a simple version of Guppy-B with two phases, which take t^B_{par} and t^B_{seq} for the parallel and sequential phases respectively. Guppy-A requires $t^A \leq t_{\mathsf{bundle}}$ whereas Guppy-B requires $t^B_{\mathsf{seq}} \leq t_{\mathsf{bundle}}$.

Guppy-B sustains any throughput given enough machines. Because aggregation is a tree, end-to-end latency grows only logarithmically with throughput.

We implement both protocols in Rust on Plonky2. The validator-side commitment for 2^{13} updates takes under 10 ms. A one-month measurement of Sui shows that a light-client service must sustain thousands of updates per second at peak. On a distributed testbed of up to 15 AWS machines, Guppy-B with 14 machines processes 2,250 updates/s at a median latency of 4 s; microbenchmarks predict 8,571 updates/s at 3.15 s with 41 machines (Section 4). Guppy also broadens what light clients can ask: the same mechanism authenticates streams of events, balances, or objects (Section 2), including queries such as all events of a given type at a checkpoint, which Ethereum's state tree cannot answer.

Contributions. We make the following contributions:

- We present Guppy, the first protocol giving light clients completeness (current and historical state of any supported stream) with negligible validator overhead by adding one header field per checkpoint.
- We introduce the techniques that make Guppy scale: hash-chain commitments that move signature checks out of the circuit, parallel base proofs, tree aggregation, and a constant-cost cyclic proof, giving latency logarithmic in throughput; many apply to other IVC-style computations.
- We implement Guppy and show thousands of updates per second at a few seconds of latency.

2 Model

2.1 Blockchain model

We model the blockchain as a sequence of *checkpoints* (or *blocks*), where each checkpoint is composed of *contents*, a *header*, and a *signature* from validators. The checkpoint contents $\mathcal{C}$ consist of a list of transactions along with their execution effects, $\{\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_n\}$, where each tx_i contains both the original transaction and its *effects* [8]. Effects can contain information such as the outcome of

the transaction's execution, its impact over the blockchain's state, and the emitted events [40]. The header $\mathcal{H}$ includes a checkpoint sequence number c unique to each checkpoint, and most commonly it also includes a commitment to the checkpoint contents; we will propose adding one field to the header.

Let $\mathsf{pk}_{\mathsf{chain}}$ be the current committee's public key. The validators in the committee jointly sign the checkpoint header to produce a signature σ that can be verified by running $\mathsf{Sig.verify}(\sigma, \mathcal{H}, \mathsf{pk}_{\mathsf{chain}})$ using an unforgeable signature scheme (EUF-CMA). Committee handoffs are authenticated at special checkpoints and the genesis committee is public, allowing anyone to verify a committee by following the sequence from genesis. We denote the contents and header of checkpoint c by $\mathcal{C}_c$ and $\mathcal{H}_c$ respectively, assuming the presence of functions $\mathsf{IsValid}(\mathcal{C}_c)$ and $\mathsf{IsValid}(\mathcal{H}_c)$ that verify their validity using the committee keys. Our model applies to any chain that has a signed-checkpoint semantics, such as Ethereum [12] and Sui [8]. Ethereum uses an account model [12] where validators commit to a Merkle tree root of all accounts and contracts at the end of every checkpoint, whereas Sui uses an object model supported by a key-value store without a state commitment to maintain high throughput.

2.2 Streams

We model blockchain data as a collection of *streams*. A stream S represents a logical view of blockchain state evolution, defined as a key-value map from unique *stream identifiers* (denoted id) to short *stream states* (denoted v_{id}) or commitments thereof, i.e., $S = \{(id_1, v_{id_1}), \ldots, (id_M, v_{id_M})\}$. Each stream evolves deterministically as new checkpoints are processed using a function A, which derives all stream updates from the current stream and a checkpoint's contents:

$$U_c^{\mathsf{bc}} \leftarrow A(S, \mathcal{C}_c).$$

The output U_c^{bc} is the stream update set, a list of (stream identifier, new state, new bit flag) tuples: $U_c^{\mathsf{bc}} = [(id_1, v'_{id_1}, e_{id_1}), \ldots, (id_n, v'_{id_n}, e_{id_n})]$. The new bit flag indicates whether the identifier is new or already exists; the updated stream S' follows directly from S and U_c^{bc}. The size of the set U_c^{bc} is a key metric when evaluating the performance of the Guppy protocols, and we use the term updates/s (updates per second) to refer to the number of stream updates produced or processed in a second.

Definition 1 (Stream). *Given an identifier type, value type, and a stream update function A, a stream S is a collection of stream identifiers and their corresponding states: $S = \{(id_1, v_{id_1}), \ldots, (id_M, v_{id_M})\}$.*

The stream abstraction is general and can represent a variety of indexing schemes. An address-balance stream that tracks the balance of each address uses the address as its identifier (e.g., $id = \texttt{0x123}$) and the current balance as its state. Its stream update function extracts all balance updates for this address $\{p_1, \ldots, p_n\}$ from the checkpoint contents and updates the balance as $v'_{id} \leftarrow v_{id} + \sum_{i=1}^{n} p_i$. An event stream tracking all events of a given type uses the event

type as its identifier (e.g., $id = \texttt{exampleContract} :: \texttt{exampleEvent}$) and a hash-chain commitment of all matching events as its state, updating by extracting matching events $\{p_1, \ldots, p_n\}$ and folding them into the hash-chain as $v'_{id} \leftarrow H(H(\ldots H(v_{id}, p_1), \ldots, p_n))$. A blockchain may support object, address-object, or custom developer-specified streams that define a stream update function over filtered event types or the union of multiple event types.

2.3 Goals

Our goal is to support verifiable queries for the state of any stream at the end of any checkpoint c, a property we call *completeness*. Let M denote the total number of stream IDs and n the number of stream IDs updated per checkpoint. Our solution should fulfill two requirements. **R1** (Completeness for varied streams): Checkpoints must contain authenticated data structures that permit querying the state of any of several streams at an arbitrary checkpoint. **R2** (Light-weight validator changes): The bandwidth, memory, and computational resources a validator spends to build a checkpoint are linear in the size of the checkpoint (i.e., $O(n)$) and in particular do not depend on M. We do not place any restrictions on the internal storage of validators, so it is acceptable if the internal storage increases by $O(M)$ to support a new stream.

2.4 API and Correctness

We next define the key APIs used by the three participants in Guppy: validators (**Val**), the ZK service (**ZKS**), and clients (**Client**). Validators produce a *stream update commitment* d_c for each checkpoint, which compactly commits to all stream updates U_c^{bc} derived from that checkpoint. The ZK service maintains a commitment over the full stream S (that consists of all stream identifiers and their corresponding states) called the *global stream commitment* D_c. Clients are interested in learning the state of a stream identifier id at a specific checkpoint c.

1. **[Val]** $d_c \leftarrow \mathsf{localCommit}(U_c^{\mathsf{bc}})$: Executed by validators before committing to a checkpoint, this commits to the stream update set U_c^{bc} of the n streams updated in checkpoint c, optionally taking prior digests as inputs.
2. **[Val]** $\mathcal{H}_c \leftarrow \mathsf{genHeader}(c, d_c, \ldots)$: The checkpoint header generating function is modified to include d_c as a new field.
3. **[ZKS]** $(D_c, \Pi_c) \leftarrow \mathsf{processUpd}(D_{c-1}, (\mathcal{H}_c, U_c^{\mathsf{bc}}))$: The ZK service processes the new checkpoint updates to advance the global commitment.
4. **[ZKS]** $(v, \pi) \leftarrow \mathsf{readState}(id, c)$: Returns the stream state after checkpoint c for the stream identifier id along with a validity proof.
5. **[Client]** $0/1 \leftarrow \mathsf{verifyState}(id, c, v, \pi)$: Verifies the validity proof assuming knowledge of the public keys of the validator quorum.

Intuitively, a stream state (id, v) is *correct* at checkpoint c if it incorporates all updates to id up to c and excludes any that occur afterward.

Definition 2 (Stream State Correctness). *A stream state* (id, v) *is* correct *at checkpoint* c *if there exists a checkpoint* $c' \leq c$ *and a sequence of stream updates* $U_{c'}^{\mathsf{bc}}, U_{c'+1}^{\mathsf{bc}}, \ldots, U_c^{\mathsf{bc}}$ *such that:*

1. *For every* $c'' \in [c', c]$*, the stream update commitment* $d_{c''} = \mathsf{localCommit}(U^{\mathsf{bc}}_{c''})$ *is included in a valid checkpoint header* $\mathcal{H}_{c''}$ *i.e.,* $\mathsf{IsValid}(\mathcal{H}_{c''}) = 1$.
2. *The stream id is updated at checkpoint* c'*, i.e.,* $(id, v, \_) \in U^{\mathsf{bc}}_{c'}$*, and remains unchanged thereafter:* $\forall\, c'' \in (c', c],\ (id, \_) \notin U^{\mathsf{bc}}_{c''}$.

Merkle trees. The Guppy protocols use three standard Merkle-tree operations: MT.vf verifies a leaf against a root, MT.vfAndUpd additionally updates the leaf and returns the new root, and MT.insert appends a leaf at a given index (specified in Appendix B.1).

3 The Guppy protocols

The ZK service maintains a verifiable commitment to all stream states $\{(id_i, v_i)\}_{i=1}^{M}$ by processing each checkpoint within a ZK circuit, enabling efficient inclusion proofs for clients to query any stream identifier at any checkpoint. Our main protocols employ a Merkle tree because it supports efficient proofs even when tracking all stream identifiers (e.g., $M \approx 2^{30}$), effectively acting as an *authenticated full node*. We first describe validator commitments (Section 3.1), then present Guppy-A (Section 3.2) for moderate throughput and Guppy-B (Section 3.3) for arbitrarily high throughput; Appendix C describes a hash-based variant for a small fixed set of streams.

3.1 Validator commitment

Validators commit to the states of all streams created or updated in the current checkpoint by partitioning the update set into fixed-size batches and hashing them into a hash chain. Recall from Section 2 that each checkpoint $\mathcal{C}_c$ yields a stream update set U^{bc}_c of size N^{bc}_c, consisting of updates $p_i = (id_i, v_i, e_i)$. Validators first partition U^{bc}_c into $n = \lceil N^{\mathsf{bc}}_c / b \rceil$ batches $\{B_1, B_2, \ldots, B_n\}$ of size b, padding the final batch if necessary. They then compute a hash $h_i = H(B_i)$ for each batch in parallel, and finally compute the stream update commitment as a hash chain over these batch hashes, where the previous checkpoint's commitment d_{c-1} initiates the chain:

$$d_c = H(\ldots, H(H(d_{c-1}, h_1), h_2), \ldots, h_n).$$

We denote this procedure compactly as $d_c = H_{\mathsf{minnow}}(d_{c-1}, U^{\mathsf{bc}}_c)$, which implements the localCommit function from the API in Section 2.4. The resulting chain head d_c is included in the checkpoint header $\mathcal{H}_c$ to fulfill the genHeader function. Batching enables parallel commitment generation and later allows parallel proof generation in the ZK pipeline.

3.2 A first attempt: Guppy-A

In Guppy-A, the ZK service processes stream updates in fixed-size bundles of $N = n \cdot b$ updates, processing multiple bundles sequentially if a checkpoint exceeds this size. The service maintains a Merkle tree over all stream states, where

each leaf stores the latest state (id, v) and empty leaves represent uninitialized streams. It updates the tree verifiably after processing each bundle. Assuming the circuit has processed all checkpoints up to c, its public inputs summarize the current system state using the latest Merkle root D_c, the number of non-empty leaves L_c, the latest validator commitment d_c, and the checkpoint number c. The circuit takes the most recent proof it has generated, the new bundle of stream updates $U_{c+1}^{\mathsf{bc}} = \{(id_i, v_i, e_i)\}$ and their corresponding Merkle paths as inputs. It first verifies the previous proof, and then applies each stream update: if an existing stream is updated ($e = 0$) it verifies the Merkle path and updates the leaf with the new value, whereas if a new stream is introduced ($e = 1$) it checks that the insertion occurs at the next available leaf index indicated by L.

Circuit parameters and inputs. At setup time we fix and hard-code into the circuit the tree depth d to support $M = 2^d$ stream IDs, the validator batch size b, and the number of batches per bundle n so one invocation processes $N = n \cdot b$ updates. Cheap recursion makes these parameters easy to change, since a new circuit simply verifies the old proof in its base case and continues from there. The private inputs are the previous proof with its public inputs along with the old leaf, the new leaf, and the Merkle path for every update, while the public inputs are $pub^c = \{D_c, d_c, L_c, c\}$. Appendix B.2 specifies the full statement S_A.

Serving proofs to clients. To answer $\mathsf{readState}(id, c)$, the ZK service returns the state together with $\pi = \{\Pi_c, \mathcal{H}_c, path_{id}\}$, which consists of the recursive proof with its public inputs, the signed checkpoint header carrying d_c, and the Merkle path of the queried stream. The $\mathsf{verifyState}$ routine checks the Merkle path against D_c, verifies the validator signature on the header and checks that it carries d_c and c, and finally checks the proof Π_c (Appendix B.3).

Security theorem. We now state and prove the security of Guppy-A.

Theorem 1 (Guppy-A Security). *Let B be a blockchain as modeled in Section 2: a sequence of valid checkpoints whose headers are signed by the committee, with committee handoffs authenticated from a public genesis. Suppose a client holding the committee key $\mathsf{pk}_{\mathsf{chain}}$ runs $\mathsf{verifyState}(id, c, v, \pi)$ against B as specified above. If the verifier outputs 1, and if the following assumptions hold:*

1. *The signature scheme is existentially unforgeable under chosen-message attack.*
2. *The hash function used in H_{minnow} and Merkle trees is collision resistant.*
3. *The employed ZK proof system is knowledge-sound for the statement S_A (including recursive verification).*

then (id, v) is correct at checkpoint c of B, except with negligible probability.

Informally, the theorem holds because the hash-chain structure of the validator commitment (H_{minnow}) ensures that verifying the signature on the final header guarantees the correctness of all prior stream update commitments; this relies on signature unforgeability and hash collision resistance. The knowledge-soundness of the recursive ZK proofs further guarantees that all updates to the

Merkle root match the stream updates. Appendix A gives the formal proof under the stated assumptions.

Cost analysis. Let h denote the constraint cost of a single hash invocation and r the cost of verifying one ZK proof. Each circuit invocation performs the ZK verification costing r constraints, Merkle operations requiring $2 \cdot N \cdot d$ hashes to verify and update each leaf, and stream update commitments needing $N + N/b$ hashes to compute H_{minnow}, bringing the total cost to approximately $r + N \cdot h \cdot (2d + 1 + 1/b)$ constraints. A key optimization in Guppy-A avoids verifying validator signatures inside the ZK circuit, which would otherwise be prohibitively expensive or require validators to adopt a ZK-friendly signature scheme.

Rather than proving each checkpoint individually, multiple checkpoints can be processed together to amortize the fixed recursion cost r. However, this sacrifices the ability to query intermediate checkpoints. Whether this is an issue depends on the use case and rate of checkpoint production. For example, the Sui blockchain produces four checkpoints per second, so we believe batching could be a reasonable approach. We explore the performance benefit of batching concretely in Section 4.

Another benefit of batching is that it allows handling dynamic variations in throughput. For example, if Guppy-A's circuits can handle N updates, and if two consecutive checkpoints have $1.5N$ and $0.5N$ updates, then processing the two checkpoints together means that the ZK service just needs to process two bundles instead of three.

Limitations. The main limitation of Guppy-A is its sequential dependency, as each proof generation depends on the previous proof as an input and must complete before the next bundle of updates arrives. If proving a bundle takes t^A and the blockchain produces a bundle every t_{bundle} seconds on average, the system requires $t^A \leq t_{\mathsf{bundle}}$; otherwise latency grows rapidly and the system stalls (Figure 1). Our evaluation in Section 4 shows that Guppy-A can keep up with roughly 290 updates per second for a Merkle tree of size 2^{30}.

3.3 Our main protocol: Guppy-B

We now present Guppy-B, an improved protocol that uses the same validator commitment function H_{minnow} as Guppy-A but splits its monolithic statement into smaller components that can be proven in parallel to sustain arbitrarily high throughput. Every invocation of Guppy-B processes a bundle, which may correspond to updates from one or several checkpoints and is subdivided into β sub-bundles that act as the units of parallel processing. Each sub-bundle contains n batches of b updates each, matching the batch size validators use. Assuming for simplicity that all updates produced by checkpoint c fit into one bundle where $N_c^{\mathsf{bc}} \leq N$, Guppy-B executes in three phases depicted in Figure 2. The parallel phase first processes sub-bundles independently with a base circuit to update the Merkle root from D_{c-1} to D_c, yielding β base proofs. The aggregation phase then aggregates these base proofs into a single proof using a tree of aggregation

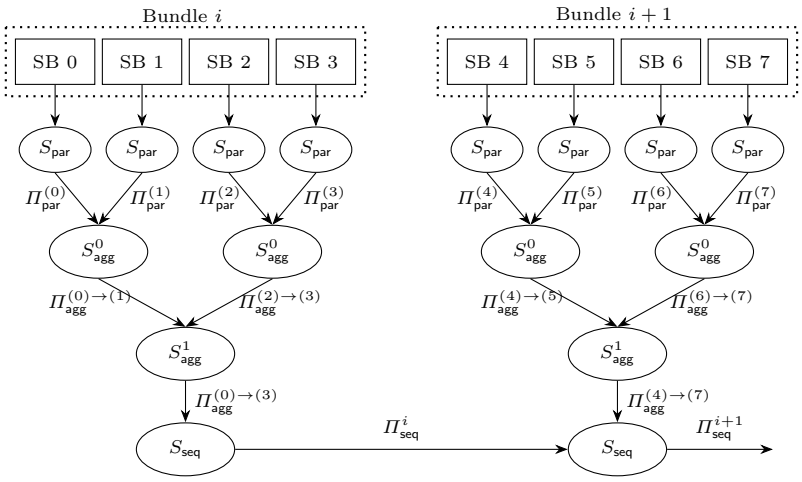

Fig. 2. Guppy-B with $\beta = 4$ base proofs and an aggregation factor of $\alpha = 2$. Processing a bundle happens in three phases: parallel phase (S_{par}), aggregation phase (S_{agg}) and a sequential phase (S_{seq}).

circuits. Finally, the sequential phase uses a cyclic circuit to verify its own previous proof together with the aggregated proof for checkpoint c, producing one final proof per checkpoint.

Parameters. The protocol defines the tree depth d, the batch size b representing updates per batch, and the sub-bundle size y where $y \bmod b = 0$ so the number of batches per sub-bundle is $n = y/b$. It also defines the bundle size N where $N \bmod y = 0$ so the number of sub-bundles per bundle is $\beta = N/y$, and the aggregation factor α which determines that the proof tree has $l = \log_\alpha(\beta)$ levels.

1) Parallel phase. This phase chunks the bundle into β sub-bundles and runs the base prover on each in parallel to prove the statement S_{par}, asserting that the sub-bundle transforms the state $\{D_{\mathsf{old}}, d_{\mathsf{old}}, L_{\mathsf{old}}\}$ into $\{D_{\mathsf{new}}, d_{\mathsf{new}}, L_{\mathsf{new}}\}$. The circuit closely mirrors Guppy-A, with the key differences that it does not recursively verify a prior proof and it exposes both the old and new states as public inputs. Assuming sufficient machines, this phase outputs β base proofs $\Pi_{\mathsf{par}}^{(i)}$ while completing in the time required to generate one base proof.

2) Aggregation phase. The aggregation phase combines the β base proofs into a single proof using an aggregation tree. Each level-i aggregation circuit takes α proofs from the level below (base proofs at level 0), verifies them, checks that the outputs of one match the inputs of the next, and emits a combined proof. The public inputs of each aggregation circuit match those of S_{par}. For example, if $\beta = 4$ and $\alpha = 2$, the system uses two level-0 aggregations that each combine two base proofs, followed by one level-1 aggregation combining those two results as shown in Figure 2. This phase is fully parallelizable, requiring β/α^{i+1} machines at level i.

3) Sequential phase. The cyclic circuit ties the system together by taking the last sequential proof Π_{seq}^{c-1} for checkpoint $c-1$ and the aggregated proof $\Pi_{\mathsf{agg}}^{(c-1)\to(c)}$ for the current checkpoint bundle, verifying both proofs, and matching their public inputs. Its statement S_{seq}, detailed in Appendix B.4, outputs the next sequential proof Π_{seq}^{c} to serve as the final proof for checkpoint c. The bottom half of Figure 1 illustrates the special case where $\beta = 1$ and $\alpha = 1$, whereas Figure 2 shows a more complex instance with $\beta = 4$ and $\alpha = 2$.

Several checkpoints per bundle. While the description above assumed Guppy-B processes a single checkpoint per run, the system can process multiple checkpoints at once by choosing a fixed time slot t_{bundle} and treating all checkpoints occurring in that slot as one bundle. As long as the updates produced in a slot remain below N, Guppy-B operates normally and only requires that the constant runtime of a single sequential phase invocation remains smaller than t_{bundle}. Because we control the slot length and the sequential phase takes roughly 0.63 s as shown in Section 4, any choice of $t_{\mathsf{bundle}} \geq 0.63\,\mathrm{s}$ avoids throughput caps. Fixing the batch size b, sub-bundle size y, and aggregation factor α to support a target throughput ρ updates per second leaves only the bundle size N, which we set to $N = y \cdot \lceil \rho \cdot t_{\mathsf{bundle}}/y \rceil$ so that it is a multiple of y.

Latency. Let t_{par}, t_{agg}, and t_{seq} denote the runtimes of the parallel, aggregation, and sequential phases respectively. The parallel phase $t_{\mathsf{par}} = O(y)$ behaves as $O(1)$ since all sub-bundles are proved in parallel, the aggregation phase $t_{\mathsf{agg}} = O(l)$ performs $O(1)$ work across $l = \log_\alpha(\beta)$ levels, and the sequential phase $t_{\mathsf{seq}} = O(1)$. Thus the end-to-end latency is

$$t_{\mathsf{e2e}} = t_{\mathsf{par}} + t_{\mathsf{agg}} + t_{\mathsf{seq}} = O(\log_\alpha(\beta)).$$

Since $\beta = \lceil N/y \rceil = \lceil \rho \cdot t_{\mathsf{bundle}}/y \rceil$, the overall latency is $t_{\mathsf{e2e}} = O(\log_\alpha(\rho))$, logarithmic in throughput. The number of machines Guppy-B needs to keep pace with the chain follows from the three phase runtimes (Appendix D.2).

4 Implementation and Evaluation

We evaluate the Guppy protocols using a Rust implementation. It includes validator-side commitment logic, Merkle-tree maintenance, and ZK circuits for Guppy-A and Guppy-B built on the Plonky2 proving system [43] with its default recursion configuration (codeword rate 1/8, no zero-knowledge), as well as orchestration and networking code for end-to-end evaluation on a multi-machine testbed. In total, we have written 10.5k lines of code (4k for circuits/logic, 6.5k for orchestration), and we are open-sourcing the entire codebase to enable full reproducibility of our experiments.[3] All benchmarks use AWS `m6a.8xlarge` instances within a single data center, where each instance provides 12.5 Gbps network bandwidth, 32 vCPUs (16 physical cores) powered by a 3.6 GHz AMD EPYC 7R13 processor, 128 GB RAM, and Ubuntu 22.04.

[3] `https://github.com/asonnino/minnow` (commit `9c3075f`)

Degree d	13	14	15	16	17
Proving time t_d^{prove} (s)	0.32	0.63	1.29	2.77	5.96

Table 1. Plonky2 proving time by circuit degree (g gates give degree $\lceil \log_2 g \rceil$).

	Degree			
Streams (M)	14	15	16	17
2^{10}	300	850	1950	4100
2^{20}	200	500	1150	2450
2^{30}	100	350	800	1750

Table 2. Max updates per Guppy-A proof at Plonky2 degrees 14–17 (batch size 50).

Plonky2 prover. We begin by benchmarking the Plonky2 prover on circuits with increasing gate counts. A circuit with g gates has polynomial degree $d = \lceil \log_2(g) \rceil$, and the corresponding proving time t_d^{prove} is shown in Table 1; for example, $t_{14}^{\mathsf{prove}} = 0.627\,\text{s}$.

Data collection. We analyze one month of Sui blockchain activity (May 25 to June 25, 2025) to estimate the throughput Guppy must sustain. We measure the average number of updates per second, our baseline, and the maximum number of updates in a single checkpoint, which multiplied by Sui's average checkpoint rate (4.12 checkpoints/s) gives a worst-case update rate. The events stream averages 86.18 events emitted per second with a worst-case 1107 events in a single checkpoint, corresponding to $1107 \times 4.12 = 4560.84$ updates/s (peak observed at checkpoint 146264685). The objects stream averages 336.09 object updates per second with a worst-case 2114 objects updated in a single checkpoint, corresponding to $2114 \times 4.12 = 8709.68$ updates/s (peak observed at checkpoint 144662971). These measurements suggest that Guppy must handle several thousand stream updates per second to support modern high-throughput chains.

4.1 Validator overhead

We measure the overhead that Guppy introduces on validators. Validators compute the Guppy commitment each checkpoint by chunking the N_c^{bc} stream updates into batches of size b, hashing each batch in parallel, and folding these batch hashes sequentially into the running commitment d_c (Section 3.1). Only this folding step is sequential, so the batch size b controls the degree of parallelism. Figure 3(a) shows the end-to-end time to compute d_c for $N_c^{\mathsf{bc}} \in \{2048, 4096, 8192\}$ using four CPU cores. A batch size of $b = 50$ strikes a good balance, keeping the overhead below 10 ms and ensuring a negligible validator-side cost for Guppy. Since the largest checkpoint in our one-month Sui measurement carried 2,114 updates, a quarter of the 8192 measured here, the per-checkpoint commitment cost is a small fraction of Sui's 240 ms checkpoint interval. We thus fix $b = 50$ for all subsequent experiments, noting that the batch size barely affects circuit size: for a base circuit processing 1000 updates, batch sizes of 10, 100, and 1000 give 50000, 49890, and 49876 gates (Table 3).

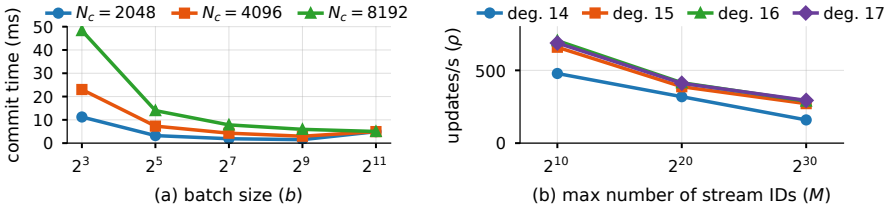

Fig. 3. (a) Time taken to compute the Guppy commitment using 4 cores for varying batch sizes and update counts. (b) Amortized throughput of Guppy-A for different maximum stream counts M. Each curve corresponds to a different proving interval, equal to the Plonky2 proving time at circuit degrees 14–17 (see Table 1).

4.2 Guppy-A

We evaluate the performance of Guppy-A, where each circuit invocation processes a fixed number of stream updates N and performs one recursive proof verification. For a given maximum number of stream identifiers M (which fixes the Merkle-tree depth $d = \log_2(M)$), increasing N increases the circuit size and thus its Plonky2 proving time. We consider three choices for M, namely $M = 2^{10}, 2^{20}, 2^{30}$, and for each M and each Plonky2 degree $d \in \{14, 15, 16, 17\}$, we determine the largest N such that the total number of gates still fits within the corresponding degree bound. We compile the circuit at increments of fifty updates and record the largest N before the gate count crosses a power-of-two threshold (Table 2).

Since one Guppy-A proof is produced every proving interval t_d^{prove} (for degree d), the amortized throughput at that configuration is $\rho \approx \frac{N}{t_d^{\mathsf{prove}}}$ updates/s. Figure 3(b) plots this throughput for each M and degree $d \in \{14, 15, 16, 17\}$ using the measured Plonky2 proving times from Table 1. When $M = 2^{30}$, Guppy-A can handle 100 updates per proof yielding ≈ 159 updates/s at degree 14, 350 updates per proof yielding ≈ 271 updates/s at degree 15, 800 updates per proof yielding ≈ 289 updates/s at degree 16, and 1750 updates per proof yielding ≈ 294 updates/s at degree 17. Two opposing effects shape these curves: batching more updates into a single proof improves amortization of the recursive-verification cost (Section 3.2), but Plonky2 proving time grows super-linearly with circuit size, meaning overly large circuits reduce throughput. The first effect is visible for $M = 2^{30}$, where increasing the proving interval monotonically raises throughput. The second dominates for $M \in \{2^{10}, 2^{20}\}$, where throughput peaks at an intermediate degree (16) and additional batching yields negative returns. For $M = 2^{30}$, the best performance is achieved by processing all the checkpoints arriving in a window of 5.96 s at once, and Guppy-A then processes about 294 updates per second; in practice one may prefer a shorter proving interval to answer queries at finer time granularity. These numbers motivate Guppy-B for high-throughput chains emitting thousands of updates per second. Plonky2 proof sizes for Guppy-A range between 120 and 150 KB depending on the number of gates; at $M = 2^{30}$ and $N = 800$ (degree 16), the proof is about 141.48 KB.

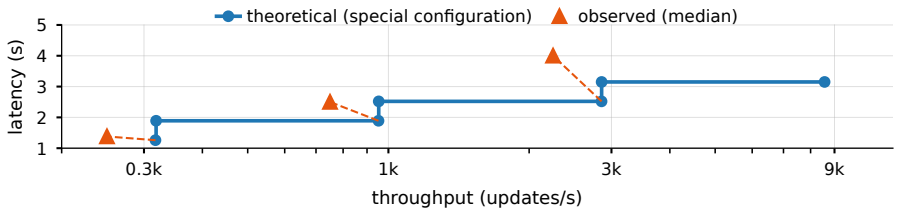

Fig. 4. Theoretical (blue) and observed (red) median latency vs. throughput for Guppy-B. The blue step curve corresponds to the special configuration in Table 5; each plateau reflects a fixed aggregation level, and each step corresponds to adding a new aggregation level.

A full Guppy proof additionally includes the checkpoint header (roughly 300 B on Sui) and a Merkle path, for a total of around 150 KB.

4.3 Guppy-B

Microbenchmarks. We configure Guppy-B with a fixed maximum number of streams $M = 2^{30}$. The parallel circuit S_{par} is structurally identical to the Guppy-A circuit except that it omits recursive proof verification, reducing its gate count and allowing it to process more updates per invocation: 200, 450, 900, and 1900 updates at degrees 14 to 17 (Table 4). Each aggregation circuit verifies α proofs and consists of approximately $4500 \cdot \alpha$ gates. The cyclic circuit verifies both the previous cyclic proof and the final aggregated proof; its roughly 10k gates give it Plonky2 degree 14 and proving time $t_{\mathsf{seq}} = 0.63\,\mathrm{s}$ (Table 1). The key design requirement is that the bundle interval exceeds this cyclic proving time, $t_{\mathsf{bundle}} \geq t_{\mathsf{seq}} = 0.63\,\mathrm{s}$.

Special configuration. For concreteness, we analyze a configuration with $t_{\mathsf{bundle}} = 0.63\,\mathrm{s}$, $b = 50$, $y = 200$, and $\alpha = 3$. With these choices, both the parallel and aggregation circuits fit comfortably within degree 14, so their proving times also equal 0.63 s. If a bundle contains β sub-bundles ($\beta = N/y$), then the aggregation tree has $l = \log_3(\beta)$ levels. Thus, the end-to-end latency is $t_{\mathsf{e2e}} = t_{\mathsf{par}} + l \cdot t_{\mathsf{agg}} + t_{\mathsf{seq}} = 0.63 \cdot (2 + l)\,\mathrm{s}$. Bundles of up to 200, 600, 1800, and 5400 updates need 0, 1, 2, and 3 aggregation levels, giving predicted latencies of 1.26, 1.89, 2.52, and 3.15 s at 317, 952, 2857, and 8571 updates/s on 2, 5, 14, and 41 machines. Latency grows logarithmically in the bundle size while throughput grows linearly, the main scalability advantage of Guppy-B; Table 5 tabulates these values and Appendix D.2 derives the machine counts.

End-to-end evaluation. We evaluate the end-to-end throughput and latency of Guppy-B on a distributed testbed of up to 15 AWS instances (one primary, up to 13 workers, and one client) to assess its performance in a realistic deployment setting. For these experiments, we fix the bundle interval to $t_{\mathsf{bundle}} = 0.8\,\mathrm{s}$, slightly larger than the theoretical minimum of 0.63 s from Section 4.3, to provide some breathing room for scheduling jitter and network variability. The implementation uses the tokio runtime [46] for asynchronous networking and

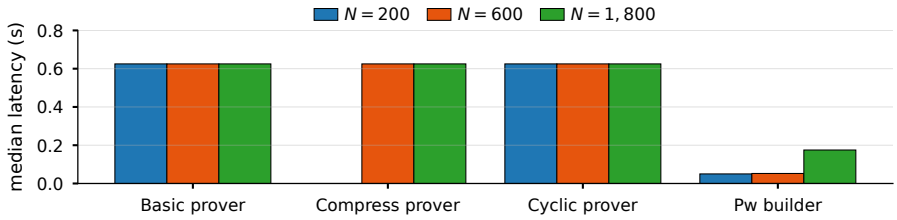

Fig. 5. Latency breakdown of Guppy-B by prover type, for bundle sizes of $N = 200$, 600, and 1,800 (1, 4, and 13 workers).

scheduling CPU-intensive tasks, as well as raw TCP sockets for inter-machine communication. The system architecture consists of a single *primary* machine and a configurable set of *worker* machines, where the primary receives bundles of state updates, constructs the corresponding witnesses, and dispatches them to the workers. Each worker computes both base and aggregation proofs, and once all intermediate proofs are returned, the primary computes the final cyclic proof, returns it to the client, and orchestrates all client communications. Operators expose only the primary and scale by adding workers.

The red triangles in Figure 4 show the observed latency and throughput of Guppy-B for each bundle size. Each point is the median over a 3-minute run; latency runs from a client's submission to the final cyclic proof. For a bundle size of 200 updates, the median latency is 1.4 s (p90: 1.5 s). Increasing the bundle size to 600 updates raises it to 2.5 s (p90: 2.9 s), and at 1,800 updates it reaches 4 s (p90: 4.9 s). The corresponding observed throughputs are 250, 750, and 2,250 updates/s (one bundle per 0.8 s). The number of machines used for each bundle size is exactly as predicted by the theoretical analysis, and the measurements confirm that the end-to-end latency of Guppy-B grows sub-linearly with the bundle size as expected.

The remaining gap between theoretical and observed latency stems primarily from networking overhead, context switching on the primary, queuing delays during bundle preparation on clients, and the time required to construct witnesses. The per-prover breakdown in Figure 5 confirms this (Appendix D.3 discusses it): the base, aggregation, and cyclic provers each take a median of about 0.63 s regardless of the bundle size, while witness construction on the primary grows from 0.05 s at 200 updates to 0.18 s at 1,800. The small throughput gap between the theoretical and observed curves is fully explained by our choice of a slightly larger bundle interval, since we send a bundle every 0.8 s in the experiments, whereas the idealized analysis earlier assumes $t_{\mathsf{bundle}} = 0.63$ s.

5 Related work

Prior light-client designs assume that validators publish, in every block header, a commitment to the full state; to our knowledge Guppy is the first to give light clients completeness on chains whose validators commit only to the updates of each checkpoint.

Light clients and state proofs. Header-chain compressions such as NIPoPoWs [23], FlyClient [14], and Blink [4] for proof of work, and PoPoS [2] for proof of stake, reduce downloads. Plumo [49] and zkBridge [52] go further with SNARK proofs of consensus, the latter proving sync-committee BLS signatures inside a distributed SNARK. Sync-committee clients include Altair [20] and Helios [1], and Chatzigiannis et al. [16] survey the space. All of these verify headers and then read state through the state root in the header, which Sui and Solana do not publish. Other designs relax trust in different ways: fraud and data-availability proofs [3] and light clients for lazy blockchains [42] need an honest full node, and BITE [31] and ZLiTE [51] rely on trusted hardware with known limitations [35]. Guppy trusts only the validator set. Mina [9,17] makes validators themselves run incrementally verifiable computation, so proving is on-chain and heavy; Guppy keeps validator overhead to one header field. Sunfish [38] is the closest work and the source of the validator-side hash-chain commitment. Its sparse client downloads and re-executes only the transactions touching a chosen sub-state and receives a succinct completeness proof that none was omitted. The value of a sub-state is thus obtained by re-execution, the cost grows with that sub-state's traffic, and there is no succinct inclusion proof for an arbitrary object at an arbitrary checkpoint. Guppy adds the untrusted ZK service that turns the same commitment into full-state, per-checkpoint inclusion proofs.

State commitments and stateless clients. Chains paying for a per-block state root use Merkle-Patricia tries (Ethereum [12]) or Jellyfish Merkle trees [22] (Aptos [26]); LVMT [29] shows this authenticated storage is the execution bottleneck. Verkle trees [13], aggregatable subvector commitments [47], and Hyperproofs [41] shrink proofs or make them maintainable, but the commitment stays on the validators' critical path. High-throughput chains drop it: Sui checkpoints carry transaction and effects digests but no state root, and the committee commits to the live object set only at epoch end [32,33]. Solana commits per block only to the accounts written in that block, and its lattice-hash successor supports no inclusion proofs [39]. That is the gap Guppy fills, without putting the commitment back on validators. Reckle trees [36] store recursive proofs at Merkle nodes to obtain succinct, updatable batch-membership and map-reduce proofs against a given root; they assume a trusted full-state digest such as Ethereum's header root and do not produce one. Guppy is complementary: its service could maintain a Reckle tree over the same leaves, adding succinct batched reads for standing queries on top of the per-checkpoint root proof.

ZK co-processors and verifiable indexers. Co-processors such as Axiom [5], Brevis [19], and Lagrange [28] prove reads of historical state and computations over it, and zkVMs such as SP1 [27] and RISC Zero [45] let developers build such indexers generically; validity rollups prove the execution of every transaction and post a state root [15]. All of them consume a state root that a header already provides. Guppy produces that root for chains that lack one, and proves only the tree update from validator-signed effects rather than re-executing transactions.

Recursion and distributed proving. Incrementally verifiable computation [48] and proof-carrying data [7] underlie Guppy; recursion was long expensive [6], and Halo [10] and Plonky2 [43] made it practical. We use Plonky2 for its FRI-based, small-field recursion and its available implementation. Folding schemes such as Nova [25], HyperNova [24], and Protostar [11] are an alternative when a proof at every checkpoint is not required; we leave exploring them to future work. Mangrove [34] obtains tree-shaped proof-carrying data with parallel proving in the folding setting, and zkTree [18] builds a recursion tree of Plonky2 proofs; both share the shape of Guppy-B's aggregation tree. Distributed provers such as DIZK [50], Pianist [30], and Hekaton [37], and aggregation schemes such as SnarkPack [21], scale one large statement across machines. Guppy-B instead exploits that its workload is naturally a tree of independent Merkle-tree updates, so it needs only small circuits and off-the-shelf recursion.

6 Conclusion

We introduced Guppy, a practical protocol for supporting efficient light clients with completeness on high-throughput blockchains. By shifting state maintenance to an off-chain prover and requiring validators to commit only to per-checkpoint state updates, Guppy avoids the heavy costs of maintaining global state commitments on-chain. Our design combines hash-chain commitments, recursive proofs, and a parallelizable proving pipeline, culminating in Guppy-B, which achieves logarithmic latency growth and sustains arbitrarily high throughput given sufficient parallelism.

Our prototype implementation using Plonky2 shows that Guppy can process thousands of state updates per second with only a few seconds of end-to-end latency, while imposing negligible overhead on validators. This demonstrates that completeness for light clients is compatible with the performance requirements of modern blockchains.

Looking ahead, Guppy opens several directions for future work, three of which we discuss below; others include additional query types such as non-inclusion proofs or per-stream proofs, and integrating state-of-the-art ZKPs [44] to further reduce latency. More broadly, the techniques developed here may benefit other incrementally-verifiable computations that require both high throughput and low latency.

Handling varying input rates. Because a blockchain emits a varying rate of stream updates while circuits process a fixed rate, both Guppy-A and Guppy-B handle this mismatch by setting a maximum bundle size. If the chain's average throughput grows beyond this limit, the system transitions to a new set of circuits supporting a larger bound, using the new sequential circuit to verify the previous sequential proof as its base case for smooth migration. This strategy allows the system to scale gradually with blockchain usage rather than requiring a large parameter choice from the start.

Temporary spikes are easily absorbed: if one slot produces more than N updates but the next slot produces fewer, the excess can be split into multiple bundles while maintaining overall latency through batching.

A drawback of fixed-size circuits is that the prover pays for N updates per slot even if fewer updates occur. One solution is to design a cyclic circuit capable of verifying aggregation circuits of multiple bundle sizes (e.g., $N \in \{5k, 10k, 15k\}$), allowing the system to choose the best parameter on a per-slot basis. We leave it to future work to investigate the efficiency of this approach.

Tracking a fixed set of streams. A useful variant of Guppy arises when the service tracks only a small fixed set of stream identifiers, which reduces prover cost while still providing completeness for selected streams.

In this case, we replace the Merkle-tree commitment used by Guppy-A and Guppy-B with a simple hash over the tracked states. If the service tracks a fixed set $\mathcal{ID} = \{id_1, \ldots, id_m\}$, then its global commitment is $D_c = H\big([(id_i, v_i)]_{i=1}^{m}\big)$. As before, the service processes the checkpoint updates to detect changes to any tracked stream and recompute the commitment. We explore this variant further in Appendix C.

Reducing validator work further. Depending on the stream type, supporting Guppy may require validators to maintain an additional key-value map. Supporting object streams on Sui requires no modification since validators already store the mapping from object IDs to their latest values, whereas supporting event streams requires introducing a new key-value store and addressing operational concerns such as state synchronization for new validators.

A natural question is whether even this overhead can be shifted to the ZK service. In principle, Guppy could be extended so that the ZK service also maintains per-stream state (e.g., rolling hash chains for event streams) in addition to maintaining global commitments. Whether this is practical depends on the cost of updating these structures inside the ZK circuit, and exploring this trade-off is an interesting direction for future work.

Acknowledgments. This work is partially funded by Mysten Labs. We thank Andrey Chursin and Michael Corey for help with ideating and data collection respectively.

References

1. a16z crypto: Helios: A fast, secure, and portable multichain light client for ethereum. https://github.com/a16z/helios (2022), accessed September 2026
2. Agrawal, S., Neu, J., Tas, E.N., Zindros, D.: Proofs of Proof-Of-Stake with Sublinear Complexity. In: 5th Conference on Advances in Financial Technologies (AFT 2023). Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2023), https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2023.14

3. Al-Bassam, M., Sonnino, A., Buterin, V., Khoffi, I.: Fraud and data availability proofs: Detecting invalid blocks in light clients. In: Financial Cryptography and Data Security - 25th International Conference, FC 2021, Part II. Lecture Notes in Computer Science, vol. 12675, pp. 279–298. Springer (2021). https://doi.org/10.1007/978-3-662-64331-0_15
4. Aumayr, L., Avarikioti, Z., Maffei, M., Scaffino, G., Zindros, D.: Blink: An optimal proof of proof-of-work. In: Financial Cryptography and Data Security - 29th International Conference, FC 2025. Lecture Notes in Computer Science, vol. 15752, pp. 173–190. Springer (2025). https://doi.org/10.1007/978-3-032-07035-7_11
5. Axiom: Axiom V2 developer docs: App architecture. https://docs.axiom.xyz/docs/axiom-developer-flow/app-architecture (2024), archived at https://web.archive.org/web/20240627181800/https://docs.axiom.xyz/docs/axiom-developer-flow/app-architecture
6. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 276–294. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
7. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for snarks and proof-carrying data. In: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing. p. 111–120. STOC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2488608.2488623, https://doi.org/10.1145/2488608.2488623
8. Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., Williams, B., Zhang, L.: Sui lutris: A blockchain combining broadcast and consensus. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024. pp. 2606–2620. ACM (2024). https://doi.org/10.1145/3658644.3670286
9. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Mina: Decentralized cryptocurrency at scale. Whitepaper, O(1) Labs (2020), https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf
10. Bowe, S., Grigg, J., Hopwood, D.: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021 (2019), https://eprint.iacr.org/2019/1021
11. Bünz, B., Chen, B.: Protostar: Generic efficient accumulation/folding for special-sound protocols. In: Advances in Cryptology - ASIACRYPT 2023, Part II. Lecture Notes in Computer Science, vol. 14439, pp. 77–110. Springer (2023). https://doi.org/10.1007/978-981-99-8724-5_3
12. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform. Whitepaper, https://ethereum.org/en/whitepaper/ (2014)
13. Buterin, V.: Verkle trees. https://vitalik.eth.limo/general/2021/06/18/verkle.html (2021), blog post, June 18, 2021. Accessed September 2026
14. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 928–946 (2020). https://doi.org/10.1109/SP40000.2020.00049
15. Chaliasos, S., Reif, I., Torralba-Agell, A., Ernstberger, J., Kattis, A., Livshits, B.: Analyzing and benchmarking zk-rollups. In: 6th Conference on Advances in Financial Technologies, AFT 2024. LIPIcs, vol. 316, pp. 6:1–6:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPIcs.AFT.2024.6

16. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: Blockchain light clients. In: International Conference on Financial Cryptography and Data Security. pp. 615–641. Springer (2022)
17. Chen, W., Chiesa, A., Dauterman, E., Ward, N.P.: Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Paper 2020/1522 (2020), https://eprint.iacr.org/2020/1522
18. Deng, S., Du, B.: zktree: A zero-knowledge recursion tree with ZKP membership proofs. Cryptology ePrint Archive, Paper 2023/208 (2023), https://eprint.iacr.org/2023/208
19. Dong, M., Liang, Q., Li, X., Liu, J.: Brevis: An omnichain ZK data attestation platform. Whitepaper v1.0, Celer Network (2023), https://get.celer.app/brevis/BrevisWhitePaper_03211833.pdf
20. Ethereum Foundation: Altair light client – sync protocol. Ethereum consensus specifications, https://ethereum.github.io/consensus-specs/specs/altair/light-client/sync-protocol/, accessed September 2026
21. Gailly, N., Maller, M., Nitulescu, A.: Snarkpack: Practical SNARK aggregation. In: Financial Cryptography and Data Security - 26th International Conference, FC 2022. Lecture Notes in Computer Science, vol. 13411, pp. 203–229. Springer (2022). https://doi.org/10.1007/978-3-031-18283-9_10
22. Gao, Z., Hu, Y., Wu, Q.: Jellyfish merkle tree. Diem technical paper (2021), https://developers.diem.com/papers/jellyfish-merkle-tree/2021-01-14.pdf
23. Kiayias, A., Miller, A., Zindros, D.: Non-interactive proofs of proof-of-work. In: Financial Cryptography and Data Security - 24th International Conference, FC 2020. Lecture Notes in Computer Science, vol. 12059, pp. 505–522. Springer (2020). https://doi.org/10.1007/978-3-030-51280-4_27
24. Kothapalli, A., Setty, S.T.V.: Hypernova: Recursive arguments for customizable constraint systems. In: Advances in Cryptology - CRYPTO 2024, Part X. Lecture Notes in Computer Science, vol. 14929, pp. 345–379. Springer (2024). https://doi.org/10.1007/978-3-031-68403-6_11
25. Kothapalli, A., Setty, S.T.V., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. In: Advances in Cryptology - CRYPTO 2022, Part IV. Lecture Notes in Computer Science, vol. 13510, pp. 359–388. Springer (2022). https://doi.org/10.1007/978-3-031-15985-5_13
26. Labs, A.: The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. Whitepaper (2022), https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf
27. Labs, S.: SP1: A performant, open-source zkVM. https://github.com/succinctlabs/sp1 (2025), accessed September 2026
28. Lagrange Labs: Lagrange ZK coprocessor and verifiable database (euclid). https://www.lagrange.dev/blog/announcing-testnet-euclid-zk-coprocessor (2024), accessed September 2026
29. Li, C., Beillahi, S.M., Yang, G., Wu, M., Xu, W., Long, F.: LVMT: an efficient authenticated storage for blockchain. In: 17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023. pp. 135–153. USENIX Association (2023), https://www.usenix.org/conference/osdi23/presentation/li-chenxing
30. Liu, T., Xie, T., Zhang, J., Song, D., Zhang, Y.: Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. In: IEEE Symposium on Security and Privacy, SP 2024. pp. 1777–1793. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00035

31. Matetic, S., Wüst, K., Schneider, M., Kostiainen, K., Karame, G., Capkun, S.: BITE: Bitcoin lightweight client privacy using trusted execution. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 783–800 (2019)
32. Mysten Labs: Checkpoint verification – sui documentation. https://docs.sui.io/concepts/cryptography/system/checkpoint-verification, accessed September 2026
33. Mysten Labs: Database snapshots – sui documentation. https://docs.sui.io/operators/snapshots, accessed September 2026
34. Nguyen, W.D., Datta, T., Chen, B., Tyagi, N., Boneh, D.: Mangrove: A scalable framework for folding-based snarks. In: Advances in Cryptology - CRYPTO 2024, Part X. Lecture Notes in Computer Science, vol. 14929, pp. 308–344. Springer (2024). https://doi.org/10.1007/978-3-031-68403-6_10
35. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel sgx. arXiv preprint arXiv:2006.13598 (2020)
36. Papamanthou, C., Srinivasan, S., Gailly, N., Hishon-Rezaizadeh, I., Salumets, A., Golemac, S.: Reckle trees: Updatable merkle batch proofs with applications. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024. pp. 1538–1551. ACM (2024). https://doi.org/10.1145/3658644.3670354
37. Rosenberg, M., Mopuri, T., Hafezi, H., Miers, I., Mishra, P.: Hekaton: Horizontally-scalable zksnarks via proof aggregation. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024. pp. 929–940. ACM (2024). https://doi.org/10.1145/3658644.3690282
38. Scaffino, G., Slowak, P., Wüst, K., Maram, D., Sonnino, A., Kokoris-Kogias, L.: Sunfish: Reading ledgers with sparse nodes. Cryptology ePrint Archive, Paper 2024/1680 (2024), https://eprint.iacr.org/2024/1680
39. Solana Foundation: SIMD-0215: Homomorphic hashing of account state (accounts lattice hash). https://github.com/solana-foundation/solana-improvement-documents/blob/main/proposals/0215-accounts-lattice-hash.md (2024), created 2024-12-20; status: activated. Accessed September 2026
40. Solidity Team: Solidity documentation 0.8.29: Contracts – events. https://docs.soliditylang.org/en/v0.8.29/contracts.html#events (2025)
41. Srinivasan, S., Chepurnoy, A., Papamanthou, C., Tomescu, A., Zhang, Y.: Hyperproofs: Aggregating and maintaining proofs in vector commitments. In: 31st USENIX Security Symposium, USENIX Security 2022. pp. 3001–3018. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/srinivasan
42. Tas, E.N., Tse, D., Yang, L., Zindros, D.: Light clients for lazy blockchains. In: Financial Cryptography and Data Security - 28th International Conference, FC 2024, Part II. Lecture Notes in Computer Science, vol. 14745, pp. 3–21. Springer (2025). https://doi.org/10.1007/978-3-031-78679-2_1
43. Team, P.Z.: Plonky2: Fast recursive arguments with plonk and fri. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf (2022), accessed: 2025-03-13
44. Team, P.Z.: Plonky3 repository. https://github.com/Plonky3/Plonky3 (2025)
45. Team, R.Z.: Risc zero repository. https://github.com/risc0/risc0 (2025)
46. Team, T.T.: Tokio. https://tokio.rs (2024)
47. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: Security

and Cryptography for Networks - 12th International Conference, SCN 2020. Lecture Notes in Computer Science, vol. 12238, pp. 45–64. Springer (2020). https://doi.org/10.1007/978-3-030-57990-6_3
48. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5. pp. 1–18. Springer (2008)
49. Vesely, P., Gurkan, K., Straka, M., Gabizon, A., Jovanovic, P., Konstantopoulos, G., Oines, A., Olszewski, M., Tromer, E.: Plumo: An ultralight blockchain client. In: Financial Cryptography and Data Security - 26th International Conference, FC 2022. Lecture Notes in Computer Science, vol. 13411, pp. 597–614. Springer (2022). https://doi.org/10.1007/978-3-031-18283-9_30
50. Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: DIZK: A distributed zero knowledge proof system. In: 27th USENIX Security Symposium, USENIX Security 2018. pp. 675–692. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/wu
51. Wüst, K., Matetic, S., Schneider, M., Miers, I., Kostiainen, K., Čapkun, S.: Zlite: Lightweight clients for shielded zcash transactions using trusted execution. In: Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23. pp. 179–198. Springer (2019)
52. Xie, T., Zhang, J., Cheng, Z., Zhang, F., Zhang, Y., Jia, Y., Boneh, D., Song, D.: zkbridge: Trustless cross-chain bridges made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022. pp. 3003–3017. ACM (2022). https://doi.org/10.1145/3548606.3560652
53. Yakovenko, A.: Solana: A new architecture for a high performance blockchain v0.8.13. Whitepaper (2018), https://solana.com/solana-whitepaper.pdf

A Proof of Theorem 1

This appendix proves Theorem 1: if a client accepts $\mathsf{verifyState}(id, c, v, \pi)$ against a blockchain B, then (id, v) is correct at checkpoint c in the sense of Definition 2, under the unforgeability of the signature scheme, the collision resistance of the hash function, and the knowledge soundness of the proof system. The proof is a sequence of hybrid games, each removing one way for the adversary to win.

Proof. Let an adversary $\mathcal{A}$ output $(id, c, (v, e), \pi)$ and auxiliary data such that the client accepts, but (id, v) is *incorrect* at c. We upper bound $\Pr[\mathcal{A}\text{ wins}]$ by the advantages of breaking our assumptions.

Game 0 (real). This is the real experiment underlying Theorem 1: the verifier runs $\mathsf{verifyState}(id, c, (v, e), \pi)$ exactly as specified. By definition, $\Pr[\text{win in } G_0] = \Pr[\mathcal{A}\text{ wins}]$.

Game 1 (header authentication). In this game, the verifier rejects if the current committee did not sign the header $\mathcal{H}_c$ of B, but the signature verification succeeds. Since the client holds the authentic committee key $\mathsf{pk}_{\mathsf{chain}}$ (committee

handoffs are authenticated from genesis, Section 2), any such header is a forgery, so

$$|\Pr[\text{win in } G_0] - \Pr[\text{win in } G_1]| \leq \mathsf{Adv}^{\mathsf{sig}}_{\mathcal{A}}.$$

Game 2 (knowledge soundness). The verifier checks $\mathsf{ZK.verify}(\Pi_c, \{D_c, d_c, L_c\})$ but aborts if the extractor fails to recover a valid witness. By knowledge soundness of the employed ZK system for all involved statements (including recursive verification), there exists a polynomial-time extractor $\mathcal{E}$ that, on any accepting transcript, outputs a concrete execution trace and witnesses: a sequence of public inputs (D_t, d_t, L_t) for all $t = 0, 1, \ldots, c$, with $(D_0, d_0, L_0) = (D_{\mathsf{init}}, d_{\mathsf{init}}, L_{\mathsf{init}})$ enforced by the circuit base case (Appendix B.2); for each checkpoint $t \in [1..c]$, the batch contents U_t such that the circuit relation $d_t = H_{\mathsf{minnow}}(d_{t-1}, U_t)$ holds; and for each $(id_j, v'_j, e_j) \in U_t$, Merkle witnesses showing that $\mathsf{MT.vfAndUpd}$ transforms (D_{t-1}, p_{id_j}) into D_t at the correct index, with the per-update consistency checks enforced by the circuit. Consequently, conditioned on knowledge soundness, (D_c, d_c, L_c) are uniquely determined by the extracted updates $\{U_t\}_{t=1}^{c}$ and the genesis inputs. Any acceptance with an incorrect (id, v) can only persist if we later find a collision in H_{minnow} or in the Merkle hash (next game), or a bad Merkle inclusion (final game). Therefore,

$$|\Pr[\text{win in } G_1] - \Pr[\text{win in } G_2]| \leq \mathsf{Adv}^{\mathsf{ksnd}}_{\mathcal{A}}.$$

Game 3 (collision resistance). Relative to G_2, we conceptually replace the hash computations by ideal bindings: we require, and check in the analysis, that $d_t = H_{\mathsf{minnow}}(d_{t-1}, U_t)$ and that $\mathsf{MT.vfAndUpd}$ yields the stated D_t for all t. In the extracted trace from G_2 these relations already hold. An adversary that changes batch contents or leaf updates while keeping the same digests produces either a checkpoint whose committed batch contents differ under the same $d_\cdot$, a collision in H_{minnow}, or a Merkle update or inclusion altered under the same root, a collision in the Merkle hash. Hence $|\Pr[\text{win in } G_2] - \Pr[\text{win in } G_3]| \leq \mathsf{Adv}^{\mathsf{crhf}}_{\mathcal{A}}$.

No win in the final game. In G_3, (i) the signed header $\mathcal{H}_c$ of B binds d_c, (ii) the extracted trace and collision resistance fix the unique sequence of updates from genesis yielding D_c, and (iii) the queried (id, v, e) is a leaf under D_c. Therefore v equals the accumulator over all updates to id up to c, so the adversary cannot win. By a telescoping sum over the games,

$$\Pr[\mathcal{A} \text{ wins}] \leq \mathsf{Adv}^{\mathsf{sig}}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{ksnd}}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{crhf}}_{\mathcal{A}} + \mathsf{negl}(\lambda),$$

which is negligible under the stated assumptions.

B Formal statements

This appendix collects the formal material that Section 2 and Sections 3.2 and 3.3 summarize: the Merkle-tree operations, the full statement of the Guppy-A circuit, the procedure for serving and verifying proofs, and the statement of the cyclic circuit of Guppy-B.

B.1 Merkle-tree API

The Guppy protocols use the following Merkle-tree operations.
- $0/1 \leftarrow \mathsf{MT.vf}(id, v, D, path)$: Verify that the value (id, v) is a leaf in a Merkle tree with root D using the provided path $path$.
- $D' \leftarrow \mathsf{MT.vfAndUpd}(id, v, v', D, path)$: Verify inclusion of (id, v) using the path like above. Then, update the stream state to the new value v' along the same path and return the new root.
- $D' \leftarrow \mathsf{MT.insert}(id, v, D, idx)$: Insert the value (id, v) at the given index idx in the Merkle tree with root D and return the new root.

B.2 The Guppy-A circuit

At setup time, we fix the following parameters and hard-code them into the circuit.[4]
1. **Tree depth** d: determines the maximum number of stream IDs supported, $M = 2^d$ (e.g., $d = 30$ for $M = 2^{30}$ stream IDs).
2. **Batch size** b: the batch size used by validators.
3. **Batches per bundle** n: the number of batches processed by a single circuit invocation (leading to a bundle of $N = n \cdot b$ updates).

Circuit inputs and Merkle-tree update. Updating the Merkle tree and deriving the circuit inputs are intertwined, so we describe them together.

The circuit processes a bundle of stream updates at a time denoted as $U^{\mathsf{cir}} = [p'_{id_1}, \ldots, p'_{id_N}]$. In theory, this could represent the updates of a single checkpoint, or even a batch of checkpoints. Suppose the Merkle tree currently represents the state after checkpoint $c-1$; we set $D \leftarrow D_{c-1}$. For each update $p'_{id_j} = (id_j, v'_j, e_j)$ in the bundle U^{cir}:
1. If $e_j = 0$, retrieve leaf (id_j, v_j) and its Merkle path $path_{id_j}$, and update:
$$D \leftarrow \mathsf{MT.vfAndUpd}(id_j, v_j, v'_j, D, path_{id_j}).$$
2. If $e_j = 1$, insert (id_j, v'_j) at the index L, and compute its corresponding path $path_{id_j}$:
$$D \leftarrow \mathsf{MT.insert}(id_j, v'_j, D, L).$$

The tuples $\{p_{id_j}, p'_{id_j}, path_{id_j}\}$ (for new streams, $p_{id_j} = \perp$) constitute the private inputs to the ZK circuit. If U^{cir} corresponds to the updates of a single checkpoint c, then the above process computes the new Merkle root after checkpoint c, i.e., $D = D_c$.

[4] Unlike many past cryptographic constructions, cheap recursion allows an easy way to change parameters: a new circuit with different parameters can simply verify the old proof in its base case, and new proofs thereafter.

Circuit statement (S_A). We now specify the Guppy-A ZK circuit's statement S_A. In the base case, it accepts $\Pi_{c'} = \perp$ and sets all the public inputs to their initial values: $c_{\mathsf{init}} = 0$, $D_{\mathsf{init}} = 0$, $d_{\mathsf{init}} = 0$, $L_{\mathsf{init}} = 0$.

- **Private inputs:**
  1. Previous proof $\Pi_{c'}$ with its public inputs $pub^{c'} = \{D_{c'}, d_{c'}, L_{c'}, c'\}$ (set to $\perp$ in the base case);
  2. Updates: $\{(p_{id_j}, p'_{id_j}, path_{id_j})\}_{j\in[N]}$.
- **Public inputs:** $pub^{c} = \{D_c, d_c, L_c, c\}$.
- **Statement logic:**
  1. *Base case:* If $c = c_{\mathsf{init}}$, set $D_c \leftarrow D_{\mathsf{init}}$, $d_c \leftarrow d_{\mathsf{init}}$, $L_c \leftarrow L_{\mathsf{init}}$, and accept. Else assert $c = c' + 1$.
  2. *Verify prior proof:* $\mathsf{ZK.verify}(\Pi_{c'}, pub^{c'})$.
  3. *Update Merkle tree:* Initialize $D \leftarrow D_{c'}$, $L \leftarrow L_{c'}$. For each update $j \in [N]$:
     - Update root: $D \leftarrow \mathsf{MT.vfAndUpd}(id_j, v_j, v'_j, D, path_{id_j})$;
     - If $e_j = 1$, assert $path_{id_j}.\text{index} = L$ and increment L by one.
     - If $e_j = 0$, assert $p_{id_j}.id = p'_{id_j}.id$.
  4. *Checks:* Assert $D = D_c$, $L = L_c$, and $d_c = H_{\mathsf{minnow}}(d_{c'}, \{p'_{id_j}\}_{j\in[N]})$.

The circuit outputs the updated proof Π_c attesting to the correctness of the new global state commitment D_c.

B.3 Serving and verifying proofs

Suppose a client queries the state of a stream identifier id at checkpoint c. The ZK service computes $(v, \pi) \leftarrow \mathsf{readState}(id, c)$, where $\pi = \{\Pi_c, \mathcal{H}_c, path_{id}\}$. The proof consists of three components:

- The recursive ZK proof Π_c with its public inputs $\{D_c, d_c, L_c, c\}$.
- The checkpoint header $\mathcal{H}_c$, which includes the stream update commitment d_c and a validator signature over the header (see Section 2).
- The Merkle path $path_{id}$ for the queried stream.

The client verifies the proof using the committee keys $\mathsf{pk}_{\mathsf{chain}}$ (Section 2) as follows:

1. Check the Merkle path: $\mathsf{MT.vf}(id, v, D_c, path_{id})$.
2. Verify the validator-signed checkpoint header: $\mathsf{Sig.verify}(\mathcal{H}_c, \mathsf{pk}_{\mathsf{chain}})$, and ensure that $\mathcal{H}_c.d = d_c$ and $\mathcal{H}_c.c = c$.
3. Verify the ZKP: $\mathsf{ZK.verify}(\Pi_c, \{D_c, d_c, L_c, c\})$.

This procedure corresponds to the API call $\mathsf{verifyState}(id, c, v, \pi)$.

B.4 The cyclic circuit of Guppy-B

The statement S_{seq} of the cyclic circuit is as follows:

- **Private inputs:** (i) last proof and inputs: $\Pi^{c'}_{\mathsf{seq}}$, $pub^{c'}_{\mathsf{seq}} = \{D_{c'}, d_{c'}, L_{c'}, c'\}$, (ii) aggregated proof for checkpoint c: $\Pi^{(c')\to(c)}_{\mathsf{agg}}$.
- **Public inputs:** $pub^{c}_{\mathsf{seq}} = \{D_c, d_c, L_c, c\}$.
- **Statement:**

1. Verify the previous sequential proof: $\mathsf{ZK.verify}(\Pi^{c'}_{\mathsf{seq}}, pub^{c'}_{\mathsf{seq}})$ and check $c = c' + 1$.
2. Construct public inputs for aggregation circuit: $pub^{(c')\to(c)}_{\mathsf{agg}} = \{D_{c'}, D_c, d_{c'}, d_c, L_{c'}, L_c\}$ and verify $\mathsf{ZK.verify}(\Pi^{(c')\to(c)}_{\mathsf{agg}}, pub^{(c')\to(c)}_{\mathsf{agg}})$.

C Guppy-Sparse

We now describe a lightweight variant of Guppy in which the ZK service tracks only a *fixed, small set* of stream IDs $\mathcal{ID} = \{id_1, \ldots, id_m\}$, rather than all streams on the chain. This is useful when an application cares about completeness only for a few selected streams and wishes to minimize proving costs.

The ZK service maintains an internal state $V^c = [(id_i, v_i)]_{i=1}^m$ after checkpoint c, and commits to it using a simple hash

$$D_c = H(V^c).$$

Validators, as in the main protocols, compute a per-checkpoint hash-chain commitment d_c over the (sorted) stream updates they derive from checkpoint contents, and include d_c in the checkpoint header.

At each checkpoint, the recursive circuit for Guppy-Sparse takes as input: (i) the previous proof and its public inputs (D_{c-1}, d_{c-1}), (ii) the validator-supplied stream updates for this checkpoint, and (iii) the previous state V^{c-1}. It checks that:

– the prior proof is valid and consistent with D_{c-1} and d_{c-1},
– applying the checkpoint's updates to V^{c-1} yields a new state V^c consistent with the tracked stream set $\mathcal{ID}$, and
– the public outputs satisfy $D_c = H(V^c)$ and $d_c = H_{\mathsf{minnow}}(d_{c-1}, U^{\mathsf{bc}}_c)$.

A client that wants the state of some $id \in \mathcal{ID}$ at checkpoint c receives: (i) the claimed value v, (ii) the full list V^c (of size m), (iii) the recursive proof for c, and (iv) the signed checkpoint header. To verify, the client (a) checks that (id, v) appears in V^c, (b) recomputes $H(V^c)$ and matches it with the proof's D_c, (c) verifies the recursive proof, and (d) verifies the validator signature on the header and the embedded d_c. For small m, this yields short proofs and constant-time verification for each query.

D Evaluation details

This appendix supplements Section 4 with the measurements and derivations behind the numbers quoted there: the circuit sizes that fix the batch and sub-bundle parameters (Appendix D.1), the number of machines that Guppy-B needs and the full special-configuration table (Appendix D.2), and the per-prover latency breakdown of the end-to-end runs (Appendix D.3).

Batch size (b)	Gates
10	50000
100	49890
1000	49876

Table 3. Size of the base circuit for varying batch sizes; the circuit processes $y = 1000$ updates (batch size times number of batches).

Degree	Sub-bundle size
14	200
15	450
16	900
17	1900

Table 4. Maximum sub-bundle size ($n \cdot b$ updates) that the base circuit S_{par} of Guppy-B handles at each Plonky2 degree, tested in increments of fifty with $M = 2^{30}$ and $b = 50$.

D.1 Circuit sizes

Two circuit-size measurements determine the parameters used in Section 4. Table 3 shows that the batch size b barely changes the size of the base circuit, so it can be chosen for the validators' convenience (Section 4.1). Table 4 gives the largest sub-bundle the base circuit of Guppy-B handles at each Plonky2 degree, which fixes the sub-bundle size y of the special configuration.

D.2 Number of machines for Guppy-B

To keep pace with the chain, Guppy-B needs

$$1 + \beta \cdot \left(\frac{t_{\mathsf{par}}}{t_{\mathsf{bundle}}}\right) + \beta \cdot \left(\frac{t_{\mathsf{agg}}}{t_{\mathsf{bundle}}}\right) \cdot \left(\frac{1 - \frac{1}{\alpha^l}}{\alpha - 1}\right).$$

machines: $\beta \cdot t_{\mathsf{par}}/t_{\mathsf{bundle}}$ for the parallel phase, $\beta \cdot (\frac{1}{\alpha} + \frac{1}{\alpha^2} + \cdots + \frac{1}{\alpha^l}) \cdot t_{\mathsf{agg}}/t_{\mathsf{bundle}}$ for the aggregation phase, and one for the sequential phase.

In the special configuration of Section 4.3, where $\beta = 3^l$, this gives

$$1 + \beta + \beta(\tfrac{1-\frac{1}{\alpha^l}}{\alpha-1}) = 1 + 3^l + (3^l - 1)/2 = (3^{l+1} + 1)/2.$$

This special configuration performed best among the parameter choices we experimented with, but we do not claim it is globally optimal. Choosing parameters for Guppy-B is a multi-dimensional optimization problem: for a given target throughput (which fixes the bundle size N), one would like to pick the batch size, sub-bundle size, and aggregation factor that minimize end-to-end latency. A natural dual formulation is to fix a cap on the number of machines and ask for the configuration that maximizes sustainable throughput. We leave a systematic exploration of this design space to future work.

D.3 Latency breakdown

This section expands on Section 4.3 by analyzing the individual components of the latency of Guppy-B. Figure 5 shows the breakdown of prover latency for bundle sizes of $N = 200$, $N = 600$, and $N = 1{,}800$. As predicted by the microbenchmarks in Section 4, the median latency of each prover stays at about 0.63 s, independent of the bundle size.

Updates (N)	Levels (l)	Latency (t_{e2e}, s)	Throughput (updates/s)	Machines
1–200	0	1.26	317	2
201–600	1	1.89	952	5
601–1800	2	2.52	2857	14
1801–5400	3	3.15	8571	41
5401–16200	4	3.78	25714	122

Table 5. Performance of Guppy-B for different bundle sizes at the special configuration (see text). Throughput and machines correspond to fully utilized bundles (largest N in each row).

The partial witness builder (*Pw builder* in the figure) is the time the primary takes to construct the partial witness. Its latency grows slightly with the number of workers, and thus with the bundle size, but remains low.

The figure corroborates the scalability claims of Section 4: prover latency is invariant in the number of workers. Consequently, any observed differences between the predicted and measured end-to-end latencies can be attributed to additional overheads, such as network latency, client-side queuing delays, context switching on the primary machine, and the time required to assemble the partial witnesses.