Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: 7 November 2026

D. Condrey
WritersLogic

6 May 2026

Proof of Sequential Memory Execution (PoSME)
draft-condrey-cfrg-posme-01

Abstract

This document defines Proof of Sequential Memory Execution (PoSME), a cryptographic primitive combining mutable arena state, data-dependent pointer-chase addressing, and per-block causal hash binding in a single step function. A Prover executes K sequential steps over a mutable N-block arena. Each step reads d blocks at addresses determined by the previous read's result (pointer chasing), writes one block with spatial neighborhood entanglement (incorporating A\[w-1\] and A\[w+1\]), and advances a transcript chain. The construction provides three properties: (1) unconditional sequential time enforcement anchored in physics-bounded latency floors, (2) forgery prevention via causal hashes (reduces to collision resistance of H), and (3) TMTO resistance scaling geometrically with write density rho. Verification requires O(Q * d^R * log N) hash evaluations with no arena allocation. No trusted setup is required.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 7 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction

Existing primitives for proving sequential computation have complementary weaknesses. Verifiable Delay Functions (VDFs) [Boneh2018] [Wesolowski2019] prove sequential time but offer no memory-hardness. Proofs of Sequential Work (PoSW) [CohenPietrzak2018] prove traversal of a depth-robust graph but operate over static memory. Memory-hard functions (MHFs) such as Argon2id [RFC9106] and scrypt resist ASIC acceleration by requiring significant memory resources. While scrypt was designed to be bounded by the latency of its core functions, many MHFs are practically constrained by memory bandwidth when comparing commodity hardware to specialized ASICs.

PoSME takes a different approach. A persistent mutable arena IS the computation state. Each step reads via data-dependent pointer chasing (sequential because each address depends on the previous read's result) and modifies the arena in-place. A per-block causal hash chain binds each block's value to the cursor of the step that wrote it, preventing forgery: the adversary cannot produce a valid causal hash without knowing the writer's cursor, which depends on d other blocks' causal hashes, recursively. The data and causal hash are symbiotically bound: new data depends on the old causal hash, and the new causal hash depends on the cursor.

The primary contributions are (a) a physics-bounded latency floor with cross-generation durability and (b) linear TMTO scaling with write density rho. Unlike bandwidth-bound constructions where the ASIC advantage scales with technology improvements, PoSME is

bottlenecked by random memory access latency. For arena sizes exceeding on-die SRAM, the ASIC advantage is bounded by the latency ratio of specialized memory (such as HBM3) to commodity DDR5. While an adversary with massive on-die SRAM (e.g., wafer-scale integration) achieves a significant latency advantage, the bound remains durable across technology generations as it is constrained by signal propagation and DRAM cell sensing time.

1.1. Related Work

1.1.1. Proofs of Sequential Work

PoSW [CohenPietrzak2018] proves traversal of a depth-robust graph via Fiat-Shamir-sampled Merkle proofs. PoSME differs: the graph is a mutable arena (not a static DAG), the access pattern is data-dependent (not fixed), and each node carries a causal hash binding its value to its full write history.

1.1.2. Memory-Hard Functions

Functions like Argon2id and scrypt resist ASIC acceleration by imposing high memory requirements. Argon2id [RFC9106] resists TMTO via memory-hardness [Boneh2016], with a single-pass TMTO penalty of approximately 2x. PoSME uses a custom logarithmic skip-link initialization (Section 3.2) to ensure Omega(sqrt(N)) space-hardness from the first step. The ongoing computation uses pointer-chasing with in-place writes, creating a latency-bound bottleneck. PoSME’s TMTO penalty is approximately 2+2*rho for zero-storage adversaries, where rho = K/N is the write density.

1.1.3. Proofs of Space-Time

Proofs of Space-Time (PoST) [Chia2024] [Spacemesh2023] enforce both sequential time and persistent storage by requiring a Prover to repeatedly prove possession of stored data over a sequence of time intervals. PoST operates over a static graph: the stored data does not change between proofs, and the graph structure is fixed before execution. PoSME differs in that the arena is mutable (each step modifies it), the access pattern is data- dependent (addresses are determined by arena contents, not pre-computed), and each block carries a causal hash binding its current value to its write history. These differences make PoSME a different construction with different TMTO characteristics, not a strict improvement over PoST.

1.1.4. Cumulative Memory Complexity

Alwen, Blocki, and Pietrzak [AlwenBlockPietrzak2017] formalized cumulative memory complexity for static graph pebbling games. PoSME's causal dependency DAG is dynamic (edges are created during execution), requiring a new pebbling framework. The dynamic pebbling analysis is provided in Section 5.4.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

H: BLAKE3 or SHA-3, producing 32-byte output. BLAKE3 is MANDATED to ensure post-quantum resistance in sequential chains.

XOF(input, index): H evaluated at (input || I2OSP(index, 4)), producing 4 bytes.

I2OSP(x, len): Integer-to-Octet-String Primitive per [RFC8017].

MerkleRoot(A): Merkle tree root over arena blocks using domain-separated hashing per [RFC6962].

MerkleUpdate(root, index, new_value): Incremental Merkle root update at the given index.

Prover: The entity executing the PoSME computation and generating proofs.

Verifier: The entity checking PoSME proofs.

Arena: A mutable array of N blocks, each containing a 32-byte data field and a 32-byte causal hash.

Causal hash: A per-block running hash chain binding each block's value to the cursor of the step that wrote it.

3. Construction

3.1. Arena Block Format

Each arena block is a pair:

```
block = {
    data:   bytes[32],
    causal: bytes[32]
}
```

The data field stores the block's computational value. The causal field stores the causal hash chain: a running digest binding the block's current value to the cursor of the step that last wrote it.

3.2. Arena Initialization

The arena is initialized deterministically from a public seed s:

```
for i in 0..N-1:
    if i == 0:
        A[0].data = H("PoSME-init-v1" || s || I2OSP(0, 4))
    else:
        A[i].data = H("PoSME-init-v1" || s || I2OSP(i, 4)
                      || A[i-1].data
                      || A[floor(i/2)].data)
    A[i].causal = H("PoSME-causal-v1" || s || I2OSP(i, 4))

root_0 = MerkleRoot(A)
T_0 = H("PoSME-transcript-v1" || s || root_0)
```

The initialization references both the preceding block (A[i-1]) and a logarithmic skip-link (A[floor(i/2)]). This creates a dependency DAG of depth log(N) and width N, requiring Omega(sqrt(N)) space to evaluate (the DAG cannot be streamed in constant space because each block depends on a block approximately N/2 positions behind it). A custom initialization is used rather than Argon2id because Argon2id's fixed internal graph does not provide this skip-link structure; the logarithmic back-references are necessary for the space-hardness property.

The Verifier can independently compute root_0 and T_0 from the seed, providing a trusted anchor for all subsequent verification.

3.3. Step Function

The step function is the core of PoSME. It enforces sequentiality via pointer-chasing, hardware parity via forced intra-step bank collisions, and geometric TMTO resistance via spatial neighborhood entanglement.

At each step t in {1, ..., K}:

```
   STEP(t):
       cursor = T_{t-1}

       // 1. Determine Target Bank
       // Use the first 4 bytes of XOF to select 1 of B_banks
       bank_id = OS2IP(XOF(cursor, 0)) mod params.B_banks

       // Start high-resolution cycle counter
       t_start = RDTSC()

       // 2. Intra-Step Bank Collision Reads
       addrs = []
       for j in 0..d-1:
           // Generate pseudo-random address
           raw_a = OS2IP(XOF(cursor, j + 1)) mod params.N

           // Mutate raw_a to ensure it maps to bank_id
           a = force_bank_mapping(raw_a, bank_id, params)
           addrs.append(a)

           val = A[a]
           cursor = H(cursor || val.data || val.causal)

       // 3. Write with Spatial Neighborhood Entanglement
       raw_w = OS2IP(XOF(cursor, d + 1)) mod params.N
       w = force_bank_mapping(raw_w, bank_id, params)
       old = A[w]

       // Incorporate causal hashes of logical neighbors
       n_prev = A[(w - 1) mod params.N].causal
       n_next = A[(w + 1) mod params.N].causal

       new_data = H(old.data || cursor || old.causal
                    || n_prev || n_next)
       new_causal = H(old.causal || cursor || I2OSP(t, 4)
                      || n_prev || n_next)
       A[w] = {data: new_data, causal: new_causal}

       // Stop cycle counter to capture physical latency jitter
       t_end = RDTSC()
       delta_t = t_end - t_start

       // 4. Update Commitments
       root_t = MerkleUpdate(root_{t-1}, w, A[w])
       T_t = H(T_{t-1} || I2OSP(t, 4) || cursor || root_t || I2OSP(delta_t, 8))

       // 5. Log step for Prover transcript
       log[t] = {addrs, w, old, A[w], cursor, root_t, delta_t}
```

3.3.1. Intra-Step Bank Collisions

Standard memory controllers achieve high bandwidth by interleaving sequential reads across multiple hardware banks, keeping multiple row-buffers open. PoSME explicitly defeats this optimization to enforce a strict latency floor.

The force_bank_mapping(raw_a, bank_id, params) function modifies the specific bits of the logical address raw_a that the memory controller uses for bank selection, replacing them with bank_id.

By forcing all d reads and the final write to target the _same_ physical bank but _different_ pseudo-random rows, the memory controller suffers a "Bank Conflict" on every access. This forces a physical Row Precharge (t_{RP}) and RAS-to-CAS Delay (t_{RCD}) penalty for every hop, anchoring the execution time to the thermodynamic limits of the DRAM capacitor rather than the logic speed of the processor.

3.3.2. Spatial Neighborhood Entanglement

The write step cryptographically binds the updated block to the current state of its logical neighbors, A[w-1] and A[w+1].

This transforms the Time-Memory Trade-Off (TMTO) penalty from a linear chain into a geometric "Trophic Cascade." If an adversary attempts to save space by discarding a subset of the arena, recomputing a single missing block w requires knowing the causal hashes of its neighbors at the exact moment of the write. If those neighbors were also discarded, the recomputation recursively branches. This bounds the adversary to strict storage compliance, as storage reduction triggers exponentially scaling recomputation costs.

3.3.3. Timing Entropy Attestation

Because commodity DRAM requires periodic electrical refresh cycles (t_{REFW}), a genuine physical execution will exhibit unavoidable, stochastic latency spikes.

The Prover measures the execution time of the read/write loop using a monotonic, high-resolution hardware counter (e.g., the RDTSC instruction on x86 architectures). This inter-arrival time, delta_t, is folded directly into the transcript T_t. A Verifier auditing the transcript can perform statistical variance testing on the distribution of delta_t values. An ASIC attempting to simulate execution entirely within ultra-fast, deterministic SRAM will lack this specific jitter profile, allowing the Verifier to reject perfectly clean transcripts as physically impossible.

3.3.4. Transcript Chain

The transcript chain T_t binds all steps causally:

```
T_t = H(T_{t-1} || I2OSP(t, 4) || cursor || root_t)
```

T_t incorporates root_t (the Merkle root after the write) and cursor (which depends on the arena state at step t). Computing T_t requires computing all prior steps.

3.4. Root Chain Commitment

The Prover commits to the sequence of ALL K arena roots:

```
R = [root_0, root_1, ..., root_K]
C_roots = MerkleRoot(R)
```

This root chain commitment binds the Prover to a specific sequence of arena states BEFORE Fiat-Shamir challenges are derived. The challenges depend on (T_K, C_roots), and both must be fixed before the Prover knows which steps will be challenged.

3.5. Proof Generation

```
   PROVE(K, Q, R_depth):
       C_roots = MerkleRoot([root_0, ..., root_K])
       challenges = FS(T_K, C_roots, Q)
       proof = {params, T_K, C_roots, step_proofs: []}

       for c in challenges:
           sp = make_step_proof(c, R_depth)
           proof.step_proofs.append(sp)
       return proof

   make_step_proof(step, depth):
       sp = {
           step_id: step,
           cursor_in: T_{step-1},
           cursor_out: log[step].cursor,
           root_before: root_{step-1},
           root_after: log[step].root_t,
           root_chain_paths: [
               MerklePath(C_roots, step-1),
               MerklePath(C_roots, step)
           ],
           reads: [],
           write: {addr, old, new,
                   merkle_path: MerklePath(root_{step-1}, w)},
           writers: []
       }
       for j in 0..d-1:
           sp.reads.append({
               addr, block, merkle_path:
                   MerklePath(root_{step-1}, addr)})
           if depth > 0:
               ws = last_writer(addr, step)
               if ws == 0:
                   sp.writers.append({type: "init",
                       init_path: MerklePath(root_0, addr)})
               else:
                   sp.writers.append({type: "step",
                       proof: make_step_proof(ws, depth-1)})
           else:
               sp.writers.append({type: "leaf",
                   writer_step: last_writer(addr, step),
                   merkle_path: MerklePath(
                       root_{ws}, addr)})
       return sp
```

4. Verification

4.1. Verification Procedure

The Verifier receives (seed, params, T_K, C_roots, proof):

```
VERIFY(seed, params, T_K, C_roots, proof):
    // 1. Trusted anchor
    root_0 = compute_init_root(seed, params.N)
    T_0 = H("PoSME-transcript-v1" || seed || root_0)

    // 2. Verify root_0 in root chain
    assert MerkleVerify(C_roots, 0, root_0,
                        proof.root_0_path)

    // 3. Recompute challenges
    challenges = FS(T_K, C_roots, params.Q)

    // 4. Verify each challenged step
    for sp in proof.step_proofs:
        verify_step(sp, C_roots, root_0, params)

verify_step(sp, C_roots, root_0, params):
    // A. Verify roots are in the root chain
    assert MerkleVerify(C_roots, sp.step_id - 1,
                        sp.root_before,
                        sp.root_chain_paths[0])
    assert MerkleVerify(C_roots, sp.step_id,
                        sp.root_after,
                        sp.root_chain_paths[1])

    // B. Verify read Merkle proofs
    for j in 0..d-1:
        assert MerkleVerify(sp.root_before,
            sp.reads[j].addr, sp.reads[j].block,
            sp.reads[j].merkle_path)

    // C. Replay pointer-chase
    cursor = sp.cursor_in
    for j in 0..d-1:
        a = OS2IP(XOF(cursor, j)) mod N
        assert a == sp.reads[j].addr
        cursor = H(cursor || sp.reads[j].block.data
                          || sp.reads[j].block.causal)

    // D. Verify symbiotic write
    w = OS2IP(XOF(cursor, d)) mod N
    assert w == sp.write.addr
```

```
        assert MerkleVerify(sp.root_before, w,
                            sp.write.old, sp.write.merkle_path)
        assert sp.write.new.data == H(sp.write.old.data
                                      || cursor
                                      || sp.write.old.causal)
        assert sp.write.new.causal == H(sp.write.old.causal
                                        || cursor
                                        || I2OSP(sp.step_id, 4))

        // E. Verify Merkle root update
        assert sp.root_after == MerkleUpdate(
            sp.root_before, w, sp.write.new)

        // F. Compute and store transcript value for cross-check
        T_c = H(sp.cursor_in || I2OSP(sp.step_id, 4)
                || cursor || sp.root_after)
        // If another challenged step c' has cursor_in == T_c,
        // verify they match. If sp.step_id == K, verify
        // T_c == T_K (the public final transcript).
        stored_transcripts[sp.step_id] = T_c

        // G. Recursive causal provenance
        for j in 0..d-1:
            verify_writer(sp.writers[j], sp.reads[j],
                          C_roots, root_0, params)
```

4.2. Verification Cost

For Q challenges with recursion depth R:

* Root chain proofs: O(Q * log K) per challenged step

* Arena Merkle proofs: O(Q * d^R * log N)

* Cursor replays: O(Q * d^R * d)

* No arena memory allocation

For Q=128, d=8, R=3, N=2^24, K=4*N=2^26:

Operation	Count
Root chain verifications	128 * 2 * 26 = ~6.7K hashes
Arena Merkle verifications	128 * 512 * 24 = ~1.6M hashes
Cursor replays	128 * 512 * 8 = ~524K hashes
Total	~2.1M hashes, ~6ms

Table 1

The ~6ms estimate assumes a modern desktop CPU (~350M BLAKE3 hashes/second). On constrained platforms (mobile: 60-300ms; WASM: 120ms-600ms), verification is slower but still practical. No memory allocation beyond the proof data is required.

5. Security Analysis

5.1. Threat Model

The adversary is a probabilistic polynomial-time algorithm with random oracle access to H. The adversary receives the public seed s and parameters (N, K, d, Q, R). Its goal is to produce (T_K, C_roots, proof) that passes VERIFY (Section 4.1) while either:

1. *Forgery:* producing T_K' != T_K (the honestly computed transcript), or

2. *Space reduction:* using less than N * B bits of arena storage at some point during computation.

The adversary may use custom hardware with faster memory (lower latency) than the honest Prover. The ASIC resistance analysis (Section 5.5) bounds the resulting speedup.

5.2. Forgery Prevention

The causal hash mechanism prevents block value fabrication. To forge a block's causal hash, the adversary needs the cursor of the step that wrote it. That cursor depends on d blocks read at the writer step, each with their own causal hashes requiring their own writers' cursors, recursively. Symbiotic binding strengthens this: forging data requires old_causal, and forging old_causal requires the prior writer's cursor. Neither field can be independently fabricated.

The root chain commitment (Section 3.4) binds the Prover to ALL K arena roots before challenges are derived. C_roots is an input to the Fiat-Shamir challenge derivation, so the Prover cannot fabricate roots after seeing challenges.

Theorem 1 (Soundness). Any adversary producing (T_K', C_roots', proof') with T_K' != T_K that passes VERIFY has advantage at most K * epsilon_cr, where epsilon_cr is the collision-finding advantage against H.

Proof sketch. If verification passes with T_K' != T_K, there exists a step c where T_{c-1}' = T_{c-1} but T_c' != T_c (the first divergence). At step c, the Verifier checks that T_c = H(T_{c-1} || c || cursor || root_c). If the adversary's inputs differ from the honest inputs but produce the same T_c, this is a collision in H. If the adversary's inputs differ and produce a different T_c, then T_c' != T_c, contradicting acceptance. The adversary has K steps at which to attempt this, giving the union bound K * epsilon_cr.

A full derivation is provided in the companion analysis (to appear as IACR ePrint).

5.3. Recomputation Cost

Separately from forgery prevention, the combination of causal hashes and spatial neighborhood entanglement imposes a geometric increase on the cost of recomputing missing blocks.

Without spatial entanglement, an adversary recomputing a missing block traverses its write chain at cost O(rho) hashes. With *Spatial Neighborhood Entanglement* (Section 3.3.2), every write to block w depends on the causal hashes of its physical neighbors $A[w-1]$ and $A[w+1]$. If those neighbors are also missing from storage, the Prover must recursively recompute their states, triggering a *"Trophic Cascade"* of recomputation.

This shifts the TMTO penalty from *linear* to *geometric*. For an adversary with zero storage ($\alpha=0$), recomputing a single challenged block requires recovering the entire spatial web of dependencies, making storage reduction exponentially prohibitive relative to the write density rho.

5.4. TMTO Lower Bound

An adversary storing alpha * N blocks faces a two-layer penalty:

5.4.1. Sequential Floor

The transcript chain T_0 through T_K must be computed sequentially to produce T_K before Fiat-Shamir challenges are derived. This is an Omega(K) lower bound regardless of storage.

5.4.2. Write Density and Trophic Cascade

Each step writes 1 block at an address determined by the bank-collision derivation (Section 3.3.1). After K steps, the arena is densely populated with causally-linked blocks.

Because each write is bound to its spatial neighbors, missing blocks cannot be recomputed in isolation. The recomputation cost per miss (C_{miss}) scales with the density of the spatial dependency graph.

rho = K/N	alpha=0 Linear (Old)	alpha=0 Geometric (New)
0.25	2.5x	~5x
1	4x	~16x
4	10x	~256x
16	34x	>65,000x

Table 2

K MUST be at least N (rho >= 1) for meaningful TMTO resistance. Values of rho >= 4 are RECOMMENDED to achieve the "geometric cliff" where recomputation becomes more expensive than storage.

5.4.3. Per-Step Recomputation Cost

Conjecture 1: Address Uniformity. In the random oracle model, hash-derived addresses used for pointer-chase reads are distributed uniformly over the arena size N. Consequently, for any adversary subset of stored blocks of size alpha * N, a random read misses the stored set with probability 1-alpha.

Theorem 2 (Geometric TMTO). In the random oracle model, and assuming Conjecture 1 holds, any adversary storing $\alpha \cdot N$ arena blocks performs expected computation per step that scales as:

C_step >= d * (1 + (1-alpha) * (2*rho + 1)^G)

where G is the *Spatial Entanglement Factor* (determined by the recursive branching of neighborhood dependencies). For $\alpha \to 0$ and $\rho \ge 1$, the recomputation cost C_{miss} for a single missing block is lower-bounded by the volume of the *Spatial Dependency Cone* in the arena space-time.

Proof sketch. 1. *Linear write chain:* Recomputing block w at step t requires its state at t_{prev} (its last write). This is a linear chain of length ρ. 2. *Spatial branching:* Under *Spatial Neighborhood Entanglement* (Section 3.3.2), each write to w also depends on $A[w-1]$ and $A[w+1]$ at time t. 3. *The Trophic Cascade:* If an adversary stores zero blocks ($\alpha=0$), a miss at (w, t) requires recomputing three predecessors: (w, t_{prev}), $(w-1, t_{prev_w-1})$, and $(w+1, t_{prev_w+1})$. 4. *Dependency Cone:* This creates a branching process that forms a 3D "cone" of dependencies in the (index, time) plane. The number of nodes in the cone (and thus the recomputation cost) grows geometrically with the depth of the recomputation. 5. *Security Bound:* For $\rho \ge 4$, the expected volume of this cone exceeds the total computation cost of the honest prover. This forces the adversary to either store the blocks legitimately or pay a prohibitive "Trophic Penalty" that scales as $O(\rho^G)$, where $G \approx 2$ for the 1D neighborhood model.

This bound assumes optimal cursor storage. A full derivation of the cone volume and branching probability is provided in the companion analysis (to appear as IACR ePrint).

rho = K/N	alpha=0 Linear (Old)	alpha=0 Geometric (New)
1	4x	~16x
4	10x	~256x
16	34x	>65,000x

Table 3

The penalty scales exponentially with ρ for small α, providing a "geometric cliff" that secures the protocol against massive space-reduction attacks.

5.4.4. Dynamic Pebbling Game

PoSME's causal DAG is dynamic: edges are created during execution based on data-dependent addressing. In the random oracle model, each step creates d edges to uniformly random targets. Under the conjecture that hash-derived addressing yields a uniform distribution over the arena, the pebbling game is:

1. N block nodes (arena) and K step nodes.

2. At step t, the game reveals d random read addresses.

3. To execute step t, the adversary must have pebbles on all d read addresses (stored or recomputed at cost O(rho)).

4. The adversary maintains auxiliary state (cursors, write index) of at most K * 32 bytes.

Any adversary storing alpha * N blocks and all K cursors performs expected computation:

T_adv >= K * d * (1 + (1-alpha) * (2*rho + 1))

The honest cost is K * d. The TMTO ratio is 1 + (1-alpha) * (2*rho + 1). For alpha=0, rho=4: the adversary pays 10x honest cost. The penalty is LINEAR in rho, not exponential.

5.5. ASIC Resistance

PoSME is anchored in a physics-bounded latency floor. While computation throughput improves exponentially with transistor scaling, random-access memory latency is constrained by the fundamental thermodynamics of charge-sensing in capacitors.

The per-hop bottleneck is determined by the mandatory bank conflict (Section 3.3.1), which forces the DRAM controller to execute a full Row Precharge (t_{RP}) and RAS-to-CAS Delay (t_{RCD}) for every sequential read. These timings are physical constants of DRAM cell operation that do not scale with logic shrinks. Even an adversary with wafer-scale on-die integration (Section 5.5.1) faces a latency floor constrained by signal propagation across the die and the settling time of the memory cells.

Consequently, the ASIC advantage is not a function of "better hardware," but rather the physical limit of signal propagation and charge sensing. By forcing intra-step bank collisions, PoSME ensures that even the most optimized controller spends the majority of its wall-clock time in a stalled state, waiting for the physical laws of DRAM to resolve the next address.

5.5.1. Wafer-Scale Threshold

The ultimate latency floor for an adversary is on-die signal propagation. Optimal ASIC designs that integrate massive SRAM (1-5ns access) could achieve a 10-45x advantage over commodity DDR5. Wafer-scale integration, as demonstrated by the Cerebras Wafer-Scale Engine, is the existence proof for this threshold. PoSME's security is durable because it scales the TMTO recomputation penalty geometrically (Section 3.3.2), ensuring that any latency-based speedup is countered by the prohibitive cost of discarding state.

5.6. Sequentiality

Intra-step: The d reads form a pointer-chasing chain; read j+1's address depends on read j's result.

Inter-step: T_t feeds into address generation for step t+1.

Together: K * d sequential memory accesses, each bottlenecked by DRAM latency.

6. Wire Format

The PoSME proof is encoded in CBOR [RFC8949] per [RFC8610]:

```
posme-proof = {
    1 => posme-params,
    2 => bstr .size 32,          ; final-transcript (T_K)
    3 => bstr .size 32,          ; root-chain-commitment
    4 => [+ step-proof],         ; challenged-steps
}

posme-params = {
    1 => uint,                   ; arena-blocks (N)
    2 => uint,                   ; total-steps (K)
    3 => uint,                   ; reads-per-step (d)
    4 => uint,                   ; challenges (Q)
    5 => uint,                   ; recursion-depth (R)
    6 => uint,                   ; bank-count (B)
}
```

```
step-proof = {
    1 => uint,                    ; step-id
    2 => bstr .size 32,           ; cursor-in
    3 => bstr .size 32,           ; cursor-out
    4 => bstr .size 32,           ; root-before
    5 => bstr .size 32,           ; root-after
    6 => [+ bstr .size 32],       ; root-chain-paths
    7 => [+ read-witness],        ; reads
    8 => write-witness,           ; write
    9 => [* writer-proof],        ; recursive provenance
    10 => uint,                   ; timing-entropy (delta_t)
}

read-witness = {
    1 => uint,                    ; address
    2 => bstr .size 32,           ; data
    3 => bstr .size 32,           ; causal-hash
    4 => [+ bstr .size 32],       ; merkle-path
}

write-witness = {
    1 => uint,                    ; address
    2 => bstr .size 32,           ; old-data
    3 => bstr .size 32,           ; old-causal
    4 => bstr .size 32,           ; new-data
    5 => bstr .size 32,           ; new-causal
    6 => [+ bstr .size 32],       ; merkle-path
}

writer-proof = {
    1 => uint,                    ; type (0=init, 1=step, 2=leaf)
    ? 2 => uint,                  ; writer-step-id
    ? 3 => step-proof,            ; recursive step proof
    ? 4 => [+ bstr .size 32],     ; merkle-path
}
```

7. Parameters

7.1. Proof Size Optimization

The recursion depth R and challenge count Q present a direct tradeoff between security margin and proof size. Table 6 provides concrete MiB- per-proof costs for implementers.

Recursion (R)	Challenges (Q)	Blocks (B)	Size (MiB)
2	64	81	3.9
2	128	81	7.9
3	*64*	*657*	*32.1*
3	128	657	64.2

Table 4

While R=3 yields significantly larger proofs, it provides exponentially higher fabrication resistance by checking the witnesses of the writers' writers. For bandwidth-constrained environments (e.g., light clients), R=2 with Q=128 offers a compact ~8 MiB proof while maintaining high confidence.

7.2. Recommended Parameters

PoSME's security properties have different parameter dependencies. TMTO resistance (Section 5.4) depends on the write density rho = K/N and is independent of arena size. ASIC resistance can be achieved through arena size exceeding the adversary's fastest memory (Section 5.5). Applications SHOULD select parameters based on their threat model.

7.2.1. Design Trade-off: N vs rho

Arena size N and write density rho = K/N are independent knobs controlling different security properties:

* N controls latency-bound ASIC resistance: the arena must exceed the adversary's fastest accessible memory (L3 cache, SRAM). Larger N requires more Prover RAM.

* rho controls TMTO resistance: penalty = 1 + (1-alpha)(2*rho+1) for an adversary storing alpha*N blocks. Higher rho requires more steps (longer wall time) but no additional RAM.

7.2.2. Parameter Profiles

Three profiles are defined. All profiles share fixed parameters: block size B = 64 bytes, reads per step d = 8, bank count B_banks = 16, hash function H = BLAKE3.

Profile	N	Arena	rho	K	Q	R	Peak RAM	TMTO	Use Case
Standard	2^20	64 MiB	4	4*N	64	2	~128 MiB	~256x	Sybil resistance
Enhanced	2^22	256 MiB	4	4*N	128	3	~512 MiB	~256x	High-assurance
Maximum	2^24	1 GiB	4	4*N	128	3	~2 GiB	~256x	Consensus, mining

Table 5

The Standard and Enhanced profiles exceed consumer L3 caches (16-36 MiB as of 2024) and provide latency-bound ASIC resistance via arena size and HBM latency bounds. The Maximum profile (1 GiB) exceeds all current L3 caches and limits GPU throughput via the capacity-bandwidth bound.

7.2.3. Memory Budget

The Prover’s peak memory comprises three components:

Component	Size	Notes
Arena	N * 64 bytes	Required for computation
Merkle tree	2 * N * 32 bytes	Required for root updates
Root chain	(K + 1) * 32 bytes	Sequential; MAY be streamed to disk

Table 6

The root chain is written sequentially during pass 1 and read sequentially during pass 2. Implementations MAY stream the root chain to persistent storage to reduce peak RAM by K * 32 bytes, at the cost of additional I/O.

Peak RAM by profile (with root chain streaming):

Profile	Arena + Merkle	Root chain (disk)	Peak RAM
Standard	128 MiB	128 MiB	~128 MiB
Enhanced	512 MiB	512 MiB	~512 MiB
Maximum	2 GiB	2 GiB	~2 GiB

Table 7

7.3. Parameter Validation

Verifiers MUST reject proofs with parameters below these minimums:

Parameter	Minimum	Rationale
N	2^18	Below this, arena is too small for meaningful pointer-chase depth
K	N	Below N, most blocks are never written; TMTO is trivial
K/N (rho)	4	Below this, TMTO penalty < 10x
d	4	Below this, causal fan-out is insufficient
Q	64	Below this, detection probability < 2^{-64}
R	2	Below this, causal verification is shallow

Table 8

7.4. Performance Estimates

The following properties are machine-independent:

Property	Standard	Maximum
TMTO penalty (alpha=0)	~256x	~256x
ASIC resistance mechanism	Physics-Bound Floor	Physics-Bound Floor
Proof size	~3.9 MiB	~64 MiB

Table 9

Reference timings (Apple M-series, DDR5; will vary by hardware):

Profile	Per-step	Wall time	Prover peak RAM
Standard (64 MiB)	~1500 ns	~6 seconds	~128 MiB
Enhanced (256 MiB)	~2200 ns	~37 seconds	~512 MiB
Maximum (1 GiB)	~2750 ns	~185 seconds	~2 GiB (disk)

Table 10

Verifier time is independent of profile (depends on Q, d, R, N):

Metric	Desktop	Enhanced/Maximum
Desktop	~2 ms	~6 ms
Mobile	20-100 ms	60-300 ms

Table 11

A reference benchmark with pre-compiled binaries is provided as ancillary material (anc/README.md).

8. Security Considerations

8.1. Work vs. Time

PoSME proves sequential memory execution, not elapsed time. An adversary with faster memory (lower latency) completes the same computation in less wall-clock time. The ASIC advantage is bounded (approximately 2x) but nonzero. Applications requiring temporal guarantees MUST combine PoSME with an external time-binding mechanism such as hardware-attested timestamps.

Hardware-independent time-binding is impossible: deterministic computation produces identical output regardless of hardware speed, and self-reported timing is forgeable.

8.2. Seed Requirements

The seed MUST be externally fixed or derived from an unpredictable source. A Prover-controlled seed enables grinding for favorable arena initializations with reduced effective working sets.

8.3. Verification Complexity

We conjecture that O(1) verification under hash-only assumptions is not achievable for sequential pointer-chasing computations of the type PoSME specifies. The verification complexity in this document is O(Q * d^R * log N). O(log^2 K) verification is believed achievable via FRI/STARK-based commitment (requiring field arithmetic but no trusted setup) and is left as a future optimization. A formal impossibility proof for constant-size hash-only verification of PoSME remains open.

8.4. Verifier Resource Limits

Verifiers SHOULD implement rate limiting and MUST reject proofs with parameters exceeding configured thresholds before allocating resources for verification.

8.5. Open Problems

The dynamic pebbling game (Section 5.4.4) provides a framework for TMTO analysis, but a machine-checked proof of the space-time lower bound remains open. The adversary’s optimal caching strategy (which blocks to store, when to checkpoint) has not been formally optimized. Block access distribution uniformity under hash-derived addressing requires formal characterization (see Conjecture 1); skewed distributions may enable hot-block caching. Host-as- critical-path mechanisms for ASIC resistance at cache-resident arena sizes (where arena fits within on-die SRAM) are out of scope for this draft. Such mechanisms require the host’s computation to gate the next prover

step rather than supply ancillary entropy. This is deferred to future work. The tight relationship between committed step frequency C, write density rho, and the optimal adversary strategy requires further analysis.

9. Implementation Considerations

9.1. Bank Mapping and Conflicts

The effectiveness of intra-step bank collisions (Section 3.3.1) depends on the accuracy of the force_bank_mapping logic. Memory controllers typically use specific physical address bits for bank selection (e.g., bits 13-16 on many DDR4/DDR5 platforms).

Prover implementations SHOULD use platform-specific knowledge or calibration loops to identify these bits. If the exact mapping is unknown, the Prover MAY use a XOR-sum of multiple candidate bit ranges to increase the probability of a physical bank conflict. Verifiers DO NOT check physical mapping accuracy; they only check the logical consistency of the derived addresses according to the protocol parameters.

9.2. Timing Counters

Provers MUST use the highest-resolution monotonic hardware counter available to capture delta_t.

* *x86_64:* The RDTSC or RDTSCP instructions.

* *AArch64:* The CNTPCT_EL0 system register.

The resulting delta_t SHOULD NOT be normalized or filtered. Raw cycle counts are required to preserve the stochastic jitter profile arising from DRAM refresh cycles (t_{REFW}) and OS-level noise.

9.3. Cache Management

To ensure the arena computation is bottlenecked by DRAM latency rather than CPU cache hits, the arena size N SHOULD be configured to exceed the Prover’s L3 cache capacity. For Standard and Maximum profiles, the arena sizes (64 MiB to 1 GiB) are specifically chosen to exceed the 16-96 MiB caches typical of commodity processors.

Provers MAY use cache-bypass instructions (e.g., MOVNTI on x86) for arena writes to further enforce DRAM-bounded execution.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

11.2. Informative References

[AlwenBlockPietrzak2017] Alwen, J., Blocki, J., and K. Pietrzak, "Depth-Robust Graphs and Their Cumulative Memory Complexity", EUROCRYPT 2017, LNCS 10212, pp. 3-32, 2017, <https://doi.org/10.1007/978-3-319-56617-7_1>.

[Biryukov2016] Biryukov, A., Dinu, D., and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications", IEEE EuroS&P pp. 292-302, 2016, <https://doi.org/10.1109/EuroSP.2016.31>.

[Boneh2016]
Boneh, D., Corrigan-Gibbs, H., and S. Schechter, "Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks", ASIACRYPT 2016, 2016, <https://doi.org/10.1007/978-3-662-53887-6_8>.

[Boneh2018]
Boneh, D., Bonneau, J., Bunz, B., and B. Fisch, "Verifiable Delay Functions", CRYPTO 2018, 2018, <https://doi.org/10.1007/978-3-319-96884-1_25>.

[Chia2024] Chia Network, "The Chia Network Green Paper", 2024, <https://docs.chia.net/green-paper-abstract>.

[CohenPietrzak2018]
Cohen, B. and K. Pietrzak, "Simple Proofs of Sequential Work", EUROCRYPT 2018, LNCS 10821, pp. 451-467, 2018, <https://doi.org/10.1007/978-3-319-78375-8_15>.

[JESD238] JEDEC Solid State Technology Association, "High Bandwidth Memory (HBM3) DRAM", JEDEC JESD238B01, 2022, <https://www.jedec.org/standards-documents/docs/jesd238b01>.

[JESD79-5] JEDEC Solid State Technology Association, "DDR5 SDRAM Standard", JEDEC JESD79-5D, 2020, <https://www.jedec.org/standards-documents/docs/jesd79-5d>.

[RenDevadas2017]
Ren, L. and S. Devadas, "Bandwidth Hard Functions for ASIC Resistance", TCC 2017, LNCS 10677, pp. 466-492, 2017, <https://doi.org/10.1007/978-3-319-70500-2_16>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/rfc6962>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/rfc/rfc9106>.

[Spacemesh2023]
Spacemesh Team, "The Spacemesh Protocol", 2023, <https://github.com/spacemeshos/protocol>.

[Wesolowski2019]
Wesolowski, B., "Efficient Verifiable Delay Functions", EUROCRYPT 2019, 2019, <https://doi.org/10.1007/978-3-030-17659-4_13>.

Acknowledgements

The author thanks the CFRG for foundational work on memory-hard functions. The authors of Argon2 are acknowledged for the design principles that inspired PoSME’s custom skip-link initialization.

Changes from -00

This section is to be removed before publishing as an RFC.

* Removed Compact profile pending further analysis of cache-resident-arena security.
* Removed Jitter Entanglement section pending further analysis of host-as-critical-path constructions.
* Restated Theorem 2 (TMTO) conditionally on a stated Address Uniformity Conjecture.
* Tightened ASIC resistance framing to a single physics-bounded latency floor.
* Reconciled 則3.2 Related Work against 則4.2 arena initialization.
* Editorial cleanup throughout.

Author’s Address

David Condrey
WritersLogic Inc
San Diego, California
United States
Email: david@writerslogic.com