Mathematical Foundations of Sibuna: Sequential Work, Symmetric Authentication, Bounded State, and Linear-Time Inspection

STATE

published

STATUS

Published

TYPE

Architectural Specification

AUTHORS

Sibuna Contributors <team@sibuna.local>

CREATED

2026-09-07

UPDATED

2026-10-15

SUMMARY

A paper-style record of the mathematics behind every Sibuna hot-path primitive: the Cohen-Pietrzak proof of sequential work and the geometric hashcash tier, keyed-hash tokens and stateless challenges with work-bounded state, GCRA rate limiting, Robin Hood spent sets, Aho-Corasick and single-pass tokenizers for inspection, read-copy-update engine slots, adaptive difficulty, and feature-hashed campaign clustering, each with definitions, lemmas, proofs, citations, and the exact pure-Zig implementation and measurement that realises it.

DISCUSSION

No external discussion URL assigned

LABELS

mathematics, cryptography, proof-of-work, posw, rate-limiting, hashing, automata

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Revision note 2
2. Abstract 3
3. Introduction and Adversary Model 4
4. Part A: The Hashcash Tier 7
5. Part B: Proof of Sequential Work 8
6. Part C: Stateless Challenges and Symmetric Tokens 10
7. Part D: Rate Limiting by the Generic Cell Rate Algorithm 13
8. Part E: Robin Hood Hashing for the Spent Set 13
9. Part F: Linear-Time Inspection 15
10. Part G: Read-Copy-Update Engine Slots 16
11. Part H: Load-Adaptive Difficulty 17
12. Part I: Campaign Clustering by Feature Hashing 17
13. Verification and Benchmark Gates 18

14. Security Considerations 18
15. Open Problems 18
16. References 19

1. Revision note

2026-10-15: Ported from the Typst original in the Zig repository. The record describes the Zig implementation as it was decided; SID 0010 maps its packages to the Rust crates of the port.

2026-10-15 (Rust port, Parts A to C): SHA-256 in `sibuna-crypto` detects the x86-64 SHA extensions and the ARMv8 SHA-2 instructions at run time, where Zig's standard library chose at build time from the target features, so one binary uses the instructions on every CPU that has them; a build without the `std` feature still decides at compile time. Digests, and therefore every Hashcash and PoSW result, are unchanged. The PoSW verifier derives the challenged leaves γ_i from one SHA-256 state that has absorbed the statement χ once, instead of rehashing χ for every i. The digests and the count of hash evaluations in Lemma 12 are the same, and verification costs t fewer SHA-256 compressions. Ed25519 tokens are verified with width-5 non-adjacent-form digits for the public key and width-8 digits over a precomputed table of base-point multiples, instead of Zig's signed 4-bit windows, and the public key is decoded once when the coordinator starts instead of on every verification. Decoding now rejects off-curve, identity and small-order keys, which Zig rejected only in `verify`. Exactly the same signatures are accepted, and verification is faster. The rule hash of Definitions 17 and 20 and the ban-table key are Wyhash; `sibuna-crypto` counts the input length in a `u64` where Zig used `usize`. The two agree bit for bit on every 64-bit target and differ only on 32-bit targets for inputs of 4 GiB or more, which Sibuna never hashes. The measurements quoted in this record are those of the Zig build.

2026-10-15 (Rust port, Parts D, E and H): the GCRA limiter, the spent set and the adaptive controller keep their definitions, table layouts and probe rules. Zig's limiter already saturated its arithmetic; the port also saturates the conversion of a window in seconds to milliseconds in its convenience entry point. The adaptive controller and the coordinator saturate where Zig would have trapped on overflow (a difficulty plus the bump past `u32::MAX`, an EWMA product, a smoothing factor `alpha_256` above 256, a time near `u64::MAX` plus a lifetime). Theorem 23 and Definition 32 hold unchanged for every value the daemon can produce.

2026-10-15 (Rust port, Part F): the automata of `sibuna-patterns` index their rows by byte class instead of by byte. A class is a byte that occurs in some pattern, with both cases of a letter forming one class under case folding, or the one class of all other bytes. Each row holds one `u32` successor per class, so construction takes $O(mc)$ time and space for $c \leq \sigma$ classes instead of $O(m\sigma)$, and a scan is still one dependent load per input byte with failure links folded in: Theorem 29 holds with c in place of σ. Match states are numbered after all other states, so the scan detects a match with one comparison. Patterns equal under case folding are all kept, and the one added first wins a first-match scan, where Zig overwrote the earlier one; Sibuna's signature tables contain no such duplicates. Tags run from 0 to 63, so a set of tags is one 64-bit word. Findings are unchanged.

2026-10-15 (Rust port, Part G): the slot protocol of Definition 30 is not ported. The daemon publishes each rebuilt engine as an owned value behind an `Arc` (SID 0010). Each request thread reads it through a cache that checks a generation counter with one acquire load per request and clones the `Arc` only after a publication; an engine is freed when the last thread holding it moves on, and no writer waits for readers to drain. An engine is never rebuilt in place, so the property of Lemma 31 holds by reference counting, and the fourth open problem does not apply to the port. The incident ring is unchanged.

2026-10-15 (Rust port, Part I): sqlite-vec is gone with Zaxonlite. The storage thread keeps the most recent 16,384 embeddings of all nodes in memory and assigns an incident to the campaign of its nearest neighbour among them (SID 0011), instead of a nearest-neighbour query over every recorded incident. An incident older than that window is not compared, so a variant that recurs after 16,384 newer incidents founds a new campaign. The embedding, the cosine threshold of 0.35 and the campaign identifiers are unchanged.

Implementation review: 2026-09-07
The cited sequential-work results apply under their random-oracle assumptions to the constructions and bounds in the papers. They are not an independent audit of this concrete SHA-256 protocol, its parameter calibration, browser timing or quantum security level. Cohen-Pietrzak (2018) and Blocki-Lee-Zhou (2021) are the primary references. Hash evaluations below must not be confused with SHA-256 compression calls: variable-length labels and padding can require multiple compressions.

2026-10-15 (Rust port, security fix, Parts A to C): the ban-table key is no longer Wyhash with a public seed; see SID 0004.
2026-10-15 (Rust port, security fix, Parts D and E): the limiter evicts the oldest-TAT cell of a full probe window instead of refusing (Theorem 23 holds for every client whose cell is not evicted; an evicted client restarts from an idle state). The spent set's capacity is configurable (`--spent-tags`, default 1,048,576 tags, 24 MiB, 24-byte slots); the Zig set's 65,536 tags were exhausted by about 220 verified solutions per second within a five-minute challenge lifetime, after which honest solutions were refused.
2026-10-15 (Rust port, security fix, Part H): the adaptive controller's input is the rate of verified solutions, not of issued challenges, which cost a flooder nothing; issuance only advances the bucket clock so the average decays. Arithmetic and bounds are unchanged.
2026-10-15 (Rust port, Parts A to C, primitives): BLAKE3 in `sibuna-crypto` is portable code only. Zig's SIMD paths hash large inputs several times faster, but Sibuna hashes only inputs under 100 bytes with BLAKE3 (token tags, fingerprints, the key schedule), where the portable code is faster. Ed25519 signing cannot fail; Zig's `sign` returned an error when the nonce reduced to zero, an event of probability about 2^{-252}. Signing keys, the master seed and the derived keys are wiped when dropped.
2026-10-15 (Rust port, Parts A and B, solvers): functions that Zig defined only for valid parameters are total and never panic. `challenge_leaf` keeps all 32 bits for depths of 32 or more, where Zig's shift would be illegal; `leaf_count` saturates; and `meets_hex` refuses a hexadecimal difficulty whose bit count overflows. Valid parameters (depth 4 to 24, 1 to 32 challenges, as in Zig) never reach these cases, and the prover keeps Zig's depth-first recursion, at most 25 frames deep. The native reference solver and the browser solver share one Hashcash search, which absorbs the challenge and the colon once and clones that SHA-256 state for each nonce, as the Zig browser solver did; the Zig reference solver rehashed the prefix for every nonce. The nonce found is the same.
2026-10-15 (Rust port, Part E): a spent-set slot no longer stores the probe distance or an occupied flag. The distance follows from the slot index and the tag's home slot, and a zero expiry marks an empty slot (an expiry of zero is stored as one), so a slot is 24 bytes. The 16 shards share the configured capacity, which is clamped to 65,536 to 16,777,216 tags and rounded up to a power of two. Definition 25 and Lemmas 26 to 28 apply unchanged, because the distances are the same numbers, computed instead of stored. Lookup is still capped at 64 probes.
2026-10-15 (Rust port, Part F, construction): the capacity of an automaton is a value given to its builder instead of a compile-time parameter, with the same failures. A failed `add` leaves the builder unchanged; Zig kept the states it had created. The built-in signature sets are sized from their pattern lengths, as before.

2. Abstract

A paper-style record of the mathematics behind every Sibuna hot-path primitive: the Cohen-Pietrzak proof of sequential work and the geometric hashcash tier, keyed-hash tokens and stateless challenges with work-bounded state, GCRA rate limiting, Robin Hood spent sets, Aho-Corasick and single-pass tokenizers for inspection, read-copy-update engine slots, adaptive difficulty, and feature-hashed campaign clustering, each with definitions, lemmas, proofs, citations, and the exact pure-Zig implementation and measurement that realises it.
Sibuna is a web firewall whose admission decisions must cost the server almost nothing and cost an unverified client a measurable, unavoidable amount of work. This record states the mathematics that makes that asymmetry a theorem rather than a hope. It gives, for every hot-path primitive, a definition, the lemmas or theorems that bound its cost and its security, a citation to the primary literature, and the exact pure-Zig function that realises it together with the number we measured on the reference host.

The central design decision is the choice of the client puzzle. SID 0002 proposed three tiers (SHA-256 hashcash, HashX, Argon2id). On review, HashX carries no formal proof and requires an interpreter, Argon2id makes verification as expensive as solving, and the memory-hard puzzle we considered as a replacement (Equihash) is a generalised-birthday construction with a cryptocurrency lineage and known quantum speedups for its underlying k-XOR problem. Sibuna therefore adopts the Cohen-Pietrzak **proof of sequential work** [4] as its research-grade tier: a construction with a sequentiality theorem in the random oracle model, a post-quantum proof in the parallel quantum random oracle model [6], deterministic solve time, $O(\log N)$ prover memory, and $O(t \log N)$ verification. Bit-granular hashcash [1] remains as the lightweight tier.

Around the puzzle, the record proves: that a 16-byte keyed-BLAKE3 tag bounds forgery by $q/2^{128}$ and makes both challenges and session tokens stateless and post-quantum; that the only server state the design keeps, the **spent set**, is bounded by work the adversary actually performed; that the GCRA limiter admits at most $N + \lfloor L/T \rfloor$ requests in any interval of length L with sixteen bytes of state per client; that Robin Hood lookups may terminate early and may overwrite expired occupants exactly when the incoming probe distance is not smaller; that the inspection pipeline is linear in the input with one automaton pass and one byte-class pass; and that the read-copy-update slot protocol never rebuilds an engine a request is still reading. Every measurement quoted is ours, from `benchmarks/results/latest.json` on an Apple M1; reference figures for other systems are labelled as models and never as measurements.

3. Introduction and Adversary Model

3.1. Setting

A request arrives at the daemon from a client that is either a human browser, an authorised crawler, or an unverified automaton. The daemon must decide, within a bounded request-classification budget, whether to forward the request to the origin, refuse it, or demand a proof of work. The client that presents a valid proof receives a session token and is not asked again for the token's lifetime.

The adversary controls unbounded parallel hardware (GPU farms, ASICs, botnets with fresh IP addresses for every request), sees every byte the daemon emits, may replay, forge, or reorder messages, and may submit arbitrary garbage to every endpoint at line rate. The adversary does not hold the daemon's master seed.

3.2. Axioms

The engineering priorities of SID 0001 (safety, then performance, then developer experience) become four axioms that every primitive in this record must satisfy. They are stated as axioms because the rest of the design is derived from them, not because they are self-evident.

Axiom 1 (Asymmetry)

For every admission mechanism, the server's cost to issue and to verify one instance is bounded by a constant c_s, while the client's expected cost to produce an accepted instance is at least W, with $W/c_s \geq 10^3$ for the lightest tier and W/c_s tunable upward without changing c_s by more than a small constant.

Axiom 2 (Zero allocation)

No code on the request path allocates from the heap. Every buffer is a fixed-size stack or static object whose size is a compile-time constant. Consequently the memory consumed by the daemon under a flood is a function of configuration only, never of traffic.

Axiom 3 (Native execution)

Verification executes as native machine code of the host, using hardware instructions where the target exposes them. The server never interprets or JIT-compiles the client's program.

Axiom 4 (Bounded, work-backed state)
Any per-client or per-challenge state the daemon must remember is bounded by a fixed table, and an adversary cannot occupy a table entry without first performing the work that entry represents.

3.3. Cost calculus

Write c_{iss} for the server cost to issue a challenge, c_{ver} for the cost to verify a submission, W for the expected client work per accepted submission, and λ for the adversary's request rate. The adversary's spend rate to obtain admission tokens is λW; the server's spend rate to resist is $\lambda(c_{\text{iss}} + c_{\text{ver}})$. Axiom 1 requires the ratio $W/(c_{\text{iss}} + c_{\text{ver}})$ to be large. With stateless challenges (Part C) c_{iss} is one keyed hash, and with the tiers of Parts A and B c_{ver} is one SHA-256 compression or $t(n+1)$ compressions respectively. Measured on the reference host the ratio is $2^{16} \cdot 300\,\text{ns}\ /62.6\,\text{ns} \approx 3 \cdot 10^5$ for the hashcash tier at 16 bits and $138\,\text{ms}\ /16.9\,\mu\text{s} \approx 8 \cdot 10^3$ for the sequential tier at depth 16 in a browser.

The figure below fixes the pipeline that the rest of the record analyses; each box names the part that proves its properties.

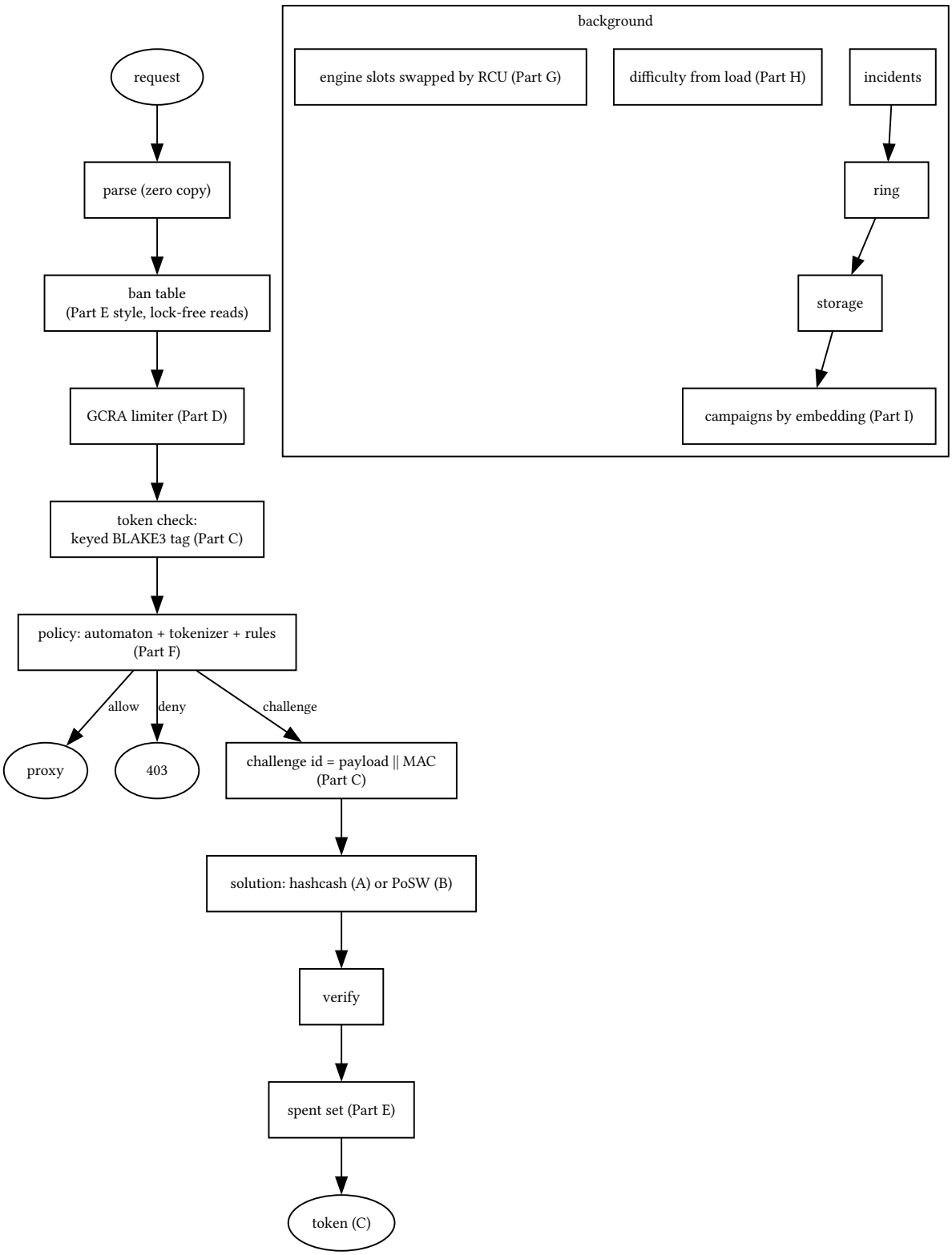


Fig. 1 The request pipeline and the part of this record that proves each stage.

4. Part A: The Hashcash Tier

Definition 5 (Hashcash puzzle)
Let H be a hash function modelled as a random oracle with 256-bit output, χ a challenge string, and $b \in \{0, \ldots, 256\}$ the difficulty in bits. A solution is a nonce $\eta \in \mathbb{N}$ such that the first b bits of $H(\chi \,\|\, : \| \,\mathrm{dec}\,(\eta))$ are zero, where dec is the decimal encoding.

Lemma 6 (Trial distribution)
Under the random oracle model each trial succeeds independently with probability $p = 2^{-b}$. The number of trials X to the first success is geometric with $\mathbb{E}[X] = 2^b$, median $\lceil 2^b \ln 2 \rceil$ (to within one), and tail $\mathbb{P}[X > k \cdot 2^b] = (1 - 2^{-b})^{k2^b} \leq e^{-k}$.

Proof of Lemma 6
Independence and uniformity of oracle outputs give $\mathbb{P}[X = j] = (1-p)^{j-1}p$. The expectation of a geometric variable is $1/p$. The median m satisfies $(1-p)^m = 1/2$, so $m = -\ln 2 / \ln(1-p) \approx 2^b \ln 2$ for small p. The tail follows from $(1-p)^{k/p} \leq e^{-k}$ since $\ln(1-p) \leq -p$. □

The tail is what an operator feels: at any difficulty, about 5% of honest clients need more than three times the mean, and 0.7% need more than five times. This variance is inherent to nonce search and is the first reason the sequential tier of Part B is preferred for the human-facing default.

Lemma 7 (Bit-granular calibration)
Raising b by one doubles $\mathbb{E}[X]$. Measuring difficulty in hexadecimal digits, allows only factors of sixteen between adjacent settings; measuring in bits allows any factor 2^k.

Quantum caveat
Grover's algorithm [24] finds a marked item among 2^b candidates in $\Theta(2^{b/2})$ oracle queries. A quantum adversary therefore solves a b-bit hashcash puzzle with quadratically fewer hash evaluations. This halves the effective difficulty rather than breaking the scheme, and it applies to **every** nonce-search puzzle including memory-hard ones; the only construction in this record immune to it is the sequential tier, whose cost is depth rather than search.

Implementation
`libs/crypto/src/pow.zig`: `computeHashcash` absorbs the challenge, a colon, and the decimal nonce; `checkDifficultyBits(digest, bits)` tests `bits/8` whole zero bytes and a mask on the remainder; `verifyHashcashBits` composes the two. The hash is `std.crypto.hash.sha2.Sha256`, which dispatches to the ARMv8 `sha256h`/`sha256h2` instructions on aarch64 targets with the `sha2` feature and to SHA-NI on x86-64 with `sha` and `avx2`. Measured verification: **62.6 ns** per solution, zero heap bytes. The browser solver in `apps/wasm-pow/src/entry.zig` (`sibuna_solve_step`) pre-absorbs the prefix once and copies the 32-byte SHA state per nonce; in V8 it runs at about 0.3 μs per hash, so 16 bits takes about 20 ms.

Rust port note (2026-10-15)
The Rust SHA-256 chooses its hardware path at run time, not at build time. Digests are unchanged. See the revision note.

5. Part B: Proof of Sequential Work

5.1. Why sequential work

Nonce search is embarrassingly parallel: an adversary with P processors divides the expected wall-clock time by P, and Grover divides the query count by $2^{b/2}$. Memory-hard puzzles raise the per-processor cost by forcing a memory footprint, but they remain parallel across processors and, as puzzles, remain nonce searches. A **proof of sequential work** changes the axis: the prover must perform N hash evaluations **one after the other**, so P processors and a quantum computer alike gain at most a constant factor from parallelism, and the honest client's time is deterministic.

5.2. The construction

We follow Cohen and Pietrzak [4]. Let n be the depth and $N = 2^{n+1} - 1$ the number of nodes.

Definition 8 (The DAG G_n)

Nodes are binary strings v of length $\mid v \mid \leq n$; the root is the empty string ε and the leaves are the 2^n strings of length n. Edges point from each child $v0, v1$ to its parent v. In addition, for every leaf $u = u_1 \ldots u_n$ and every position i with $u_i = 1$, there is an edge from $u_1 \ldots u_{i-1}0$ (the left sibling of u's ancestor at depth i) to u.

Definition 9 (Labelling and commitment)

Fix a statement χ (Sibuna uses the 70-byte challenge identifier of Part C) and a hash H. The label of a node is $\ell_v = H(\chi \,\|\, \mathrm{enc}\ (v) \,\|\, \ell_{v_1} \,\|\, \ldots \,\|\, \ell_{v_d})$ where $v_1, \ldots, v_d$ are the parents of v in a fixed order and $\mathrm{enc}\ (v)$ is the depth byte followed by the little-endian index within the level. The commitment is $\varphi = \ell_\varepsilon$.

The extra leaf edges are the whole point: leaf u cannot be labelled until every left sibling of its ancestors is labelled, which in turn requires every leaf to the left of u. A depth-first post-order traversal is therefore the only order in which the labels can be computed, and it performs exactly N hash evaluations one after the other.

Definition 10 (Openings and the non-interactive proof)

After computing φ, the prover derives t leaves $\gamma_i = H(\chi \,\|\, \varphi \,\|\, i) \bmod 2^n$ (Fiat-Shamir). For each γ_i it emits the leaf label and the n sibling labels along the root path. The proof is φ followed by the t openings: $32(1 + t(n + 1))$ bytes.

The verifier recomputes the leaf label from the siblings that the leaf's extra edges name (exactly the left siblings at positions where γ_i has a 1 bit, all of which are among the transmitted siblings), then folds upward, hashing the pair of children at each level, and accepts if the result equals φ for every i.

5.3. Theorems

Theorem 11 (Scope of the sequentiality results [4, 6])

The cited papers establish sequential-work security in their classical and quantum random-oracle models, with bounds depending on the adversary's query budget, sequential rounds and protocol parameters. Their full bounds, including Fiat-Shamir query effects, must be used for calibration; this record does not derive a concrete bound for Sibuna.

For intuition only, if a fixed commitment has an invalid fraction α of leaf openings and t independent uniformly sampled leaves are checked, the chance of missing all invalid leaves is $(1 - \alpha)^t$. Sibuna derives samples from the commitment. An adversary can try many commitments and reuse work, so this elementary fixed-commitment calculation does not prove that cheating is unprofitable, nor that two openings

suffice. The previous unconditional "cheating never pays" lemma is withdrawn. Default $t = 16$ is an engineering parameter requiring independent adversarial calibration, not a 128-bit soundness guarantee.

Lemma 12 (Verification cost)
Verifying a proof costs exactly $t(n + 1)$ hash evaluations of at most $32(n + 2) + 37$ input bytes each, plus t evaluations to derive the challenges, and needs $O(1)$ working memory beyond the proof itself.

Proof of Lemma 12
Per opened leaf: one leaf hash over at most n sibling labels, then n parent hashes over two labels each. The verifier walks the proof in place and keeps one 32-byte running label. □

Rust port note (2026-10-15)
The Rust verifier derives the t challenges from one SHA-256 state that has absorbed χ once. The count of hash evaluations is unchanged, and verification costs t fewer compressions.

Lemma 13 (Prover memory)
With the top m levels of labels retained, the prover needs $2^{m+1} - 1 + (n + 1)$ labels of memory, performs N sequential hashes for the commitment, and an additional $t(2^{n-m+1} - 1)$ hashes to open the t challenges.

Proof of Lemma 13
During the depth-first pass the only labels needed for future leaves are the left siblings of the current path, one per depth: a stack of $n + 1$ labels. The top m levels are copied out as they are produced. To open leaf γ, the prover restores the stack entries above depth m from the retained levels and recomputes the subtree rooted at γ's depth-m ancestor, capturing the siblings and the leaf on the way; that subtree has $2^{n-m+1} - 1$ nodes. □

With $m = 10$ the retained labels occupy $2047 \cdot 32 = 64$ KB, and for $n = 16, t = 16$ the openings add $16 \cdot 127 = 2032$ hashes to the 131071 of the commitment: 1.6% overhead. The prover therefore fits in under 90 KB including the proof buffer, which is what makes it acceptable on a phone and what makes the WebAssembly module 8.8 KB.

Theorem 14 (Post-quantum security, Blocki-Lee-Zhou [6])
The Cohen-Pietrzak proof of sequential work remains a secure proof of sequential work against quantum adversaries in the parallel quantum random oracle model: any quantum prover making fewer than $N(1 - \alpha)$ sequential rounds of (superposition) queries is accepted with probability negligible in t and the output length, up to polynomial loss.

We cite the theorem rather than reprove it; its proof uses the compressed-oracle technique to track which labels a quantum adversary has "computed". This motivates the sequential tier under the cited oracle assumptions. It does not independently certify Sibuna's concrete protocol or parameter choices.

5.4. Why not the alternatives

Construction	Server verify	Parallel/quantum	Proof status	Verdict
Hashcash [1]	1 hash	fully parallel; Grover $\sqrt{\cdot}$	random oracle, folklore	Kept as Tier 1: cheapest verification, high variance

Construction	Server verify	Parallel/quantum	Proof status	Verdict
Cohen-Pietrzak PoSW [4]	$t(n+1)$ hashes	sequential; QROM-secure [6]	EUROCRYPT 2018 theorem; ITC 2021 quantum	**Adopted as Tier 2**
Equihash [7]	2^k hashes	parallel; GPU-friendly; k-XOR quantum speedups [9]	NDSS 2016; Wagner [8] analysis; cryptocurrency lineage	Rejected: memory hardness does not stop parallelism and the nonce search inherits Grover
Argon2id (RFC 9106) [10]	full memory-hard evaluation	parallel across instances	PHC winner; MHF proofs	Rejected as a puzzle: verification costs as much as solving, violating Axiom 1
HashX (Tor)	program generation + interpretation	parallel	no formal proof	Rejected: needs an interpreter on the server (Axiom 3)
RSA / class-group VDFs	1 modexp with trapdoor	sequential	group-of-unknown-order assumptions	Rejected: Shor [25] breaks the group assumptions

The Argon2id row deserves emphasis. A puzzle whose verification is the same memory-hard function forces the server to spend the same 16 to 64 MB and tens of milliseconds as the client on every submission. Since submissions are free to send, an adversary who fetches challenges and posts garbage converts each of its cheap requests into a full memory-hard evaluation on the server: the inverse of Axiom 1.

5.5. Implementation and calibration

Implementation

`libs/crypto/src/posw.zig`: `Params{depth, challenges}` with `validate`, `proofSize`; `challengeLeaf` derives γ_i; `verify` implements the verifier in `t(n+1)` hashes with no allocation; `Workspace` holds the retained levels (`top`, 2^{11} labels), the sibling `stack`, and the proof buffer; `solve` runs the depth-first prover with the `Prover.labelOf` recursion and `open` for captures. The identical source is compiled for `wasm32-freestanding` and exported by `apps/wasm-pow/src/entry.zig` as `sibuna_posw_solve`; `apps/web/src/worker.js` carries a JavaScript prover that we verified produces byte-identical proofs to the WASM module (Node/V8 harness). The coordinator maps work bits b to depth $n = b - 3$ (`ChallengeSpec.posw_depth_offset`) so that the two tiers cost a browser about the same wall-clock time at equal b.

The reproducible suite measures native verification at depth 13 with sixteen openings. Browser solve-time calibration from earlier drafts depended on a scratch harness absent from this repository, so those timing tables are withdrawn. Work-bit parity between the two tiers is an engineering approximation and must be calibrated on target client hardware.

6. Part C: Stateless Challenges and Symmetric Tokens

6.1. Pseudorandom functions and message authentication

Definition 15 (PRF and MAC)

A keyed function F_K is a (q, ε)-PRF if no adversary making q queries distinguishes it from a random function with advantage more than ε. A MAC built as $\mathrm{Tag}_K(m) = \mathrm{trunc}_\tau(F_K(m))$ is existentially unforgeable: an adversary that sees q valid tags and outputs a fresh (m, tag) succeeds with probability at most $\varepsilon + 2^{-\tau}$ per attempt.

Sibuna uses BLAKE3 in keyed mode [13] as F_K. BLAKE3's keyed mode is its compression function with the key in place of the IV, the same structural argument that makes HMAC a PRF under the assumption that the compression function is one [12]; the BLAKE3 specification states the keyed mode as a PRF directly.

Lemma 16 (Tag truncation)
With $\tau = 128$ tag bits, an adversary that submits q forgeries has total success probability at most $q/2^{128} + \varepsilon_{\mathrm{PRF}}$.

Proof of Lemma 16
Each forgery attempt against a fresh message succeeds only by guessing the 128-bit truncation of a value the adversary cannot distinguish from uniform; union bound over q attempts. □

At the observed verification rate of $7.5 \cdot 10^6$ tags per second, $q = 2^{64}$ guesses would take 78 000 years and succeed with probability 2^{-64}. Comparison is performed with `std.crypto.timing_safe.eql`, so no byte-position information leaks through response time.

6.2. Stateless challenge identifiers

Definition 17 (Challenge identifier)
The identifier is base64url $(P \,\|\, \mathrm{Tag}_{K_c}(P))$ where the 36-byte payload P is

$$P = \mathrm{ver}_1 \,\|\, \mathrm{alg}_1 \,\|\, \mathrm{diff}_1 \,\|\, t_1 \,\|\, \mathrm{issued}_8 \,\|\, \mathrm{fp}_8 \,\|\, \mathrm{nonce}_8 \,\|\, \mathrm{rule}_8$$

(subscripts are byte lengths), K_c is the challenge key, and the tag is 16 bytes: 70 URL-safe base64 characters in all.

Lemma 18 (Unpredictability and uniqueness)
With $\mathrm{nonce} = \mathrm{trunc}_{64}(F_{K_c}(\mathrm{ctr} \,\|\, \mathrm{now} \,\|\, \mathrm{fp} \,\|\, \mathrm{nonce}))$ for a process-unique counter, identifiers issued to different requests are distinct with probability $1 - 2^{-64}$ per pair and are unpredictable to any party without K_c.

Proof of Lemma 18
The counter makes the PRF inputs distinct, so outputs are independent uniform 64-bit values under the PRF assumption; distinctness follows from the birthday bound and unpredictability from PRF security. No entropy syscall is needed on the request path. □

Because the payload carries the issue time, difficulty, algorithm, opening count, client fingerprint, and the hash of the policy rule that demanded the challenge, the verifier needs no lookup to know what a submission must satisfy, and a client cannot lower its own difficulty without breaking the tag. Verification order matters for Axiom 1: tag, then expiry, then fingerprint, then the proof. All three cheap checks run before a single proof hash is computed, so garbage submissions cost the server one keyed hash.

Theorem 19 (Work-bounded state)
Let T be the challenge lifetime and $S(\tau)$ the number of entries in the spent set at time τ. Then $S(\tau) \le |\,\{\text{challenges solved and verified in } (\tau - T, \tau]\}\,|$. In particular an adversary cannot add an entry without producing an accepted proof of work.

Proof of Theorem 19
Entries are inserted only by `verifyAndMint` after `checkSolution` succeeds, and only for identifiers whose tag verified, so every entry corresponds to one accepted proof for one honestly issued challenge. An entry expires at issued $+ T \le \tau$, after which it is reclaimable by construction of Part E. Hence live entries are a subset of the accepted proofs of the last T seconds. □

This theorem is the discharge of Axiom 4. In the design it replaces (SID 0002's original coordinator) the store was written at **issuance**, so 30 free requests per second filled a 16 384-entry table in nine minutes and denied service to everyone. Now the table fills only at the rate the adversary pays for admission tokens, which is the rate the puzzle already prices.

6.3. Session tokens

Definition 20 (Token)
A token is $\mathrm{base64url}\,(Q \,\|\, \mathrm{Tag}_{K_t}(Q))$ with a 40-byte payload $Q = \mathrm{ver}_1 \,\|\, \mathrm{alg}_1 \,\|\, \mathrm{bits}_1 \,\|\, 0_5 \,\|\, \mathrm{ts}_8 \,\|\, \mathrm{exp}_8 \,\|\, \mathrm{rule}_8 \,\|\, \mathrm{fp}_8$ and a 16-byte tag: 75 characters. The work level $(\mathrm{alg}, \mathrm{bits})$ is what admission compares against a route's demand (amended 2026-10-01).

The fingerprint binds the token to the client identity the daemon can observe (address and User-Agent, hashed under a third key with length separation so `("ab","c")` and `("a","bc")` differ), which is what defeats the solver-farm pattern in which one strong machine solves puzzles and distributes cookies.

Lemma 21 (Symmetric versus asymmetric tokens)
When the issuer and every verifier share K_t, a MAC token is at least as secure as a signature token against a classical adversary and strictly stronger against a quantum one, and its verification cost is that of one keyed hash rather than one group operation.

Proof of Lemma 21
Unforgeability of both rests on the adversary lacking a secret; for the MAC that secret is a 256-bit key attacked only by Grover-accelerated search (2^{128}), while Ed25519's secret is recoverable from the public key by Shor's algorithm [25] in polynomial time. Cost: measured **134 ns** for the MAC token against **52.8 microseconds** for the Ed25519 token on the reference host. □

Ed25519 tokens remain available (`--token-scheme ed25519`) for the one deployment shape where verifiers must not be able to mint: they are then a deliberate trade of 400 × verification cost and post-quantum security for key separation.

Rust port note (2026-10-15)
The 52.8 microseconds and the factor of 400 are Zig figures. The Rust verifier uses non-adjacent-form digits and decodes the public key once, so it is faster; it accepts exactly the same signatures.

Implementation
`libs/crypto/src/keys.zig` derives four purpose keys from one 32-byte seed by keyed BLAKE3 over a domain string (`sibuna/token/v1`, `sibuna/challenge/v1`, `sibuna/ed25519/v1`, `sibuna/fingerprint/v1`), so a leak of one purpose key reveals nothing about another. The seed comes from `--secret-file` or `SIBUNA_SECRET`, else a fresh random value per process. `libs/crypto/src/token.zig`: `MacToken.mint/verify`, `Token` (Ed25519), `computeFingerprintKeyed`. `libs/challenge/src/coordinator.zig`: `createChallengeWithSpec`, `decode`, `verifyAndMint`, `verifyCookie`.

7. Part D: Rate Limiting by the Generic Cell Rate Algorithm

Definition 22 (GCRA, virtual scheduling form [18])
Fix a rate N per window W. For $N > 0$, let $T = \max(1, \lceil W/N \rceil)$ be the integer emission interval and $\tau = (N-1)T$ the tolerance. Each client keeps one theoretical arrival time TAT, initially 0. An arrival at time t **conforms** iff $\mathrm{TAT} \le t + \tau$; on conformance set $\mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T$, otherwise leave it unchanged and report a retry delay of $\mathrm{TAT} - \tau - t$.

Theorem 23 (Interval bound)
For any client and any interval $(t_1, t_1 + L]$ that begins with a conforming arrival, at most $N + \lfloor L/T \rfloor$ arrivals conform.

Proof of Theorem 23
Number the conforming arrivals in the interval $1, \ldots, k$ at times $t_1 \le \ldots \le t_k$ and write TAT_j for the value after the j-th. Since $\mathrm{TAT}_j = \max(\mathrm{TAT}_{j-1}, t_j) + T \ge \mathrm{TAT}_{j-1} + T$ and $\mathrm{TAT}_1 \ge t_1 + T$, induction gives $\mathrm{TAT}_j \ge t_1 + jT$. Arrival k conformed, so $\mathrm{TAT}_{k-1} \le t_k + \tau \le t_1 + L + \tau$. Combining, $t_1 + (k-1)T \le t_1 + L + \tau$, i.e. $k - 1 \le (L + \tau)/T = L/T + N - 1$, hence $k \le N + L/T$ and, k being an integer, $k \le N + \lfloor L/T \rfloor$. $\square$

Corollary 24 (Burst then pace)
From idle a client may issue N requests at once; thereafter one request conforms per T. A fixed-window counter admits up to $2N$ across a boundary; a sliding log admits exactly N per W but stores N timestamps per client. GCRA stores one integer.

The theorem is deliberately stated as an inequality rather than the marketing phrase "sliding window". SID 0004 described the earlier fixed-window implementation as sliding; the present record corrects that: the guarantee is the token-bucket bound above, which is what NGINX's `limit_req` and Envoy's local limiter also provide, with the difference that Sibuna's state is a 16-byte cell in a lock-striped open-addressed table with lazy reclamation (a cell whose TAT is older than $t - \tau$ is fully drained and free).

Implementation
`libs/store/src/rate_limiter.zig`: `Limits{rate, window_ms}` with `emissionInterval` and `burstTolerance`; `Shard.locate` probes at most 16 slots and reclaims drained cells; `RateLimiter.check` returns `Decision{limited, retry_after_ms, remaining}`, which the server turns into HTTP 429 with `Retry-After`. Measured **4.8 ns** per check.

8. Part E: Robin Hood Hashing for the Spent Set

Definition 25 (Robin Hood open addressing [14])
Keys hash to a home slot in a table of size M and probe linearly. The **distance** of an occupant is the number of probes from its home. On insertion, whenever the incoming key's current distance exceeds the occupant's, the two swap and the displaced key continues probing.

Lemma 26 (Invariant)
After any sequence of insertions, for every slot s with occupant distance d_s and every stored key k whose probe path passes through s (home $\leq s <$ position, cyclically), the distance of k at s is at most d_s.

Proof of Lemma 26
Suppose k passed s with distance $p > d_s$. At the moment k probed s, the occupant's distance was $d' \leq d_s$ (distances of a fixed occupant never change, and later swaps only place occupants with distance at least the displaced one's at that slot). Then $p > d'$ and the insertion rule would have swapped k into s, contradiction. □

Lemma 27 (Early termination)
A lookup for k that stops at the first slot s where s is empty or $d_s <$ (probe distance of k at s) is correct.

Proof of Lemma 27
If k were stored beyond s, its path would pass s with distance exceeding d_s, contradicting the invariant; an empty slot terminates every linear-probing path. □

Lemma 28 (Safe overwrite of expired occupants)
Replacing the occupant of slot s (distance d_s) by an incoming key with distance $d \geq d_s$ preserves the invariant; replacing it with $d < d_s$ may make a stored key unreachable.

Proof of Lemma 28
Every key passing s has distance at most $d_s \leq d$, so the invariant holds with the new occupant. If $d < d_s$, a key k passing s with distance d_s (which the invariant permits) would now terminate early at s and be lost. □

Insertion first simulates the bounded displacement path without mutation and refuses if no safe destination exists. This prevents a failed insertion from losing a live tag. This lemma is why the insert path overwrites an expired entry only when `carry.dist >= e.dist` and otherwise swaps and drops the expired key when it becomes the carried one; both cases reclaim without a sweeper and without ever moving a live key backwards. Expected probe costs depend on load and hash distribution; results for other probing models do not prove a universal variance bound for this linear-probing implementation. Lookup is capped at 64 probes. Insertion preflight can inspect up to one shard and refuses before mutation when it cannot preserve all live keys. Saturation is a correctness case, not something an average-case bound allows us to ignore.

Implementation
`libs/store/src/challenge_store.zig`: 16 shards × 4096 entries of `{tag: [16]u8, expires_at, dist, occupied}`; `Shard.find` implements early termination, `Shard.insert` the swap-and-drop rule; `markSpent` refuses live duplicates with `DoubleSpendAttempt`. Measured **22.8 ns** per spend-and-lookup pair. The ban table `ban_list.zig` uses versioned key/expiry snapshots; readers retry concurrent replacement.

9. Part F: Linear-Time Inspection

9.1. Multi-pattern matching

Theorem 29 (Aho-Corasick [16])
For a pattern set $\mathcal{P}$ of total length m over an alphabet of size σ and a text of length M, the Aho-Corasick automaton is built in $O(m\sigma)$ time and space (dense) and reports whether any pattern occurs in $O(M)$ time, independent of $|\ \mathcal{P}\ |$.

Sibuna uses the dense form: 256 successors of two bytes each per state, 512 bytes per state, failure links folded into the table so that scanning is exactly one load per input byte and never follows a failure chain. Case folding uses a comptime table so the automaton is byte-oriented and branch-free. Every pattern carries a tag (category), so one pass over a field returns both the match and its class. Each automaton capacity is one plus the sum of its pattern lengths; both live inside the `Engine` value so an engine can be rebuilt and swapped as one object (Part G).

Rust port note (2026-10-15)
The Rust automata have one `u32` successor per state and byte class, not 256 two-byte successors per state. Construction is $O(mc)$ for $c \leq \sigma$ classes, and a scan is still one load per byte. See the revision note.

9.2. The semantic layer

Signatures alone are either too loose (a bare `/*` flags every browser's `Accept: */*`, which is how the first WAF blocked every real browser) or too tight. The structural layer adds three single-pass recognisers:

1. A **byte-class scan**: one loop that ORs a 256-entry class table over the field, yielding eight booleans (quote, equals, <, shell separator, percent, NUL, canonicalisation-needed, double space). Every structural detector and the decision to run canonicalisation are gated on these bits, so a field without a quote or an equals sign never enters the SQL tokenizer at all.
2. A **SQL word tokenizer**: one pass extracting alphanumeric words, folding case into a 7-byte buffer, looking each up in a 21-entry keyword list pruned by length, and, on `or`/`and`, checking the tautology grammar `literal = literal` in place. The verdict is a small score: a quote is mandatory, up to two keywords, a comment marker, and an equals sign add one point each, and four points deny. Prose such as *it's a group order from the shop* scores three; `name='x' union all from t--` scores four.
3. Tag/attribute and separator/command recognisers for XSS and command injection that fire only in attribute or separator position.

This is a deliberate middle between two extremes. Regular-expression rule sets (backtracking engines) run hundreds of backtracking engines per request, each its own pass with its own match state, and are neither linear nor allocation-free. Full SQL/HTML parsers are linear but carry parser state, memory for ASTs, and a large surface of grammar edge cases; their precision advantage is real for a general-purpose WAF, but for an edge gate the fingerprint approach pioneered by libinjection captures most of it with a fixed-size tokenizer state.

9.3. Vectorised matchers

Hyperscan [17] and its portable fork Vectorscan reach multi-gigabyte-per-second literal scanning with SIMD prefilters (Teddy, FDR) that test several pattern prefixes per 16- or 32-byte block using byte shuffles, then verify candidates. Three facts decide against them here. First, our inputs are short: a User-Agent is about 120 bytes and the whole set of non-structural headers a few hundred; the automaton scans a User-Agent in **85 ns**, below the cost of one cache miss, so the prefilter's constant setup cost dominates. Second, portable Zig `@Vector` has no runtime byte shuffle, so Teddy needs per-architecture inline assembly and cannot run in the WASM solver. Third, Hyperscan is a multi-megabyte C++ dependency with a compile-time pattern database, which contradicts the single-binary and pure-Zig constraints. Where SIMD would pay, on 8 KB bodies that we scan at about 2.9 ns per byte (**23.7 microseconds**), the body scan is already three orders of magnitude below origin latency, and the body inspection is bounded at 8 KB by design.

Implementation

`libs/policy/src/aho_corasick.zig`: `Automaton(max_states)` with `addPatternTagged`, `build`, `findFirstTagged`. `libs/policy/src/waf.zig`: `buildSignatures`, `scanClasses`, `scanSql`, `tautologyAfter`, `checkSqliStructure`, `checkXssStructure`, `checkRceStructure`, `inspectRequest`; structural headers such as `Accept` are exempt by name. Measured: gate profile (no WAF) full classification **295 ns**; shield profile with the WAF over a seven-header browser request **1.46 microseconds**; naive sequential substring search over the same 40 bot signatures **4.2 microseconds** against the automaton's **85 ns**.

10. Part G: Read-Copy-Update Engine Slots

The policy engine is a fixed-capacity value (rules, both automata, the reputation trie). Dynamic policy from Part I's storage layer must replace it without stopping workers. Sibuna uses the read-copy-update discipline [23] with an explicit reader count per slot.

Definition 30 (Slot protocol)

A slot is a pair (engine, readers). Readers: load the active slot pointer, increment its reader count, reload the pointer, and if it changed decrement and retry. Writers: publish the new slot by an atomic swap, then wait until the old slot's reader count is zero before rebuilding into it.

Lemma 31 (No reader on a rebuilt slot)

A writer never begins rebuilding a slot while a reader that will use it holds it.

Proof of Lemma 31

Consider a reader R and the swap S that retires slot A. If R's increment and successful pointer recheck on A precede S in the sequentially consistent order, the writer's subsequent zero-check observes the increment and waits until R releases. If R's increment follows S, then R's reload of the pointer (which follows its increment) observes the swapped value $B \neq A$, so R decrements and retries; R never uses A. All pointer publication, pointer recheck, counter increments/decrements and writer zero-checks use sequential consistency. In that single total order, either the reader pins and rechecks before retirement (the writer sees its count), or its recheck sees retirement and retries. Acquire/release alone on separate atomics is not sufficient for this argument. □

Rust port note (2026-10-15)

The Rust daemon does not use this slot protocol. It publishes each engine behind an `Arc` and never rebuilds one in place, so no request can read an engine that is being rebuilt or freed. See the revision note.

The cost includes an atomic increment and decrement per classification on a cache-line-aligned counter; contention must be measured. The response copies its rule name before releasing the slot, so origin I/O does not prolong the writer's wait. Incidents flow from workers to the storage thread through a bounded multi-producer single-consumer ring (Vyukov's sequence-stamped design [19]): each slot carries a sequence number that tells producers whether it is free and the consumer whether it is full, so neither side locks, and a full ring drops the newest record rather than blocking a response.

Implementation

`apps/sibuna/src/server.zig`: `EngineSlot`, `AppState.acquireEngine`, `releaseEngine`, `publishEngine`. `libs/store/src/ring.zig`: `BoundedQueue(T, capacity)`. `apps/sibuna/src/persistent.zig`: the storage thread's

`tick` drains the ring, detects policy changes by a change stamp, rebuilds the spare slot from file policy, database policy, and reputation, and publishes it.

11. Part H: Load-Adaptive Difficulty

Definition 32 (Adaptive bump)
Let λ be the exponentially weighted moving average of the challenge issue rate over one-second buckets with smoothing a, and λ_0 the configured baseline. The difficulty bump in bits is $\beta = \min(\beta_{\max}, \lceil \log_2(1 + \lambda/\lambda_0) \rceil)$ when $\lambda > \lambda_0$ and 0 otherwise.

Lemma 33 (Exponential cost growth)
Under the bump, the expected client work per challenge at observed rate λ is at least $2^b(1 + \lambda/\lambda_0)/2$ up to the cap, so the aggregate work an adversary must perform to sustain rate λ grows at least quadratically in λ beyond the baseline.

Proof of Lemma 33
$2^{b+\beta} \geq 2^b \cdot 2^{\log_2(1+\lambda/\lambda_0)-1}$ since $\lceil x \rceil \geq x$ costs at most one bit in the other direction; multiply by λ for aggregate work. □

The cap $\beta_{\max}$ (default 6 bits, 64 ×) bounds what a human on a slow device is ever asked, so a flood degrades the human experience by at most a fixed factor rather than locking humans out. The state is three atomics and needs no lock.

Implementation
`libs/challenge/src/adaptive.zig`: `Adaptive.observe(now_ms)` rolls buckets with a compare-and-swap; `bump()` computes β. Applied in `Coordinator.createChallengeWithSpec`. The fixed-point ratio is rounded upward before the integer ceiling-logarithm; flooring 15/10 previously returned one bit instead of two.

12. Part I: Campaign Clustering by Feature Hashing

Incidents recorded by the WAF are embedded so that the storage layer can group variants of one attack template without a model.

Definition 34 (Trigram feature hashing)
Fold digits to `0` and letters to lowercase. For each byte trigram g of the payload let $h(g)$ be a 64-bit hash; add sign $(h(g)) \in \{-1, +1\}$ (the top bit) to coordinate $h(g) \bmod 64$. Normalise to unit length.

This is the hashing trick of Weinberger et al. [20] over Broder's shingles [21]; the random sign makes the estimator of the inner product unbiased, and the fold makes `id=1'` and `id=42'` share almost all trigrams. Cosine similarity is then a dot product, and sqlite-vec's cosine KNN (the vector table is declared with `distance_metric=cosine`) returns the nearest recorded incident. An incident joins the nearest campaign when the cosine distance is at most 0.35 and otherwise founds a new campaign whose identifier is its own. The storage test in `apps/sibuna/src/persistent.zig` verifies that two SQL injection variants against different paths share a campaign and a honeypot hit does not.

Rust port note (2026-10-15)
The Rust storage searches only the most recent 16,384 embeddings of all nodes, kept in memory, instead of every recorded incident through sqlite-vec (SID 0011). See the revision note.

Implementation
`libs/policy/src/embedding.zig`: `embed`, `cosine`, `toBytes` (64 float32 little-endian); `apps/sibuna/src/persistent.zig`: `nearestCampaign`, `insertIncident`, the `incidents_vec` vec0 table with the `embedding_coarse` bit column Zaxonlite's typed search expects.

13. Verification and Benchmark Gates

`zig build test` runs library, server, solver, storage and live HTTP regressions. The standalone primitive harness records seven batches after untimed warmup and resets mutable state before each batch. Failed insertions are errors, not accepted benchmark samples. Both bot algorithms search the same patterns with any-match semantics. Compiler barriers keep pure calls in their loops without per-operation clocks. The harness does not measure heap allocations; its zero allocation expectation is based on the inspected APIs and sources.

See `benchmarks/results/latest.json` for primitive measurements and exact `@sizeOf` values, and `benchmarks/results/distributed-latest.json` for three-process loopback measurements. Older numerical examples in this paper describe the pre-review build and are illustrative, not current performance claims. Unsupported fixed competitor cost rows were removed.

The external distributed harness checks session sharing, authenticated WAF denial, issuer-bound challenge rejection, ban propagation, and HTTP service with one member down. It does not establish WAN latency, production mTLS performance, global quotas or durable replay protection.

14. Security Considerations

- **Trust boundary for client identity.** The fingerprint uses the peer address unless `--trust-forwarded` is set, in which case `X-Forwarded-For` is trusted; that flag must be set only when an ingress that overwrites the header is the sole peer, as in forward-auth mode where it defaults on.
- **Secret handling.** Without `--secret-file` or `SIBUNA_SECRET` the seed is random per process: tokens and challenges do not survive restarts and are not shared across nodes. The banner says so. A cluster must share one seed.
- **Replay.** Challenges are single-use within a process lifetime. Cluster node ids derive distinct challenge keys, so verification must return to the issuing node; tokens remain cluster-wide. Spent state is not durable across restarts; tokens are bound to the fingerprint and expire; both carry the issue time so clock skew beyond sixty seconds forward is rejected.
- **Amplification.** Every endpoint is bounded: heads over 16 KB are refused, bodies over 64 KB are refused for the verify endpoint, proofs are length-checked before any hash, and the expensive verification is reached only after the constant-cost tag, expiry, and fingerprint checks.
- **Side channels.** Tag comparison is constant-time. Hashing time is independent of secrets. Response codes distinguish malformed, expired, mismatched, and invalid submissions for the honest client's benefit; none of these outcomes depends on secret material beyond the tag verdict, which is uniform.
- **WAF false negatives.** The structural layer is a fingerprint classifier, not a parser; it will miss injections that avoid keywords, quotes, and separators entirely. Operators needing parser-grade coverage should treat Sibuna's shield profile as the first layer and keep an application-level defence.

15. Open Problems

1. **Memory-hard and asymmetric with proof.** No construction we know is simultaneously memory-hard for the prover, cheap for the verifier, non-parallelisable, and proven in the quantum random oracle

model. Depth-robust graph labelling with sequential openings is the natural candidate; without a proof it stays out of the daemon.
2. **JA4H fingerprints** as a policy signal (not as a token binding, because browsers vary header sets between navigation and fetch).
3. **HTTP/2 and TLS termination** in the reverse-proxy mode; forward-auth mode already delegates both to the ingress.
4. **Formal proof of the RCU slot protocol** in a model checker; the argument in Part G is a paper proof.

16. References

1. A. Back. *Hashcash: A Denial of Service Counter-Measure.* Technical report, 2002 (announced 1997).
2. C. Dwork and M. Naor. *Pricing via Processing or Combatting Junk Mail.* CRYPTO 1992.
3. A. Juels and J. Brainard. *Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks.* NDSS 1999.
4. B. Cohen and K. Pietrzak. *Simple Proofs of Sequential Work.* EUROCRYPT 2018.
5. M. Mahmoody, T. Moran, and S. Vadhan. *Publicly Verifiable Proofs of Sequential Work.* ITCS 2013.
6. J. Blocki, S. Lee, and S. Zhou. *On the Security of Proofs of Sequential Work in a Post-Quantum World.* ITC 2021.
7. A. Biryukov and D. Khovratovich. *Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem.* NDSS 2016.
8. D. Wagner. *A Generalized Birthday Problem.* CRYPTO 2002.
9. L. Grassi, M. Naya-Plasencia, and A. Schrottenloher. *Quantum Algorithms for the k-XOR Problem.* ASIACRYPT 2018.
10. A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications.* RFC 9106, 2021.
11. M. Bellare, R. Canetti, and H. Krawczyk. *Keying Hash Functions for Message Authentication.* CRYPTO 1996.
12. M. Bellare. *New Proofs for NMAC and HMAC: Security without Collision-Resistance.* CRYPTO 2006.
13. J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn. *BLAKE3: One Function, Fast Everywhere.* Specification, 2020.
14. P. Celis. *Robin Hood Hashing.* PhD thesis, University of Waterloo, 1986.
15. L. Devroye, P. Morin, and A. Viola. *On Worst-Case Robin Hood Hashing.* SIAM Journal on Computing 33(4), 2004.
16. A. V. Aho and M. J. Corasick. *Efficient String Matching: An Aid to Bibliographic Search.* Communications of the ACM 18(6), 1975.
17. X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu. *Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs.* NSDI 2019.
18. ATM Forum. *Traffic Management Specification Version 4.0*, af-tm-0056.000, 1996 (the Generic Cell Rate Algorithm).
19. D. Vyukov. *Bounded MPMC Queue.* 1024cores.net, 2010 (sequence-stamped ring used here in its MPSC form).
20. K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. *Feature Hashing for Large Scale Multitask Learning.* ICML 2009.
21. A. Z. Broder. *On the Resemblance and Containment of Documents.* SEQUENCES 1997.
22. M. Charikar. *Similarity Estimation Techniques from Rounding Algorithms.* STOC 2002.
23. P. E. McKenney and J. D. Slingwine. *Read-Copy Update: Using Execution History to Solve Concurrency Problems.* PDCS 1998; P. E. McKenney et al., *Read-Copy Update*, Ottawa Linux Symposium 2001.
24. L. K. Grover. *A Fast Quantum Mechanical Algorithm for Database Search.* STOC 1996.
25. P. W. Shor. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.* SIAM Journal on Computing 26(5), 1997.

Lifecycle event

> 2026-10-15: prediscussion -> discussion. Ported from the Zig repository's
> Typst record

Lifecycle event

> 2026-10-15: discussion -> published. State carried over from the Zig
> repository's registry