Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: 1 November 2026

D. Condrey
WritersLogic

30 April 2026

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

Abstract

This document defines Proof of Sequential Memory Execution (PoSME), a cryptographic primitive combining mutable arena state, data-dependent pointer-chase addressing, per-block causal hash binding, and execution entropy entanglement 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 symbiotic binding (new data depends on old causal hash; new causal hash depends on cursor), and advances a transcript chain. At regular intervals, CPU jitter entropy is folded into the transcript, binding the proof to physical execution on a general-purpose system. The construction provides four properties: (1) unconditional sequential time enforcement (Omega(K) computation regardless of storage), (2) forgery prevention via causal hashes (reduces to collision resistance of H), (3) TMTO resistance scaling linearly with write density rho = K/N (34x penalty at rho=16 for a zero-storage adversary), and (4) execution- environment ASIC resistance (the adversary must execute on a general-purpose CPU with an operating system to produce valid entropy, limiting the advantage to commodity hardware variation). A compact parameter profile achieves these properties with approximately 32 MiB peak RAM. 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/.

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] resist ASIC acceleration but are single-evaluation primitives with no chain proof system. Composing these (e.g., chaining Argon2id with Merkle sampling) produces a construction where sequentiality and memory-hardness are independent properties; neither reinforces the other.

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 contribution is latency-bound ASIC resistance. Each pointer-chase iteration is bottlenecked by random DRAM access (~45ns on DDR5 [JESD79-5]), with hash computation (~3ns via BLAKE3) as a minor component. The ASIC advantage is bounded by the memory latency ratio (approximately 2x for DDR5 vs HBM3), tighter than the 8-16x bandwidth bounds of Argon2id [Biryukov2016]. Memory latency improves more slowly than bandwidth across technology generations (constrained by signal propagation and DRAM cell sensing time), making this bound more durable than bandwidth-based resistance.

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

Argon2id [RFC9106] resists TMTO via bandwidth-hardness [Boneh2016], with a single-pass TMTO penalty of approximately 2x. PoSME uses Argon2id only for arena initialization. The ongoing computation uses pointer-chasing with in-place writes, creating latency-hardness (2x ASIC bound vs Argon2id's 8-16x). PoSME's TMTO penalty is approximately 2+2*rho for zero-storage adversaries, where rho = K/N is the write density (10x at rho=4 vs Argon2id's 2x).

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 in XOF mode, producing 32-byte output.

XOF(input, index): BLAKE3 XOF 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

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

```
STEP(t):
    // 1. Pointer-chase reads (data-dependent)
    cursor = T_{t-1}
    addrs = []
    for j in 0..d-1:
        a = OS2IP(XOF(cursor, j)) mod N
        addrs.append(a)
        val = A[a]
        cursor = H(cursor || val.data || val.causal)

    // 2. Write with symbiotic binding
    w = OS2IP(XOF(cursor, d)) mod N
    old = A[w]
    new_data = H(old.data || cursor || old.causal)
    new_causal = H(old.causal || cursor || I2OSP(t, 4))
    A[w] = {data: new_data, causal: new_causal}

    // 3. Update commitments
    root_t = MerkleUpdate(root_{t-1}, w, A[w])
    T_t = H(T_{t-1} || I2OSP(t, 4) || cursor || root_t)

    // 4. Log step
    log[t] = {addrs, reads, w, old, A[w], cursor, root_t}
```

3.3.1. Pointer-Chase Addressing

Read addresses are data-dependent: address j+1 depends on the value and causal hash of the block read at address j. This creates a pointer-chasing chain within each step that is inherently sequential.

3.3.2. Symbiotic Binding

The data update incorporates the old causal hash:

```
new_data = H(old_data || cursor || old_causal)
```

This creates bidirectional dependency: data depends on causal history, and causal hash depends on cursor (which depends on data). Neither can be independently fabricated. An adversary forging data must know old_causal; forging causal must know the cursor; computing the cursor requires reading d blocks with their causal hashes.

3.3.3. 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.

3.3.4. Jitter Entanglement

At regular intervals during execution, the Prover samples CPU jitter entropy and folds it into the transcript chain. This binds the proof to physical execution on a general-purpose system with an operating system.

```
ENTANGLE(T_t, m, K):
    interval = K / m
    jitter_samples = []

    // At steps t where t mod interval == 0:
    j = sample_cpu_jitter()
    jitter_samples.append((t, j))
    T_t = H("PoSME-entangle-v1" || T_t || j)
```

The jitter source MUST be CPU execution timing variance (e.g., rdtsc delta across a calibration loop), not a pseudorandom generator. CPU jitter arises from instruction pipeline stalls, cache miss variability, branch misprediction recovery, OS scheduler preemption, and memory controller contention. These are emergent properties of general-purpose CPU execution under an operating system that cannot be replicated by fixed-function hardware.

The Prover MUST include at least m = 16 jitter samples in the proof. The number of samples and the injection interval are included in the proof and verified.

The Verifier checks:

1. Each jitter sample is incorporated into the transcript at the claimed step via the entanglement hash.

2. The jitter samples exhibit sufficient min-entropy (at least 1 bit per sample, measured via the NIST SP 800-90B most common value estimator).

3. The final transcript T_K depends on all jitter samples.

An adversary using fixed-function hardware (ASIC/FPGA) cannot produce genuine CPU jitter. Injecting pseudorandom values fails the min-entropy test when the Verifier cross-checks against the expected statistical properties of OS scheduling jitter (quantized intervals, interrupt clustering, sub-microsecond variance). The adversary's only option is to execute on a general-purpose CPU with an OS, limiting the hardware advantage to commodity CPU-to-CPU performance variation (approximately 2x).

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, causal hashes impose a constant-factor increase on the cost of recomputing missing blocks. Without causal hashes, an adversary recomputing a missing block traverses its write chain at cost O(rho) hashes (one per write in the chain). With causal hashes, the adversary must traverse both the data chain and the causal chain, doubling the cost to O(2*rho) per miss.

This is a MODERATE improvement: a 2x constant factor on write chain traversal, not an exponential blowup. The TMTO penalty table in Section 5.4 incorporates this factor. The causal hash mechanism's primary contribution is soundness (Section 5.2), not TMTO amplification.

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 Arena Coverage

Each step writes 1 block at a uniformly random address (in the ROM). After K steps, the fraction of blocks written at least once is phi = 1 - e^{-rho}, where rho = K/N is the write density.

A block that was never written retains its initialization value, recomputable in O(1) from the seed. Only written blocks require write chain traversal for recomputation. The expected write chain length for a modified block is rho, costing O(rho) hashes.

rho = K/N	Blocks modified	Recomp cost/miss	alpha=0 TMTO ratio
0.25	22%	O(1)	2.5x
1	63%	O(1)	4x
4	98%	O(4)	10x
16	~100%	O(16)	34x

Table 2

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

5.4.3. Per-Step Recomputation Cost

Theorem 2 (TMTO). In the random oracle model, any adversary storing alpha * N arena blocks and all K cursors (K * 32 bytes) performs expected computation at least:

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

giving a TMTO ratio of T_adv / T_honest >= 1 + (1-alpha) * (2*rho + 1).

Proof sketch. The adversary computes the K-step transcript chain (Theorem 1 of the companion analysis: Omega(K) sequential floor). At each step, d blocks are read at addresses uniform in [N] (random oracle property). Each read misses the stored set with probability 1-alpha. Each miss requires traversing the block’s write chain (expected length rho) for both data and causal hash, costing 2*rho + 1 hashes. The per-step cost is d * (1 + (1-alpha) * (2*rho + 1)). Summing over K steps gives the bound.

This bound assumes optimal cursor storage (adversary stores all K cursors). A full derivation and optimality analysis for alternative cursor strategies is provided in the companion analysis (to appear as IACR ePrint).

rho = K/N	alpha=0 penalty	alpha=0.5 penalty
1	4x	2.5x
4	10x	5.5x
16	34x	17.5x

Table 3

The penalty scales linearly with rho.

This bound assumes the adversary stores all K cursors (K * 32 bytes; 512 MiB for K = 2^26). Storing all cursors is optimal for the adversary: it eliminates the need for sequential replay from checkpoints. An adversary who stores cursors at intervals of L steps instead pays an additional L * d * (1 + (1-alpha) * (2*rho + 1)) hashes per cursor miss. Sparse cursor storage strictly increases the adversary's cost; the bound above is a LOWER bound on the TMTO penalty.

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. The pebbling game:

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 designed to be latency-dominated: hash computation constitutes less than 10% of per-step cost, making the bottleneck memory random-access latency rather than computation throughput. This is a structural property of the construction (it holds for any instantiation where H is significantly faster than a DRAM random read).

The specific advantage ratio depends on the memory technology available to the adversary. Current measurements:

Component	Consumer DDR5	ASIC (HBM3)	Ratio
Memory read	~45ns	~20ns	2.25x
BLAKE3 hash	~3ns	~0.3ns	10x
Total	*~48ns*	*~20.3ns*	*~2.4x*

Table 4

These are empirical values for 2024-era memory technology, not formal bounds. The approximately 2x ratio reflects the current DDR5-to-HBM3 latency gap [JESD79-5] [JESD238]. With aggressive controller optimization: up to 3x.

An adversary with on-die SRAM (~1-5ns access) could cache a fraction of hot blocks. However, with 1 GiB arena and typical ASIC SRAM budgets (8-32 MiB), the SRAM hit rate for uniform random addresses is 0.8-3.1%, providing negligible benefit. Larger SRAM allocations are economically prohibitive (1 GiB of on-die SRAM costs orders of magnitude more than 1 GiB of DRAM).

For comparison, bandwidth-hard constructions (Argon2id) have empirical ASIC advantage of 8-16x [Biryukov2016] [RenDevadas2017]. The structural argument for PoSME's durability: memory latency is constrained by DRAM cell sensing time and signal propagation, which improve more slowly than bandwidth (wider buses, more channels) across technology generations.

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
    ? 5 => [+ jitter-sample],    ; jitter entanglement (REQUIRED if N < 2^20)
}

jitter-sample = {
    1 => uint,                    ; step-id (injection point)
    2 => bstr .size 32,           ; jitter-value
}

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)
}

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
}

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 5

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 two complementary mechanisms: (a) arena size exceeding the adversary's fastest memory (Section 5.5), or (b) mandatory jitter entanglement requiring a general-purpose CPU with an OS (Section 3.3.4). 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.

For applications where ASIC resistance is provided by jitter entanglement (Section 3.3.4) rather than arena size, N can be reduced while rho is increased. This yields strictly superior TMTO resistance with dramatically lower RAM. The Compact profile exploits this trade-off.

7.2.2. Parameter Profiles

Four profiles are defined. All profiles share fixed parameters: block size B = 64 bytes, reads per step d = 8, hash function H = BLAKE3. Jitter entanglement (Section 3.3.4) is REQUIRED for the Compact profile and RECOMMENDED for all profiles.

Profile	N	Arena	rho	K	Q	R	Peak RAM	TMTO	Use Case
Compact	2^18	16 MiB	16	16*N	64	2	~32 MiB	34x	Attestation, mobile, resource-constrained
Standard	2^20	64 MiB	4	4*N	64	2	~128 MiB	10x	General attestation, Sybil resistance
Enhanced	2^22	256 MiB	4	4*N	128	3	~512 MiB	10x	High-assurance with latency-bound ASIC resistance
Maximum	2^24	1 GiB	4	4*N	128	3	~2 GiB	10x	Consensus, mining, custom hardware adversaries

Table 6

The Compact profile uses a small, cache-resident arena with high write density. Because the arena fits in L3 cache, per-step latency is lower (~300 ns vs ~2750 ns), but TMTO resistance is 3.4x stronger than the Maximum profile (34x vs 10x). ASIC resistance is provided by mandatory jitter entanglement rather than arena size: any adversary must execute on a general-purpose CPU with an OS to produce valid entropy samples, limiting the hardware advantage to commodity CPU-to-CPU variation (~2x).

The Compact profile's rho MAY be increased beyond 16 for stronger TMTO at the cost of additional wall time:

rho	TMTO (alpha=0)	K (N=2^18)	Wall time (~300 ns/step)
16	34x	4.2M	~1.3 seconds
64	130x	16.8M	~5 seconds
256	514x	67M	~20 seconds

Table 7

The Standard and Enhanced profiles exceed consumer L3 caches (16-36 MiB as of 2024) and provide latency-bound ASIC resistance via arena size. 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 8

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
Compact	32 MiB	128 MiB	~32 MiB
Standard	128 MiB	128 MiB	~128 MiB
Enhanced	512 MiB	512 MiB	~512 MiB
Maximum	2 GiB	2 GiB	~2 GiB

Table 9

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 10

When N < 2^20 (arena fits in consumer L3 cache), jitter entanglement (Section 3.3.4) is REQUIRED. The Verifier MUST reject proofs with N < 2^20 that do not include valid jitter samples.

7.4. Performance Estimates

The following properties are machine-independent:

Property	Compact	Standard	Maximum
TMTO penalty (alpha=0)	34x	10x	10x
ASIC resistance mechanism	Jitter	Arena + Jitter	Arena
Proof size	~3.9 MiB	~3.9 MiB	~64 MiB

Table 11

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

Profile	Per-step	Wall time	Prover peak RAM
Compact (16 MiB, rho=16)	~300 ns	~1.3 seconds	~32 MiB
Compact (16 MiB, rho=256)	~300 ns	~20 seconds	~32 MiB
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 12

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

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

Table 13

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 fundamentally 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; skewed distributions may enable hot-block caching. The tight relationship between committed step frequency C, write density rho, and the optimal adversary strategy requires further analysis.

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. 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>.

[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>.

10.2. 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>.

[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, and the authors of Argon2 for the initialization primitive used in PoSME.

Author’s Address

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