Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 16 November 2026

D. Condrey
WritersLogic

15 May 2026

Proof of Sequential Memory Execution (PoSME)
draft-condrey-posme-00

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 as $1/\alpha$ with spatial entanglement, where α is the adversary's storage fraction. 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 16 November 2026.

Copyright Notice

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 [RFC7914] resist ASIC acceleration by requiring significant memory resources. On commodity architectures, scrypt is primarily bottlenecked by the sequential computation latency of its internal Salsa20/8 core, while scaling memory size to deny low-memory parallelization. MHFs in general impose high

cumulative memory-time cost, but their ASIC resistance ultimately depends on the specific hardware cost model (area, bandwidth, or latency) that dominates the adversary's design.

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) TMTO resistance that scales as ρ/α under spatial entanglement. 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. Applicability Statement

PoSME is NOT a general-purpose password-hashing function (PHF) or key derivation function (KDF). It does not compete with, and MUST NOT be used as a substitute for, Argon2id [RFC9106] or scrypt [RFC7914] in password storage, credential stretching, or key derivation contexts.

PoSME is a specialized hardware-discriminator and latency-hard asymmetry boundary designed for content provenance frameworks. Its intended use is as the cryptographic foundation for a C2PA [C2PA] "process-proof assertion": a proof that a specific sequence of K sequential memory-execution steps was performed on a single computing device over a corresponding interval of wall-clock time. PoSME proves sequential computation, not the nature of the content produced during that computation; binding PoSME proofs to specific content requires an application-layer protocol (e.g., C2PA manifest assertions) that is outside the scope of this document. The construction exploits the physics-bounded latency floor of random DRAM access to create an asymmetry between honest execution on commodity hardware and attempted forgery on specialized hardware.

The following table highlights the divergent design optimization vectors:

Property	Argon2id	scrypt	PoSME
Primary goal	Password hashing / KDF	Password hashing / KDF	Sequential process proof
Hardness type	Memory-hard (time * space)	Memory-hard (time * space)	Latency-hard (sequential DRAM access)
ASIC resistance vector	Side-channel resistance; GPU-area hardness	Memory-size scaling to deny low-memory parallelization	Sequential DRAM-latency floor via pointer-chasing
Access pattern	Data-independent (2id hybrid)	Data-dependent ROMix with Salsa20/8	Data-dependent pointer-chase
Arena mutability	Static (read-only after fill)	Static (read-only after fill)	Mutable (in-place writes)
Side-channel profile	2id mode resists side-channels	Data-dependent arena reads; Salsa20/8 core is data-independent	Data-dependent; leaks full access pattern (see Section 8)
Verification model	Recompute from password	Recompute from password	Succinct proof (no recomputation)
Target deployment	Authentication servers	Authentication servers	Content provenance (C2PA), sequential process attestation

Table 1

Implementations MUST NOT deploy PoSME outside of its intended provenance and proof-of-process scope without a dedicated security analysis addressing the side-channel and threat model considerations documented in Section 8.

1.2. Related Work

1.2.1. 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.

1.2.2. Memory-Hard Functions

Functions like Argon2id and scrypt resist ASIC acceleration by imposing high memory requirements. Argon2id [RFC9106] [Biryukov2016] 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 $1 + 2\rho(1-\alpha)^2/\alpha$ for an adversary storing αN blocks, where $\rho = K/N$ is the write density.

1.2.3. 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.

1.2.4. 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 5.5.

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. Implementations MUST use BLAKE3 for sequential chains to ensure post-quantum resistance; SHA-3 MAY be used as an alternative where BLAKE3 is unavailable.

XOF(input, index): H evaluated at (input || I2OSP(index, 4)), truncated to 8 bytes and interpreted as a big-endian unsigned integer.

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 TMTO resistance via spatial neighborhood entanglement.

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

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

       // 1. Determine Target Bank
       bank_id = 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 = 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 = 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. Addressing Function and Bad Locality Requirement

The addressing function that maps the output of the step hash to the next memory chunk address MUST produce addresses that are approximately uniform over the allocated memory arena. Formally, for a hash output h of length L bits and arena size N, the derived index:

a = OS2IP(h[0..7]) mod N

where OS2IP is the Octet String to Integer Primitive per [RFC8017], interpreting the first 8 octets of h as an unsigned big-endian integer, MUST produce addresses that are within statistical distance $N/2^{64}$ of uniform over $[0, N)$ (established by Theorem 2, Section 5.5.3).

This access pattern MUST ensure a uniform distribution of jumps across the entire memory layout, rendering modern hardware caching layers (L1, L2, L3 caches) and hardware pre-fetch engines entirely ineffective. Every single step index transition MUST result in an effective cache miss with overwhelming probability, forcing a cold random access to physical memory cells to preserve the sequential execution time delay. For the recommended arena sizes ($N \ge 2^{20}$, arena $\ge$ 64 MiB), the arena exceeds the largest commodity L3 caches, and the pseudo-random addressing ensures that successive accesses have negligible spatial locality.

Implementations MUST NOT reorder, batch, or prefetch arena accesses. Each pointer-chase hop MUST complete (the read value MUST be available to the CPU) before the next address can be computed.

3.3.1.1. Type Conversion Primitives

Whenever octet string arrays output from the pseudo-random step transformations (BLAKE3 XOF) are converted into integer array indices for chunk addressing, the conversion MUST use the OS2IP (Octet String to Integer Primitive) per [RFC8017], interpreting octets as a big-endian unsigned integer. This encoding is REQUIRED for interoperability: a Verifier MUST be able to reproduce the Prover’s address derivations from the proof without out-of-band negotiation of byte order.

The XOF function defined in Section 2 truncates the hash output to 8 bytes and interprets the result via OS2IP as a big-endian unsigned integer. All pseudocode in this document uses this convention. Implementations MUST NOT use alternative byte orderings (e.g., little-endian) for index derivation.

3.3.2. 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.

3.3.3. 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 self-contained write chain into a spatial cascade. If an adversary discards 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 propagates outward: $w-1 \to w-2 \to \cdots$ and $w+1 \to w+2 \to \cdots$, until reaching stored blocks on each side. For an adversary storing αN blocks, the expected cascade width is $2(1-\alpha)/\alpha$, and each block costs ρ hash evaluations to replay (see Theorem 3, Section 5.5.3). This cascade is sequential (Theorem 4, Section 5.5.4), adding directly to the adversary’s critical path.

3.3.4. Timing Entropy Attestation

Because commodity DRAM requires periodic electrical refresh cycles (t_{REFW}), a genuine physical execution will exhibit unavoidable, stochastic latency spikes. These timing variations arise from multiple independent physical sources: DRAM refresh interference, OS scheduler preemption, thermal throttling, and crossed clock- domain jitter between the CPU core clock, memory controller clock, and DRAM internal timing.

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.

3.3.4.1. CPU Jitter Entropy Calibration

The min-entropy available from CPU timing jitter depends on the Over-Sampling Ratio (OSR) configured in the collection loop. Implementations MUST NOT assume a flat 1-bit min-entropy per time delta sample under default configurations.

Under a standard execution configuration with an Over-Sampling Ratio (OSR) of 3, the estimated min-entropy rate is approximately 0.33 bits per raw time delta sample; equivalently, approximately 3 raw samples are required to extract 1 bit of conditioned output entropy. This relationship follows from the Jitter RNG analysis methodology: the OSR determines the ratio of raw samples to conditioned output bits.

If a target deployment profile strictly mandates a 1-bit min-entropy baseline per sample, the implementation initialization routines MUST explicitly override default execution bounds to enforce a significantly higher OSR value (typically OSR >= 10-20, depending on the platform’s jitter characteristics). The required OSR SHOULD be determined by platform-specific entropy assessment using established methodologies (e.g., NIST SP 800-90B health tests).

3.3.4.2. Embedded Platform Entropy Warning

WARNING: CPU jitter entropy MAY degrade significantly or collapse entirely on low-power, deeply embedded, single-clock- domain microcontroller architectures. Basic microcontrollers and simple RISC-V cores that lack complex, asynchronous, crossed clock-domain jitter patterns found in modern commodity consumer or enterprise CPUs (multi-GHz, out-of-order, superscalar designs with independent memory controller clocks) may produce timing deltas with near-zero min-entropy.

Implementations targeting such platforms MUST perform a platform-specific entropy assessment before relying on delta_t values for any security-relevant purpose. Implementations SHOULD fall back to a hardware random number generator (HWRNG) or external entropy source if the platform cannot provide adequate CPU jitter entropy. If no adequate entropy source is available, the delta_t values MUST be treated as non-cryptographic metadata (useful for Verifier plausibility checks but not as a source of secret randomness).

3.3.5. 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: w, old: log[step].old, new: A[w],
                   merkle_path: MerklePath(root_{step-1}, w),
                   neighbor_prev: {
                       addr: (w-1) mod N,
                       block: A[(w-1) mod N],
                       merkle_path:
                           MerklePath(root_{step-1}, (w-1) mod N)},
                   neighbor_next: {
                       addr: (w+1) mod N,
                       block: A[(w+1) mod N],
                       merkle_path:
                           MerklePath(root_{step-1}, (w+1) mod N)}},
           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 = 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 write with spatial neighbors
        w = XOF(cursor, d) mod N
        assert w == sp.write.addr
        assert MerkleVerify(sp.root_before, w,
                            sp.write.old, sp.write.merkle_path)

        // D1. Verify spatial neighbor Merkle proofs
        w_prev = (w - 1) mod N
        w_next = (w + 1) mod N
        np = sp.write.neighbor_prev
        nn = sp.write.neighbor_next
        assert np.addr == w_prev
        assert nn.addr == w_next
        assert MerkleVerify(sp.root_before, w_prev,
                            np.block, np.merkle_path)
        assert MerkleVerify(sp.root_before, w_next,
                            nn.block, nn.merkle_path)
        n_prev = np.block.causal
        n_next = nn.block.causal

        // D2. Verify symbiotic write with spatial entanglement
        assert sp.write.new.data == H(sp.write.old.data
                                      || cursor
                                      || sp.write.old.causal
                                      || n_prev || n_next)
        assert sp.write.new.causal == H(sp.write.old.causal
                                        || cursor
                                        || I2OSP(sp.step_id, 4)
                                        || n_prev || n_next)

        // 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=2^25, K=4*N=2^27:

Operation	Count
Root chain verifications	128 * 2 * 27 = ~6.9K hashes
Arena Merkle verifications	128 * 512 * 25 = ~1.6M hashes
Cursor replays	128 * 512 * 8 = ~524K hashes
Total	~2.2M hashes, ~6ms

Table 2

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. Assumption Registry

All security claims in this document are conditional on the assumptions listed below. Each assumption is given a label (A1-A4) referenced by the theorems that depend on it.

A1 (Random Oracle Model). The hash function H (BLAKE3 or SHA-3) is modeled as a random oracle: on any fresh input, H returns an independent, uniformly random 256-bit output. This assumption underlies address uniformity (Theorem 2) and the independence of pointer-chase targets.

A2 (Collision Resistance). Finding $x \ne x'$ with $H(x) = H(x')$ requires $\Omega(2^{128})$ queries (birthday bound for a 256-bit hash). Theorem 1 (Soundness) reduces forgery to collision-finding with a factor-K loss.

A3 (Preimage Resistance). Given y, finding x with $H(x) = y$ requires $\Omega(2^{256})$ queries. This assumption is used implicitly in the causal hash mechanism: reversing a causal hash chain to recover a prior cursor requires a preimage query.

A4 (Uniform and Independent Addressing). Under A1, the derived addresses $a_{t,j} = \text{XOF}(\text{cursor}_t, j) \bmod N$ are pairwise independent and within statistical distance $N/2^{64}$ of uniform over $[0, N)$. This is formally established as Theorem 2 (Section 5.5.3). The TMTO bounds in Theorems 3-5 are conditional on A4 holding; if H deviates from the random oracle model in a way that induces address clustering, the TMTO bounds may not hold as stated.

The following table summarizes the dependency:

Theorem	Depends on	Claim
1 (Soundness)	A2	Forgery advantage $\le K \cdot \epsilon_{\text{cr}}$
2 (Address Uniformity)	A1	Addresses pairwise independent, near-uniform
3 (Spatial Cascade TMTO)	A1, A3, A4	Per-step cost $\ge d(1 + 2\rho(1-\alpha)^2/\alpha)$
4 (Sequential Cascade)	A3, A4	Cascade is sequential ($L \cdot \rho$ critical path)
5 (Checkpoint Dominance)	A4	Partial checkpoints strictly suboptimal

Table 3

Scope limitation. No composability (UC-security) claim is made. PoSME is analyzed as a standalone primitive in the random oracle model. Applications that compose PoSME with other protocols (e.g., combining with timestamps per Section 9.1) MUST perform a separate composition analysis; the security guarantees stated here do not automatically carry through sequential or parallel composition.

5.2. 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 5.6) bounds the resulting speedup.

5.3. 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). Let $\mathcal{A}$ be any adversary producing $(T_K', C_{\text{roots}}', \text{proof}')$ with $T_K' \ne T_K$ that passes VERIFY. There exists a reduction $\mathcal{B}$ to collision-finding in H such that:

Adv^{forge}(A) <= K * Adv^{coll}(B)

where K is the number of steps. The factor K arises from a union bound over the K steps at which the transcript chain may diverge, and represents a reduction loss of factor K. For the recommended parameter range $K \le 2^{26}$, this loss is acceptable against a 256-bit hash ($K \cdot \epsilon_{\text{cr}} \le 2^{26} \cdot 2^{-128} = 2^{-102}$ for birthday-bound collision resistance).

Proof sketch. If verification passes with $T_K' \ne T_K$, there exists a first divergence step c where $T_{c-1}' = T_{c-1}$ but $T_c' \ne T_c$. The reduction $\mathcal{B}$ guesses this step c uniformly at random from ${1, \ldots, K}$ (success probability $1/K$, the source of the reduction loss). At step c, the Verifier checks that $T_c = H(T_{c-1} | c | \text{cursor} | \text{root}_c)$. If the adversary's inputs differ from the honest inputs but produce the same T_c, $\mathcal{B}$ outputs a collision in H. If the adversary's inputs differ and produce a different T_c, then $T_c' \ne T_c$, contradicting acceptance. The union bound over K guesses yields the stated loss.

Remark. The factor-K loss is loose: it reflects the reduction strategy (random guess of the divergence point), not necessarily a real attack. Tightening this to $O(\log K)$ via a binary-search reduction or to $O(1)$ via a rewinding argument is an open problem. See also Section 9.5, item 1.

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

5.4. Recomputation Cost

Separately from forgery prevention, spatial neighborhood entanglement (Section 3.3.3) imposes a storage-dependent penalty on recomputation. Without spatial entanglement, an adversary recomputing a missing block traverses its temporal write chain at cost $O(\rho)$ hashes; the chain is self-contained because each write depends only on the block's own previous state and the cursor (which is stored). With spatial entanglement, each write also depends on the causal hashes of neighbors $A[w-1]$ and $A[w+1]$ at the time of the write. These are historical states that cannot be derived from the neighbors' current values (hash chains are irreversible). Therefore, recomputing a missing block requires replaying the full temporal chains of its spatial neighbors, which in turn require their own neighbors, creating a spatial cascade.

The cascade extends outward from the missing block until reaching a stored block on each side. For an adversary storing αN blocks, the expected cascade width is $2(1-\alpha)/\alpha$ (geometric distribution), and each block in the cascade costs ρ hash

evaluations to replay. The per-miss recomputation cost is therefore $\Theta(\rho/\alpha)$, compared to $\Theta(\rho)$ without spatial entanglement. The $1/\alpha$ factor means that reducing storage becomes increasingly expensive: halving storage more than doubles recomputation cost.

5.5. TMTO Lower Bound

An adversary storing $\alpha \cdot N$ blocks faces a two-layer penalty:

5.5.1. 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.

5.5.2. Spatial Cascade TMTO

Each step writes 1 block at a uniformly random address (Theorem 2). After K steps with write density $\rho = K/N$, each block has been written ρ times on average. Because each write is bound to its spatial neighbors' causal hashes, missing blocks cannot be recomputed in isolation.

Theorem 3 (Section 5.5.3) establishes the TMTO ratio:

TMTO(alpha) = 1 + 2*rho*(1-alpha)^2 / alpha

rho	alpha=0.5	alpha=0.25	alpha=0.1	alpha=0.01
1	2x	5x	17x	197x
4	5x	19x	65x	785x
16	17x	73x	257x	3,137x

Table 4

K MUST be at least N ($\rho \ge 1$) for meaningful TMTO resistance. Values of $\rho \ge 4$ are RECOMMENDED.

5.5.3. Per-Step Recomputation Cost

Theorem 2 (Address Uniformity). In the random oracle model, the addresses $a_{t,j} = \text{XOF}(\text{cursor}_t, j) \bmod N$ used for pointer-chase reads satisfy:

1. *Independence.* For distinct pairs $(t_1, j_1) \ne (t_2, j_2)$, the addresses a_{t_1,j_1} and a_{t_2,j_2} are pairwise independent, except with probability at most $\binom{Kd}{2} / 2^{257}$ (birthday bound on cursor collisions in the 256-bit hash space).

2. *Uniformity.* Each address $a_{t,j}$ has statistical distance at most $N / 2^{64}$ from the uniform distribution over $[0, N)$. For $N \le 2^{48}$, this bound is at most 2^{-16}.

Consequently, for any adversary subset of stored blocks of size $\alpha \cdot N$, a random read misses the stored set with probability $(1 - \alpha) \pm N/2^{64}$.

Proof. In the random oracle model, H maps distinct inputs to independent, uniformly random 256-bit outputs. Define $X_{t,j} = H(\text{cursor}_t | \text{I2OSP}(j, 4))$. The inputs are distinct across j (for fixed t) by the index suffix, and across t (for any j) whenever $\text{cursor}_t \ne \text{cursor}_{t'}$. Since $\text{cursor}_t = H(\text{cursor}_{t-1} | \cdots)$ is itself a hash output, cursor collision probability across all K steps is at most $\binom{K}{2} / 2^{256}$, which is negligible. Given distinct inputs, the outputs $X_{t,j}$ are independent uniform 256-bit strings, establishing claim (1).

For claim (2): XOF truncates $X_{t,j}$ to its first 8 bytes, yielding a uniform value U in $[0, 2^{64})$. The address is $U \bmod N$. The number of integers in $[0, 2^{64})$ mapping to any particular residue $r \in [0, N)$ is either $\lfloor 2^{64}/N \rfloor$ or $\lceil 2^{64}/N \rceil$. The maximum deviation from the ideal probability $1/N$ is at most $1/2^{64}$, giving statistical distance at most $N/2^{64}$ over the full distribution. For $N \le 2^{48}$, this is at most 2^{-16}, which is negligible for all recommended profiles.

Theorem 3 (Spatial Cascade TMTO). In the random oracle model, under Theorem 2, consider an adversary that maintains $\alpha \cdot N$ arena blocks in working memory ($0 < \alpha < 1$) and stores all K transcript values. The adversary's expected computation per step is:

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

where $\rho = K/N$ is the write density.

Proof. The proof proceeds in three parts.

Part 1: Write chain cost. When a read at step t targets a block w not in the adversary's working memory, the adversary must reconstruct w's current state. Block w was written ρ times on average (each of K steps writes to a uniformly random block by Theorem 2). The adversary knows the cursor at every step (stored), so replaying one write requires one hash evaluation given the block's previous state and the cursor. Tracing w's temporal write chain from initialization to the current epoch costs ρ hash evaluations.

Part 2: Spatial cascade. Each write to block w at step t' depends on the causal hashes of $A[w-1]$ and $A[w+1]$ at time t' (Section 3.3.3). These are historical states: the causal hash of w's neighbor at the moment w was written, not the neighbor's current state. Even if $w-1$ is currently in the adversary's working memory, its state at time t' is not recoverable from its current state (hash chains are irreversible). Therefore, to replay w's write chain, the adversary must also replay the full temporal chain of $w-1$ (so that $w-1$'s state at each of w's write times is available).

Block $w-1$'s temporal chain depends on $w-2$'s historical causal hashes (its own spatial neighbor). If $w-2$ is not stored, this cascades further: $w-2 \to w-3 \to \cdots$. The cascade extends outward in one direction until reaching a block that is in the adversary's working memory; that block's full temporal chain is available (the adversary maintains it by construction).

The same cascade extends in the opposite direction: $w+1 \to w+2 \to \cdots$. The cascade width in each direction follows a geometric distribution with success probability α (each successive neighbor is stored with probability α, independently by Theorem 2's uniformity guarantee on write addresses). The expected cascade width in one direction is $(1-\alpha)/\alpha$. For all recommended profiles ($N \ge 2^{20}$), this is negligible relative to N for any $\alpha > 2^{-19}$; ring wrap-around does not affect the bound in practice. Each block in the cascade requires ρ hash evaluations to replay its temporal chain.

Part 3: Expected per-step cost. At each of the K steps, the Prover makes d reads. Each read targets a uniformly random block, which is absent from the adversary's working memory with probability $(1-\alpha)$. Each miss triggers a spatial cascade of expected total width $2(1-\alpha)/\alpha$ blocks, each costing ρ hash evaluations.

The expected computation per step is:

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

For $(1-\alpha) \approx 1$ (small α), this simplifies to $d \cdot (1 + 2\rho/\alpha)$. The TMTO ratio (adversary cost divided by honest cost d) is:

TMTO(alpha) = 1 + 2*rho*(1-alpha)^2 / alpha

rho	alpha=0.5	alpha=0.25	alpha=0.1	alpha=0.01
1	2x	5x	17x	197x
4	5x	19x	65x	785x
16	17x	73x	257x	3,137x

Table 5

For $\alpha < 1/(2\rho)$, the TMTO ratio exceeds $2\rho^2$, making space reduction more expensive than honest execution with full storage.

5.5.4. Sequential Cascade Latency

Theorem 4 (Sequential Cascade Latency). The spatial cascade of Theorem 3 adds to the adversary's sequential critical path. An adversary resolving a cascade of width L blocks incurs at least $L \cdot \rho$ sequential hash evaluations that cannot be parallelized.

Proof. To replay block w's temporal chain, the adversary needs the causal hashes of $w-1$ at each of w's write times. These causal hashes are outputs of $w-1$'s own temporal chain. Therefore, $w-1$'s full temporal chain (ρ sequential hash evaluations) must complete before w's chain can begin. By the same argument, $w-2$'s chain must complete before $w-1$'s, and so on. For a cascade of width L blocks, the critical path is L sequential temporal chains of ρ links each, totaling $L \cdot \rho$ sequential hash evaluations. Parallel hardware does not reduce this latency because each chain link depends on the output of the previous link (hash chaining) and the completion of the adjacent block's chain (spatial dependency).

This result is significant because it means spatial entanglement converts a work penalty into a latency penalty: the adversary not only performs more total computation but takes more wall-clock time, directly undermining the sequential execution guarantee.

5.5.5. Checkpoint Dominance

Theorem 5 (Checkpoint Dominance). Under spatial entanglement, partial-arena checkpoints are strictly suboptimal. For any adversary using checkpoints of $\alpha \cdot N$ blocks ($\alpha < 1$) at interval C steps, the space-time product $S \times T$ satisfies:

S * T >= (2*rho*(1-alpha) + alpha) * S_full * T_full

where $S_{full} \times T_{full}$ is the space-time product for full-arena checkpoints. For $\rho \ge 1$, this exceeds $S_{full} \times T_{full}$, with the gap increasing linearly in ρ.

Proof. A full-checkpoint adversary stores all N blocks at interval C, giving storage $S_{full} = (K/C) \cdot N \cdot B$ and replay cost $T_{full} = Q \cdot C \cdot d / 2$ per proof generation. The product $S_{full} \cdot T_{full}$ is independent of C (the standard time-space tradeoff).

A partial-checkpoint adversary stores αN blocks at interval C. Storage: $S = \alpha \cdot S_{full}(C)$. When replaying from a partial checkpoint, each of the $C/2$ replayed steps incurs spatial cascade overhead per Theorem 3. Replay cost per challenge: $T = T_{full}(C) \cdot (1 + 2\rho(1-\alpha)/\alpha)$.

The product:

```
S * T = alpha * S_full * T_full * (1 + 2*rho*(1-alpha)/alpha)
      = S_full * T_full * (alpha + 2*rho*(1-alpha))
      = S_full * T_full * (2*rho + alpha*(1 - 2*rho))
```

For $\rho \ge 1$: the coefficient $(1 - 2\rho) \le -1$, so the product is minimized at $\alpha = 1$ (full checkpoints), where it equals $S_{full} \times T_{full}$. Any $\alpha < 1$ strictly increases the product. For $\alpha \to 0$, the ratio approaches 2ρ, meaning the adversary's space-time product is 2ρ times worse than with full checkpoints.

Corollary. Spatial entanglement forces the adversary into an all-or-nothing checkpointing strategy: either store the complete arena at each checkpoint or forgo checkpointing entirely. There is no useful middle ground.

5.5.6. Dynamic Pebbling Game

PoSME's causal DAG is dynamic: edges are created during execution based on data-dependent addressing. In the random oracle model (Theorem 2), each step creates d edges to uniformly random targets. 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 and on the write target's spatial neighbors (stored or recomputed via spatial cascade).

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

Without spatial entanglement, the per-miss recomputation cost is ρ (linear write chain), giving a TMTO ratio of $1 + (1-\alpha) \cdot (2\rho + 1)$. With spatial entanglement, Theorem 3 establishes the tighter bound $1 + 2\rho(1-\alpha)^2/\alpha$, and Theorem 4 proves this overhead is sequential (cannot be parallelized). Theorem 5 further shows that partial-arena checkpoints are strictly dominated by full-arena checkpoints.

5.6. ASIC Resistance

PoSME is anchored in a physics-bounded latency floor, distinct from the bandwidth-hard approach [RenDevadas2017]. 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 3.3.2), 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 5.6.2) 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.

5.6.1. Hardware Latency Parameters

The following table summarizes random-access latency for memory technologies relevant to PoSME's ASIC resistance analysis. All latencies are for random reads with a bank conflict (row miss), which is the bottleneck PoSME forces via Section 3.3.2.

Technology	Random Read Latency	Capacity	Arena Threshold
CPU L3 cache (SRAM)	3-12 ns	16-96 MiB	Below Standard profile
DDR5-5600 ($t_{RP}+t_{RCD}$) [JESD79-5]	26-28 ns	16-256 GiB	Honest Prover baseline
HBM3 ($t_{RP}+t_{RCD}$) [JESD238]	14-17 ns	16-48 GiB (per stack)	1.6-2.0x advantage over DDR5
Custom ASIC SRAM	1-5 ns	64-512 MiB (die area limited)	5-28x advantage over DDR5
Wafer-scale SRAM	1-5 ns (local), 10-20 ns (cross-die)	1-40 GiB	1.4-28x advantage over DDR5

Table 6

The "Arena Threshold" column indicates when the arena exceeds the technology's practical capacity, forcing fallback to slower memory. For the Standard profile (64 MiB arena), the arena fits in large SRAM; for Enhanced (256 MiB) and Maximum (2 GiB), even wafer-scale SRAM is capacity-constrained.

The ASIC advantage ratio for a given adversary memory technology is:

R_ASIC = t_honest / t_adversary

where t_{honest} is the per-hop latency on commodity DDR5 (~27 ns) and $t_{\text{adversary}}$ is the adversary's per-hop latency. For HBM3, $R_{\text{ASIC}} \approx 1.7$x. For on-die SRAM at arena-exceeding sizes (requiring partial SRAM + fallback), R_{ASIC} depends on the SRAM hit rate, which is bounded by the arena size exceeding SRAM capacity.

5.6.2. Wafer-Scale Threshold

The ultimate latency floor for an adversary is on-die signal propagation. Optimal ASIC designs that integrate massive SRAM (1-5 ns access) could achieve a 5-28x advantage over commodity DDR5. Wafer-scale integration, as demonstrated by the Cerebras Wafer-Scale Engine, is the existence proof for this threshold; however, cross-die signal propagation (10-20 ns) narrows the advantage for large arenas that span multiple die regions. PoSME's security is durable because spatial entanglement (Theorem 3) imposes a TMTO recomputation penalty that scales as $1/\alpha$, ensuring that any latency-based speedup is countered by the prohibitive cost of discarding state. Furthermore, the cascade latency (Theorem 4) is sequential, so the adversary cannot hide recomputation behind parallelism.

5.6.3. ASIC SRAM-Packing Vulnerability Analysis

A well-funded adversary can engineer a custom ASIC that integrates massive, high-density on-die SRAM blocks directly adjacent to custom computational execution cores on a single silicon die. Within this unified die area, signal propagation delays across vias and local metal interconnect operate at the speed of light in silicon (~6.7 cm/ns), completely bypassing the external pin-count, bus-width, and motherboard trace latency constraints associated with discrete DRAM or HBM3 interfaces. Such an adversary eliminates the off-chip memory latency that PoSME exploits for honest-execution parity.

However, monolithic on-die SRAM is constrained by three physical limits:

1. *Die area:* 6T-SRAM bit cells occupy approximately 0.021-0.050 um^2 per bit at leading-edge nodes (5 nm and below). Including peripheral circuitry (row/column decoders, sense amplifiers, power distribution), effective SRAM area is approximately 20-40% larger than the raw bit-cell array. Fabricating 512 MiB of on-die SRAM requires approximately 100-230 mm^2 of total die area, consuming a substantial fraction of a reticle-limited die (~800 mm^2 maximum).

2. *Yield rate:* Total defect count scales with die area (Poisson yield model). Leading-edge designs currently integrate up to approximately 200 MiB of on-die SRAM (e.g., large GPU register files and L2 caches); beyond this scale, yield losses are likely to dominate fabrication economics, requiring increasing redundancy overhead that further consumes die area.

3. *Economic cost:* At leading-edge process nodes (5 nm and below), wafer costs exceed $20,000 per wafer. A single die integrating 2 GiB of SRAM would exceed a single reticle and is likely to yield poorly, resulting in per-die costs orders of magnitude above commodity DRAM modules providing equivalent capacity.

PoSME counters the SRAM-packing attack vector by defining standard operational parameter profiles (Section 7.2.2) whose allocation footprints scale intentionally beyond these physical, economic, and yield-rate limits of monolithic on-die SRAM fabrication. For the Enhanced profile (256 MiB arena) and Maximum profile (2 GiB arena), the arena size forces the adversarial ASIC to offload memory states to external commodity memory architectures (DDR5 or HBM3), re-introducing the physical latency bottleneck that the SRAM-packing design sought to eliminate. Formally, let S_{SRAM} denote the maximum economically viable on-die SRAM capacity. For any arena size $N \cdot B > S_{\text{SRAM}}$, the adversary's per-hop latency reverts to $t_{\text{external}} \ge 14$ ns (HBM3) rather than $t_{\text{SRAM}} \approx 1\text{-}5$ ns, bounding the ASIC advantage ratio to:

R_ASIC <= t_honest / t_external (for N*B > S_SRAM)

For DDR5-to-HBM3, this yields $R_{\text{ASIC}} \le 2.0$x. Parameter profiles SHOULD be selected such that $N \cdot B$ exceeds the anticipated S_{SRAM} threshold for the target threat model.

5.7. 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
       7 => read-witness,           ; neighbor-prev (w-1)
       8 => read-witness,           ; neighbor-next (w+1)
   }

   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 7

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 5.5) 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 5.6). 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 SHOULD exceed the adversary's fastest accessible memory (L3 cache, SRAM). Larger N requires more Prover RAM.

* rho controls TMTO resistance: penalty = 1 + 2*rho*(1-alpha)^2/ alpha for an adversary storing alpha*N blocks (Theorem 3). Higher rho requires more steps (longer wall time) but no additional RAM.

7.2.2. Parameter Profiles

Four standard parameter 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 (alpha=0.1)	Use Case
Minimal	2^19	32 MiB	4	4*N	64	2	~64 MiB	65x	Constrained environments, logging micro-services
Standard	2^20	64 MiB	4	4*N	64	2	~128 MiB	65x	Sybil resistance, lightweight provenance
Enhanced	2^22	256 MiB	4	4*N	128	3	~512 MiB	65x	High-assurance content provenance
Maximum	2^25	2 GiB	4	4*N	128	3	~4 GiB	65x	Enterprise/ desktop content authenticity workflows

Table 8

The Minimal profile (32 MiB) is intended for highly constrained execution environments where memory is scarce but a baseline sequential execution proof is still required. Implementations SHOULD use the Standard profile or higher unless resource constraints prevent it. The Standard and Enhanced profiles exceed consumer L3 caches (16-96 MiB as of 2025) and provide latency-bound ASIC resistance via arena size. The Maximum profile (2 GiB) exceeds all current on-die SRAM capacities (see Section 5.6.3) and is RECOMMENDED for enterprise and desktop content authenticity workflows where the Prover has sufficient RAM.

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 9

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
Minimal	64 MiB	64 MiB	~64 MiB
Standard	128 MiB	128 MiB	~128 MiB
Enhanced	512 MiB	512 MiB	~512 MiB
Maximum	4 GiB	4 GiB	~4 GiB

Table 10

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 < 5x at alpha=0.5
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 11

7.4. Performance Estimates

The following properties are machine-independent:

Property	Standard	Maximum
TMTO penalty (alpha=0.1)	65x	65x
ASIC resistance mechanism	Physics-Bound Floor	Physics-Bound Floor
Proof size	~3.9 MiB	~64 MiB

Table 12

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

Profile	Per-step	Wall time	Prover peak RAM
Minimal (32 MiB)	~1200 ns	~3 seconds	~64 MiB
Standard (64 MiB)	~1500 ns	~6 seconds	~128 MiB
Enhanced (256 MiB)	~2200 ns	~37 seconds	~512 MiB
Maximum (2 GiB)	~3200 ns	~430 seconds	~4 GiB

Table 13

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 14

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

8. Side-Channel and Platform Engineering Considerations

8.1. Data-Dependent Access Pattern Disclosure

PoSME features a data-dependent memory access pattern: at each step, the addresses read from the arena are determined by the values previously read (pointer-chasing). This architecture inherently leaks the access execution history via multiple observable channels:

* *Timing side-channels:* The sequence of memory access latencies (including cache hits, bank conflicts, and DRAM refresh collisions) reveals the addresses accessed.

* *Cache-state monitoring:* An adversary sharing a physical CPU (e.g., a co-located process on a multi-tenant cloud server) can observe L1/L2/L3 cache eviction patterns via Flush+Reload, Prime+Probe, or similar attacks to reconstruct the Prover’s access sequence.

* *Electromagnetic and power analysis:* The physical memory bus activity pattern is correlated with the data-dependent address sequence.

These channels are fundamental to PoSME's design; they cannot be eliminated without abandoning the pointer-chasing construction that provides the sequential latency guarantee.

8.2. Security Boundary Framework

Under PoSME's targeted threat model (Section 5.2), the data-dependent access leakage does NOT constitute a vulnerability. The executing local user (the Prover) is actively attempting to prove their own sequential process to an external Verifier. The Prover is not attempting to conceal a static secret cryptographic root-key from a co-located malicious tenant.

The proof itself discloses read addresses and Merkle paths for the Q challenged steps (Section 3.5). A side-channel adversary co-located on the same physical CPU could observe the complete K-step access trace, which is strictly more information than the Q-step subset in the proof. However, this additional information does not compromise security under the intended threat model: the seed is public, the arena contents are deterministically derived from the seed and the execution transcript, and the full access trace is a deterministic function of these public values. An observer who knows the seed and transcript can reconstruct the full trace without side-channel access.

The threat model assumes the Prover controls the execution environment. If the Prover's inputs contain no secret material (which is the case for the provenance use case, where the seed is public), then side-channel observation reveals no information beyond what is already publicly derivable.

8.3. Deployment Restriction

WARNING: If PoSME is deployed outside of its intended provenance and proof-of-process boundaries, the data-dependent memory access pattern creates exploitable side-channels. Specifically:

Implementations MUST NOT use PoSME to process, derive, wrap, or protect high-value long-term secret cryptographic keys.

Implementations MUST NOT deploy PoSME as a key derivation function, password hash, or any construction where the input to the arena initialization or step function contains secret material that must be protected from co-located observers.

If PoSME is executed within a multi-tenant virtualization layer, shared cloud instance, or any environment where an adversary may share physical CPU or memory resources with the Prover, the Prover MUST assume that the full memory access trace is observable by the adversary. In such environments, PoSME MUST NOT be used to process secret inputs.

9. Security Considerations

9.1. 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 by the ratio of the adversary's memory latency to the honest Prover's (see Section 5.6), but is nonzero. Applications requiring temporal guarantees MUST combine PoSME with an external time-binding mechanism such as hardware-attested timestamps. Note that no universal composability (UC) proof is provided for PoSME (see Section 5.1); the security of any such combination requires a dedicated composition analysis and cannot be inferred from the standalone properties proven in this document.

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

9.2. 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.

9.3. 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.

9.4. Verifier Resource Limits

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

9.5. Open Problems

The following problems remain open:

1. *Machine-checked proofs.* The TMTO bounds in Theorems 3-5 rely on pen-and-paper arguments. A machine-checked formalization (e.g., in Lean or Coq) of the spatial cascade lower bound and the checkpoint dominance result would strengthen confidence in the security claims.

2. *Optimal eviction policy.* Theorem 5 establishes that partial-arena checkpoints are dominated by full-arena checkpoints. However, the adversary's optimal online eviction policy (which blocks to retain in working memory during sequential execution) has not been formally characterized. The analysis in Theorem 3 assumes independent miss probabilities; correlated access patterns or adaptive eviction may yield tighter or looser bounds.

3. *Succinct verification.* Verification complexity remains $O(Q \cdot d^R \cdot \log N)$. Reducing this to $O(\log^2 K)$ via FRI/STARK-based commitment (requiring field arithmetic but no trusted setup) is left as future work.

4. *Host-as-critical-path for small arenas.* ASIC resistance at cache-resident arena sizes (where the arena fits within on-die SRAM) requires mechanisms where the host's computation gates the next prover step rather than supplying ancillary entropy. This is deferred to future work.

5. *Spatial entanglement width.* The current construction uses 2 spatial neighbors (w-1, w+1). Wider entanglement (e.g., w-2, w-1, w+1, w+2) would increase the cascade branching factor and tighten the TMTO bound, at the cost of additional hash evaluations per step. The optimal neighbor count has not been analyzed.

6. *Composability.* PoSME is analyzed as a standalone primitive. No universal composability (UC) framework or simulation-based proof is provided. A UC-secure formulation would enable modular composition with timestamping, consensus, or authentication protocols without requiring per-application composition arguments.

7. *Tighter soundness reduction.* The factor-K reduction loss in Theorem 1 arises from a union-bound guess of the divergence step. A tighter reduction (e.g., $O(\log K)$ via binary search or $O(1)$ via rewinding) would improve the concrete security margin for large K.

10. Implementation Considerations

10.1. Bank Mapping and Conflicts

The effectiveness of intra-step bank collisions (Section 3.3.2) 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.

10.2. 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.

10.3. 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 (32 MiB to 2 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.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. 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>.

12.2. 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>.

[C2PA] Coalition for Content Provenance and Authenticity, "C2PA Technical Specification v2.1", 2024, <https://c2pa.org/specifications/specifications/2.1/specs/C2PA_Specification.html>.

[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>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/rfc/rfc7914>.

[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 authors of Argon2 and scrypt for the design principles that informed PoSME's memory-hard foundations. Thanks to participants in the CFRG for constructive review feedback on earlier versions of this document.

Changes from draft-condrey-cfrg-posme-02

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

This document replaces draft-condrey-cfrg-posme, resetting the version counter to -00 under the Independent Submission Stream.

* Migrated from IRTF (CFRG) to Independent Submission Stream; changed submissiontype to "independent" and category to "Experimental".

* Added Applicability Statement scoping PoSME as a specialized hardware-discriminator for content provenance, not a general-purpose PHF or KDF.

* Corrected scrypt hardware characterization: sequential Salsa20/8 latency, not external bandwidth, is the primary bottleneck on commodity architectures.

* Added ASIC SRAM-packing vulnerability analysis with mathematical mitigation rationale for arena sizing.

* Formalized "bad locality" requirement: addressing function MUST behave as a pseudo-random permutation defeating hardware prefetch and caching.

* Added explicit OS2IP / endianness type conversion declarations for octet-string-to-integer index derivation.

* Added top-level Side-Channel and Platform Engineering Considerations section with data-dependent access disclosure, threat model boundary, and implementer warning.

* Defined formal Standard Parameter Profiles table from 32 MiB (Minimal) to 2 GiB (Maximum).

* Corrected CPU Jitter entropy baseline: ~1/3 bits per time delta at OSR=3 (not 1 bit flat).

* Added embedded microcontroller entropy degradation warning.

* Audited all RFC 2119/8174 modal verbs for correctness.

Prior history (draft-condrey-cfrg-posme)

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

Changes from -00 to -02

* Removed Compact profile pending further analysis of cache-resident-arena security.

* Removed Jitter Entanglement section pending further analysis of host-as-critical-path constructions.

* Promoted Address Uniformity from conjecture to Theorem 2, with full ROM proof.

* Widened XOF output from 4 bytes to 8 bytes to eliminate modular bias for non-power-of-2 N.

* Replaced hand-wavy "Trophic Cascade" TMTO claims with rigorous cascade-width analysis (Theorem 3), proving per-miss cost of Theta(rho/alpha).

* Added Theorem 4 (Sequential Cascade Latency): spatial cascade adds to critical path.

* Added Theorem 5 (Checkpoint Dominance): partial-arena checkpoints strictly suboptimal under spatial entanglement.

* Replaced misleading alpha=0 TMTO table with honest comparison across meaningful alpha values.

* Tightened ASIC resistance framing to a single physics-bounded latency floor.

* Reconciled Related Work against arena initialization.

* Rewrote Open Problems to reflect resolved and remaining items.

Author's Address

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