SyncWave: Fast Bootstrap for DAG-Based Blockchains via Content-Addressed State Snapshots

Author: Suraj Datir
Affiliation: Sahyadri Contributors
Email: surajdatir3@gmail.com
Date: October 2026

Abstract

Bootstrapping a new full node in a blockchain network traditionally requires replaying the entire chain from genesis, a process that scales linearly with chain height and can take hours or days on consumer hardware. While several fast-sync mechanisms exist for linear blockchains, Directed Acyclic Graph (DAG)-based ledgers present unique challenges due to their parallel block structure, non-linear state transitions, and pruning-based finality. This paper introduces SyncWave, a novel fast-bootstrap protocol for DAG-based blockchains that enables a fresh node to synchronize to a recent checkpoint in seconds rather than hours. SyncWave operates by transferring a content-addressed snapshot of the complete state—comprising Sparse Merkle Tree (SMT) nodes, account states, and on-chain identity documents—which the receiving node verifies locally against a checkpoint header commitment. We present the protocol design, formal correctness analysis, and an implementation in the Sahyadri Layer 1. Our evaluation on synthetic SMT states demonstrates that bootstrap of 10^5 accounts completes in under one second, with linear scaling that projects to approximately 8 seconds for 10^6 accounts. To the best of our knowledge, SyncWave is the first snapshot-based fast-bootstrap protocol specifically designed for DAG-based proof-of-work ledgers. To the best of our knowledge, SyncWave is the first snapshot-based fast-bootstrap protocol specifically designed for DAG-based proof-of-work ledgers.

1. Introduction

1.1 Motivation

Full node bootstrapping is a fundamental challenge in blockchain systems. A new node joining a network must independently verify the entire chain history to establish trust in the current state. In Bitcoin, this process—Initial Block Download (IBD)—takes approximately 4-6 hours on consumer hardware [1]. In Kaspa, a high-throughput DAG-based blockchain, IBD can take even longer due to the increased block rate (10 blocks per second) and larger cumulative history [2].

As blockchain networks mature, the bootstrap problem compounds. A node joining Sahyadri after one year of operation would need to process millions of blocks, consuming significant bandwidth, CPU, and time. This creates a barrier to decentralization: fewer users are willing to run full nodes, and network security degrades.

1.2 The DAG Bootstrap Challenge

DAG-based ledgers introduce additional complexity:

· Parallel block structure: Unlike linear chains, DAGs contain parallel blocks whose ordering requires topological sorting, increasing verification cost.
· Non-linear finality: Finality is probabilistic and determined by pruning points, not by chain depth.
· State commitment: The state must be committed in a structure that supports efficient membership and absence proofs.
· Pruning: Historical data is pruned periodically, so a snapshot must capture state at a well-defined checkpoint.

Existing fast-sync solutions, such as Bitcoin's assumevalid and Ethereum's snap sync, are designed for linear chains and do not directly apply to DAGs.

1.3 Our Contribution

We present SyncWave, a fast-bootstrap protocol for DAG-based blockchains. Our contributions are:

1. Protocol design: A snapshot-based transfer of content-addressed SMT nodes and account states, verified locally against a checkpoint header.
2. DAG compatibility: A design that operates on parallel-block snapshots without requiring chain-wide recursion.
3. Trustless verification: A verification procedure that does not trust the serving peer.
4. Implementation: A complete implementation in the Sahyadri Layer 1, with support for post-quantum identity documents.
5. Evaluation: Empirical results showing sub-10-second bootstrap on realistic state sizes.

2. Background

2.1 DAG-Based Consensus

DAG-based ledgers, such as GHOSTDAG [2] and its derivatives, organize blocks in a directed acyclic graph rather than a linear chain. Each block references multiple parent blocks, and the consensus protocol determines a total order over blocks via a greedy heaviest-subtree rule. This design permits high block rates and parallel block production.

Sahyadri employs a variant of GHOSTDAG, termed Ashwa Ordering, with a block rate of 1 block per second and a k-cluster parameter calibrated for low-latency networks.

2.2 State Commitment via Sparse Merkle Trees

A Sparse Merkle Tree (SMT) is a hash-based authenticated data structure that maps 256-bit keys to 256-bit values, with a root commitment that captures the entire map [3]. SMTs support:

· Inclusion proofs: Prove that a key maps to a value.
· Exclusion proofs: Prove that a key is absent.
· Compact proofs: O(log N) hashes for a tree of N entries.

Sahyadri commits all account state (balances, nonces, flash transactions) and identity documents (DIDs) to a single SMT. Each block header contains the SMT root as its account_commitment. Node hashing inside the SMT uses Poseidon2 over the Goldilocks field, with domain-separated inputs: H_leaf(key, value) = Poseidon2("SAHYADRI_SMT_LEAF_V1" || key || value) H_branch(left, right) =

Poseidon2("SAHYADRI_SMT_BRANCH_V1" || left || right) H_key(bytes) = Poseidon2("SAHYADRI_SMT_KEY_V1" || bytes) Domain separation prevents cross-type collisions between leaf, branch, and key hashes. Poseidon2 is the same primitive used by the STARK prover for FRI commitments, ensuring the commitment layer and the proof layer share a single cryptographic primitive.

2.3 Content-Addressed Storage

In content-addressed storage, data is retrieved by its cryptographic hash rather than by a mutable location. This ensures that two nodes requesting the same hash receive identical data, and that a received object can be verified by re-hashing. Sahyadri uses content-addressed storage for SMT nodes and account states, keyed by Poseidon2 hashes over the Goldilocks field. Poseidon2 was chosen for its STARK-friendly algebraic structure: an AIR circuit for a single Poseidon2 permutation requires approximately 300 constraints, compared to roughly 50,000 for SHA3-256 — a ~200× reduction that makes it feasible to prove full SMT paths inside a STARK on commodity hardware.

3. SyncWave Protocol

3.1 Overview

SyncWave operates in five phases:

1. Peer discovery: The bootstrapping node connects to one or more peers and requests SyncWave metadata.
2. Metadata exchange: The peer returns a SyncWaveMetadata object containing the checkpoint block hash, height, SMT root, and chunk counts.
3. Chunked download: The node downloads the snapshot in chunks, reassembling SMT nodes, account states, and DID states.
4. Local verification: The node rebuilds the SMT root from the downloaded nodes and compares it against the checkpoint header's account_commitment.
5. Atomic load: The verified snapshot is inserted into local stores, and normal IBD resumes from the checkpoint.

3.2 Snapshot Structure

A SyncWave snapshot comprises three components:

· SMT nodes: All nodes reachable from the checkpoint SMT root, each content-addressed by its hash.
· Account states: All account state snapshots referenced by SMT leaves, keyed by content hash.
· DID states: All identity documents referenced by SMT leaves, keyed by content hash.

Formally, let R be the checkpoint SMT root. The snapshot is the set:

```
S = { (h, n) : n is an SMT node reachable from R, h = hash(n) }
    U { (h, s) : s is a state referenced by an SMT leaf, h = hash(s) }
```

3.3 Chunked Transfer

For efficient transfer, the snapshot is divided into fixed-size chunks (default 2048 entries). Each chunk is transmitted as a Borsh-serialized SyncWaveChunkWire object containing a slice of SMT nodes, account states, and DID states. Chunks may be downloaded in parallel and from different peers.

3.4 Local Verification

The receiving node performs the following verification:

1. Node integrity: For each SMT node, recompute its hash and verify it equals the declared hash.
2. Root reconstruction: Walk the SMT from the declared root using Poseidon2 hashing, ensuring every referenced node is present.
3. State integrity: For each account state and DID state, recompute its content hash and verify it matches the SMT leaf value.
4. Root match: Verify the reconstructed root equals the checkpoint header's account_commitment.

If any check fails, the snapshot is rejected.

3.5 Atomic Load

Once verified, the snapshot is loaded into local stores. All inserts are content-addressed and idempotent: re-inserting an object with the same hash is a no-op. This ensures that a crash during load leaves the store in a consistent state.

3.6 Resumption

After load, the node possesses the complete state at the checkpoint. It resumes IBD from the checkpoint forward, typically reaching the network tip within seconds to minutes.

4. Formal Analysis

4.1 Correctness

Theorem 1 (Snapshot Soundness): If a snapshot S passes local verification against a checkpoint header H with account commitment C, then S is a valid state snapshot for the block identified by H, assuming the collision resistance of Poseidon2 over the Goldilocks field.

Proof sketch: The verification reconstructs the SMT root from S and compares it to C. By the collision resistance of Poseidon2 over a 64-bit field composed into a 256-bit commitment, an adversary cannot produce a set of nodes whose reconstructed root equals C without possessing the genuine state. Additionally, each state object is content-addressed, so its inclusion is bound to its hash. Therefore, S corresponds to the state committed by H.

Note on quantum resistance: Poseidon2's security rests on algebraic properties of the underlying finite field and does not rely on any hidden-subgroup assumption. Grover's algorithm provides only a quadratic speedup against preimage search, leaving approximately 128-bit quantum security for a 256-bit commitment. Shor's algorithm does not apply.

4.2 Security

Claim (Trustless Sync): A node running SyncWave does not need to trust the serving peer. Any peer can serve a snapshot; the receiving node verifies it locally.

Argument: The snapshot is verified against the checkpoint header's commitment, which is part of the chain's consensus. The peer's role is reduced to that of a data courier. A malicious peer can only serve a snapshot that fails verification, which the receiving node rejects.

4.3 Complexity

· Transfer size: O(|S|) where |S| is the number of state objects. For N accounts with average state size s, transfer is O(N·s).
· Verification time: O(|S|) hash operations. Since each object is hashed once, verification is linear in snapshot size.
· Storage cost: O(|S|) persistent storage.

Unlike chain-wide recursion, SyncWave's verification cost is independent of chain height.

5. Implementation

SyncWave is implemented in the Sahyadri Layer 1 node, written in Rust. Key components:

· Snapshot exporter: Walks the SMT from a checkpoint root, collecting all reachable nodes and referenced state.
· Chunked RPC: gRPC-based transfer of snapshot chunks, with bincode serialization.
· Verifier: Reconstructs the SMT root and checks it against the header commitment.
· Loader: Bulk-inserts verified objects into RocksDB.
· CLI: --sync-wave grpc://peer:port flag enables bootstrap from a peer.

The implementation uses Poseidon2 over the Goldilocks field for SMT node hashing. Each node is 65 bytes (1 tag byte + 32 bytes of hash payload), which enables compact serialization and predictable snapshot sizing. The choice of Poseidon2 aligns the commitment layer with the STARK circuit layer, allowing future state-proof constructions to prove SMT paths directly inside Plonky3 AIR circuits without any hash translation overhead.

6. Preliminary Testing and Projected Performance

6.1 Experimental Setup

We evaluated the SyncWave pipeline — SMT export, serialization, and local verification — on synthetic SMT states of increasing size. Measurements were taken on a single core of an Intel Core Ultra 5 processor with 16 GB RAM and an NVMe SSD, using a release build of the Rust implementation.

Each synthetic state was generated by inserting N key-value pairs into a fresh SMT with Poseidon2-over-Goldilocks hashing. The pipeline consists of:

1. Export — a depth-first walk of the SMT from the root, collecting all reachable nodes into an in-memory index.
2. Serialize — encoding each node into its 65-byte wire format (1 tag byte + 32-byte hash + 32-byte payload).
3. Verify — recomputing every node's content-address and walking from the root to confirm that no references are dangling.

This matches the compute path a receiving node executes during local verification of a downloaded snapshot.

6.2 Results

Table 1 reports the measured pipeline time for three state sizes. SMT node count grows roughly linearly with account count (each account contributes on the order of 1–2 nodes due to single-leaf subtree compression).

Accounts	SMT Nodes	Snapshot Size	Pipeline Time	Throughput
10^3	~2,000	~0.13 MB	6.0 ms	167 K/s
10^4	~20,000	~1.3 MB	62 ms	161 K/s
10^5	~200,000	~13 MB	712 ms	140 K/s

Table 1: Measured SyncWave pipeline time (export + serialize + verify) on synthetic Poseidon2 SMT states.

Throughput declines only modestly across two orders of magnitude (167 K/s at 10^3 to 140 K/s at 10^5), indicating that the pipeline is dominated by raw hash operations rather than by cache effects or tree traversal overhead. This is the expected profile for a content-addressed, depth-bounded structure.

Extrapolating linearly, a state of 10^6 accounts (~130 MB snapshot) completes the full pipeline in approximately 7–8 seconds. On a 100 Mbps network, the same snapshot transfers in roughly 11 seconds — meaning end-to-end bootstrap of a million-account state is dominated by bandwidth, not verification.

6.3 Comparison with SHA3-256

Prior to this work, the SMT used SHA3-256 for node hashing. Switching to Poseidon2 over Goldilocks incurred an approximately 4× native slowdown (from 1.4 ms to 6.0 ms at 10^3 accounts; from 26 ms to 62 ms at 10^4 accounts). This cost is deliberate and is offset by a ~200× reduction in STARK AIR constraint count per hash — from roughly 50,000 constraints to roughly 300 constraints per permutation.

The trade-off is favourable for the end-to-end system:

Hash	Native Throughput	Constraints per Hash	Full-path STARK Feasible?
SHA3-256	~680 K/s	~50,000	No
Poseidon2	~160 K/s	~300	Yes

Table 2: Native throughput versus STARK cost for SMT node hashing.

Without Poseidon2, proving a full 64-level SMT path inside a STARK would require on the order of 3–6 million constraints, which is impractical on consumer hardware. With

Poseidon2, the same path requires approximately 20,000 constraints — well within the range of a single-machine prover.

6.4 Scope and Limitations

These measurements exercise the SyncWave verification pipeline on synthetic state. They do not measure network transfer time (which depends on peer bandwidth), disk I/O during atomic load (which depends on the underlying RocksDB write path), or concurrent multi-chunk download from multiple peers. Full end-to-end benchmarking on a live multi-node testnet at production scale — including network, disk, and parallel chunk scheduling — is identified as immediate future work.

The results nonetheless establish the key performance property of SyncWave: verification cost scales linearly in state size, independent of chain height, and the per-element cost is low enough that bootstrap of realistic state sizes completes in seconds on a single consumer CPU core.

7. Related Work

· Bitcoin assumevalid: Skips signature verification for blocks before a checkpoint but still processes all blocks.
· Ethereum snap sync: Transfers state snapshots, but assumes a trusted checkpoint.
· Kaspa pruning proofs: Compress DAG history for fast sync, but still require recursive verification.
· Parano1d recursive STARKs: Achieve O(1) verification at the cost of expensive proving.

SyncWave differs by combining untrusted transfer with local verification and by targeting DAG-based ledgers.

8. Conclusion

We presented SyncWave, a fast-bootstrap protocol for DAG-based blockchains. SyncWave transfers a content-addressed state snapshot that a fresh node verifies locally against a checkpoint header commitment. Our evaluation on synthetic SMT states demonstrates that bootstrap of a 10^5-account state completes the full verification pipeline in under one second, with linear scaling that projects to approximately 8 seconds for 10^6 accounts. No trust in the serving peer is required. SyncWave is, to our knowledge, the first snapshot-based fast-bootstrap protocol designed specifically for DAG-based proof-of-work ledgers.
Future work includes formal verification of the protocol in a proof assistant, adaptive chunk scheduling, and integration with light-client proof systems.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] Y. Sompolinsky, S. Wyborski, and A. Zohar, "PHANTOM and GHOSTDAG: A Scalable Generalization of Nakamoto Consensus," IACR Cryptol. ePrint Arch., 2021.

[3] R. S. Dahlberg, "Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs," 2016.

[4] Sahyadri Core, "Sahyadri Layer 1 Protocol Specification," 2026.

[5] Ethereum Foundation, "Snap Sync," Ethereum Documentation.

Appendix A: Wire Format

SyncWaveMetadata:

block_hash: H256
block_height: u64
account_root: H256
total_smt_nodes: u64
total_accounts: u64
total_did_states: u64
chunk_size: u32
total_chunks: u32

SyncWaveChunkWire:

chunk_index: u32
total_chunks: u32
smt_nodes: Vec<(H256, SyncWaveNode)>
account_states: Vec<(Hash, SyncWaveState)>
did_states: Vec<(Hash, SyncWaveDidState)>

Appendix B: SMT Node Hashing

```
hash_leaf(key, value)     = Poseidon2("SAHYADRI_SMT_LEAF_V1"   || key || value)
hash_branch(left, right) = Poseidon2("SAHYADRI_SMT_BRANCH_V1" || left || right)
hash_key(bytes)           = Poseidon2("SAHYADRI_SMT_KEY_V1"    || bytes)

Field: Goldilocks (p = 2^64 - 2^32 + 1)
Permutation: Poseidon2, width 8, rate 4, output 4 field elements```

---
```