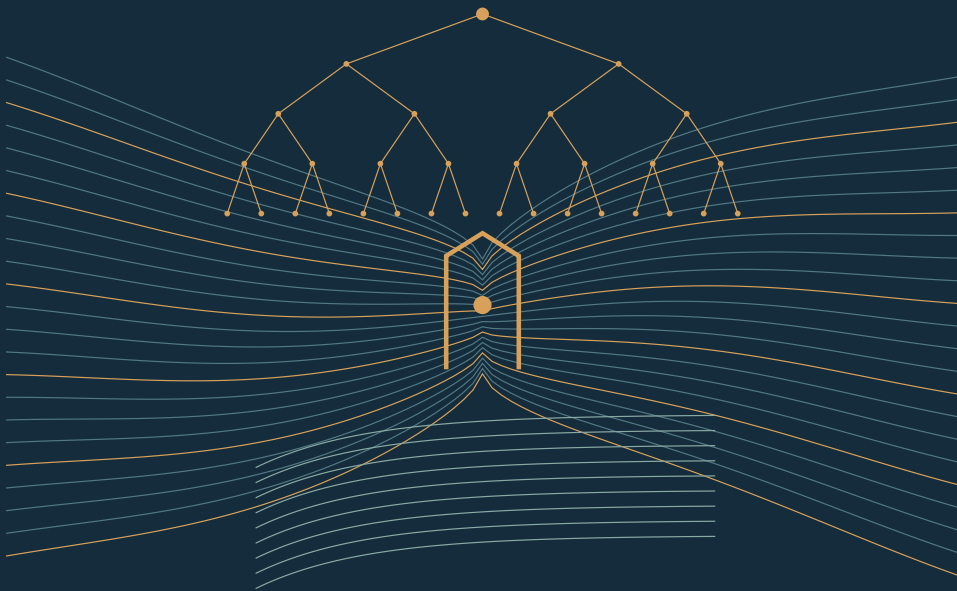


THE BOOK OF

SIBUNA

The mathematics and engineering of web defense



WORK · TRUST · BOUNDED STATE

Vikrant Rathore

With assistance from Ronak Rathore

Preface

A request arrives with a method, a path, some headers, and a claim to the server's attention. None of those bytes establishes that serving it is worthwhile. Sibuna places a decision between the request and the origin: admit it, refuse it, or ask its sender to do verifiable work. This book follows that decision from its mathematical model to the bytes in memory.

The system has two surfaces. **Gate** provides proof-of-work admission and local flood controls. **Shield** adds application inspection and policy decisions. Either can use replicated storage for policy, reputation, and incident history; we call that deployment **Edge**. A valid session establishes admission; it does not exempt the request from inspection. Replication distributes durable state; it does not turn a local rate counter into a global quota.

The central engineering question is not whether a hash is fast. It is whether the complete path remains bounded when an untrusted client chooses the input. How many bytes can a parser inspect? How much memory can a challenge consume? How many connections can a client hold open? Which writes may be retried? What remains available after a leader disappears? Each answer must name both an invariant and its boundary.

The mathematics serves the same purpose. An expected cost is not a latency guarantee. A sampling argument for a fixed commitment is not a proof against every adaptive prover. A queue absorbs a burst; it cannot compensate for a permanently slower consumer. We derive small models, work examples by hand, and then ask where the implementation departs from them.

The book is written to be used three ways. Read in order, Parts I to VII are a course: each chapter opens with what you should be able to do afterwards, works an example by hand, and closes with an exercise and an invariant to explain in your own words; selected solutions are at the back. Parts VIII to X are the reference: measurements with their provenance, every flag and topology, every endpoint, status, error, and table. Part XI is a two-page card for the terminal. A glossary and a bibliography close the book.

Part II places Sibuna beside Anubis, SafeLine, and the Cloudflare WAF, feature by feature, and Part VIII measures Sibuna and Anubis as whole products under the same load generator. Both are written so the reader can check the claims: the comparison names its sources and the measurements are rendered from committed result files.

The reader should know basic programming, logarithms, and conditional probability. Zig is introduced through ownership and data layout rather than a language survey. The chapters can be read in order: the cost model motivates the protocol, the protocol determines the state, and the state determines the concurrency and storage design. Later chapters turn those invariants into measurements and operating procedures.

This is an implementation book, not a claim that computational puzzles eliminate automated traffic. Clients can buy compute, addresses can be shared, and application syntax is richer than a bounded detector. The useful result is a system whose costs and limitations can be examined, tested, and changed without hiding them behind a slogan.

Contents

Preface	2
How to Read This Book	5
1 The Cost of a Request	7
1.1 Admission Is an Economic Decision	7
1.2 What a Proof of Work Establishes	7
1.3 Hash Search as a Random Variable	8
1.4 Sessions Amortise Work	8
1.5 Bounded State Is a Second Budget	9
1.6 The Invariants We Will Carry Forward	9
2 Prior Art and the Design Space	10
2.1 Where Sibuna Comes From	10
2.2 Three Ideas in One Binary	11
2.3 Feature Comparison	12
3 Foundations from Cryptography Research	15
3.1 Hashcash: The Mathematics of Tier One	15
3.2 Proof of Sequential Work: Tier Two	16
3.3 Symmetric Authentication: Challenges and Sessions	19
4 The Sibuna Protocol	21
4.1 The Challenge Round Trip	21
4.2 The Stateless Challenge Record	22
4.3 Verification Order and Single Use	23
4.4 The Session Cookie	24
5 Deterministic Memory and Zero-Allocation Engineering	26
5.1 The Connection Loop	26
5.2 Concurrent State Without Allocation	28
6 Automata, Tries, and the Semantic Firewall	33
6.1 Single-Pass Multi-Pattern Matching	33
6.2 The Radix Trie for IPv4 and IPv6	34
6.3 The Semantic Firewall	35
6.4 The Declarative Policy Engine	36
7 The Browser Proof-of-Work Engine	38
7.1 One Source, Two Targets	38
7.2 The Worker Protocol	39
7.3 Calibration	40
7.4 The Interstitial	40
8 Empirical Evaluation	41
8.1 Methodology	41
8.2 Primitive Latencies	42
8.3 Whole-Product Comparison	44
8.4 Admission Comparison	48
8.5 A Cluster of Three	48
8.6 Distributed Measurement	50
9 Operations and Deployment	51
9.1 Surfaces and Configuration	51
9.2 Console Preview	53
9.3 Deployment Topologies	65
9.4 Declarative Policy	68
9.5 Persistent Storage and Clustering	68
9.6 Packaging	71
10 Desk Reference and Diagnostics	73
10.1 Endpoints	73
10.2 Metrics	74
10.3 Status Codes	74
10.4 Error Catalog	75
10.5 Policy Schema	76
10.6 Storage Tables	76
10.7 Build Targets	76
11 Quick Reference Card	78
Selected Solutions	81
11.1 Part I: Cost and Probability	81
11.2 Part II: Prior Art	81
11.3 Part III: Cryptography	81
11.4 Part IV: Protocol	82

11.5 Part V: State and Ordering	82
11.6 Part VI: Algorithms	82
11.7 Part VII: Browser Engine	82
11.8 Part VIII: Evaluation	83
11.9 Part IX: Operations	83
11.10 Part X: Reference	83
11.11 Continue the Investigation	83
Glossary	84
Bibliography	86

How to Read This Book

Begin with a single question: what is the cheapest safe decision the server can make now? Keep that question beside you through the protocol and implementation chapters. A cheap operation that admits the wrong request is a defect; a correct operation with unbounded cost is another kind of defect.

Question	Where the answer develops
What does a puzzle buy?	Parts I–III: cost, probability, and the limits of proof.
How does Sibuna compare?	Part II: lineage and the feature table; Part VIII: measurements.
What does a session mean?	Part IV: bindings, verification order, expiry, and replay.
Where does the memory go?	Parts V–VI: buffers, threads, tables, automata, and publication.
How does the browser solve?	Part VII: worker messages, memory, and calibration.
How do we know it works?	Part VIII: four harnesses and what each excludes.
How do I run it?	Part IX: flags, topologies, policy, storage, clustering, packaging.
What was that error?	Part X: endpoints, statuses, errors, schema; Part XI: the card.

0.0.1 Three Reading Paths

- **Learning the design.** Parts I–IV in order, then the worked trace in Part VI, then Part VII. Do the exercises; compare with the solutions at the back only afterwards. Return to Part V when a trace reaches a shared table or a borrowed slice.
- **Operating a deployment.** Part IX, then Part XI, with Part X open for lookups. Read the "Feature Comparison" in Part II before choosing a surface, and the "Whole-Product Comparison" in Part VIII before choosing worker and connection limits.
- **Changing the code.** Parts V and VI, the source anchors in every chapter, and the SID records they cite. Part VIII explains which harness will catch a regression in what you touched; Part II lists the constraints a change must keep.

0.0.2 Conventions

A **worked example** shows the intermediate states, not just the answer. An **exercise** asks you to change one assumption. Hints suggest a first step; selected solutions at the end of the book make the reasoning checkable. Diagrams distinguish the request path from background work. A source anchor names the implementation to inspect when prose and code appear to

disagree. Boxes headed "Implementation" name a function and its file; boxes headed "Explain the invariant" ask you to teach the idea back.

In equations, b is Hashcash difficulty in bits, n is sequential-work depth, t is the number of openings, p is success probability per trial, K is the number of trials, T is a rate limiter's emission interval, τ its burst tolerance, and Q is a queue capacity. A symbol is local to its section unless stated otherwise. Nanoseconds and requests per second in Part VIII are measurements from a named run. Numbers in worked examples are chosen inputs, not benchmark claims. Statistical models state their assumptions before drawing conclusions.

1 The Cost of a Request

Before choosing an algorithm, identify the resource it is meant to protect. This chapter builds the small models that every later design decision is measured against: origin work, puzzle work, session amortisation, and the memory an untrusted client can make a server hold.

In this chapter By the end of this chapter, you should be able to write the inequality that decides whether a gate pays for itself, state what a proof of work does and does not establish, derive the expectation and tail of hash search, explain how sessions amortise work, and list the five invariants the rest of the book maintains.

1.1 Admission Is an Economic Decision

Suppose an origin performs a database lookup and renders a page for each admitted request. Let c_o be that work, c_g the gate's work per request, and r the arrival rate. With no gate, the origin must sustain rc_o units of work per second. A gate that admits fraction a changes that demand to rac_o , while spending rc_g itself. The gate is useful only when the saved origin work exceeds its own cost and the cost it imposes on legitimate visitors.

$$rc_g + rac_o < rc_o \Leftrightarrow c_g < (1 - a)c_o.$$

This is a capacity model, not a measurement. It omits bandwidth, connection state, storage, and client delay. Its value is that every missing term is visible. A cache hit may make c_o small; a costly query may make it large. There is no universal requester-to-server cost ratio.

Worked example: when a gate pays for itself Choose $c_o = 100$ work units and $c_g = 1$. If half the requests are rejected, the gated system spends $1 + 0.5 \times 100 = 51$ units per arrival instead of 100. If only one in a thousand is rejected, it spends 100.9: the gate costs more than it saves in this model. These selected numbers are not timings. They show why the workload belongs in every performance claim.

1.2 What a Proof of Work Establishes

A client puzzle establishes that somebody found an input satisfying a public verification rule. It does not establish humanity, identity, or good intent. A requester can rent compute, reuse a valid session within its lifetime, or distribute work across machines. The puzzle changes the admission cost; policy and inspection still decide what admitted requests may do.

Address-based limits are complementary. They constrain one address, but shared addresses can represent many people and one requester can use many addresses. Neither a puzzle nor an address is an identity oracle. Sibuna therefore keeps admission, inspection, and reputation as distinct decisions.

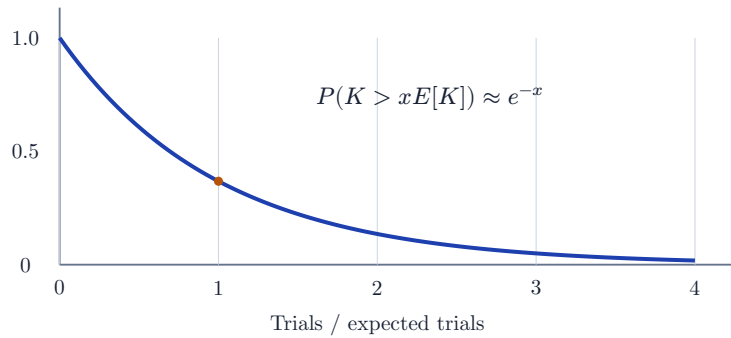


Figure 1: Survival probability of hash search. The horizontal axis is trials divided by expected trials; the continuous curves use the large-work approximation $P(K > \frac{x}{p}) \approx e^{-x}$.

1.3 Hash Search as a Random Variable

For an ideal 256-bit digest, requiring b leading zero bits gives success probability $p = 2^{-b}$ for each independent trial. Let K count trials through the first success. Then

$$P(K = k) = (1 - p)^{k-1}p, \quad P(K > k) = (1 - p)^k, \quad E[K] = \frac{1}{p} = 2^b.$$

The expectation follows from the tail sum:

$$E[K] = \sum_{k=0}^{\infty} P(K > k) = \sum_{k=0}^{\infty} (1 - p)^k = \frac{1}{p}.$$

A difficulty step doubles expected trials. It does not promise that every puzzle takes twice as long. Some succeed on the first trial; some take far longer than the mean. A client that measures only one solve has mostly measured this randomness.

Worked example: expectation is not a deadline At $b = 16$, the expected work is 65,536 hashes. The probability of still searching after that many trials is approximately $e^{-1} = 0.368$. The 95th-percentile trial count is $\left\lceil \frac{\ln(0.05)}{\ln(1-2^{-16})} \right\rceil \approx 196327$, nearly three times the mean. A user interface should tolerate that spread rather than announcing failure at the expected completion time.

1.4 Sessions Amortise Work

Suppose a session permits m requests before expiry, a puzzle costs c_p , verification costs c_v , and a session check costs c_s . Ignoring unsuccessful attempts, the amortized gate cost per request is $\frac{c_v}{m} + c_s$, while the client's puzzle cost is $\frac{c_p}{m}$. Increasing the session lifetime helps people and automated clients alike. Choosing it is a policy decision, not a cryptographic optimization.

Exercise 1.1. A client gets 100 requests per session and performs a puzzle costing one million trials. What is the amortized work per request? What changes if it shares the session with a second process?

Hint: Distinguish the accounting model from the token's actual bindings.

1.5 Bounded State Is a Second Budget

Moving work off the request path does not make it disappear. Let incidents arrive at rate λ , let the storage thread persist them at average rate μ , and let the queue hold Q records. When $\lambda > \mu$, a fluid approximation gives

$$t_{\text{fill}} \approx \frac{Q}{\lambda - \mu}.$$

A larger queue buys time; it does not create throughput. Batching amortizes transaction cost, but retained records still need memory and eventual service. Under overload Sibuna drops new incident records and counts them. It continues making request decisions from in-memory state. The operator must monitor the lost evidence as well as the HTTP success rate.

Exercise 1.2. A queue holds 512 records. Arrival rate is 1,000/s and drain rate is 600/s. How long can an initially empty queue absorb the excess? Why is the answer only an approximation?

Hint: Use the difference of rates, then consider bursts and batch commits.

1.6 The Invariants We Will Carry Forward

1. Untrusted requests cannot create unbounded server state.
2. A valid admission token does not bypass an application denial.
3. A buffer remains alive for every slice borrowed from it.
4. A durable retry cannot multiply an incident or its reputation effect.
5. A published policy snapshot is immutable until its last reader releases it.

The rest of the book derives the mechanisms that make these statements true, and the tests that would reveal a violation. A fast path is useful only while those statements remain true.

Explain the invariant. State the gate inequality $c_g < (1 - a)c_o$ in words, then explain why a measured verification latency alone cannot tell an operator whether the gate is worth running.

2 Prior Art and the Design Space

We place Sibuna among the systems it learned from: client puzzles and sequential work from the cryptography literature, signature and semantic application firewalls, proof-of-work interstitials, and hosted edge networks. We then compare the four tools an operator is most likely to weigh against each other, feature by feature, and record which facts were checked.

2.1 Where Sibuna Comes From

In this chapter By the end of this chapter, you should be able to trace the lineage of client puzzles from 1992 to the sequential-work constructions of 2018, name the three architectural ideas Sibuna combines, place Sibuna, Anubis, SafeLine, and the Cloudflare WAF in one design space, and say which comparison claims were verified and which were only read from documentation.

2.1.1 Client Puzzles and Sequential Work

Dwork and Naor's "Pricing via Processing" (CRYPTO 1992) proposed that a service require a moderately hard, easily checked computation before doing work for a requester. Back's Hashcash (1997, written up in 2002) made the puzzle a partial hash inversion whose difficulty is one integer, and Juels and Brainard's client puzzles (NDSS 1999) applied the idea to connection floods, where the server must not remember anything about an unsolved puzzle. All three share the shape Sibuna keeps: a server-chosen statement, a solution whose cost is tunable, and verification that is orders of magnitude cheaper than solving.

Hash search is embarrassingly parallel, so a requester with more cores finishes sooner. Mahmoody, Moran, and Vadhan (CRYPTO 2013) defined proofs of **sequential** work, in which the prover must perform a long chain of dependent steps whatever its core count, and Cohen and Pietrzak (EUROCRYPT 2018) gave the simple hash-graph construction whose verifier needs only $O(t \log N)$ work. Blocki, Lee, and Zhou (ITC 2021) proved the same construction sound against quantum provers in the quantum random-oracle model. Part III derives Sibuna's Tier Two from that line of work; SID 0006 records the assumptions.

2.1.2 Application Firewalls

The first generation of web application firewalls matched request bytes against regular expressions. ModSecurity (2002) and the OWASP Core Rule Set made that approach open and widely deployed, and also made its costs familiar: hundreds of backtracking patterns per request, anomaly scores tuned by hand, and a long tail of false positives on ordinary input. libinjection (Hanson, 2012) replaced pattern lists with a tokenizer that asks whether a string **parses** as SQL, cutting both cost and false positives. Chaitin's SafeLine carries that semantic idea into a full product, and Hyperscan (Wang et al., NSDI 2019) showed how far vectorised literal matching can be pushed when inputs are long. Part VI explains why Sibuna chose a dense automaton plus small structural tokenizers over either regular expressions or a SIMD engine for the short fields it inspects.

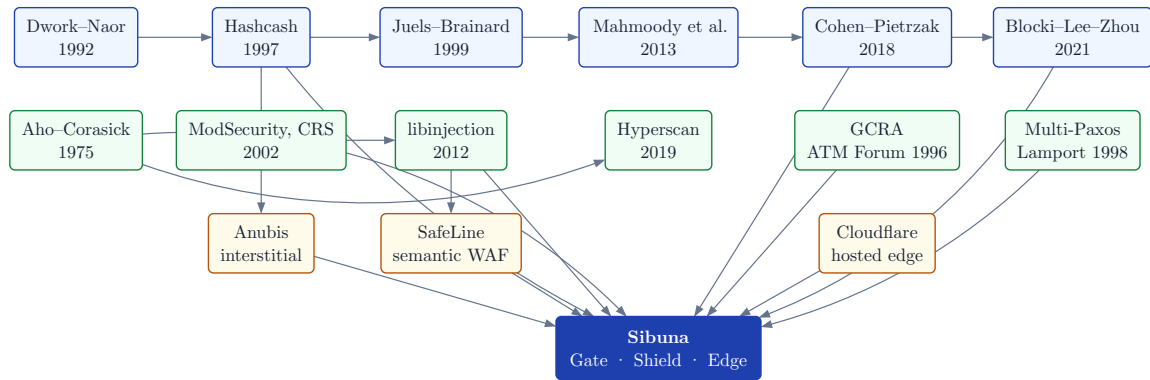


Figure 2: Lineage of the ideas Sibuna combines. Solid arrows are direct descent; the shaded row is the system layer where Sibuna sits.

2.1.3 Interstitials and Edges

The proof-of-work interstitial in front of a website became common in 2025, when scraper traffic feeding language-model training made ordinary origins unaffordable to run. Anubis (Techaro) is the best-known open implementation: a Go reverse proxy that serves a JavaScript puzzle, signs a session token, and otherwise passes traffic through. Sibuna's Gate surface solves the same problem with a stateless challenge, a symmetric token, and a second work tier; the comparison below records where the two differ.

At the other end of the scale, hosted edges such as Cloudflare place inspection, rate limiting, and bot scoring in hundreds of points of presence with a global control plane. Sibuna's Edge deployment borrows the shape (a replicated policy and reputation plane feeding local, immutable snapshots) while staying self-hosted, which is why the storage layer is an embedded consensus database rather than a service call.

2.2 Three Ideas in One Binary

Sibuna combines a browser work challenge for admission, bounded application-payload inspection, and a replicated control plane that publishes local policy snapshots. Gate and Shield are the two product surfaces; storage and distributed deployment are options for either.

Property	Gate	Shield
Admission	Native Hashcash or sequential-work verification	Same
Session	Keyed BLAKE3; optional Ed25519	Same
Local controls	Rules, CIDRs, GCRA, bans	Same
Inspection	Disabled	Bounded signatures and structural tokenizers
State	Optional replicated policy and reputation	Also inspection forensics

The browser solver compiles from the same proof sources as the native verifier. Dense literal matching avoids backtracking and keeps scan work linear in input length. Immutable policy snapshots isolate request handling from SQL and consensus. None of these choices establishes universal algorithmic optimality: short headers, long bodies, pattern diversity, core count and cache size change the tradeoffs.

Design constraints adopted

1. No heap allocation on the request path; every table has a fixed capacity and a stated saturation behaviour.
2. Issuing a challenge writes nothing; only a paid-for solution occupies memory.
3. A session admits; it never exempts a request from inspection or an explicit denial.
4. Every claimed number is either measured by a committed harness or labelled as a model.
5. Cryptographic choices cite a venue and a proof, and state their quantum posture.

2.3 Feature Comparison

The table compares Sibuna with the three tools most often mentioned alongside it. Facts about the other products were read from their public documentation and release artefacts in September 2026: Anubis 1.27.0 (MIT licence, Go), SafeLine community edition 9.x (GPL-3.0 management code around closed detector images, Docker Compose), and the Cloudflare WAF developer documentation. Anubis was also run on the benchmark host; SafeLine and Cloudflare were not (Part VIII explains why). A dash means the product does not offer the feature; a plan name means the feature exists but only on that plan.

Feature	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Runs as	One static binary; reverse proxy or forward auth; optional embedded database	One Go binary; reverse proxy or forward auth	Seven Docker containers (tengine, detector, mgt, luigi, fvm, chaos, PostgreSQL)	Hosted network; DNS points at Cloudflare
Language, runtime	Zig, no garbage collector, no libc when storage is off	Go, garbage collected	C/C++ proxy and detector, Go management, PostgreSQL	Proprietary edge software
Licence	Source in the repository	MIT	GPL-3.0 management code; closed detector images	Proprietary service
Host footprint	33 MiB binary; 10.2 MiB idle RSS; Linux fixture, console off	38.1 MiB binary; 21.2 MiB idle RSS; Linux fixture, console off	1 CPU core, 1 GB RAM, 5 GB disk minimum, Docker 20.10+	None on premises
Proof-of-work admission	Hashcash (bit-level) and Cohen-Pietrzak sequential work, chosen per rule; WebAssembly with JavaScript fallback	SHA-256 Hashcash in hex nibbles (fast , slow); non-work metarefresh and preact challenges	JavaScript anti-bot challenge and CAPTCHA; not work-bound	Managed challenge, JS challenge, Turnstile; Bot Fight Mode issues a compute challenge
Challenge state on the server	None until solved; solved tags in a fixed Robin Hood table	Store backend: memory, bbolt, Valkey, or S3	Managed by the stack	Managed by Cloudflare

Feature	Sibuna	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Session token	Keyed BLAKE3 tag, 75-character cookie bound to the work paid; Ed25519 optional	JSON Web Token signed with Ed25519; HS512 optional	Cookie issued by the stack	<code>cf_clearance</code> cookie
Post-quantum posture of the default token	Symmetric: only Grover's quadratic speedup applies	Ed25519: broken by Shor's algorithm	Not documented	Not documented
Policy language	JSON rules: path, User-Agent, headers, IPv4/IPv6 CIDR; ALLOW, DENY, CHALLENGE, WEIGH with thresholds; dynamic rules from the database	YAML/JSON bots: regex on UA, path, headers; CIDR; CEL expressions; WEIGH with thresholds	Console-defined custom rules, ACLs, IP groups	Rules language expressions; 5 / 20 / 100 / 1000 <i>custom rules by plan</i>
SQLi, XSS, RCE, traversal inspection	Shield: tagged automaton plus structural tokenizers, first 8 KB of body	—	Semantic analysis engine over OWASP categories, also SSRF, XXE, CRLF	Managed rulesets (<i>Pro and above</i>) OWASP Core Ruleset; attack score (<i>Business and above</i>)
Rate limiting	GCRA per client, local to a node	— (system load is exposed to CEL rules)	Per IP, path, session	1 / 2 / 5 / 100 <i>rules by plan</i> IP-only keys below Enterprise
Bans and reputation	Honeypot bans; reputation trie replicated across the cluster	DNSBL; ASN and GeoIP through Thoth (<i>paid, closed</i>)	IP groups and blacklists; threat intelligence (<i>Pro</i>)	IP lists; managed lists and bot score (<i>Enterprise</i>)
Forensics	Embedded SQLite: full-text search over incidents, vector campaign clustering	Prometheus metrics only	PostgreSQL attack log with console	Security Events; sampled on Free
Multi-node	Multi-Paxos replication of policy and reputation; shared seed; sticky challenge verification	Shared signing key; shared Valkey store	One stack per host	Global anycast
Metrics	Prometheus at <code>/__sibuna/metrics</code>	Prometheus on a separate port	Console dashboards	Dashboard and analytics
TLS termination	— (ingress)	— (ingress)	Yes	Yes
Management console	Opt-in; SID 0007 Committed	—	Yes	Yes
Measured on the benchmark host	Yes	Yes	— (Docker only)	— (hosted)

Three differences carry most of the weight in a selection decision.

- **Where the work goes.** Anubis and Sibuna Gate make the **client** pay before the origin does anything. SafeLine and Cloudflare inspect first and challenge selectively; their default posture is to admit and filter. Sibuna Shield does both, in that order: inspection can deny a request that already holds a valid session.
- **What the server remembers.** Anubis keeps challenge state in a store (memory, bbolt, Valkey, or S3) and signs a JSON Web Token with Ed25519; Sibuna issues a self-authenticating record and remembers only solved tags in a fixed table. SafeLine keeps attack logs in PostgreSQL; Cloudflare keeps everything, with sampled visibility on the Free plan.
- **What you have to run.** Sibuna is one static binary with an optional embedded database; Anubis is one Go binary; SafeLine is seven containers behind a Docker daemon on a Linux

host with at least one core, one gigabyte of memory, and five gigabytes of disk; Cloudflare is a DNS change and a subscription.

Current product boundary Sibuna leaves ingress TLS termination to the deployment proxy and does not score bots with a trained model or publish paid signatures. Its opt-in console provides authenticated dashboards, country enrichment, policy editing, investigation and cluster management. Country lookup and storage work run outside request classification. SID 0007 is Committed; functional verification is complete. The fresh container measurements do not establish the console's strict performance isolation target; Part VIII records that distinction.

Exercise 2.1. Explain why compiling one verifier source to native code and WebAssembly reduces protocol drift. Which properties still require independent testing or cryptographic review?

Exercise 2.2. Using only the comparison table, write down the smallest deployment that gives a static site (a) a proof-of-work gate, (b) SQL-injection inspection, and (c) a cluster-wide ban. Which rows forced each choice?

Hint: Start from the "runs as" and "multi-node" rows.

Explain the invariant. Describe how Gate, Shield, and optional replicated storage divide admission, inspection and policy distribution. Then name one row of the comparison table you would want to re-verify before quoting it to a customer, and say how you would verify it.

3 Foundations from Cryptography Research

We derive the cost models behind Sibuna's two proof-of-work tiers, present the Cohen–Pietrzak Proof of Sequential Work with its security argument, and justify keyed-hash authentication for challenges and sessions. SID 0006 records the assumptions and references.

3.1 Hashcash: The Mathematics of Tier One

In this chapter By the end of this chapter, you should be able to state the expected work, variance, and tail probability of a bit-level Hashcash puzzle, explain why the server's verification cost bounds the attacker's advantage, and describe the two limits that motivate a second tier.

3.1.1 Definition and Cost

Sibuna's Tier One puzzle for challenge string C and difficulty b bits is: find $N \in \mathbb{N}$ such that $\text{SHA-256}(C \parallel \text{dec}(N))$ has b leading zero bits. The decimal rendering of the nonce is deliberate; it keeps the wire format identical between the WebAssembly solver, the JavaScript fallback, and the native verifier without a binary encoding step.

The unit of difficulty is an expected **hash trial**, not a compression function call. An input can occupy several SHA-256 blocks. Prefix precomputation can also change the work per nonce. The wire rule remains the digest predicate; implementations must agree on the statement and nonce encoding, not on an estimated CPU cost.

For independent ideal digests, the trial count X is geometric with $p = 2^{-b}$. Its median is $\left\lceil \frac{\ln(\frac{1}{2})}{\ln(1-p)} \right\rceil$, approximately $2^b \ln 2$ for small p , and its variance is $\frac{1-p}{p^2}$. The verifier checks one candidate digest rather than performing the search. Parsing, challenge authentication, fingerprint checks, spent-state insertion, and the response are additional work. A primitive timing cannot be inverted into a server's flood capacity.

3.1.2 Two Limits of Search

The first is variance: a long search is not necessarily a broken worker. The second is parallelism: independent nonce trials can run concurrently. More cores reduce elapsed search time even when the expected aggregate trial count remains the same. Hardware throughput therefore matters to the admission policy. Neither a claimed hash rate nor a single benchmark can bound every requester's advantage.

Exercise 3.0. Using the geometric tail, estimate the probability of needing more than 4.6×2^b trials. Why is the expected trial count unchanged by dividing the nonce space among several workers?

Hint: Count aggregate trials, not elapsed seconds.

3.2 Proof of Sequential Work: Tier Two

In this chapter Understand the Cohen–Pietrzak labelling, why it forces sequential computation, what the verifier checks, the conditional sampling bound $(1 - \alpha)^t$ and its limitations, the post-quantum result, the prover's memory bound, and the calibration data behind the work-bit mapping.

3.2.1 Why Sequential Work

A sequential-work construction introduces dependencies between computations: a later label needs a value produced earlier. This addresses a different quantity from aggregate work, namely the depth of the computation. Independent puzzles can still run in parallel, and hardware hash latency still matters.

Sibuna follows the hash-based construction described by Cohen and Pietrzak. Its graph has $2^{n+1} - 1$ labels at depth n , but a fixed label count is not a deterministic browser latency: labels can span several compression blocks, openings may recompute subtrees, and scheduling adds variation. Security statements from oracle-model analyses must not be turned into an unqualified claim about every concrete implementation or adversary.

3.2.2 The Construction

Let n be the depth and consider the complete binary tree with nodes named by binary strings of length at most n (root ε , children $v0$ and $v1$, leaves of length n). Each node v carries a 32-byte label $\ell_v = H(\chi, v, \text{parents}(v))$, where χ is the challenge statement and H is SHA-256 with χ absorbed once (the same prefix trick as the Hashcash solver):

- an internal node hashes its two children: $\ell_v = H(\chi, v, \ell_{v0}, \ell_{v1})$;
- a leaf $u = u_1 \dots u_n$ hashes the **left siblings of its ancestors** wherever the root path turned right: $\ell_u = H(\chi, u, \{\ell_{u_1 \dots u_{i-1}0} : u_i = 1\})$.

The left-sibling edges are what make the computation sequential: leaf u cannot be labelled until every subtree to its left is complete, so a depth-first post-order traversal is forced and the root label $\varphi = \ell_\varepsilon$ depends on all $2^{n+1} - 1$ labels in sequence.

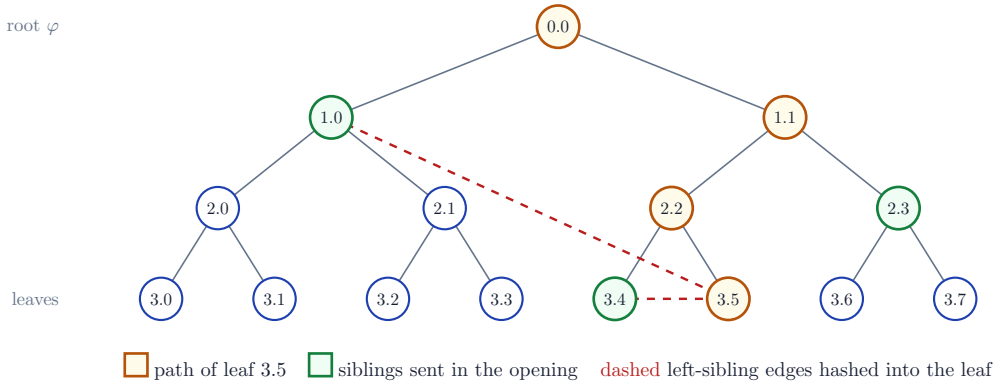


Figure 3: The depth-3 tree: an opening of leaf 3.5 sends the leaf label and the three sibling labels; the dashed edges are the left-sibling dependencies that make labelling sequential

Openings. After computing φ , the prover derives t leaves by Fiat–Shamir, $\gamma_i = H(\chi \parallel \varphi \parallel i) \bmod 2^n$, and for each sends the leaf label and the n sibling labels along its root path. The verifier recomputes the leaf from the siblings that are its parents, hashes upward through the path, and checks that it arrives at φ . The verifier's cost is $O(tn)$ label computations with bounded workspace. A label computation may span multiple hash blocks; counting labels is not counting compression calls.

Implementation: `posw.verifyOpening` Recomputes one opened leaf from its sibling labels and walks the path to the root. in `libs/crypto/src/posw.zig`

```
fn verifyOpening(base: *const Sha256, n: u8, phi: *const Label, gamma: u32,
    opening: []const u8) bool {
    const leaf: *const Label = opening[0..label_len];
    var h = nodeHasher(base, n, gamma);
    var d: u8 = 1;
    while (d <= n) : (d += 1) {
        if (pathBit(gamma, n, d) == 1) h.update(siblingAt(opening, n, d));
    }
    var computed: Label = undefined;
    h.final(&computed);
    if (!std.mem.eql(u8, &computed, leaf)) return false;

    var cur = leaf.*;
    d = n;
    while (d >= 1) : (d -= 1) {
        const sib = siblingAt(opening, n, d);
        var parent = nodeHasher(base, d - 1, gamma >> @intCast(n - d + 1));
        if (pathBit(gamma, n, d) == 0) { parent.update(&cur); parent.update(sib); }
        else { parent.update(sib); parent.update(&cur); }
        parent.final(&cur);
    }
    return std.mem.eql(u8, &cur, phi);
}
```

3.2.3 Sampling, Soundness, and Their Limits

Consider a fixed commitment with an independently detectable bad fraction α . If each opening samples uniformly and independently, the chance of missing every bad location in t openings is $(1 - \alpha)^t$. This elementary calculation explains why more openings help. It is not, by itself, a security proof for a sequential-work protocol.

Worked example: a conditional sampling bound With a fixed bad fraction $\alpha = 0.1$ and $t = 16$ independent openings, the chance of missing every bad location is $0.9^{16} \approx 0.185$. With 32 openings it is about 0.034. The premise fixes the commitment before the samples are chosen. If a prover can grind many roots, choose which commitment to reveal, or correlate failures, this calculation does not account for that strategy.

The referenced sequential-work construction has a security analysis in an oracle model. Sibuna's domain separation, parameter mapping, truncation, and implementation must still be checked against the construction's assumptions. Fiat–Shamir sampling makes openings

reproducible from a commitment; reproducibility alone does not prove resistance to grinding. We make no unconditional claim that cheating is never cheaper.

Exercise 3.1. If an attacker tries g independent commitments, each accepted with probability q , derive the probability that at least one is accepted. Which costs are missing from that expression?

Hint: First calculate the probability that all g fail.

3.2.4 Prover Memory

A naive prover stores every label ($2^{n+1} - 1$ of them). Sibuna's prover keeps a stack of the current path's left-sibling labels ($n + 1$ labels) during the first pass and retains only the top $m = \min(n, 10)$ levels ($2^{m+1} - 1$ labels, at most 64 KB). To open a leaf it recomputes the subtree of the leaf's depth- m ancestor, $2^{n-m+1} - 1$ hashes, seeding the stack from the retained levels. Total prover cost is $2^{n+1} - 1 + t(2^{n-m+1} - 1)$ hashes and the whole workspace, including the output proof, is under 100 KB: a browser tab and a native test share the same Workspace type.

3.2.5 Parameters Are Not Timings

Sibuna maps work bits to depth using $n = b - 3$, clamped to the supported interval $[4, 24]$. This is a work-scale convention, not a guarantee of equal wall-clock time between algorithms. A leaf can hash several parent labels, while Hashcash repeatedly tests a nonce. Compiler, browser, processor, and parameter choices all affect their relative costs. The committed primitive benchmark measures verification; it does not establish browser solve latency.

Proof size is $32(1 + t(n + 1))$ bytes. At depth 13 with sixteen openings, this is $32(1 + 16 \times 14) = 7200$ bytes. Raising the opening count increases both the proof and the verifier's work, even if the dominant cost of constructing the tree stays the same.

Exercise 3.2. Compute the proof size at depth 17 with 16 openings. Then double the opening count. Which term doubles and which term remains fixed?

3.2.6 Choosing a Construction

The relevant comparison is a resource budget, not an algorithm's reputation. Record the prover's memory, the verifier's worst-case work, the proof bytes, and the assumptions needed for the claimed property. Memory-hard search can be useful, but a verifier that must repeat expensive work gives an attacker another place to spend the server's resources. Publicly verifiable delay constructions serve a different trust model from a shared-secret gate.

Sibuna retains cheap hash verification and a hash-based sequential-work option. The latter requires more proof bytes and more careful parameter analysis. Neither choice removes the need to reject malformed proofs before entering expensive verification.

Explain the invariant. Explain the distinction between an authentication tree, which proves that a label belongs to a commitment, and dependency edges, which constrain how labels are computed.

3.3 Symmetric Authentication: Challenges and Sessions

In this chapter Explain why a keyed pseudorandom function is the correct primitive when issuer and verifier are the same trust domain, state the forgery bound for a 128-bit tag, describe the stateless challenge record and the spent-set bound, and read the key schedule.

3.3.1 Issuer Equals Verifier

A digital signature lets **anyone** holding the public key verify, which is the right tool when verifiers must not be able to mint. Sibuna's verifier **is** the issuer (or a cluster sharing one seed). A message authentication code built from a pseudorandom function supplies authentication within that trust domain. It does not supply public verification: anyone holding the MAC key can also mint tokens. The benchmark chapter compares the implemented token operations.

Lemma 3 (Forgery bound) With BLAKE3 in keyed mode modelled as a PRF and a 16-byte tag, an adversary making q verification queries forges a valid token with probability at most $q \cdot 2^{-128} + \epsilon_{\text{PRF}}$. At a million queries per second that is 2^{-108} per second.

A second consequence matters for the long term: Ed25519 rests on discrete logarithms in an elliptic-curve group, which Shor's algorithm breaks; a keyed hash rests only on the hash function, against which quantum algorithms give at most a quadratic speedup. Sibuna keeps Ed25519 as an option (`--token-scheme ed25519`) for deployments that need non-minting verifiers.

3.3.2 The Key Schedule

One 32-byte master seed (from `--secret-file`, `SIBUNA_SECRET`, or a fresh random value) derives every purpose key with keyed BLAKE3 and a domain string:

```
pub fn derive(seed: *const [32]u8) Keys {
    return .{
        .token = subkey(seed, "sibuna/token/v1"),
        .challenge = subkey(seed, "sibuna/challenge/v1"),
        .ed25519_seed = subkey(seed, "sibuna/ed25519/v1"),
        .fingerprint = subkey(seed, "sibuna/fingerprint/v1"),
    };
}
```

Domain separation prevents accidental reuse across purposes; compromise of the master seed compromises every derived key. A cluster only has to agree on one seed.

3.3.3 Stateless Challenges

A challenge identifier is a self-authenticating record: version, algorithm, difficulty, opening count, issue time, the client's keyed fingerprint, a PRF-derived nonce, and the hash of the policy rule that demanded the challenge, followed by a 16-byte tag under the challenge key. Issuing one writes nothing. Verification decodes the record, checks the tag in constant time, checks the age against the challenge TTL, checks the fingerprint against the submitting client, and only then examines the proof.



Challenge identifier: 36-byte payload + 16-byte keyed tag = 52 bytes, 70 URL-safe base64 characters



Session token: 40-byte payload + 16-byte keyed tag = 56 bytes, 75 URL-safe base64 characters

Figure 4: Wire formats of the challenge identifier and the session token

Lemma 4 (State grows only with paid work) Let S be the set of challenge tags the daemon remembers. A tag enters S only after a valid proof for it was accepted, and it leaves after the challenge TTL. Hence $|S| \leq r \cdot \text{TTL}$ when r bounds the rate of accepted distinct solutions over that interval. A tag is not allocated on a free challenge fetch. The implementation also has a fixed capacity and can reject a valid solution when saturated. This is a state bound, not a lower bound on every adversary's computational strategy.

The spent set is 16 shards of Robin Hood open addressing over the 16-byte tags (chapter 5). The fingerprint is a keyed hash of the client address and User-Agent, length-separated so no two inputs collide by concatenation, and keyed so fingerprints of other clients cannot be computed offline.

Exercise 3.3. A token payload carries a work level, `timestamp`, `expiry`, `rule_hash`, and `fingerprint`. Explain what each field prevents if it were removed, and why the tag must cover all five.

Explain the invariant. A colleague proposes switching all tokens to Ed25519 "because it is stronger". Explain the trust-domain argument, the cost difference, and the post-quantum consideration.

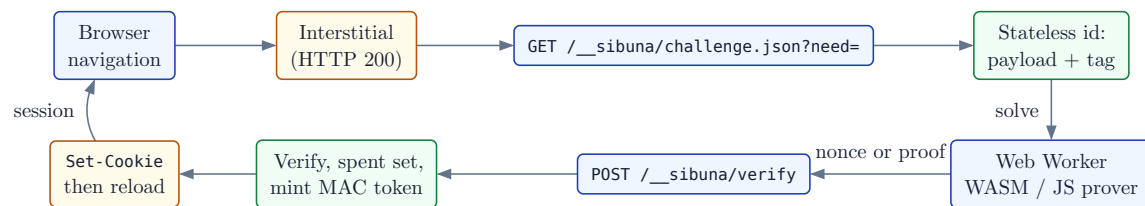


Figure 5: The challenge round trip

4 The Sibuna Protocol

We specify the wire protocol as implemented: the interstitial round trip, the stateless challenge record, both solution formats, verification order, single-use enforcement, and the session cookie.

4.1 The Challenge Round Trip

In this chapter By the end of this chapter, you should be able to trace a browser from its first request to a minted session, name every endpoint and JSON field involved, and explain why the challenge carries the protected path's policy decision.

1. A navigation with no valid session reaches the policy engine and is classified **CHALLENGE**. Because the request accepts `text/html`, the daemon answers **200** with the interstitial page and the header `X-Sibuna-Status: CHALLENGE`; an API client that does not accept HTML receives **401** with a JSON body whose `challenge` URL is ready to fetch. Either way the response carries a **requirement ticket**: the challenge this decision demands (algorithm, work bits, openings) and the rule's hash, sealed for this client with a keyed BLAKE3 tag under its own derived key and valid for one challenge lifetime.
2. The interstitial fetches `GET /__sibuna/challenge.json?path=<original URL>&need=<ticket>`. A valid ticket decides the challenge: the work is fixed by the decision that demanded it, not recomputed from a reported URL and the fetch's own headers, which differ from the navigation's (`Accept`, `Sec-Fetch-*`). Without a valid ticket (an API client that ignores the URL, a page left open past the lifetime) the server evaluates the reported URL, split into path and query by the same function that reads the request line; a URL longer than 8 KiB is refused with **414**, never truncated. The ticket is advisory: admission still checks the session's work level, so a replayed or forged ticket can only cost its holder work.
3. The response is the challenge record:


```

{"id": "AQEND...70
chars", "algorithm": "posw", "difficulty": 13, "challenges": 16, "expires_at": 1757241234}

```

 For hashcash, `difficulty` is the bit count and `challenges` is 0.
4. A Web Worker solves it with the WebAssembly module, or the JavaScript prover if WebAssembly is unavailable (Part VII), and posts either


```

{"challenge_id": "...", "nonce": "90766"}

```

 or


```

{"challenge_id": "...", "proof": "<base64url, 32(1 + t(n+1)) bytes>"}

```

to POST `/__sibuna/verify`.

5. On success the server answers 200 {"status":"ok"} with a Set-Cookie header, and the page reloads its original location. Every later request carries the cookie and is admitted before policy evaluation.

Why the path travels with the challenge The interstitial is served at the protected URL, so the browser knows where it is. Sending the path lets the server pick the **per-rule** difficulty and algorithm (a checkout route may demand 20 work bits of sequential work while a blog demands 16 bits of Hashcash) and binds the rule hash into the token, where X-Sibuna-Rule-Hash reports it upstream.

4.2 The Stateless Challenge Record

In this chapter Read the challenge encoder, understand every field's purpose, and see how the per-challenge nonce is derived without an entropy syscall on the hot path.

Implementation: Coordinator.createChallengeWithSpec Builds the 36-byte payload, tags it under the challenge key, and base64url-encodes the 52 bytes into a 70-character identifier. in `libs/challenge/src/coordinator.zig`

```
pub fn createChallengeWithSpec(self: *Coordinator, client_ip: []const u8,
    user_agent: []const u8,
    now: u64, spec: ChallengeSpec, rule_hash: u64) ChallengePayload {
    self.adaptive.observe(now * 1000);
    var effective = spec;
    effective.difficulty += self.adaptive.bump();
    const fp = self.fingerprint(client_ip, user_agent);
    const difficulty: u8 = switch (effective.algorithm) {
        .hashcash => @intCast(effective.hashcashBits()),
        .posw => effective.poswDepth(),
    };
    const challenges: u8 = if (effective.algorithm == .posw)
        effective.posw_challenges else 0;

    var raw: [id_raw_len]u8 = undefined;
    raw[0] = version;
    raw[1] = @intFromEnum(effective.algorithm);
    raw[2] = difficulty;
    raw[3] = challenges;
    std.mem.writeInt(u64, raw[4..12], now, .little);
    std.mem.writeInt(u64, raw[12..20], fp, .little);
    std.mem.writeInt(u64, raw[20..28], self.nextNonce(now, fp), .little);
    std.mem.writeInt(u64, raw[28..36], rule_hash, .little);
    raw[payload_len..id_raw_len].* = self.tagFor(raw[0..payload_len]);
    // ... base64url encode and return the payload view
}
```

Three details deserve attention:

- **Adaptive difficulty.** The coordinator keeps an exponentially weighted moving average of the challenge issue rate in three atomics. Above a baseline of 50 challenges per second, each doubling of the rate adds one work bit, capped at six. A flood pays exponentially more while quiet traffic keeps the base cost.
- **The nonce is a PRF output,** keyed by the challenge key over a counter, the clock, and the fingerprint. It is unique (counter) and unpredictable (key), so an adversary cannot precompute solutions for identifiers it has not been issued, and the hot path never calls into the operating system for entropy.
- **Difficulty travels inside the tag.** A client cannot lower the difficulty by editing the record: the tag would fail before any proof is examined.

4.3 Verification Order and Single Use

In this chapter Understand why the cheap checks run before the proof, why the challenge is marked spent only after the proof is valid, and how two concurrent submissions of one solution are resolved.

Implementation: `Coordinator.verifyAndMint` Decodes and authenticates the identifier, checks age and client binding, verifies the proof for the recorded tier, records the tag as spent, and mints the session token. in `libs/challenge/src/coordinator.zig`

```
pub fn verifyAndMint(self: *Coordinator, challenge_id: []const u8, solution:
Solution,
    client_ip: []const u8, user_agent: []const u8, now: u64) VerifyError!
VerifiedResult {
    const decoded = try self.decode(challenge_id);
    const expires_at = decoded.issued_at + self.challenge_ttl;
    if (now > expires_at or decoded.issued_at > now + 60) return
error.ChallengeExpired;
    if (decoded.fingerprint != self.fingerprint(client_ip, user_agent)) {
        return error.FingerprintMismatch;
    }
    try checkSolution(challenge_id, decoded, solution);
    self.spent.markSpent(&decoded.tag, expires_at, now) catch |err| switch (err) {
        error.DoubleSpendAttempt => return error.DoubleSpendAttempt,
        else => return error.StoreFull,
    };
    return self.mintToken(now, decoded.rule_hash, decoded.fingerprint);
}
```

The order is deliberate. Tag, age, and binding are constant-time checks that reject garbage before any hashing. A wrong proof must not consume the challenge, so the spent set is written only after `checkSolution` succeeds. Two threads that both verify the same valid solution race on `markSpent`; the shard spinlock serialises them, exactly one wins, and the other receives `DoubleSpendAttempt`. The `issued_at > now + 60` clause rejects records minted by a node whose clock runs ahead by more than a minute, which would otherwise extend a challenge's life.

Errors surface to the client as 400 with an Elm-style diagnostic (Part X): the end-to-end tests assert on DOUBLE SPEND, FINGERPRINT MISMATCH, WRONG SOLUTION TYPE, and MALFORMED CHALLENGE in the response body.

4.4 The Session Cookie

In this chapter Read the keyed-hash token, understand the cookie attributes, and explain the client binding.

Implementation: MacToken.mint and MacToken.verify Serialises the 40-byte payload, appends the first 16 bytes of keyed BLAKE3 over it, and verifies with a constant-time comparison. in `libs/crypto/src/token.zig`

```
pub fn verify(key: *const [32]u8, token_str: []const u8, now: u64,
    expected_fingerprint: ?u64) TokenError!Payload {
    if (token_str.len != encoded_size) return error.InvalidTokenLength;
    var raw: [raw_size]u8 = undefined;
    b64.Decoder.decode(&raw, token_str) catch return error.InvalidEncoding;
    const expected = tag(key, raw[0..payload_size]);
    if (!std.crypto.timing_safe.eql([tag_size]u8, expected,
        raw[payload_size..raw_size].*)) {
        return error.InvalidTokenSignature;
    }
    const payload = Payload.deserialize(raw[0..payload_size]);
    try payload.check(now, expected_fingerprint);
    return payload;
}
```

The cookie is emitted as

```
Set-Cookie: __sibuna_token=<75 chars>; Path=/; Max-Age=86400; HttpOnly;
    SameSite=Lax[; Secure]
```

HttpOnly keeps it out of page scripts, SameSite=Lax stops cross-site replay while allowing top-level navigation, and Secure is added with `--secure-cookie` when TLS terminates in front of the daemon. Because the payload carries the keyed fingerprint of address and User-Agent, a cookie copied to another machine fails with `TokenBoundAddressMismatch`; the end-to-end test "the cookie is bound to the client identity" exercises exactly that path.

The payload begins with a version byte and the **work level** the holder paid: the mechanism (Hashcash or PoSW) and the work bits actually solved, including any load-adaptive bump. A session clears a challenge only when its level reaches what the route demands, computed from the rule's difficulty through the same clamping the challenge itself applies. Levels are ordered, not named: a cookie earned at 24 bits also covers every 16-bit route, while a 16-bit cookie presented to a 24-bit route falls through to the stronger interstitial and the new cookie replaces the old one. `rule_hash` remains the audit identity of the rule that issued the challenge; it is reported upstream, never used for admission. WAF findings and explicit denials are never cleared by a session. The end-to-end test "a session earned on a cheaper route does not admit a route that demands more work" exercises both directions.

Forwarded addresses Behind an ingress the client address arrives in `X-Forwarded-For`. Sibuna trusts that header only with `--trust-forwarded` (implied in `forward-auth` mode). A reverse proxy exposed directly to the internet must leave it off, or any client could pick its own binding and its own rate-limit bucket.

Exercise 4.1. Sketch the sequence of two browsers behind one NAT address with identical User-Agents. Which checks do they share, which do they not, and what would an attacker on that NAT need to reuse the other browser's cookie?

Explain the invariant. Explain to an operator why the daemon does not "remember" outstanding challenges and why that is a security feature rather than a shortcut.

5 Deterministic Memory and Zero-Allocation Engineering

We follow a connection from accept to response through one stack buffer, and examine the concurrent data structures that let many worker threads share state without heap allocation on the request path.

5.1 The Connection Loop

In this chapter By the end of this chapter, you should be able to explain how one 64 KB buffer serves an entire keep-alive connection, how the head and body are located without copying, why every connection gets its own bounded thread, and how the proxy learns the origin's framing so a proxied client connection can stay open.

5.1.1 One Thread per Connection, Bounded

Accept threads (`--workers`, one per CPU by default) share the listening socket. Each accepted connection is handed to its own thread with a one-megabyte stack, and the accept loop goes straight back to `accept`. The alternative, serving a connection to completion on the accept thread, was the first design: it measured well on a single client and badly on sixty-four, because sixty of them waited for a worker to finish its 256-request quota. The tail latency under that design was over 100 ms at 64 connections; with a thread per connection it is under one millisecond (Part VIII).

The number of connection threads is bounded by `--max-connections` (1,024 by default). Past the bound the accept loop answers `503 Service Unavailable` on the new socket and closes it without spawning anything, and counts the event in `sibuna_overloaded_total`. Memory is bounded the same way: a connection thread touches about 100 KB of its stack, so the worst case is a known number rather than a function of how many sockets a client can open. The idle reaper (below) closes connections that stop sending, so a slow client cannot pin a thread past `--idle-timeout`.

5.1.2 One Buffer per Connection

Every dynamic allocation on a request path is a denial-of-service lever: fragmentation over days, allocator lock contention across threads, and amplification by attackers who craft requests that maximise allocation. Sibuna's rule is absolute: classification, verification, and proxy hand-off run with zero heap allocation. The connection handler owns a 64 KB buffer on its thread's stack and a 16 KB write buffer; everything else is a slice into them.

```
pub fn handleConnection(stream: Io.net.Stream, io: Io, state: *AppState) void {
    defer stream.close(io);
    var conn_buf: [max_request_bytes]u8 = undefined; // 64 KB
    var reader = stream.reader(io, &conn_buf);
    var writer_buf: [16 * 1024]u8 = undefined;
```

```

var writer = stream.writer(io, &writer_buf);
// ... format the peer address once, then serve up to 4096 requests
while (served < max_requests_per_connection) : (served += 1) {
    const keep = serveOne(&conn) catch break;
    if (!keep) break;
}
}

```

`readHead` fills the buffer until the blank line appears, refusing heads over 16 KB with 431. The parser produces a `Request` whose method, path, query, headers, and cookies are slices of the buffer; the declared body is filled up to what fits and sliced after the head; then `toss(body_end)` advances the reader so the next keep-alive request starts cleanly. Internal routes are length-delimited and keep the connection open. Proxied requests keep it open too, provided the origin's response is framed; the next section shows how the proxy decides.

5.1.3 The Proxy Head Rewrite

The head is not forwarded verbatim. `writeHead` re-emits the request line and each header, dropping hop-by-hop fields (Connection, Transfer-Encoding, any incoming X-Forwarded-For) and appending the audit set. Body framing is generated, never copied. A Content-Length is re-emitted. A chunked body that ended within the connection buffer is decoded in place and sent with its length. A longer one is announced as Transfer-Encoding: chunked and re-chunked one read at a time (SID 0009). The audit set:

```

Connection: close
X-Forwarded-For: <client>
X-Real-IP: <client>
X-Sibuna-Status: PASS
X-Sibuna-Rule: session | robots-txt | ip/cidr-trie | ...

```

Bodies larger than the buffer are relayed in 16 KB chunks from the client reader to the origin writer before the response is streamed back. The unit test in `proxy.zig` asserts the rewrite drops a spoofed X-Forwarded-For and injects the audit fields.

5.1.4 Relaying the Origin Response

The first proxy streamed the origin's bytes until the origin closed and then closed the client: correct framing with no parsing, at the price of a new TCP connection per proxied request. Under a load generator that price was visible as tens of thousands of sockets in `TIME_WAIT` and, on loopback, exhausted ephemeral ports. The proxy now reads the origin's head (at most 16 KB) and classifies the body by RFC 9112's rules:

Implementation: `proxy.parseResponseHead` Returns the status and one of four framings: no body (HEAD, 1xx, 204, 304), a Content-Length, chunked transfer coding, or close-delimited. in `libs/net/src/proxy.zig`

```

pub const Framing = union(enum) { none, length: u64, chunked, until_close };

pub fn parseResponseHead(head: []const u8, head_request: bool) ?ResponseHead {
    if (head.len < 12 or !std.mem.startsWith(u8, head, "HTTP/1.")) return null;
    const status = std.fmt.parseInt(u16, head[9..12], 10) catch return null;
    var framing: Framing = .until_close;

```

```

var chunked = false;
// ... one pass over the header lines for Transfer-Encoding and Content-Length
if (head_request or status / 100 == 1 or status == 204 or status == 304) framing
= .none;
if (chunked) framing = .chunked;
return .{ .status = status, .framing = framing };
}

```

The head is re-emitted to the client with the origin's `Connection` headers replaced by Sibuna's own decision, and the body is relayed exactly: `streamExact` for a length, chunk by chunk (size line, data, trailers) for chunked coding, `streamRemaining` for the legacy case. Only the legacy case closes the client.

5.1.5 The Origin Pool

Parsing the framing also tells the proxy whether the **origin** socket can be used again: an HTTP/1.1 response without `Connection: close` (or an HTTP/1.0 one with `keep-alive`) whose body was fully consumed leaves the socket at a clean request boundary. Such sockets go into a fixed pool of 256 idle origin connections guarded by a spinlock; the next proxied request takes one instead of connecting. The measurement that forced this (Part VIII) was blunt: with a new origin connection per request, a four-core proxy managed about 1,400 requests per second before loopback ran out of ephemeral ports.

A pooled socket may have been closed by the origin while idle. The proxy notices in one of two ways, a failed write or an end-of-stream before the first response byte, and in both cases nothing has reached the client yet, so it closes the socket and retries once on a fresh connection, never on another pooled one: after an idle period every pooled socket may be stale, and the first version of this retry, which took a second pooled socket, answered 502 to the first request after every quiet spell. The retry is allowed only when the request body was fully buffered; a body that was relayed in chunks cannot be sent again, and the client gets 502. Unit tests relay fixed byte strings through the same function and assert the output is byte-identical apart from the connection header; an end-to-end test sends two proxied requests on one client socket and checks, through a sequence header the stub origin adds, that the second reused the pooled origin connection.

Exercise 5.1. A client sends a 200 KB upload. Trace which bytes live in the 64 KB buffer, which are relayed by `relayBody`, and why the proxy must write the head to the origin **before** relaying.

5.2 Concurrent State Without Allocation

In this chapter Analyse the spinlock, the Robin Hood spent set, the GCRA limiter, the lock-free ban table, the MPSC incident ring, and the read-copy-update engine slot.

5.2.1 Spinlocks and Sharding

The critical sections in Sibuna's tables are tens of instructions long, far shorter than a kernel futex round trip, so shards are guarded by a two-state atomic spinlock with `spinLoopHint` in

the wait loop. Tables are split into 16 shards by key hash so two operations contend with probability $\frac{1}{16}$.

5.2.2 The Robin Hood Spent Set

Only accepted proofs enter the spent set (chapter 3). Each shard is an open-addressed array of 4096 entries of {tag: [16]u8, expires_at: u64, dist: u8, occupied: bool}. Robin Hood insertion (Celis, 1986) displaces any occupant closer to its home than the incoming key, which keeps probe lengths tightly clustered; lookups stop as soon as they meet an entry nearer its home than they are to theirs.

```
fn insert(self: *Shard, tag: *const Tag, expires_at: u64, now: u64) StoreError!void {
    if (!self.canInsert(tag, now)) return error.StoreFull;
    var carry = Entry{ .tag = tag.*, .expires_at = expires_at, .dist = 0,
        .occupied = true };
    var idx = home(tag);
    while (carry.dist < MAX_PROBE) {
        const e = &self.entries[idx];
        if (!e.occupied) { e.* = carry; self.live += 1; return; }
        // Overwriting an expired occupant is safe only when the new
        // distance is not smaller, so keys probing past this slot still
        // pass the early-termination test.
        if (e.expired(now) and carry.dist >= e.dist) { e.* = carry; return; }
        if (carry.dist > e.dist) {
            std.mem.swap(Entry, &carry, e);
            if (carry.expired(now)) return;
        }
        idx = (idx + 1) % SHARD_CAPACITY;
        carry.dist += 1;
    }
    return error.StoreFull;
}
```

The expired-slot rule is the subtle part: an expired entry may be overwritten in place only by a key at least as far from its home, otherwise a later key that probes past the slot would stop early and be lost. Expired entries that get displaced are simply dropped. No sweeper thread is needed. Insertion first checks that bounded displacement can succeed without losing a live tag.

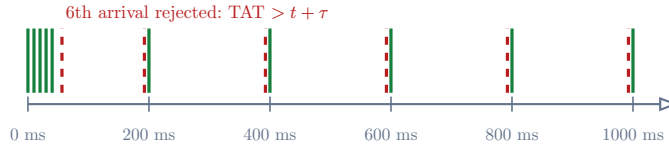
5.2.3 GCRA: Rate Limiting in One Integer

The Generic Cell Rate Algorithm (ATM Forum, 1996) is the virtual-scheduling form of the leaky bucket. Per client it stores one **theoretical arrival time** (TAT). With emission interval $T = \max(1, \lceil \frac{W}{N} \rceil)$ for a positive limit N per window W , and burst tolerance $\tau = (N - 1)T$:
 arrival at t conforms $\Leftrightarrow \text{TAT} \leq t + \tau$, then $\text{TAT} \leftarrow \max(\text{TAT}, t) + T$.

Theorem (token-bucket bound) After the first conforming arrival at t_1 , the k -th conforming arrival satisfies $t_k - t_1 \geq (k - 1)T - \tau$. Hence any interval of length L contains at most $N + \lfloor \frac{L}{T} \rfloor$ conforming arrivals: a burst of N , then one per T . There is no fixed-window boundary artefact beyond the defined burst, and no per-request history.

```
fn check(self: *Shard, key: u64, now_ms: u64, limits: Limits) Decision {
    const interval = limits.emissionInterval();
```

rate = 5 per 1000 ms, so $T = 200$ ms and $\tau = 800$ ms: a burst of five, then one admission per T



green: admitted; dashed red: rejected. TAT after the k -th admission is $t_1 + kT$, so any interval of length L admits at most $N + \lfloor L/T \rfloor$

Figure 6: GCRA admission for a limit of 5 per second: a burst of five, then one every 200 ms

```
const tau = limits.burstTolerance();
self.lock.lock();
defer self.lock.unlock();
const cell = self.locate(key, now_ms, tau) orelse return .{
    .limited = true, .retry_after_ms = interval, .remaining = 0,
};
const tat = @max(cell.tat_ms, now_ms);
if (tat > now_ms +| tau) {
    return .{ .limited = true, .retry_after_ms = tat - tau - now_ms, .remaining =
0 };
}
cell.tat_ms = tat +| interval;
// ... remaining = floor((t + tau - TAT) / T) + 1
}
```

Cells are 16 bytes in 16 shards of 512 slots with a 16-slot probe window; a cell whose TAT is older than $t - \tau$ has drained and is reclaimed on the spot. The **Retry-After** header is computed from the same arithmetic. Saturated probe windows refuse new clients rather than evicting active quota state; a zero configured rate refuses requests.

5.2.4 The Versioned Ban Table

Bans use 4096 slots with atomic key, expiry and version fields. A writer brackets updates with version increments; readers accept only a stable even version and retry concurrent changes. Sequential consistency prevents mixing an old identity with a replacement expiry. Readers take no mutex, but can wait for a writer and are not wait-free. These atomic operations have real cost, measured indirectly in the HTTP harness rather than assumed free.

5.2.5 The MPSC Incident Ring

When the Shield surface denies a request or the honeypot fires, the incident is copied into a fixed 1.3 KB record and pushed onto a bounded multi-producer single-consumer ring (Vyukov's sequence-stamped design, 512 slots). Producers are worker threads; the consumer is the storage thread. A full ring drops the newest record rather than blocking a response, an explicit loss policy rather than a durability guarantee. The drop counter makes that loss observable. After moving a record into a pending batch, the storage thread retains it until commit is confirmed; it does not silently discard it on a database error.

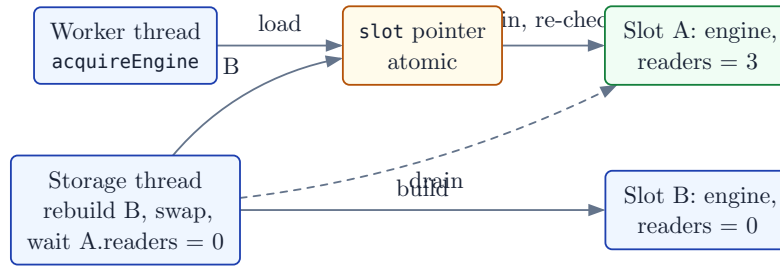


Figure 7: Engine slot swap: workers pin a slot; the storage thread swaps the pointer and waits for the old slot's readers to drain before rebuilding into it

5.2.6 Read-Copy-Update Engine Slots

The policy engine has fixed-capacity automaton tables and is rebuilt off the hot path when policies change. Workers must never observe a half-built engine, and the builder must never overwrite an engine a request is still reading. Sibuna uses two slots with reader counts:

```

pub fn acquireEngine(self: *AppState) *EngineSlot {
    while (true) {
        const slot = self.slot.load(.seq_cst);
        _ = slot.readers.fetchAdd(1, .seq_cst);
        if (self.slot.load(.seq_cst) == slot) return slot;
        _ = slot.readers.fetchSub(1, .seq_cst);
    }
}

pub fn publishEngine(self: *AppState, fresh: *EngineSlot) *EngineSlot {
    const old = self.slot.swap(fresh, .seq_cst);
    while (old.readers.load(.seq_cst) != 0) std.atomic.spinLoopHint();
    return old;
}

```

Sequentially consistent pointer and reader-count operations establish one total order across the two atomics. Merely using acquire/release on separate objects is insufficient. The re-check after incrementing closes the race in which a writer swaps between the reader's load and its increment and observes zero readers: the reader notices the pointer changed, releases, and retries on the new slot. The test suite includes a scenario that deadlocked when a test held a slot across a rebuild, which is exactly the guarantee working as designed.

Each physical slot owns its own allocation arena, so a rebuild into the spare slot cannot free strings the active engine still references. A request copies the matched rule name into a bounded buffer and releases its slot **before** proxy I/O begins; otherwise a slow origin would hold a reader count and stall the next publication for as long as the origin took to answer.

Exercise 5.2. Using the GCRA theorem, compute the maximum number of requests a single client can get through in the first 3 seconds after being idle, for `--rate-limit 100 --rate-window 10`. Compare with a fixed 10-second window counter and with a true sliding-window log.

Explain the invariant. Explain why an expired Robin Hood entry can be overwritten only by a key with a distance at least as large, using a three-slot example.

6 Automata, Tries, and the Semantic Firewall

We examine the algorithms behind sub-microsecond classification: the tagged Aho–Corasick automaton, the IPv4/IPv6 radix trie, the byte-class tokenizers of the semantic WAF, and the declarative policy engine with WEIGH scoring.

6.1 Single-Pass Multi-Pattern Matching

In this chapter By the end of this chapter, you should be able to derive the Aho–Corasick automaton's linear-time guarantee, explain the dense-table layout and comptime case folding, and argue why a SIMD literal engine was not adopted.

6.1.1 Aho–Corasick as a Dense DFA

Sibuna has two literal-pattern sets: about forty bot signatures and about a hundred attack signatures. Scanning a field against N patterns one at a time is $O(NM)$; the automaton of Aho and Corasick (1975) scans it once in $O(M)$ after building a trie with failure links. Sibuna folds the failure links into the transition table at build time, so scanning is one table load per input byte with no branches on the pattern set:

```
fn findFirstImpl(self: anytype, haystack: []const u8) ?Match {
    var state: u16 = 0;
    for (haystack) |byte| {
        state = self.transitions[state][toLower(byte)];
        const pid = self.match_id[state];
        if (pid != no_match) {
            return .{ .name = self.pattern_names[pid], .tag =
self.pattern_tags[pid] };
        }
    }
    return null;
}
```

The automaton is generic over its state budget. The implementation reserves at most one root plus the sum of all signature lengths, a safe upper bound on trie states. This avoids historically oversized fixed tables while retaining a dense transition for every byte. Each pattern carries an 8-bit category tag; case folding uses a 256-byte compile-time table. Exact engine sizes and the comparison with sequential substring search are recorded in chapter 8. The latter uses the same patterns and any-match semantics.

6.1.2 Regular Expressions, SIMD Engines, and the Choice

Three families were weighed for the signature stage:

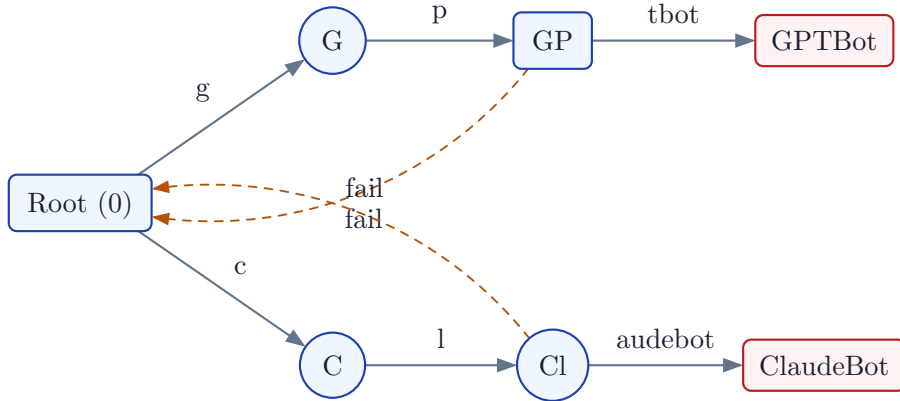


Figure 8: Trie states and failure edges for two signatures

- **Backtracking regular expressions** (a backtracking rule-set style) run hundreds of patterns per request and allocate per match; they are the slowest option and the most prone to false positives.
- **SIMD literal engines** (Hyperscan, Vectorscan) prefilter blocks of 16–32 bytes with shuffle-based nibble masks (the "Teddy" algorithm) and verify candidates with automata. They excel on long inputs with large literal sets. The cost is a multi-megabyte C++ dependency, a compile-time pattern database, no WebAssembly target, and, for inputs of a hundred bytes, no advantage: the whole field scans in less time than one cache miss.
- **Dense Aho–Corasick plus small structural tokenizers**, the libinjection lineage, gives exact linear-time behaviour in a fixed table and needs nothing outside the binary.

Sibuna's fields are short, so the third family wins on both memory and speed; chapter 8 records 2.9 ns per byte on an 8 KB body, the one workload where a SIMD prefilter would pay, and SID 0006 keeps that as an open item.

6.2 The Radix Trie for IPv4 and IPv6

In this chapter Trace a longest-prefix lookup through the 128-bit trie, explain the IPv4 root shortcut, and see how the trie doubles as the reputation store fed by the cluster.

One binary trie over 128-bit keys serves both families: IPv6 prefixes are inserted as-is, IPv4 prefixes are mapped into `::ffff:0:0/96` so a /24 becomes a depth-120 path. Nodes are flat arrays indexed by `u16` with index zero as both root and "no child", so a lookup is at most 128 dependent loads with no pointer chasing.

A first implementation walked the 96 mapped-prefix levels for every IPv4 lookup and measured 257 ns; the trie now records the node at depth 96 at initialisation and starts IPv4 lookups there:

```
if (address >> 32 == v4_mapped_prefix >> 32) {
    current = self.v4_root;
    i = 96;
}
```

IPv4 lookups cost 45 ns and IPv6 lookups 80 ns. The engine consults the trie twice: terminal **allow/deny** verdicts before the rule table, so a cluster-wide ban beats every rule, and **challenge** verdicts after it.

6.3 The Semantic Firewall

In this chapter Understand the byte-class pre-scan, the tagged signature automaton, the SQL tokenizer with tautology detection, the HTML and shell structure checks, canonicalisation, and the false-positive discipline that a real browser request must pass untouched.

6.3.1 A Lesson in False Positives

The first version of the WAF matched a flat substring list against every header. The list contained `/*`, and every browser sends `Accept: text/html,...,*/;q=0.8`. Every real browser was classified as SQL injection and received **403**. The lesson shaped the design: a detector may fire only on **structure**, and the test suite now passes a complete Chrome request, including a comment body containing `select`, `--`, and an apostrophe, through the engine and asserts it comes out clean.

6.3.2 Byte Classes in One Pass

Each inspected field is scanned once to collect which byte classes occur: quotes, `=`, `<`, shell separators, `%`, `NUL`, canonicalisation triggers, and double spaces. Every detector consults the classes instead of rescanning, and the canonicalised second pass runs only when a trigger was seen. The scan is a 256-entry table lookup per byte:

```
fn scanClasses(text: []const u8) Classes {
    var bits: u8 = 0;
    var prev: u8 = 0;
    var double_space = false;
    for (text) |c| {
        bits |= class_table[c];
        if (c == ' ' and prev == ' ') double_space = true;
        if (c == '*' and prev == '/') bits |= 64;
        prev = c;
    }
    // ... unpack bits into the Classes struct
}
```

6.3.3 Detectors

- **Signatures.** All strong signatures of every category live in one tagged automaton; a hit is a terminal **deny** named `waf:sqli`, `waf:xss`, `waf:path-traversal`, or `waf:rce`.
- **SQL injection.** A single-pass word scanner looks up each alphanumeric word (at most seven bytes) in the keyword table and, on `or` / `and`, checks for a `literal = literal` tautology; `id=1` or `1=1` is terminal. Otherwise a quote byte is mandatory and the evidence must score at least 4 out of: quote (1), up to two keywords (2), a comment marker (1), an `=` (1). `admin'--` is terminal by itself. "it's a group order from the shop" scores 3 and passes.

- **Cross-site scripting.** Requires `<`; fires when a tag that can host script co-occurs with an `on<event>=` attribute in attribute position. `onboarding=1` does not count.
- **Command injection.** Requires a separator; fires when `;`, `|`, `&&`, `$()`, a backtick, or a newline is followed by a known command name. `status ok; done | next` passes.
- **Path traversal.** Signatures (`../`, `..%2f`, `/etc/passwd`, `id_rsa`, ...) plus NUL and `%00`.

Structural headers whose grammar legitimately contains quotes and stars (`Accept*`, `Content-Type`, `Sec-*`, `If-*`, ...) are skipped; the path, query, User-Agent, other headers, and the first 8 KB of the body are inspected, and fields with percent escapes, plus signs, comment openers, or collapsible whitespace are canonicalised into an 8 KB stack buffer and inspected again, so `1%27/**/UnIoN/**/SeLeCt` and `%252e%252e/` collapse onto the raw signatures.

6.3.4 Cost

The inspection cost depends on the fields and bytes presented to the engine, not just the number of requests. The benchmark chapter reports separate Gate, Shield, and 8 KB body cases. The 8 KB bound is an inspection limit, not a claim that the rest of an upload is safe. An ingress using forward-auth must pass the relevant request information; omitted bytes cannot be inspected by this process.

6.4 The Declarative Policy Engine

In this chapter Read the rule model, the evaluation order, WEIGH scoring with thresholds, and the JSON policy grammar.

A rule is a conjunction of optional criteria (path pattern, User-Agent pattern, up to four headers, up to eight IPv4/IPv6 CIDR blocks) with an action `ALLOW`, `DENY`, `CHALLENGE`, or `WEIGH`, optional per-rule difficulty in work bits and algorithm, and a signed `weight`. Evaluation order:

1. Semantic WAF (Shield only): violation is a terminal `deny`.
2. Reputation trie: `deny` or `allow` prefixes are terminal.
3. Rules in order: the first terminal match returns; `WEIGH` matches accumulate.
4. Score: negative totals `allow`; totals ≥ 40 `deny`; totals ≥ 10 `challenge` with one extra work bit per 5 points above the threshold, capped at six.
5. Static bypass paths, then trie `challenge` verdicts, then the bot automaton.
6. Default action, `challenge` unless the policy file says otherwise.

Built-in rules allow `/.well-known/*`, `/favicon.ico`, `/robots.txt`, and `/__sibuna/*`, deny CF-Worker clients and Amazonbot, and challenge every Mozilla User-Agent: a browser string is free to forge, so admitting it would make the gate decorative.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY" },
  "rules": [
    { "name": "internal-api", "path": "/api/*", "headers": { "X-API-Key": ".*" },
      "action": "ALLOW" },
    { "name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
  ]
}
```

```

    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight":
30 },
    { "name": "no-language", "headers": { "Accept-Language": "" },
      "action": "WEIGH", "weight": -5 }
  ]
}
```

6.4.1 Worked Trace: an Encoded Attack with a Valid Session

Consider GET /search?q=%2527%2520OR%25201%253D1 from a client holding a valid session. First the HTTP parser separates the path and query without copying their bytes. Local limits run, then Shield inspects the request. The percent sign marks the field for canonicalization. One decoding pass exposes %27%20OR%201%3D1; a second exposes ' OR 1=1.

The tokenizer now sees a quote, the keyword OR, and equal literal operands. A denial is a terminal policy result. The valid session does not turn it into an allow: the session check only resolves a challenge decision. The storage hook copies a bounded incident record; SQL runs later. The response path need not hold the engine snapshot while an origin connection is idle.

Exercise 6.1. Replace the payload with 0'Reilly and then with order=1. Why should neither alone establish SQL injection? Give an example of a false positive a substring-only detector could produce.

Exercise 6.2. Write the byte-class table entries needed to add a detector for LDAP injection (|(|(patterns), and explain which existing class bits it can reuse.

Explain the invariant. Explain to a reviewer why Accept: */* was a false positive, what structural property the new SQL detector requires, and how the automaton and the tokenizer divide the work.

7 The Browser Proof-of-Work Engine

We look at the 8,831-byte WebAssembly module that shares its source with the server's verifiers, the Web Worker protocol, the JavaScript fallback provers, and the interstitial.

7.1 One Source, Two Targets

In this chapter By the end of this chapter, you should be able to explain how the browser module is built from the server's own `pow.zig` and `posw.zig`, name the exported functions, and describe the memory the sequential-work prover needs in a tab.

7.1.1 Build Wiring

The browser solver imports the exact files the server verifies with. `build.zig` creates `wasm32-freestanding` modules from `libs/crypto/src/pow.zig` and `libs/crypto/src/posw.zig` and hands them to the solver entry point:

```
const wasm_pow = b.addExecutable(.{
    .name = "sibuna-pow",
    .root_module = b.createModule(.{
        .root_source_file = b.path("apps/wasm-pow/src/entry.zig"),
        .target = wasm_target,
        .optimize = .ReleaseSmall,
        .imports = &.{
            .{ .name = "pow", .module = wasm_pow_mod },
            .{ .name = "posw", .module = wasm_posw_mod },
        },
    }),
});
```

There is no second implementation to drift. The module measures 8,831 bytes with both tiers and is served from the daemon with a one-hour cache header.

7.1.2 Exports

Export	Purpose
<code>sibuna_get_buffer_ptr() / _len()</code>	256-byte input buffer the worker writes the challenge string into
<code>sibuna_solve_step(ptr, len, bits, start, max_steps)</code>	Hashcash: search <code>max_steps</code> nonces from <code>start</code> ; returns the nonce or the all-ones sentinel
<code>sibuna_posw_solve(len, depth, challenges)</code>	Sequential work: runs the prover over the buffer and returns the proof length
<code>sibuna_posw_proof_ptr()</code>	Start of the proof bytes in the prover workspace

The prover's workspace, the retained top levels, the sibling stack, and the proof, is a static 90 KB region; no allocator is linked.

7.1.3 Prefix Pre-Hashing

The Hashcash inner loop absorbs the challenge and the colon once and copies the SHA-256 state per nonce, so each candidate costs one compression rather than two:

```
var base_hasher = Sha256.init({});
base_hasher.update(challenge);
base_hasher.update(":");
while (step < max_steps) : (step += 1) {
    var hasher = base_hasher;           // value copy of the absorbed prefix
    hasher.update(nonce_buf[0..nonce_len]);
    hasher.final(&digest);
    if (pow.checkDifficultyBits(digest, difficulty_bits)) return nonce;
    nonce += 1;
}
```

The sequential-work prover uses the same trick with the statement χ absorbed once in baseHasher.

7.2 The Worker Protocol

In this chapter Read the message protocol, understand the fallback hierarchy, and see the sentinel bug that the signed-integer boundary between WebAssembly and JavaScript caused.

The page posts one message and receives progress, fallback, and result messages:

```
// page -> worker
{ challenge: spec.id, algorithm: spec.algorithm, difficulty: spec.difficulty,
  challenges: spec.challenges }

// worker -> page
{ type: 'progress', iterations: 40000 }
{ type: 'fallback', message: 'Wasm unavailable; running JS prover' }
{ type: 'solved', challenge: spec.id, solution: { nonce: '90766' } } //
hashcash
{ type: 'solved', challenge: spec.id, solution: { proof: '<base64url>' } } // posw
{ type: 'error', message: '...' }
```

The worker tries WebAssembly first and falls back to pure JavaScript when WebAssembly is absent, blocked by policy, or fails to instantiate. The JavaScript provers are line-for-line ports of the Zig code and produce **byte-identical** output: a Node.js harness compares the two paths for both tiers. They are slower, about 60 times for the sequential prover, but they guarantee that every browser can pass.

The signed sentinel WebAssembly returns a 64-bit integer to JavaScript as a **signed** BigInt. The Hashcash export signals "batch exhausted" with all ones, which JavaScript sees as -1n, not 18446744073709551615n. The original worker compared against the unsigned value, so the loop exited after the first batch with a bogus nonce. The fix is one call: `BigInt.asUintN(64, result)`. The lesson generalises: every u64 crossing the boundary must be normalised.

7.3 Calibration

In this chapter Relate difficulty settings to wall-clock time in a browser engine and on native silicon.

Do not choose a difficulty from one laptop's average solve. Hashcash has a geometric tail; browser clocks include scheduling, thermal state, startup and compilation. A deployment calibration should record the browser version, device, algorithm, parameters, and a distribution of repeated solves. Keep module fetch and compilation separate from steady-state hashing.

Worked calibration plan Hold b fixed, warm the module, and solve at least 100 independently issued puzzles on each target device class. Record median and high-percentile durations, failures and proof sizes. Repeat at $b + 1$. Expected trial count doubles; an individual sample need not. Choose a setting against the slow-device budget, then test the complete fetch–solve–verify flow.

The repository's native primitive measurements are not evidence for a promised phone or browser solve time. A WASM module can run the same arithmetic while having different startup, execution, and memory costs. The worker's messages form the boundary at which those costs can be measured without adding a timer to the server's request handler.

7.4 The Interstitial

The page is a single embedded HTML file with a card, an indeterminate progress bar, and a status line. It requests the challenge for its own location (`?path=`), spawns the worker, posts the solution with `fetch`, and reloads on success; `fetch` calls `retry` with jittered exponential backoff, and a 4xx verification answer is shown with its diagnostic title. An off-screen honeypot link (`/__sibuna/honeypot`) is invisible to people and to script-running browsers; blind crawlers that follow every `href` ban themselves.

Exercise 7.1. The JavaScript sequential prover is about 60 times slower than WebAssembly. Estimate the fallback solve time at the default difficulty and propose a policy for clients that report the fallback (hint: the worker posts a `fallback` message the page can forward).

Explain the invariant. Explain why compiling the browser module from the server's verifier sources is a stronger guarantee than testing two implementations against each other.

8 Empirical Evaluation

We present the measurements recorded by five committed performance harnesses: primitive latencies, an admission-only comparison with Anubis, a whole-product comparison under an external load generator with CPU and memory accounting, distributed behavior and local replicated-cluster costs. The console has a separate isolation acceptance matrix. Every number in this part is rendered from a results file at build time. Its header identifies the tested revision and host. Historical September records predate the request-path changes of 1 October 2026. The primitive suite was refreshed on 3 October with Zig 0.17; the other families retain their own earlier revisions and do not qualify the current release. A functional pass and an inconclusive isolation measurement answer different questions.

8.1 Methodology

In this chapter By the end of this chapter, you should be able to reproduce every number in this part, say what each harness includes and excludes, explain what the spread column means, and tell a measured figure from a published one.

Five performance harnesses live under **benchmarks/**:

Harness	What it measures	Results file
<code>benchmark.zig</code> via <code>run-all.sh</code>	Primitive latencies in ReleaseFast, seven batches, median with min–max spread; host, binary and module sizes, idle memory	<code>results/latest.json</code>
<code>compare.py</code>	Admission operations per second against a real Anubis binary over loopback HTTP: session check, unauthenticated check, challenge bootstrap, proof verification	<code>results/admission-comparison-latest.json</code>
<code>tools.py</code>	Whole products under wrk: throughput, latency percentiles, CPU time per request, peak resident memory, for Sibuna Gate, Sibuna Shield, and Anubis in forward-auth and reverse-proxy modes	<code>results/tools-comparison-latest.json</code>
<code>distributed.py</code>	Three daemons with six client processes: Gate, Shield, and replicated Shield; session porta-	<code>results/distributed-latest.json</code>

Harness	What it measures	Results file
	bility, WAF denial with a session, ban propagation, leader loss	
<code>cluster.py</code>	One node and three local replicated nodes under <code>wrk</code> , including idle CPU, memory, transport and failover checks	<code>results/cluster-latest.json</code>

The primitive suite times batches of many thousand operations; per-operation percentiles are not reported because a clock read costs as much as the work. The HTTP harnesses report what `wrk` reports: request counts, and per-request latency percentiles that `wrk` computes from its own histogram. CPU time is the product process's accumulated user and system time from `ps`, read before and after each run; memory is the peak resident set sampled every 100 ms. On the Linux containers used for the release review, `ps` reports whole CPU seconds, so short runs cannot resolve small changes in CPU cost. Request rate and latency come from the load generator's independent wall clock and histogram; CPU-accounting precision does not change those measurements. Container permissions do not grant control over the host's processor governor or other tenants. Resource conditions and uncertainty belong with each record rather than being assumed to match a dedicated machine.

The Linux admission and whole-product comparisons give every product thread the same allowed CPU set before warmup: two CPUs for admission and four by default for whole products. A Sibuna accept-thread count is not a CPU budget equivalent to Go's `GOMAXPROCS`. Each product row records its affinity; the origin and load generator keep native scheduling. Admission issuance reserves a large challenge allowance to measure successful operations rather than the production limiter's default exhaustion behavior. The comparison records that allowance explicitly. An idle CPU figure of zero over ten seconds means `ps` did not cross a whole CPU second; it does not prove that a node performed no background work.

Measurement scope Only local measurements are emitted. Third-party products are measured only when their binary can run on the host: Anubis can, SafeLine (Docker only) and Cloudflare (hosted) cannot, and both appear in a "not measured" table with the published facts that stand in for a measurement. Timers surround batches; state resets and warmup are untimed. Allocation counts are not instrumented: the allocation-free primitive contract is based on API and source review. No harness links into the daemon or adds a hook to the request path.

8.2 Primitive Latencies

Recorded 2026-10-03T08:28:22.541301+00:00 · paxos-zig · AMD Ryzen 9 5950X 16-Core Processor · Linux-7.0.0-28-generic-x86_64-with-glibc2.41 · revision `b1da948f7da9ba7dfd6b81b8005a9966dfd355ad` · Zig 0.17.0 · ReleaseFast · 7 batches, median with min-max spread

95.7 ns**Hashcash verify**

Hardware SHA-256, zero allocation

158.3 ns**Session token check**

Keyed BLAKE3, 75-character cookie

9.8 MB**Idle resident memory**

Daemon after start, storage off

33.9 MB**Daemon binary**

WASM solver 8716 bytes

Measured latency per operation

Only measured primitive latencies are shown. Allocation activity is not instrumented.

Workload	Median	Spread	ops/s
Hashcash verify (16 bits)	95.7 ns	95.1 ns – 97.2 ns	10,448,667
PoSW verify (depth 13, t = 16)	24.25 µs	22.8 µs – 26.46 µs	41,231
Bot signatures (40, Aho-Corasick)	75.1 ns	74.8 ns – 76.2 ns	13,309,623
IPv4 CIDR lookup	48.2 ns	47.9 ns – 53.9 ns	20,729,306
IPv6 CIDR lookup	107.6 ns	102.7 ns – 111.6 ns	9,297,147
Session token (keyed BLAKE3)	158.3 ns	157.7 ns – 159.1 ns	6,319,030
Session token (Ed25519)	51.16 µs	51.01 µs – 56.41 µs	19,547
Spent set (Robin Hood)	28.8 ns	28.7 ns – 39.7 ns	34,720,263
Rate limiter (GCRA)	5 ns	4.9 ns – 7.2 ns	199,690,679
HTTP parse + cookie	1.05 µs	1.04 µs – 1.05 µs	956,055
Classification, Gate profile	166.2 ns	165.2 ns – 169.6 ns	6,017,273
Classification, Shield profile	1.27 µs	1.26 µs – 1.29 µs	787,794
WAF body scan (8 KB)	19.14 µs	18.85 µs – 19.22 µs	52,244

For scale, the same 40 bot signatures scanned by sequential substring search on this host cost 1.03 µs against 75.1 ns for the automaton.

Measured latency on a logarithmic scale

Lower is better · log-10 scale · every bar is a measurement from the run above



8.2.1 Proof-of-Work Verification

Hashcash verification hashes a challenge, separator and decimal nonce. The sequential-work verifier checks graph openings; hash invocations and SHA-256 compression counts differ because inputs vary in length. The table gives current measurements without treating hardware results as mathematical constants.

8.2.2 Matching, State, and Memory

Dense Aho–Corasick and sequential substring search use identical patterns and any-match semantics. State benchmarks reset before every batch, so spent-set measurements include real insertions rather than duplicate rejection. Fixed-capacity structures need no allocator on these primitive APIs; the harness records unknown allocation counts rather than fabricating measurements. Engine and table byte counts come directly from `@sizeof`. Resident memory includes runtime and thread costs and is measured separately from the primitives.

8.3 Whole-Product Comparison

In this chapter Read the throughput, latency, CPU, and memory of Sibuna and Anubis as complete processes, understand the four workloads, and see the two design changes this measurement forced.

`python3 benchmarks/tools.py --anubis <binary>` starts an origin stub (caddy respond), then each product in turn, obtains a valid session by solving its challenge exactly as a browser would, and drives four workloads with `wrk` over keep-alive connections:

- **Admitted:** a valid session cookie on a protected path. In reverse-proxy mode the request reaches the origin and its answer is relayed; in forward-auth mode the product answers **200** itself.
- **Challenged:** no cookie and a browser User-Agent. The product serves its challenge page (**200** with HTML in reverse-proxy mode, **401** in forward-auth mode).
- **Allowed static path:** `/robots.txt` from a non-browser client, which both products admit without a challenge.
- **SQL injection with session:** a valid cookie plus `q=' OR 1=1--`. Only an inspecting product refuses it; the status column records what each product did.

Sibuna uses four accept threads (`--workers 4`) and Anubis uses `GOMAXPROCS=4`; these settings alone do not impose equal CPU budgets because Sibuna serves bounded connections on separate threads. On Linux the harness pins every product thread to the same four allowed logical CPUs before warmup and records the affinity. Other platforms retain native scheduling. Hashcash difficulty is matched (8 zero bits; 2 zero hex digits), with one fixed signing secret. Every figure is the median of three five-second runs.

Recorded 2026-10-02T01:34:49.465274+00:00 · paxos-zig · AMD Ryzen 9 5950X 16-Core Processor · Linux-7.0.0-28-generic-x86_64-with-glibc2.41 · revision 2e1a7f8d3835b79f94ae55c860cb53ca05370570 · wrk debian/4.1.0-4+b1 [epoll]
Copyright (C) 2012 Will Glozer · 2 threads, 64 connections, 5 s × 3 repetitions, median · origin: caddy respond (static 200) · Anubis v1.27.0

8.3.1 Forward-Auth Mode

The ingress asks the product whether to admit a request; there is no origin and no body relay, so this is the purest measure of admission cost.

Product	Workload	Status	req/s	p50	p99	CPU µs/req	Cores	Peak RSS
Sibuna Gate	Admitted (session)	200	192,235	164 µs	357 µs	13.5	2.59	28.5 MiB
Sibuna Gate	Challenged (no session)	401	181,567	175 µs	380 µs	14.3	2.59	30 MiB
Sibuna Gate	Allowed static path	200	198,253	160 µs	349 µs	12.1	2.4	28.5 MiB
Sibuna Gate	SQL injection with session	200	186,418	171 µs	359 µs	13.9	2.59	28.5 MiB
Sibuna Shield	Admitted (session)	200	181,000	177 µs	373 µs	14.6	2.79	28.9 MiB
Sibuna Shield	Challenged (no session)	401	180,924	176 µs	384 µs	14.5	2.59	30.1 MiB
Sibuna Shield	Allowed static path	200	200,595	158 µs	351 µs	12.5	2.55	28.9 MiB
Sibuna Shield	SQL injection with session	403	193,759	168 µs	359 µs	13.4	2.59	29.6 MiB
Anubis	Admitted (session)	200	35,879	1.65 ms	7.25 ms	107.9	3.79	32.1 MiB
Anubis	Challenged (no session)	401	113,901	506 µs	3.59 ms	33.3	3.79	31.1 MiB
Anubis	Allowed static path	200	93,836	607 µs	4.24 ms	40.1	3.73	33.2 MiB
Anubis	SQL injection with session	200	36,711	1.63 ms	7 ms	108.5	3.92	32.3 MiB

8.3.2 Reverse-Proxy Mode

The product sits in front of the origin stub and relays admitted requests and responses. Reverse-proxy rows therefore include the origin's own cost and one extra loopback hop.

Product	Workload	Status	req/s	p50	p99	CPU µs/req	Cores	Peak RSS
Sibuna Gate	Admitted (session)	200	94,847	554 µs	2.06 ms	36.2	3.39	29.8 MiB
Sibuna Gate	Challenged (no session)	200	196,893	163 µs	377 µs	14.2	2.6	30.3 MiB
Sibuna Gate	Allowed static path	200	100,774	503 µs	2.01 ms	33.4	3.39	29.8 MiB
Sibuna Gate	SQL injection with session	200	91,123	557 µs	1.99 ms	36.9	3.39	29.8 MiB
Sibuna Shield	Admitted (session)	200	91,216	573 µs	2.1 ms	37.6	3.4	30.1 MiB
Sibuna Shield	Challenged (no session)	200	215,739	143 µs	356 µs	13.9	2.99	30.3 MiB
Sibuna Shield	Allowed static path	200	100,108	521 µs	1.97 ms	34.2	3.39	30.1 MiB
Sibuna Shield	SQL injection with session	403	214,540	145 µs	326 µs	12.2	2.79	29.8 MiB
Anubis	Admitted (session)	200	17,807	3.43 ms	11.56 ms	213.2	3.79	35.3 MiB
Anubis	Challenged (no session)	200	29,288	2.03 ms	9.39 ms	140.6	3.99	506.9 MiB
Anubis	Allowed static path	200	28,087	2.03 ms	18.11 ms	134.9	3.79	593.1 MiB
Anubis	SQL injection with session	200	20,700	2.87 ms	13.32 ms	189.4	3.92	589.7 MiB

8.3.3 Footprint

Product	Binary	Idle RSS	RSS after workloads
Sibuna Gate	33,766 KiB	10.2 MiB	27 MiB
Sibuna Shield	33,766 KiB	10.4 MiB	27.1 MiB
Anubis	39,064 KiB	21.2 MiB	30.7 MiB

8.3.4 What the Measurement Changed

The first run of this harness measured a version of Sibuna in which each accept thread served one connection to completion and every proxied response closed the client connection. Under 64 concurrent connections the throughput looked healthy but the 99th-percentile latency was over 130 ms, because sixty connections waited for four workers, and the reverse-proxy runs exhausted the host's ephemeral ports with sockets in `TIME_WAIT`. Neither defect was visible in the primitive suite or in the two-client admission harness. Three changes followed, all described in Part V: a bounded thread per connection with a 503 overload path, origin response framing so proxied connections stay open, and a pooled origin connection (the second run, with framing but a fresh origin connection per request, managed about 1,400 proxied requests per second before exhausting ephemeral ports). The tables above are from the run after those changes; the superseded runs are not retained as result files, which is why their figures appear only in this paragraph, labelled as such.

A later run pinned every product to the same four CPUs for an equal budget. It showed Sibuna's reverse-proxy 99th percentile at 35–45 ms against Anubis's 11 ms, with occasional stalls of over 100 ms. The cause was the locks on the request path: the origin pool, the rate

limiter's shard, the idle table and the origin attachment were all spinlocks. Every benchmark request comes from one address, so they all hit one rate-limiter shard. With about seventy connection threads on four CPUs, a thread could be preempted while holding a lock, and the waiters then spun through their time slices while the holder waited behind them. These locks now spin briefly and then sleep on a futex (`core.Lock`), so a preempted holder gets its CPU back.

The same change enabled `TCP_NODELAY` on client and origin sockets. Without it, an origin that flushes its head before its body costs every response a delayed acknowledgement, about 40 ms on Linux. The tables above are from the run after both changes. A separate A/B run measured the reverse proxy at an equal open-loop rate of 20,000 requests per second with `wrk2`. The 99th percentile was 17–33 ms before the change, 1.9 ms after it, and 5.7 ms for Anubis. Those equal-rate figures are not in a result file, which is why they appear only in this paragraph.

Reading the Anubis rows Anubis is a capable, widely deployed product and this is not a claim that it is slow. It runs a garbage-collected runtime, verifies an Ed25519 signature per session check, and keeps challenge state in a store; those are design choices with benefits this harness does not measure, such as a smaller dependency on the host's threading model. The rows show what a request costs each product on one host under one load shape. Its reverse-proxy rows are far below its forward-auth rows, with peak memory in the hundreds of megabytes; the most likely cause is origin-connection churn under 64 concurrent clients (Go's HTTP transport keeps only two idle connections per host by default, and Anubis was run with its default flags), but the harness did not confirm that and records only what it observed.

8.3.5 Not Measured

Product	Why it is not measured here	Published facts a reader can check
safeline-ce 9.x community edition (chaitin/SafeLine, September 2026)	Ships only as a Docker Compose stack (engine, detector, mgt, luigi, fvm, chaos, postgres); no Docker daemon on this host and the detector image is not an open-source build that can be compiled here.	minimum host: Linux x86_64 or arm64 with SSSE3, 1 CPU core, 1 GB RAM, 5 GB disk, Docker 20.10.14+ and Compose 2.0+ false positive rate: 0.07% (balance) / 0.22% (strict), vendor-reported detection rate: 71.65% (balance) / 76.17% (strict), vendor-reported
cloudflare-waf developers.cloudflare.com/ waf, September 2026	Hosted service on Cloudflare's network; it cannot be installed on a host, and any loopback measurement would measure the network path, not the WAF.	custom rules: Free 5, Pro 20, Business 100, Enterprise 1000 rate limiting rules: Free 1, Pro 2, Business 5, Enterprise 100 managed rulesets: Free Managed Ruleset on all plans; Cloudflare Managed and OWASP Core rulesets from Pro; Sensitive Data Detection on Enterprise

A row in this table is not a claim about the product's performance. SafeLine's own numbers concern detection quality, not throughput; Cloudflare publishes plan quotas rather than per-request costs. Anyone with a Docker host can run SafeLine behind the same `wrk` workloads; the harness accepts any product that can be started as a process and solved as a browser.

8.4 Admission Comparison

`python3 benchmarks/compare.py --anubis <binary>` measures admission operations with two Python clients against forward-auth endpoints, so its absolute numbers are limited by the clients rather than the products. It exists to compare the **shape** of the four operations across products and token schemes: session check, unauthenticated check, challenge bootstrap (interstitial plus challenge record), and verification of a fresh proof prepared outside the timed batch.

Product (token)	Session check ops/s	Unauthenticated ops/s	Bootstrap ops/s	Proof verification ops/s	RSS
sibuna (blake3)	23,016	22,966	12,228	19,339	9.9 MiB
Anubis (ed25519)	10,202	18,419	7,964	11,017	28.8 MiB
Anubis (hs512)	13,777	18,661	8,239	13,055	29.7 MiB

8.5 A Cluster of Three

In this chapter Answer the operator's question directly: does a three-node Sibuna cluster keep the security properties, throughput, and latency of one node, and what do replication and storage cost in CPU and memory?

`python3 benchmarks/cluster.py` runs four cases with the same Shield forward-auth configuration and two workers per node: one node without storage, one node with the embedded database, and three replicated nodes over a loopback pre-shared key and then over mutual TLS with a temporary certificate authority. For each case `wrk` drives node 1 alone and then all nodes at once (one load generator per node, requests summed, CPU summed over nodes, latency the worst of the three). Before any load the harness measures ten idle seconds, so the cost of consensus heartbeats and storage polling appears as a percentage of one core per node.

Recorded 2026-10-01T17:04:05.435533+00:00 · paxos-zig · AMD Ryzen 9 5950X 16-Core Processor · revision 54f2e6a37f97e69fcaec3341e58d0fe539d45014 · wrk debian/4.1.0-4+b1 [epoll] Copyright (C) 2012 Will Glozer · 2 threads, 32 connections per node, 4 s × 2 repetitions, median · two workers per node · Shield, forward auth

Case	Load on	Workload	req/s	p99	CPU µs/req	Cores	Peak RSS per node
single, no storage	node 1 alone	Admitted (session)	228,129	155 µs	15	3.41	21.6 MiB
single, no storage	node 1 alone	Challenged	230,294	164 µs	15	3.46	21.3 MiB
single, no storage	node 1 alone	SQL injection, session	235,878	157 µs	14	3.29	21.4 MiB
single, embedded storage	node 1 alone	Admitted (session)	225,436	159 µs	15.2	3.41	38.1 MiB
single, embedded storage	node 1 alone	Challenged	228,128	158 µs	14.6	3.33	37.8 MiB
single, embedded storage	node 1 alone	SQL injection, session	232,497	162 µs	14.3	3.33	39.4 MiB
cluster of 3, PSK	node 1 alone	Admitted (session)	219,730	160 µs	15.9	3.5	40.7 MiB / 31.4 MiB
cluster of 3, PSK	node 1 alone	Challenged	229,247	168 µs	14.7	3.37	40.4 MiB / 31.4 MiB

Case	Load on	Workload	req/s	p99	CPU µs/req	Cores	Peak node	RSS	per
cluster of 3, PSK	node 1 alone	SQL injection, session	226,962	165 µs	14.7	3.34	40.9 MiB / 31.6 MiB	37.1 MiB	
cluster of 3, PSK	all nodes at once	Admitted (session)	550,389	236 µs	19.5	10.73	41.2 MiB / 41 MiB	47.1 MiB	
cluster of 3, PSK	all nodes at once	Challenged	566,299	205 µs	17.2	9.76	40.8 MiB / 40.6 MiB	46.7 MiB	
cluster of 3, PSK	all nodes at once	SQL injection, session	548,272	266 µs	18.2	10	41.3 MiB / 41.3 MiB	53.4 MiB	
cluster of 3, mutual TLS	node 1 alone	Admitted (session)	219,521	170 µs	16.3	3.58	44.9 MiB / 35.8 MiB	40.4 MiB	
cluster of 3, mutual TLS	node 1 alone	Challenged	224,891	172 µs	15.4	3.46	44.5 MiB / 35.8 MiB	40.5 MiB	
cluster of 3, mutual TLS	node 1 alone	SQL injection, session	226,811	159 µs	14.7	3.34	45 MiB / 36 MiB	44.2 MiB	
cluster of 3, mutual TLS	all nodes at once	Admitted (session)	546,936	222 µs	19	10.37	45.3 MiB / 45.4 MiB	54.2 MiB	
cluster of 3, mutual TLS	all nodes at once	Challenged	555,768	221 µs	17.3	9.63	44.9 MiB / 45.1 MiB	53.8 MiB	
cluster of 3, mutual TLS	all nodes at once	SQL injection, session	561,462	249 µs	18.7	10.49	45.3 MiB / 45.7 MiB	59.4 MiB	

The security checks run on every case before the load: a session minted by node 1 is accepted by every node (shared seed); a request carrying that valid session but an SQL-injection query is refused by every node; a solution already spent on node 1 is rejected when replayed to node 2 (challenges are issuer-bound); a honeypot hit on node 1 bans the address on the other nodes within the propagation time shown; and after the elected leader is stopped the survivors keep admitting requests and still propagate a fresh ban.

Case	Idle CPU per node	Idle RSS per node	Session on every node	WAF denies with session	Replay on other node rejected	Ban propagation	Leader stopped	Storage log clean
single, no storage	0 %	12.3 MiB	passed	passed	-	-	-	passed
single, embedded storage	0 %	27.7 MiB	passed	passed	-	-	-	passed
cluster of 3, PSK	0 / 0 / 0 %	30.9 MiB / 31 MiB / 32.4 MiB	passed	passed	passed	104 ms	node 3 stopped: 384,603 req/s, ban 62 ms	passed
cluster of 3, mutual TLS	0 / 0 / 0 %	35.1 MiB / 35.4 MiB / 37.5 MiB	passed	passed	passed	83 ms	node 3 stopped: 397,873 req/s, ban 82 ms	passed

What the cluster costs Per-node throughput and tail latency in the cluster rows should be read against the single-node rows in the same table, not against the four-worker figures earlier in this part. The differences that matter to an operator are the idle CPU and resident memory columns, which are what replication and the embedded database add to a quiet node, and the all-nodes rows. Three daemons and three load generators share the benchmark host's CPU, memory and loopback network; the aggregate does not

establish throughput on three separate hosts. CPU microseconds per request and idle costs also depend on that host and the accounting resolution. Read the measured rows and their provenance without assuming linear scaling or identical CPU cost across transports. Cross-host acceptance additionally tests the actual network, peer authentication, failover and quorum loss.

8.6 Distributed Measurement

`python3 benchmarks/distributed.py` starts three real daemons, first Gate and Shield without replication, then Shield with replicated storage over a loopback pre-shared key and over mutual TLS. Six external client processes generate keep-alive forward-auth requests. Seven batches report throughput including client and loopback costs; response statuses are checked. Session portability, WAF denial with a valid session, issuer-bound challenge rejection, replicated ban propagation, and serving after the elected leader is stopped are checked separately and appear in the last columns.

Profile	Replicated	Nodes × workers	Admitted req/s	Challenged req/s	RSS per node	Ban propagation	req/s, one node down	Checks
gate	No	3 × 2	62,167	63,037	11.9 MiB / 11.8 MiB	-	-	passed
shield	No	3 × 2	61,982	66,037	11.8 MiB / 11.9 MiB	-	-	passed
shield	Yes, loopback PSK	3 × 2	61,325	69,497	29.6 MiB / 32 MiB	124 ms	46,180	passed
shield	Yes, loopback mTLS	3 × 2	68,891	68,310	34.4 MiB / 36.8 MiB	166 ms	46,091	passed

The harness adds no per-request instrumentation to Sibuna. It does not remove the daemon's production metrics, locks, snapshot atomics or incident enqueue costs. WAN behaviour, global quotas, durable replay state, reverse-proxy origin latency and sustained overload remain outside these measurements.

Exercise 8.1. Run `sh benchmarks/run-all.sh` on your machine, rebuild the book, and compare the Hashcash verification and token verification rows with the reference host. Which of the two depends most on hardware hash instructions, and why?

Exercise 8.2. In the forward-auth table, divide CPU microseconds per request by the number of cores busy and compare with the measured latency. Why is the per-request CPU cost higher than the primitive classification cost from the first table, and which components account for the difference?

Hint: List everything between `accept` and the flushed response that the primitive suite does not time.

Explain the invariant. Explain the difference between a median-of-batches figure and a per-request p99, why the primitive table reports the former and the product table the latter, and what each can and cannot reveal about a tail-latency defect.

9 Operations and Deployment

We cover the two surfaces, every command-line flag, forward-auth recipes, policy files, persistent storage and clustering with Zaxonlite, and packaging.

9.1 Surfaces and Configuration

9.1.1 Release Packages and Licenses

Version 0.2.0 packages include persistent storage, the browser solver and the optional management console. Linux x86-64 and ARM64 packages link musl statically; macOS packages cover Intel and Apple Silicon, require macOS 15 or later, and are unsigned. Windows packages contain a native x86-64 executable for Windows 10 / Server 2019 or later. Use Ctrl+C for ordered shutdown and restrict credential and data files with Windows ACLs. Clustering requires a separate `-Dcluster=true` source build with OpenSSL 3.

Download from [GitHub Releases](#), verify the archive against SHA256SUMS, extract it and run `sibuna --version`. Each package contains license texts, dependency notices and a `sibuna.build.json` manifest identifying the commit, target, compiler, build options and executable digest. The release workflow tests the actual packaged executables before publication; these checks do not establish performance acceptance.

The engine is LGPL-3.0. The console, including its WebAssembly interface, is AGPL-3.0; the default combined executable is distributed under AGPL-3.0. Build the engine without the console using `-Dconsole=false`. Directory boundaries and third-party exceptions are in `LICENSE` and `NOTICE`; full terms are in `LICENSES/`. Every release tag includes the corresponding source and build scripts, and the console links to that source. Companies seeking a version under terms other than LGPL or AGPL can contact the authors, Vikrant Rathore and Ronak Rathore, about alternative licensing. Third-party libraries remain subject to their respective licenses.

In this chapter By the end of this chapter, you should be able to run Sibuna as a reverse proxy or a forward-auth validator, choose a surface, set the proof-of-work tier and difficulty, and manage the master secret.

9.1.2 Choosing a Surface

- **Gate** (`--gate`, alias `--no-waf`): proof-of-work admission, sessions, declarative rules, reputation, GCRA limits, bans.
- **Shield** (default, `--shield`): Gate plus the semantic firewall.
- **Distributed deployment** (an option for either surface): `--data-dir` (persistent policies, reputation, forensics) and, with a cluster build, `--cluster-*` flags for replication.

9.1.3 Command-Line Reference

Flag	Default	Meaning
<code>--port, -p</code>	8080	Listening port
<code>--host, -h</code>	0.0.0.0	Listening address
<code>--upstream-host</code>	127.0.0.1	Origin host (reverse proxy mode)
<code>--upstream-port, -u</code>	3000	Origin port
<code>--mode, -m</code>	reverse_proxy	reverse_proxy or forward_auth
<code>--workers, -w</code>	CPU count	Accept threads sharing the listening socket; each connection then gets its own thread
<code>--max-connections</code>	1024	Connections served concurrently; further ones are refused with a bounded 503 reply
<code>--idle-timeout</code>	15	Longest silence in seconds. A request head must arrive within it. An origin response refreshes it on every read, so a slow stream lives while bytes flow and a silent origin is cut on both sockets. A proxied upload must deliver 16 KiB per period (a minimum rate against slow-body attacks)
<code>--trust-forwarded</code>	off; on in forward-auth	Honour X-Forwarded-For / X-Real-IP from the peer
<code>--algorithm, -a</code>	posw	posw or hashcash
<code>--difficulty, -d</code>	16	Work bits: Hashcash zero bits, or PoSW depth plus three
<code>--posw-challenges</code>	16	Openings per sequential-work proof
<code>--token-scheme</code>	mac	mac (keyed BLAKE3) or ed25519
<code>--token-ttl</code>	86400	Session lifetime, seconds
<code>--challenge-ttl</code>	300	Challenge lifetime, seconds
<code>--secret-file, -s</code>	random	64 hex characters or 32 raw bytes; SIBUNA_SECRET also accepted
<code>--cookie-name</code>	__sibuna_token	Session cookie name
<code>--secure-cookie</code>	off	Add the Secure attribute
<code>--gate / --shield</code>	shield	Surface selection

Flag	Default	Meaning
<code>--rate-limit</code>	100	Requests per window per client (GCRA burst)
<code>--rate-window</code>	10	Window in seconds
<code>--challenge-rate-limit</code>	30	Challenge issuances and verifications per window per client, separate from the request budget
<code>--ban-seconds</code>	3600	Honeypot ban duration
<code>--policy-file, -P</code>	none	Declarative JSON policy
<code>--data-dir, -D</code>	none	Zaxonlite data directory; enables persistent storage
<code>--cluster-node</code>	0	This node's id; non-zero enables replication
<code>--cluster-listen</code>	none	This node's <code>host:port</code> for peers
<code>--cluster-peer</code>	none	<code>id@host:port[/role]</code> , repeatable
<code>--cluster-secret-file</code>	none	Shared PSK for loopback development clusters
<code>--cluster-tls-cert/key/ca</code>	none	Mutual TLS identity for production clusters
<code>--storage-poll-ms</code>	500	Storage thread cadence
<code>--verbose, -v</code>	off	Verbose logging

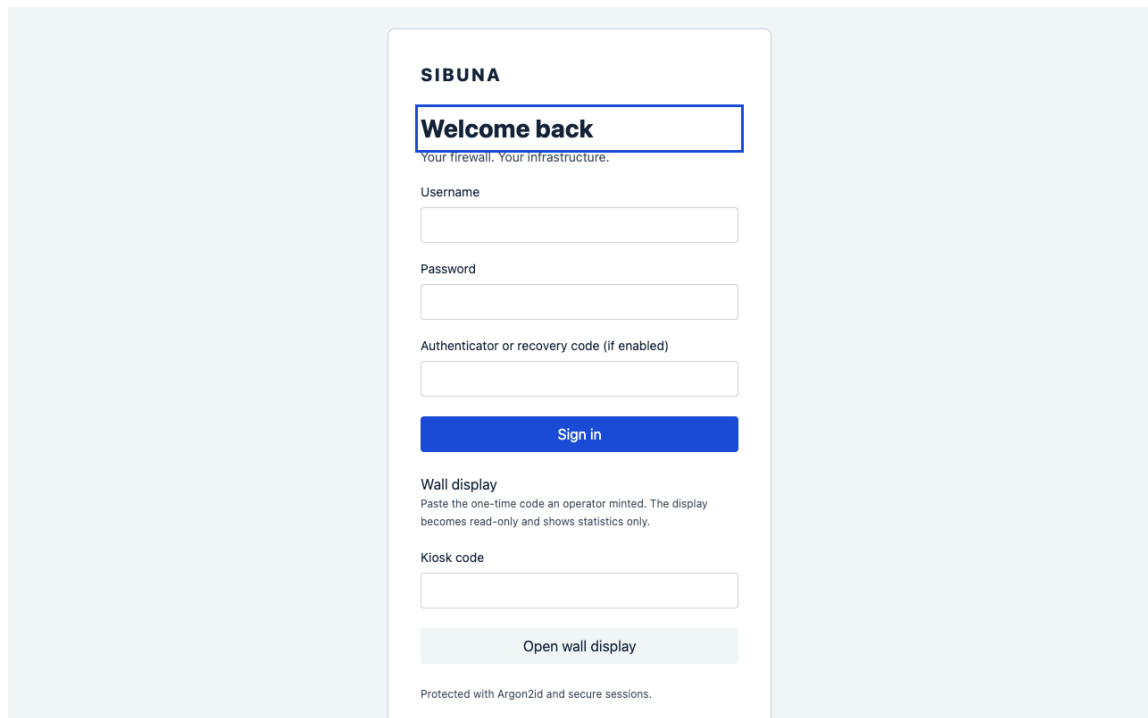
The master secret Without `--secret-file` or `SIBUNA_SECRET` the daemon draws a random seed and prints "random (set `--secret-file`)". Tokens and challenges then die with the process, and a cluster whose nodes hold different seeds will reject each other's cookies. Generate one with `head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret` and mode 0600.

9.2 Console Preview

The console is composed into the daemon when storage is compiled in, but starts only with `--console`. Use `-Dconsole=false` to compile it out; `storage-off` builds default it off too. Bootstrap locally while the daemon is stopped:

```
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
```

Open `http://127.0.0.1:19446/console/` and replace the generated temporary password. The signed-in navigation provides the animated country globe, request and challenge statistics, incident investigation, policy and inspection editors, users, scoped API tokens, audit and



SIBUNA

Welcome back

Your firewall. Your infrastructure.

Username

Password

Authenticator or recovery code (if enabled)

Sign in

Wall display

Paste the one-time code an operator minted. The display becomes read-only and shows statistics only.

Kiosk code

Open wall display

Protected with Argon2id and secure sessions.

Figure 9: Sign-in. The public shell loads no globe geometry, GeoIP data or telemetry; the second form exchanges a wall-display code for a read-only statistics session.

serving-node controls. The authentication shell loads no globe geometry or telemetry. Committed CSS and the Zig/Wasm interface ship with ordinary builds; npm is needed only when regenerating style assets.

9.2.1 A Tour of the Console

The screenshots below come from the built console on one loopback node fed with synthetic traffic (addresses from documentation ranges, a honeypot and injection probes), so every number is illustrative. Each page answers one question stated in its title and carries the product, the serving node, the page and the way back in the same place.

The optional `--console-location <latitude,longitude>` declares this node's position, for example `1.3521,103.8198` for a deployment in Singapore. The globe initially centers there; Center Sibuna returns to that point after rotation. Country activity follows animated great-circle arcs toward the node, with rear-hemisphere and flat-map dateline clipping. Country positions are representative centroids, and arrows represent the observed sixty-second window rather than individual connections. An unset server position stays explicitly unknown. The configured marker remains visible through telemetry outages; stale traffic does not animate. Policy previews build a private candidate including configured file rules. Saves compare the expected revision and require confirmation of a field comparison. Historical reverts compare against the current rule and create a new revision. Committed and locally applied revisions are distinct. Audit details retain bounded decision changes and the effective acting role. Matcher values are redacted and old records with missing context remain explicitly absent. Drain,

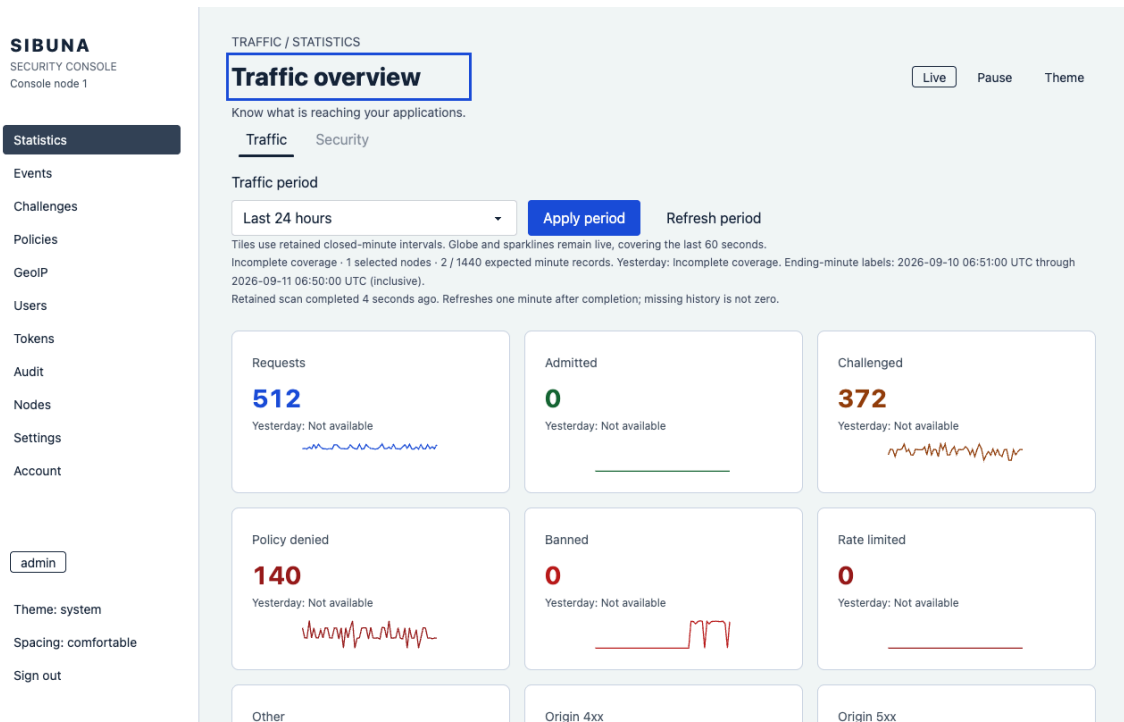


Figure 10: Statistics · Traffic. Tiles show retained closed-minute counts with a deviation against the same window yesterday and a live 60-second sparkline; decision colours (admitted green, challenged amber, denied red, banned dark red) are the same in every panel. Two further tiles are levels rather than counts: active ban entries on contributing nodes and nodes healthy from this console's own probes, each with a sparkline of observed snapshots. Below the timeline, sampled panels rank request paths and referring hosts and count client operating systems, browsers and response status over the same one-in-64 samples; retained rankings compare the same dimensions across windows.

resume and clear-local-bans require a command preview. Inspect the durable receipt after a lost response before retrying; a committed intent alone does not establish that the runtime effect finished.

For remote operation, use an HTTPS proxy and configure `--console-origin`, `--console-behind-proxy`, explicit `--console-trusted-proxy` CIDRs, and `--console-key-file`. The key file contains 64 hexadecimal characters, has owner-only permissions and protects stored second-factor secrets; retain it across restarts separately from the challenge seed. The configured HTTPS origin and proxy allowlist are mandatory for off-loopback access.

The native CLI accesses the running console through the same authorization and storage contracts. It reads credentials from private files, not command-line values:

```
sibuna console geoip update --version 2026-09-09 \
  --origin http://127.0.0.1:19446 --username admin \
  --password-file ./admin-password
```

Use an owner-only password file, complete password setup first, and add `--factor-file` for an authenticator or recovery code when required. `geoip status` reads the active generation. Updates download the selected provider's published files over HTTPS (the public-domain user-country dataset by default, or DB-IP Lite with `--provider dbip --version YYYY-MM`),

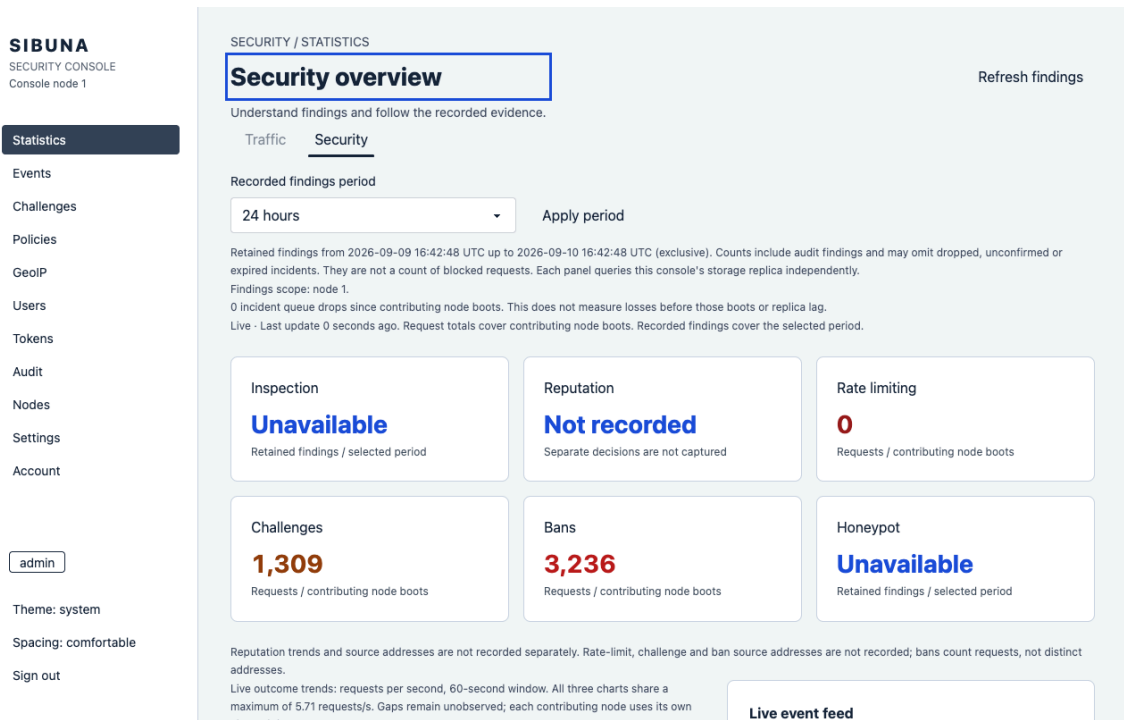


Figure 11: Statistics · Security. One tile per module, findings over sixty equal buckets, the live event feed, attack categories and attacked paths; values that are not recorded say so instead of showing zero.

verify the publisher's checksums, validate bounded ranges through `libs/geoip`, persist the new generation and activate it locally. Failed updates preserve the previous generation. A CLI timeout stops waiting; it does not cancel submitted storage work. DB-IP Lite requires CC BY 4.0 attribution, shown only while its data is active. A build with `-Dgeoip-data` embeds a validated snapshot until the first durable import. Unknown addresses, sample loss and stale data remain visible; importing country data does not create traffic or enrich already expired samples.

Cluster members announce themselves through the replicated membership table and appear on every console's Nodes page; configure `--console-probe <node-id>=<http://ip:port>` for the peers whose data-plane listeners this console should health-check and `--console-advertise <origin>` for the link other consoles show. Direct live telemetry uses `--console-peer <node-id>=<https://origin>` (repeatable, at most eight) and a dedicated `--console-peer-key-file`. Both endpoints must list each other and use trusted HTTPS ingress. The peer key is an owner-only file containing 64 hex characters, provisioned independently of console encryption, challenge and consensus keys. `--console-peer-ca-file` optionally supplies PEM trust anchors for a private management PKI; otherwise the client uses system roots. Certificate hostname validation always applies. Use a DNS name in each management origin, resolvable by its peers and listed in the certificate's DNS subject alternative names. The pinned Zig 0.17 verifier also supports IP subject alternative names for numeric-IP origins. An address must match the certificate's IP alternative name; DNS names must match a DNS alternative name. Private certificates must also have the normal CA constraints, key usages and authority identifiers.

SIBUNASECURITY CONSOLE
Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

Country at persistence

Not recorded

Response

Local denial: status 403 selected; no origin response. Delivery is not recorded.

Matched rule and score terms

Not recorded

JA4 fingerprint

Not recorded: requires bounded capture at a trusted ingress that overwrites spoofed headers.

Evidence metadata v1

Firewall response selected: 403. Delivery is not recorded.

Query: 18 bytes. Received body: 0 bytes. Declared body: 0 bytes.

Query values, body contents, cookies and other headers are omitted from this evidence view. This is not a reconstructed request.

No capture truncation recorded.

Request · Response

Redacted before capture: credential headers and every query value read [redacted]. Rendered as UTF-8 (invalid bytes shown as U+FFFD). UTF-8 Latin-1

Request head

GET /__sibuna/honeypot?token=[redacted] HTTP/1.1

Host: 127.0.0.1:18080

Accept-Encoding: identity

Origin: http://127.0.0.1:18080

X-Forwarded-For: 13.107.4.143

User-Agent: Mozilla/5.0 (X11; Linux x86_64) Gecko/20100101 Firefox/129.0

Referer: https://www.google.com/

Response head

Local response: no origin head; the selected status is shown above.

Copy as cURL

```
curl -X 'GET' 'http://127.0.0.1:18080/__sibuna/honeypot?token=[redacted]' \
-H 'Host: 127.0.0.1:18080' \
-H 'Accept-Encoding: identity' \
-H 'Origin: http://127.0.0.1:18080' \
```

Figure 12: Events. Each recorded incident opens inline with its evidence: the selected local response, byte lengths, campaign candidate, and explicit “not recorded” entries for the matched rule, score terms and JA4 fingerprint. With `--console-capture-heads` the redacted request head (and, for audited admissions through the reverse proxy, the origin response head) opens on request, rendered as UTF-8 or Latin-1, with the response condition stated (captured, local, forward-auth unobserved, origin unavailable) and a copy-as-cURL command; only listed header values are kept, and `--console-capture-header <name>` adds to that list. Otherwise heads read as not recorded. Deny and allow actions open the IP groups form with the address drafted.

Key rotation requires coordinated restart. The Nodes API and its subscription report receipt age, clock skew, boot changes and sampling loss. Missing observations remain unavailable and disconnected values remain stale; received statistics never become another node's own contribution. The dashboard's **Live traffic scope** selects one node or combines the configured nodes. **Node coverage and locations** identifies missing, stale and clock-skewed sources, observation times and configured destinations. Country rankings show the uncertainty from omitted source rows. Arrows retain their receiving node, and missing node locations are not invented. The combined rate stays unobserved when any contributing source lacks a consecutive interval. Select one node to inspect its retained seconds and current-minute path rankings. Remote queries use the authenticated peer connection; missing peers stay unavailable and history cursors cannot cross a node restart. Minute history keeps its own node selection and per-node rows. The navigation and mobile header identify the serving console node independently of the selected traffic source. Local commands remain per node. `zig build console-impact` runs the data-plane isolation matrix (`-- --quick` for a smoke run, `-- --mode reverse_proxy` against a local origin, `-- --capture-heads` to store heads on the enabled consoles and add an audited-admission workload) and writes `benchmarks/results/console-impact-latest.json` with the mode and capture setting in its provenance.

SIBUNASECURITY CONSOLE
Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

SINGLE NODE / PROOF VERIFICATION

Challenges[Back to dashboard](#)

Understand proof issuance and verification.

Refresh observations

Retained window

Window

Last 24 hours

Load window

2 recorded minutes (1 complete) of 1440 in the window; unrecorded minutes are not zero activity. Retention 90 days. Received 2 seconds ago.

Window flow

Issued 0

Submitted 0

Accepted 0

Rejected 0

Sums of per-minute deltas across contributing nodes and boots. Retries and cross-window solutions prevent a cohort conversion rate.

Rejection causes

Address banned 0

Body too large 0

Missing challenge ID 0

Malformed solution 0

Malformed challenge 0

Invalid challenge tag 0

Expired challenge 0

Fingerprint mismatch 0

Figure 13: Challenges. Issued, submitted, accepted and rejected counts by cause, the configured and most recently issued parameters, and the accepted solve-time histogram partitioned by algorithm and parameter bin. A retained window (one hour, one day or seven days) sums durable per-minute records and states how many minutes were recorded and complete; live totals since this boot stay separate. Below them, adaptive-difficulty transitions and per-address records (with deny and allow shortcuts) cover the same window, retained seven days.

Traffic tiles open on the last 24 hours of retained closed-minute records for the selected nodes. The period control also offers an hour, seven or ninety days, and live boot totals. Yesterday deviations require complete matching coverage from every selected node. The globe and sparklines still describe their separate live 60-second window. A retained scan refreshes one minute after completion, keeping its previous values and age visible until replacement; changing the source or period discards the old scope. Missing history never becomes zero.

Compare retained traffic displays two closed minute windows beside each other: the same window yesterday, the preceding period, or a second node. Duration and end-offset controls freeze both UTC ranges. Each action reads at most sixteen compact pages per side, with 96 stored records per page. This covers a 24-hour window without overlapping restarts in one batch; Continue retains the same boundaries for longer scans. Complete coverage, incomplete coverage and missing history remain explicit. Rate percentages require complete non-overlapping intervals in equal UTC windows, normalized by observed milliseconds; a zero reference with new traffic shows “New”. The primary Traffic period remains independent. The minute format does not record historical proxy mode, so origin-response comparisons remain unavailable rather than interpreting forward-auth zeros as observed responses.

Compare retained path rankings selects two closed windows or nodes. Node 0 includes all retained nodes, including retired members. An action reads at most sixteen immutable archives,

SIBUNASECURITY CONSOLE
Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

POLICY / APPLIED ENGINE

Policies

Evaluate a request

Refresh policies

Managed rules

Inspect the rules this node currently evaluates.

Live - Last update 1 seconds ago.

Current committed revision 600; applied revision 600. Refresh to review newer policy. An open draft keeps its reviewed revision.

Applied configuration

Surface: Shield (challenge plus semantic inspection)

Committed revision 600 - Applied revision 600 - 7 rules

Inspection: Enabled - Default algorithm: posw

Configured default difficulty: 16. This is not a measured solve time.

Node-local limits: 100 requests per 10 s per client address; honeypot ban 3600 s. Per-rule limits below override the global limiter on this node.

Rules include database configuration and file or built-in fallbacks. Inspection and terminal reputation decisions run before this ordered rule list. Header and CIDR matcher values are omitted from this summary; their counts are shown.

1. well-known

allow

Path: ^/well-known/*\$

User agent: Any user agent

0 header matchers - 0 address matchers

Algorithm: Inherit - WEIGH score: 0

Difficulty: inherit

Recorded hits today (UTC)

Incomplete

Snapshot: 2026-09-10 16:42:59 UTC. Refresh policies for newer observations.

Figure 14: Policies. The applied engine names its surface and node-local limits, then lists rules in evaluation order with hits today, followed by the inspection mode matrix, the request tester and IP groups.

checks their identities and checksums, and merges every counter before selecting display rows. Archives seal after the 60-second late-sample window, so the newest closed minute may still be pending. Continue preserves the original windows and cursor. Bounds describe sampled path prefixes, not exact request totals. The page shows partial scans, truncation, retention

Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

POLICY / MANAGEMENT

Policies

Applied rules

Managed rules

Manage rules, preview decisions and review previous revisions.

Live - Last update 0 seconds ago.

Current committed revision 2; applied revision 2. Refresh to review newer policy. An open draft keeps its reviewed revision.

Rule editor

Based on committed revision 2. Preview runs privately. Save creates a revision; this node applies it on its next successful policy reload. Disable a rule to stop enforcing it while keeping its history.

Reviewing historical revision 2. Saving creates a new revision from these fields.

Review policy changes

Existing rule - Based on committed revision 2.

Only changed fields appear below. Confirming validates the complete candidate and creates a new revision. The node applies it after a successful reload. Matcher values belong to this private editor and are omitted from audit summaries.

Field	Before	After
Path pattern	/internal	/internal/*

Back to editor

Confirm save

Figure 15: Reviewing a rule edit before saving (dark theme). Only changed fields are listed; confirming validates the whole candidate and creates a new revision.

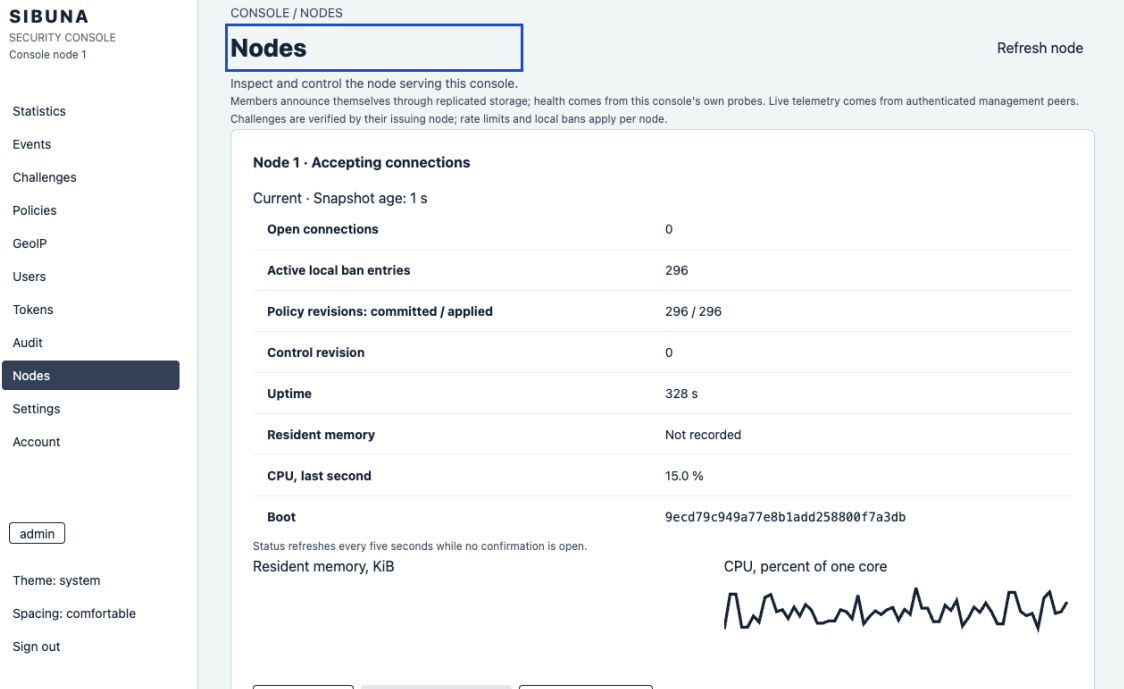


Figure 16: Nodes. The serving node's status and commands, with resident memory and CPU of the last second and their sixty-second sparklines, then every announced member with its applied revision, log slots, replication lag and probe result.

boundaries and reported queue loss; missing archives cannot establish zero traffic or complete coverage.

Security opens on the last 24 hours and uses sixty equal time buckets over the selected period. Its aggregates run on the storage owner under a fixed SQLite step budget (`--console-query-steps`, 100,000 to 50,000,000, default four million). Set it alongside `--console`; invalid or repeated values are refused. This allowance applies to local embedded queries. Cluster queries use Zaxonline's server-side ten-million-step bound instead; changing this option does not change that RPC limit. Exhausted queries ask for a narrower period. Step counts bound query work, not elapsed time. Charts share a scale; expand Trend values for the UTC interval starts and exact grouped counts. These are retained findings, so missing incident coverage cannot be interpreted as absence of attacks.

Wall displays use kiosk sessions. Under Account, an operator names the display and selects Create display code. The display pastes the one-time code into the sign-in form within ten minutes; its session is read-only, limited to statistics and expires within twelve hours. The account page erases the displayed code on navigation or when hidden. Hiding does not revoke an unused grant; the exchange deadline still applies. Codes never belong in URLs. Traffic and Security share the display: Security shows aggregate module trends and request outcomes without incident addresses or payload evidence. Automatic cycling is optional, off initially and suspended with reduced motion, stale data or Pause.

Figure 17: GeoIP. Provider, licence, active generation digest, load time and the import form; a failed import never replaces the active generation.

Notification destinations (signed webhooks and syslog for denial spikes, bans, unreachable members and leader changes) live under Settings; webhook secrets need `--console-key-file`. Each destination has its own cooldown and three-attempt retry budget. Delivery audit records show outcomes; completed queue history is bounded to seven days and 4,096 events. Webhooks include a stable `Idempotency-Key` so receivers can suppress repeated effects after uncertain network completion. Syslog's UDP/TCP selection applies only to outbound notifications, independently of protected web traffic. Manual tests record intent and completion in Audit and refresh the destination outcome; an unconfirmed audit completion is shown explicitly before an operator retries.

Settings also controls retention: one to 90 days for minute history, one to seven for rankings, one to 30 for incidents and one to 365 for audit. The upper values are the defaults; rankings keep their 512 MiB quota. Saving a retention value requires confirmation because cleanup permanently removes older records in bounded batches. Increasing the value later does not restore deleted history. Stale edits are refused and successful changes appear in Audit.

Response pages (challenge, denied, rate limited, banned, overloaded) are editable under Settings as bounded HTML with fixed placeholders; drafts preview in a sandboxed tab and saved pages are served from the next policy snapshot.

Policy workflows on the Policies page: reorder managed rules, replay a draft against retained inspection findings, manage IP groups and country blocks pinned to the active GeoIP generation, and export or atomically import the managed set (also `sibuna console policies export` and `sibuna console policies import --file <set.json>`). Applied rules show recorded

SIBUNA
 SECURITY CONSOLE
 Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

CONSOLE / SETTINGS

Settings

Refresh

Notifications, retention and response pages.

Notification destinations

At most eight destinations. Webhooks receive a JSON body with an HMAC-SHA256 signature when a secret is set; syslog receives RFC 5424 lines over UDP or framed TCP, selected explicitly below. Secrets are stored sealed and never shown again.

Label	Kind	Target	Events	Last outcome
-------	------	--------	--------	--------------

New destination

Next page

New destination

Kind

Webhook (HTTPS POST)

Syslog transport

UDP datagrams

For outbound syslog notifications only. Webhooks use HTTPS (local HTTP for testing). This does not change protected web traffic.

Label

Target

Figure 18: Settings. Notification destinations, denial-spike thresholds, retention with an explicit acknowledgement per window, response-page templates and About.

hits today, hourly sparklines and an accessible table. A hit is a successful declarative matcher evaluation: matching WEIGH rules and the first terminal match count; an earlier inspection or reputation decision may prevent evaluation. Private tests do not increment these counters. **Compare rule hits** freezes two closed periods for one recorded node. Revision history offers **Compare hits around this edit**, excluding the edit minute and using equal available periods up to the chosen duration. Optional applied-revision filters keep unrelated generations out of the comparison. Startup, cutover, missing writes and retention leave visible coverage gaps; percentages require complete coverage on both sides. Rule observations and hourly/daily summaries follow the minute-retention setting. An observed change around an edit is a comparison, not evidence that the edit caused the traffic change.

Importing a later GeoIP generation does not automatically refresh existing country-derived reputation rows. Preview the country action to compare added, retained and removed prefixes; page through the reviewed diff before applying it. The replacement removes obsolete rows owned by that country and preserves independently managed prefixes. A changed generation or policy revision requires a fresh preview, and overlapping independent edits are refused. The Events page contains retained WAF findings and honeypot incidents; it is not a complete access log. With GeoIP loaded, the storage worker records each incident's country and generation in the incident transaction. This is attribution at persistence, not a reconstructed request-time location. Later imports leave recorded mappings unchanged. The country filter accepts an uppercase two-letter code, `unknown` for an address absent from the loaded generation, or `not_recorded` for an incident without mapping data. Source groups report mixed countries or

SIBUNASECURITY CONSOLE
Console node 1

Statistics

Events

Challenges

Policies

GeoIP

Users

Tokens

Audit

Nodes

Settings

Account

admin

Theme: system

Spacing: comfortable

Sign out

Apply audit filters

Live - Last update 0 seconds ago.

0 recent summaries newer than this page. Rows stay in place while you read. Load latest records

At most 64 summaries retained in the live feed; this is not a total count. 0 unavailable source IDs, including expired history. Storage observation age: 1 seconds.

Recorded activity

Window: 2026-09-03 16:43:12 UTC – 2026-09-10 16:43:12 UTC

Time (UTC)	Actor / role	Action	Subject	Record
2026-09-10 16:41:43 UTC	1 / Not recorded	session.create	1	Open 6
2026-09-10 16:27:16 UTC	1 / Not recorded	session.create	1	Open 5
2026-09-10 16:27:16 UTC	1 / Not recorded	session.create	1	Open 4
2026-09-10 16:27:16 UTC	1 / admin	user.password_change	1	Open 3
2026-09-10 16:27:15 UTC	1 / Not recorded	session.create	1	Open 2
2026-09-10 16:27:14 UTC	0 / Not recorded	user.create	1	Open 1

Page 1

[Previous audit page](#)[Next audit page](#)[Export page metadata](#)

Export contains this filtered page's metadata. Open a record for available redacted summaries. Export itself creates an audit record.

Figure 19: Audit. Append-only history with actor and role, action, subject and a detail view with redacted before and after summaries; refused sign-ins are recorded too. Every management mutation and authentication row records the client address that presented the credential; rows the system writes on its own show it as not recorded. A policy record offers **Revert this change**, which restores the document recorded before that revision as a new, audited revision after a confirmation, and is refused if the rule set has moved on.

coverage explicitly. The globe's **View events** action keeps the selected node and opens the country's retained incidents for the last hour. The Statistics page's **Security** view combines live rate-limit, challenge and ban rates with retained inspection and honeypot findings over one hour, one day, seven days or thirty days. Category, source and path links open the normal incident workflow with the same node and a fixed time boundary. Applying incident filters starts a new period. Findings include audit records and are distinct from blocked-request totals; absent reputation and rule-hit attribution is shown as not recorded. Event and audit filters keep labels with their controls, align the Apply action separately and adapt their columns to the available content width. One authenticated WebSocket survives navigation and carries statistics, incident summaries, node status, policy revisions, challenges and audit summaries. New incident and audit records wait behind **Load latest records** so the table stays in place while it is read. Policy updates show current committed and applied revisions without replacing an open draft. Flow counters update live; refresh a selected non-default challenge timing partition explicitly. Historical queries, detail reads and mutations remain HTTP requests. Page fragments can be bookmarked; Back and Forward reopen authenticated pages. The sidebar also remembers theme and spacing choices in this browser, with System as the default theme. SID 0007 is Committed. Its pages and management workflows passed the October Linux and connected Chrome review, including an actual cluster on three physical hosts. Performance acceptance is separate: all eight fresh single-node and three-host impact matrices

are inconclusive, with each raw sample, coverage check and formal verdict retained under **benchmarks/results/**. The September paired reading was accepted by the owner as a historical exception, not a formal pass, and it does not apply automatically to the updated request path or to new measurements. These unprivileged containers cannot control the processor governor or other host activity; that limits acceptance without establishing a cause for measured throughput differences. The corrected harness includes each dashboard's stream, rankings and retained-timeline queries; a full acceptance run requires a production GeoIP snapshot and documented host conditions. Direct TLS peer transport and the combined dashboard have separate coverage and freshness contracts; interface review and browser acceptance remain tracked in SID 0007. The complete Wasm application warns above 640 KiB and has a 768 KiB uncompressed ceiling. These project limits leave room for console workflows; they do not replace browser loading and responsiveness measurements or change the explicit 4 MiB linear-memory allocation.

9.3 Deployment Topologies

In this chapter Deploy the autonomous reverse proxy and the forward-auth validator behind Nginx or Caddy.

9.3.1 Reverse Proxy

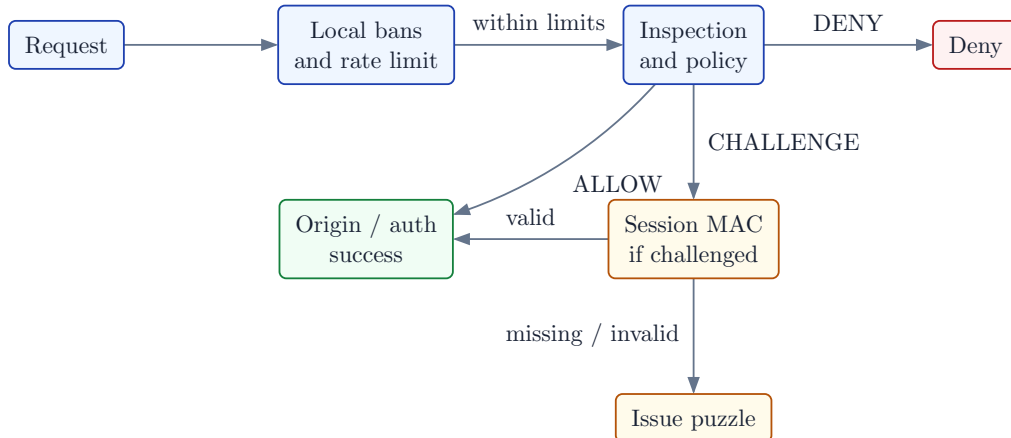


Figure 20: Request routing on the Shield surface

```
sibuna --port 80 --upstream-host 127.0.0.1 --upstream-port 3000 \
  --secret-file /etc/sibuna/secret --difficulty 16 --algorithm posw
```

Admitted requests reach the origin with X-Forwarded-For, X-Real-IP, X-Sibuna-Status, and X-Sibuna-Rule headers; hop-by-hop headers and any incoming forwarded-for value are stripped. HTTP/1.1 WebSocket upgrades pass through admission and policy checks before the origin's handshake is accepted. Subprotocol and extension negotiation remains end-to-end; two fixed 16 KiB buffers relay bytes in both directions, including prefetched bytes and half-closes. Upgraded sockets remain within the normal connection quota and shutdown registry, but use `--websocket-idle-timeout` (300 seconds by default) independently of the HTTP idle deadline. Traffic in either direction refreshes that bound. Frames after the handshake are not WAF inspection inputs. HTTPS/WSS uses a TLS-terminating ingress in front of Sibuna's private HTTP/1.1 listener; Sibuna does not terminate browser TLS itself.

The ingress can negotiate HTTP/2 with browsers and forward HTTP/1.1 to Sibuna. Native HTTP/2 support in Sibuna is deferred to a later update; this arrangement does not provide end-to-end HTTP/2 semantics for applications such as native gRPC.

Content-Length request bodies stream through fixed buffers, preserving bytes and MIME headers. Multipart uploads retain boundaries, repeated field names, filenames and part types. The WAF examines at most the first 8 KiB of the body: multipart metadata and non-file fields remain text inspection inputs, while file payloads are opaque. Recognized binary top-level MIME types (images except SVG, audio, video, PDF, ZIP, gzip, 7z, protobuf and octet-stream) are opaque too. JSON, XML, SVG, URL-encoded forms and unknown types retain text inspection. Ambiguous or malformed MIME metadata falls back to text inspection. Multipart parsing is bounded to 32 parts and 2 KiB of headers per part within that same prefix.

This is upload compatibility, not file validation or malware scanning. The backend must enforce accepted media types rather than trust a client's Content-Type declaration. Fields after the inspection prefix, including those after a large uploaded file, are not inspected. Compressed payloads are not decompressed for inspection. Uploads still pass admission, path, query and header checks and remain subject to connection deadlines. Chunked request bodies are decoded inside the connection buffer before inspection (SID 0009). A body that ends within the buffer reaches the backend with Content-Length. A longer one is re-chunked by Sibuna, one chunk per read, so the backend never sees the client's chunk sizes, extensions or trailers. A chunk line with a lone CR or LF, whitespace around the size, a malformed extension or more than 4 KiB is refused with 400. So is a trailer section over 16 KiB. Transfer-Encoding with another coding receives 501. Backends that cannot parse chunked requests still receive Content-Length for bodies under about 44 KiB. Clients waiting for 100-continue receive it locally before sending the body; unsupported expectations receive 417. The Expect header is consumed before forwarding to the backend.

9.3.2 Forward Auth Behind an Ingress

In `--mode forward_auth` the daemon answers the ingress's subrequest with 200 (plus the audit headers), 401 for a challenge, 403 for a denial, or 429 when rate limited. The ingress must forward the client address and original URL; forward-auth mode trusts them by default. Bind this listener privately so only the ingress can connect. X-Forwarded-Uri (Caddy) or X-Original-URI (Nginx), and X-Forwarded-Method, restore the application request for policy evaluation. Duplicate or conflicting original-URL fields are rejected. Internal daemon routes always use the actual request URI. Forward-auth does not inspect a body the ingress omits.

```
map $http_upgrade $sibuna_connection_upgrade {
    default upgrade;
    '' close;
}
server {
    listen 443 ssl;
    location / {
        auth_request /__sibuna_auth;
        auth_request_set $sibuna_auth_status $upstream_status;
        auth_request_set $sibuna_retry_after $upstream_http_retry_after;
        auth_request_set $sibuna_status $upstream_http_x_sibuna_status;
        auth_request_set $sibuna_rule $upstream_http_x_sibuna_rule;
        auth_request_set $sibuna_rule_hash $upstream_http_x_sibuna_rule_hash;
        error_page 401 = @sibuna_challenge;
        error_page 500 = @sibuna_auth_error;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $sibuna_connection_upgrade;
        proxy_set_header Host $host;
        proxy_set_header X-Sibuna-Status $sibuna_status;
        proxy_set_header X-Sibuna-Rule $sibuna_rule;
        proxy_set_header X-Sibuna-Rule-Hash $sibuna_rule_hash;
        proxy_read_timeout 300s;
        proxy_pass http://127.0.0.1:3000;
    }
}
```

```

location = /__sibuna_auth {
    internal;
    proxy_pass http://127.0.0.1:8080/;
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
    proxy_set_header X-Forwarded-Method $request_method;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Forwarded-Uri "";
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header User-Agent $http_user_agent;
    proxy_set_header Cookie $http_cookie;
}
location @sibuna_challenge {
    rewrite ^ /__sibuna/challenge break;
    proxy_pass http://127.0.0.1:8080;
    proxy_pass_request_body off;
    proxy_set_header Content-Length "";
    proxy_set_header X-Original-URI $request_uri;
    proxy_set_header X-Forwarded-Method $request_method;
    proxy_set_header X-Forwarded-For $remote_addr;
    proxy_set_header X-Forwarded-Proto $scheme;
}
location @sibuna_auth_error {
    add_header Retry-After $sibuna_retry_after always;
    if ($sibuna_auth_status = 429) { return 429; }
    return 503;
}
location /__sibuna/ {
    proxy_pass http://127.0.0.1:8080;
    proxy_set_header X-Forwarded-For $remote_addr;
}
}
example.com {
    handle /__sibuna/* {
        reverse_proxy localhost:8080
    }
    handle {
        forward_auth localhost:8080 {
            uri /
            header_up X-Forwarded-For {remote_host}
            header_up -X-Original-URI
            copy_headers X-Sibuna-Status X-Sibuna-Rule X-Sibuna-Rule-Hash
        }
        reverse_proxy localhost:3000
    }
}

```

The `/__sibuna/*` namespace (interstitial, challenge, verify, solver assets) must reach the daemon directly in both recipes. The nginx error page passes the original URI and method so the interstitial carries the requirement the authorization decided; without them the page still works, and issuance falls back to evaluating the URL the browser reports. Caddy supplies original URI/method metadata itself. The exclusive `handle` blocks ensure that challenge

and verification routes do not enter the forward-auth precheck. Nginx requires the explicit Upgrade/Connection headers shown above for application WebSockets. The Nginx error handler renders the internal challenge route without making a second authorization decision or consuming the upload body. Its auth module accepts only 2xx, 401 and 403 directly, so the example translates a rate-limited auth error back to 429 with Retry-After; other authorization failures remain closed with 503. Nginx supplies its own 403 body, whereas Caddy forwards Sibuna's denial page. Both recipes replace incoming Sibuna audit headers with the actual authorization response. Configure normal TLS certificates and application upload/deadline limits at the ingress; the examples show the routing logic.

9.4 Declarative Policy

In this chapter Write a policy file with rules, weights, thresholds, IP rules, and the WAF switch.

```
{
  "default_action": "CHALLENGE",
  "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5 },
  "ip_rules": { "10.0.0.0/8": "ALLOW", "192.0.2.0/24": "DENY", "2001:db8::/32":
"DENY" },
  "rules": [
    { "name": "deny-cf-workers", "headers": { "CF-Worker": ".*" }, "action":
"DENY" },
    { "name": "deny-amazonbot", "user_agent": "Amazonbot", "action": "DENY" },
    { "name": "api-with-key", "path": "/api/*", "headers": { "X-Api-Key": ".*" },
"action": "ALLOW" },
    { "name": "protect-checkout", "path": "/checkout/*", "action": "CHALLENGE",
"challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight":
30 },
    { "name": "internal-vpc", "remote_addresses": ["10.0.0.0/8", "fd00::/8"],
"action": "ALLOW" }
  ]
}
```

The loader fails closed: an unknown key, a misspelled action, a malformed address or an out-of-range number rejects the whole file, the diagnostic names the rule and field, and the daemon does not start. A command line with an unknown option or an out-of-range value is refused the same way. `rules` replaces the built-in table when present; `ip_rules` feeds the reputation trie, which scales to thousands of prefixes; `waf: false` selects the Gate surface from the file.

9.5 Persistent Storage and Clustering

In this chapter Enable Zaxonline storage, add a dynamic policy and a ban with SQL, query incident forensics, understand campaign clustering, and run a replicated cluster.

`--data-dir /var/lib/sibuna` opens an embedded Zaxonline node (journal, payload store, and SQLite image in one directory) and starts the storage thread. On start and whenever the

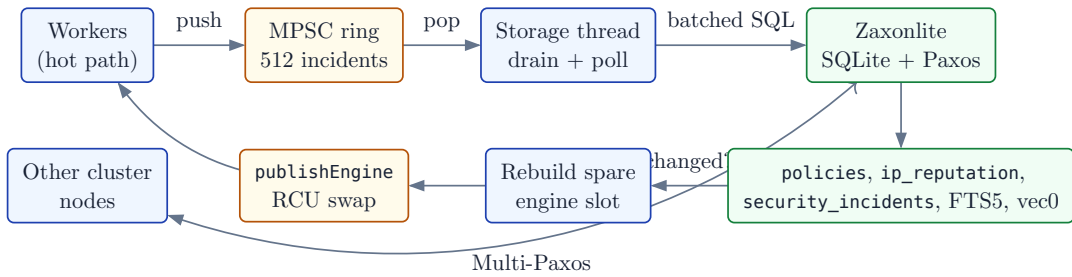


Figure 21: The storage architecture: workers never block on the database

policies or ip_reputation tables change, the thread rebuilds the spare engine slot from the policy file plus the database and publishes it; requests never wait on SQL.

9.5.1 Schema

Table	Purpose and key
policies	Ordered dynamic rules, keyed by id.
ip_reputation	Scores, hit counts and expiry, keyed by address or prefix.
security_incidents	One bounded event record, keyed by issuer and sequence.
incidents_fts	Full-text index over paths and payloads.
incidents_vec	64-component embeddings for nearest-campaign lookup.
sibuna_meta	Policy revision and per-issuer incident commit receipts.

The complete schema is `apps/sibuna/src/persistent.zig`. Runtime migrations and indexes are part of the same storage transaction; an abbreviated printed schema is not an upgrade script.

9.5.2 Operating It

Use the `zaxon` CLI from the Zaxonlite release, or any SQLite client on the materialised `current.db` for reads:

```

-- Add a dynamic rule; healthy nodes poll after committed changes.
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty,
algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
VALUES ('p-checkout', 'protect-checkout', 10, '/checkout/*', 'CHALLENGE', 20, 'posw',
  '{"X-API": "v2"}', ["10.0.0.0/8"], 0, 1, unixepoch(), unixepoch());

-- Ban an address cluster-wide for a day.
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until,
  trigger_rule, hits, last_seen)
VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());

-- Forensics: full-text search over recorded payloads.
SELECT s.id, s.client_ip, s.violation_category, s.path, s.campaign_id
FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;

```

Scores at or below `-50` become `deny` prefixes in the trie, scores at or above `50` become `allow`, and `banned_until` bounds the ban. Honeypot hits insert a `-100` record automatically, so a trap sprung on one node bans the address on all of them.

9.5.3 A Transaction Receipt Makes Retry Safe

The storage thread collects at most 32 pending records and builds one SQL transaction. The transaction inserts incidents, updates the text and vector indexes, and applies honeypot reputation changes. Each issuer has a monotonically increasing cursor in `sibuna_meta`. Every data-changing statement is guarded by that cursor; the transaction advances it last.

Worked example: the acknowledgement disappears Suppose issuer 2 prepares sequences 101 through 132, ending at cursor 133. The leader commits all records and cursor 133, then disappears before replying. The storage thread retains the exact SQL and retries. The receipt already equals 133, so the guarded writes do nothing; neither incidents nor honeypot hit counts are doubled. If the original transaction rolled back, the old cursor remains and the retry applies all writes once.

A failed commit retains the pending batch in memory and increments `incident_write_failures`. The tick still attempts policy polling. A full 512-record queue counts `incidents_dropped`; it never blocks an HTTP response waiting for disk. The pending batch adds space for 32 records. Process death before commit can lose queued data: this is a bounded asynchronous forensic path, not a durable message queue at enqueue time.

Exercise 9.1. Move the receipt update outside the data transaction. Construct one crash schedule that duplicates a reputation effect and another that loses an incident.

9.5.4 Campaign Clustering

Each incident payload is embedded as a 64-dimensional unit vector by the hashing trick over byte trigrams (digits folded, case folded) and stored in `incidents_vec`. Before insertion the transaction queries the vector table for the nearest existing incident, including earlier records in the same batch; if its cosine distance is below 0.35 the new incident joins that incident's `campaign_id`, otherwise it starts a campaign. Similarity is a heuristic: the regression examples group related SQL payloads, but that is not a guarantee that every pair of attacks shares a campaign. No model runs and nothing is trained.

9.5.5 Clustering

Build with `-Dcluster=true` (links OpenSSL 3 for Zaxonlite's mutual TLS) and start each member with the full static membership:

```
sibuna --data-dir /var/lib/sibuna --cluster-node 1 --cluster-listen 10.0.0.1:9901 \
  --cluster-peer 2@10.0.0.2:9901 --cluster-peer 3@10.0.0.3:9901 \
  --cluster-tls-cert n1.crt --cluster-tls-key n1.key --cluster-tls-ca ca.crt \
  --secret-file /etc/sibuna/secret
```

Every member must pass the identical member list and the same master secret. Writes go to the elected leader and replicate as SQLite page images by Multi-Paxos; each node's storage

thread sees the committed change and rebuilds its engine. For local experiments a loopback cluster may use `--cluster-secret-file` (a pre-shared key) instead of certificates.

9.5.6 Cluster Challenge Routing

Challenge keys are bound to `--cluster-node`; keep challenge issuance and verification on the same member. Tokens use the shared seed and work on other members. Local rate quotas do not become global quotas, and process restarts clear spent sets. See the distributed benchmark results for loopback throughput, replicated ban propagation and one-member-loss coverage. The storage transport authenticates each node certificate using the common name `zaxon-node-<id>` (matching `--cluster-node`), signed by the configured CA. A certificate with an arbitrary common name does not authenticate a storage member. The distributed harness creates temporary CA-signed identities and exercises this mutual-TLS transport.

9.6 Packaging

In this chapter Build the binary for a target, package it, and run it under systemd.

- `zig build -Doptimize=fast` produces a binary with storage compiled in (it links `libc` for SQLite).
- `zig build -Doptimize=fast -Dstorage=false` produces a fully static binary with no `libc` dependency, suitable for a `scratch` container image; `--data-dir` is then refused at start.
- `zig build -Dcluster=true` adds replication and requires OpenSSL 3 at build and run time.

[Unit]

`Description=Sibuna Web Firewall`

`After=network.target`

[Service]

`User=sibuna`

`ExecStart=/usr/local/bin/sibuna --port 8080 --upstream-port 3000 \`
`--secret-file /etc/sibuna/secret --data-dir /var/lib/sibuna`

`Restart=always`

`LimitNOFILE=65535`

`ProtectSystem=strict`

`ReadWritePaths=/var/lib/sibuna`

[Install]

`WantedBy=multi-user.target`

Exercise 9.2. Write the policy file and the `ip_reputation` rows needed so that a staging network (10.20.0.0/16) bypasses challenges, `/admin/*` demands 20 work bits of sequential work, and a partner scraper identified by `X-Partner-Key` is admitted at 30 requests per 10 seconds.

Hint: Rate limits are global per client; use a rule for the partner and the `daemon` flags for the limit.

Explain the invariant. Explain to an operator why adding a row to `policies` takes effect without a restart and without a request ever waiting, in terms of the storage thread and the engine slots.

10 Desk Reference and Diagnostics

Endpoints, metrics, error catalog, policy schema, and storage tables in one place.

10.1 Endpoints

In this chapter Know every route under `/__sibuna/` and what each returns.

Endpoint	Method	Function
<code>/__sibuna/challenge</code>	GET	The interstitial page
<code>/__sibuna/challenge.json?need=&path=</code>	GET	Issues a stateless challenge: the requirement ticket from the challenged response decides it, otherwise the reported URL is evaluated (414 above 8 KiB). Returns <code>{id, algorithm, difficulty, challenges, expires_at}</code>
<code>/__sibuna/verify</code>	POST	Accepts <code>{"challenge_id", "nonce"}</code> or <code>{"challenge_id", "proof"}</code> ; 200 with Set-Cookie, or 400 with a diagnostic
<code>/__sibuna/wasm/sibuna-pow.wasm</code>	GET	The 8,831-byte solver module, cacheable
<code>/__sibuna/worker.js</code>	GET	The Web Worker with WASM and JavaScript provers, cacheable
<code>/__sibuna/honeypot</code>	GET	Bans the caller for <code>--ban-seconds</code> and records an incident; 403
<code>/__sibuna/health</code>	GET	JSON liveness status with engine name, version, proxy mode and proof algorithm
<code>/__sibuna/metrics</code>	GET	Prometheus text format

Challenge responses carry `X-Sibuna-Status: CHALLENGE`; admitted requests carry `X-Sibuna-Status: PASS` and `X-Sibuna-Rule` upstream (and `X-Sibuna-Rule-Hash` in forward-auth replies). The separate opt-in management listener serves `/console/`. Initialize its administrator locally with `sibuna init-admin`, then replace the temporary password on first sign-in. `/console/api/stats` and `/console/ws` require an authenticated session. The WebSocket multiplexes statistics, events, nodes, policy, challenges and audit with bounded snapshots, deltas and gap recovery. `/console/stream` retains the earlier statistics-only protocol. Origin error counters are not observed in forward-auth mode. Part IX documents the console's workflows, HTTPS deployment and current SID 0007 limits.

10.2 Metrics

Counters exposed as `sibuna_<name>_total` in Prometheus text format:

Counter	Incremented when
<code>requests</code>	A request head was parsed
<code>allowed</code> , <code>denied</code> , <code>challenged</code>	A policy decision was made (admitted, refused, challenge issued or reissued)
<code>challenges_issued</code>	<code>/__sibuna/challenge.json</code> minted a challenge record
<code>solutions_accepted</code> , <code>solutions_rejected</code>	<code>/__sibuna/verify</code> accepted or rejected a proof
<code>rate_limited</code>	GCRA refused a request (429)
<code>banned</code>	A banned address was refused, or the honeypot banned one
<code>proxied</code> , <code>upstream_errors</code>	A request was relayed to the origin, or the origin failed (502)
<code>parse_errors</code>	A malformed head was refused (400)
<code>overloaded</code>	A connection beyond <code>--max-connections</code> was answered 503
<code>incidents_persisted</code> , <code>incident_batches</code>	Records and transactions whose commit the storage thread confirmed
<code>incidents_dropped</code> , <code>incident_write_failures</code>	Queue pushes rejected because the ring was full, and failed commit attempts

A retry can increment `incident_write_failures` without losing records, because the pending batch is retained. Monitor increments over an interval; totals alone are not a queue depth.

10.3 Status Codes

Code	When
200	Admitted (forward-auth), interstitial (HTML navigation needing a challenge), internal routes
302	Not used by the current protocol; verification answers 200 and the page reloads
400	Malformed request, or a rejected solution with an Elm-style diagnostic
401	Challenge required for a client that does not accept HTML, or in forward-auth mode
403	Policy or WAF denial, banned address, honeypot
413	Solution body larger than the 64 KB connection buffer
417	Unsupported request expectation; 100-continue is handled locally
429	GCRA limit exceeded; <code>Retry-After</code> in seconds

Code	When
431	Request head over 16 KB
502	Origin unreachable, or its response head was malformed or larger than 16 KB
503	Connection limit (<code>--max-connections</code>) reached; the socket is closed after the reply

10.4 Error Catalog

`INVALID COMMAND LINE` stops startup for any option the daemon does not recognise, any value outside its documented range, and an invalid, missing or duplicate mode selection. The block names the option, the value given, the range expected and the error (for example `UnknownOption`, `InvalidValue`, `InvalidMode`, `DuplicateMode`, `TooManyPeers`). Supply `--mode reverse_proxy` or `--mode forward_auth` once; `-m` is the equivalent short option.

Implementation: `core.explainError` Maps every domain error to a boundary line, an explanation, and a `Hint::` in `libs/core/src/errors.zig`

Error	Cause	Hint
<code>MalformedChallenge</code>	The identifier is not a well-formed challenge record	Fetch a fresh challenge and submit it unchanged
<code>InvalidChallengeTag</code>	The tag does not authenticate; not issued by this cluster or edited	Challenges cannot be forged; request a new one
<code>ChallengeExpired</code>	Older than the challenge TTL, or minted in the future	Request a new challenge
<code>FingerprintMismatch</code>	Submitted from a different address or <code>User-Agent</code>	Submit from the client that fetched it
<code>DifficultyNotMet</code>	Hashcash nonce lacks the required zero bits	Keep searching nonces
<code>InvalidProof</code>	The sequential-work proof does not open the committed labels	Run the prover to completion for the issued depth and openings
<code>WrongSolutionType</code>	A nonce for a PoSW challenge or a proof for Hashcash	Match the solution field to the algorithm
<code>DoubleSpendAttempt</code>	The challenge was already spent	Challenges are single use
<code>StoreFull</code>	Spent set shard exhausted	Lower the challenge TTL or raise capacity

Error	Cause	Hint
InvalidTokenSignature	Cookie tag or signature fails	Re-authenticate through the interstitial
TokenExpired	Cookie past its expiry	Re-authenticate
TokenBoundAddressMismatch	Cookie presented from a different client identity	Cookies cannot be shared

10.5 Policy Schema

```

{
  "default_action": "ALLOW" | "DENY" | "CHALLENGE",
  "waf": true | false,
  "thresholds": { "challenge_at": int, "deny_at": int, "bits_step": int },
  "ip_rules": { "<cidr>": "ALLOW" | "DENY" | "CHALLENGE", ... },
  "rules": [
    {
      "name": "<kebab-case>",
      "path" | "path_regex": "<pattern>",
      "user_agent" | "user_agent_regex": "<pattern>",
      "headers" | "headers_regex": { "<Header>": "<pattern>" }, // up to 4
      "remote_addresses" | "cidrs": [ "<cidr>", ... ], // up to 8, IPv4 or
IPv6
      "action": "ALLOW" | "DENY" | "CHALLENGE" | "WEIGH",
      "weight": int, // WEIGH only
      "challenge": { "difficulty": <work bits>, "algorithm": "hashcash" | "posw" }
    }
  ]
}

```

Pattern grammar: `.*` or `*` match anything; `^...$` anchors an exact path; a trailing `*`, `/*`, or `.*` is a prefix; a pattern starting with `/` is an exact path; anything else is a case-insensitive substring.

10.6 Storage Tables

policies, ip_reputation, security_incidents, incidents_fts (FTS5 over path and payload), incidents_vec (vec0, 64-float cosine embeddings), and sibuna_meta. Incident ids are `node_id < 40` | sequence, unique across a cluster without coordination.

10.7 Build Targets

Command	Result
<code>zig build</code>	Daemon with storage, benchmark binary, browser module
<code>zig build -Dstorage=false</code>	Fully static daemon without Zaxonlite
<code>zig build -Dcluster=true</code>	Daemon with Multi-Paxos replication (needs OpenSSL 3)
<code>zig build test</code>	Unit tests, solver tests, end-to-end tests against a live daemon, storage tests

Command	Result
<code>zig build fmt</code>	<code>zig fmt --check</code> plus the 70-line / 99-column style gate
<code>zig build wasm</code>	The browser module only
<code>zig build benchmark</code>	<code>run-all.sh</code> : ReleaseFast benchmarks with host metadata
<code>zig build book / zig build sid</code>	This book / the SID records

Exercise 10.1. A client receives 400 with the title `CLIENT FINGERPRINT MISMATCH` after switching from Wi-Fi to cellular mid-solve. Explain the cause from the token and challenge formats, and propose the smallest change to the interstitial that recovers gracefully.

Explain the invariant. Without looking, list the endpoints a reverse proxy in front of Sibuna must route to the daemon rather than to the origin, and say why each is needed.

11 Quick Reference Card

Keep beside a terminal: build, run, configure, route, observe, and diagnose. Part X holds the full tables; this card holds what an operator types.

Build

```
zig build -Doptimize=fast          # daemon + storage
zig build -Doptimize=fast -Dstorage=false # static, no libc
zig build -Dcluster=true           # Multi-Paxos, needs OpenSSL 3
zig build test                     # native, UI and live daemon tests
zig build console-test             # console workflows through a live daemon
zig build fmt                      # zig fmt + 70-line / 99-column gate
zig build book sid                 # this book, the SID records
zig build benchmark                # primitives -> results/latest.json
python3 benchmarks/tools.py --anubis <binary> # whole products under wrk
```

Run

```
# Shield (default): PoW + WAF, proxy to :3000
sibuna -p 8080 -u 3000 -s /etc/sibuna/secret
# Gate: PoW only
sibuna --gate -p 8080 -u 3000 -s /etc/sibuna/secret
# Forward auth behind Nginx/Caddy (trusts X-Forwarded-For)
sibuna -m forward_auth --host 127.0.0.1 -p 8080 -s /etc/sibuna/secret
# Edge: persistent policies, reputation, forensics
sibuna -D /var/lib/sibuna -s /etc/sibuna/secret
head -c 32 /dev/urandom | xxd -p -c 64 > /etc/sibuna/secret
```

Console (opt-in preview)

```
# Initialize while the daemon is stopped, then start the console
sibuna init-admin admin --data-dir ./data
sibuna --data-dir ./data --console 127.0.0.1:19446
Open http://127.0.0.1:19446/console/ and replace the temporary password. Add --console-location
1.3521,103.8198 to place the node on the globe. Remote access requires an HTTPS proxy, an explicit origin and trusted-
proxy CIDRs; follow Part IX. SID 0007 is Committed; functional verification is complete, while the console's performance
isolation target remains unproved by the container measurements.
```

Flags that matter first

-d, --difficulty	16	work bits (PoSW depth = bits - 3)
-a, --algorithm	posw	posw or hashcash
--posw-challenges	16	openings per proof
--token-ttl	86400	session seconds
--challenge-ttl	300	challenge seconds
--token-scheme	mac	mac or ed25519
--rate-limit / --rate-window	100 / 10	GCRA burst per window
--ban-seconds	3600	honeypot ban
-w, --workers	CPUs	accept threads
--max-connections	1024	then 503
--idle-timeout	15	reaper seconds
-P, --policy-file	—	JSON policy
--trust-forwarded	off	on in forward auth
--secure-cookie	off	add Secure

Endpoints under /__sibuna/

challenge	interstitial HTML
challenge.json? need=&path=	{id, algorithm, difficulty, challenges, expires_at}
verify (POST)	{challenge_id, nonce proof} → 200 + Set-Cookie, else 400
wasm/sibuna- pow.wasm	8,831-byte solver
worker.js	worker with WASM and JS provers
honeypot	bans caller, 403
health	{"status": "ok", ...}
metrics	Prometheus text

Route all of them to the daemon, never to the origin.

Status codes

200	admitted (forward auth), interstitial, internal routes
400	malformed request or rejected solution (diagnostic body)
401	challenge required: non-HTML client or forward auth
403	policy or WAF denial, ban, honeypot
413	solution body over the 64 KB buffer
417	unsupported request expectation
429	GCRA limit, Retry-After seconds
431	head over 16 KB
502	origin unreachable or malformed
503	--max-connections reached

Headers

Upstream (proxy): X-Forwarded-For, X-Real-IP, X-Sibuna-Status: PASS, X-Sibuna-Rule: <rule>; hop-by-hop and incoming forwarded-for dropped.
Forward-auth reply: X-Sibuna-Status, X-Sibuna-Rule, X-Sibuna-Rule-Hash.
Challenge reply: X-Sibuna-Status: CHALLENGE.
Cookie: __sibuna_token=<75 chars>; Path=/; Max-Age=<ttl>; HttpOnly; SameSite=Lax[; Secure]

Policy file skeleton

```
{ "default_action": "CHALLENGE", "waf": true,
  "thresholds": { "challenge_at": 10, "deny_at": 40, "bits_step": 5},
  "ip_rules": { "10.0.0.0/8": "ALLOW", "2001:db8::/32": "DENY"},
  "rules": [
    { "name": "api", "path": "/api/*", "headers": { "X-API-Key": ".*" }, "action": "ALLOW"},
    { "name": "checkout", "path": "/checkout/*", "action": "CHALLENGE",
      "challenge": { "difficulty": 20, "algorithm": "posw" } },
    { "name": "headless", "user_agent": "Headless", "action": "WEIGH", "weight": 30 } ] }
```

Patterns: */.* any · ^...\$ exact · trailing * prefix · leading / exact path · else case-insensitive substring. Order: WAF
→ trie deny/allow → rules → score → bypass → trie challenge → bots → default.

Storage (Edge) in SQL

```
INSERT INTO policies (id, name, priority, path_pattern, action, difficulty, algorithm,
  header_matchers, cidr_matchers, weight, enabled, created_at, updated_at)
VALUES ('p1', 'protect', 10, '/checkout/*', 'CHALLENGE', 20, 'posw', '{}', '[]', 0, 1,
  unixepoch(), unixepoch());
INSERT INTO ip_reputation (ip_or_cidr, reputation_score, banned_until, trigger_rule, hits,
  last_seen)
VALUES ('198.51.100.7', -100, unixepoch() + 86400, 'analyst', 1, unixepoch());
SELECT s.client_ip, s.violation_category, s.path, s.campaign_id
FROM incidents_fts f JOIN security_incidents s ON s.id = f.rowid
WHERE incidents_fts MATCH 'union' ORDER BY rank LIMIT 20;
```

Score ≤ -50 → deny prefix; ≥ 50 → allow. Cluster: same member list and secret on every node; -Dcluster=true, --cluster-node N --cluster-listen host:port --cluster-peer id@host:port, certificates zaxon-node-<id> or --cluster-secret-file for loopback.

Diagnostics

MALFORMED CHALLENGE	id edited or truncated; fetch a new one
INVALID CHALLENGE TAG	not minted by this seed or node
CHALLENGE EXPIRED	older than TTL or clock skew > 60 s
FINGERPRINT MISMATCH	address or User-Agent changed
DIFFICULTY NOT MET	nonce lacks zero bits
INVALID PROOF	openings do not reach the root
WRONG SOLUTION TYPE	nonce for PoSW or proof for Hashcash
DOUBLE SPEND	challenge already used
STORE FULL	spent shard saturated; lower TTL
TOKEN ... MISMATCH	cookie copied to another client

Metrics to alert on: `denied`, `rate_limited`, `overloaded`, `upstream_errors`, `incidents_dropped`, `incident_write_failures` (all `sibuna*_total`).

Numbers to remember

Challenge id 70 chars (36-byte payload + 16-byte tag) · token 75 chars (40 + 16) · proof bytes $32(1 + t(n + 1))$ · PoSW depth = bits - 3, range 4-24 · request buffer 64 KB · head limit 16 KB · body inspected 8 KB · spent set $16 \times 4,096$ · rate cells 16×512 · bans 4,096 · incident ring 512 · batch 32 · campaign distance 0.35 · keep-alive 4,096 requests per connection · origin pool 256 sockets.

Ingress routing

Use the complete, live-tested Nginx or Caddy recipe in Part IX, Forward Auth Behind an Ingress. Route `/__sibuna/*` directly; authorize application requests with the original URI, method and trusted client address. Replace incoming Sibuna audit headers with the authorization result. Nginx needs explicit challenge, 429 and unavailable-auth handling. `reverse_proxy` carries admitted HTTP/1.1 bodies and WebSockets; `forward_auth` grants admission and the ingress carries them. Omitted auth bodies and origin responses cannot be inspected or counted by Sibuna. TLS terminates at the ingress; native HTTP/2 is deferred. Uploads use Content-Length; inspection sees at most an 8 KiB prefix, with file bytes opaque.

Selected Solutions

These solutions are checks on the argument, not substitutes for the exercises. If your result differs, first compare assumptions: endpoints of time intervals, what is counted as work, and whether a write has committed are common sources of disagreement.

11.1 Part I: Cost and Probability

1.1. One million trials divided among 100 requests is 10,000 trials per request. Sharing a session does not change that arithmetic if the aggregate request count stays 100. The actual system must decide whether a second process can satisfy the fingerprint and routing bindings; those conditions are outside the amortization equation.

1.2. The excess is $1000 - 600 = 400$ records/s. An empty queue of 512 records buys $\frac{512}{400} = 1.28$ seconds in the fluid model. Real producers arrive in bursts and the consumer commits batches, so the exact first drop depends on arrival and commit times.

11.2 Part II: Prior Art

2.1. One source compiled twice cannot disagree with itself about the statement, the nonce encoding, the tree shape, or the opening layout; drift would require a compiler bug. It says nothing about whether the construction is sound (cryptographic review), whether the browser runtime executes it correctly (a byte-identical comparison between the WebAssembly and JavaScript provers covers that), or whether the parameters are calibrated (measurement).

2.2. (a) Sibuna Gate or Anubis, one binary; (b) Sibuna Shield, SafeLine, or Cloudflare (Pro plan or above); (c) Sibuna Edge with `-Dcluster`, Anubis with a shared Valkey store (for state, not for bans), or Cloudflare. The only single product meeting all three without a Docker host or a subscription is Sibuna Shield with `--data-dir` and cluster replication; the "runs as", "inspection", and "multi-node" rows decide it.

11.3 Part III: Cryptography

3.0. The tail estimate is $e^{-4.6} \approx 0.010$. Dividing the nonce space changes the rate at which trials are completed, not the success probability of each independent trial. Duplicate trials or coordination work can make a real parallel implementation less efficient.

3.1. If each commitment succeeds with probability q , all g fail with probability $(1 - q)^g$. At least one succeeds with probability $1 - (1 - q)^g$. The formula does not count commitment construction, query budgets, dependencies between trials, or the work needed to find a commitment with a particular acceptance probability.

3.2. At depth 17 and 16 openings, the proof has $32(1 + 16 \times 18) = 9248$ bytes. At 32 openings it has $32(1 + 32 \times 18) = 18464$ bytes. The opening bytes double; the 32-byte root is still sent once.

3.3. Without the work level a session earned on a cheap route would admit a request to an expensive one that demanded more work; without `timestamp` a token could be minted "in the future" and outlive its policy; without `expiry` it would never die; without `rule_hash` the

upstream could not learn which rule admitted the client; without `fingerprint` a cookie copied to another machine would be accepted. The tag must cover all five because any field left outside it could be edited freely, and the verifier could not tell.

11.4 Part IV: Protocol

4.1. Both browsers share the address and User-Agent, so they share the fingerprint, the rate-limit cell, and the ban slot. They do not share the challenge (each fetches its own record with its own nonce), the spent-set entry, or the cookie. To reuse the other browser's cookie an attacker on the NAT would need to read it from the other machine: the fingerprint would then match, which is why `HttpOnly` and `SameSite` matter more than the binding on a shared address.

11.5 Part V: State and Ordering

5.1. The head and the first part of the body (up to what fits after the head in 64 KB) are in the connection buffer and are sent with `writeAll(req.body)`. The remaining bytes never enter the buffer whole: `relayBody` reads them in 16 KB pieces from the client reader and writes each to the origin. The head must go first because the origin needs `Content-Length` before the body, and because the audit headers are part of the head.

5.2. The emission interval is 100 ms and burst tolerance is 9,900 ms. An idle client may send 100 requests immediately, then one per 100 ms. Through the inclusive endpoint at 3,000 ms, the bound is 130. A fixed 10-second counter would allow 100 in the first window and 100 more the instant the window rolls over; a sliding log would allow exactly 100 in any 10 seconds but must store 100 timestamps per client.

11.6 Part VI: Algorithms

6.1. An apostrophe in a name and an ordinary parameter named `order` do not establish an SQL expression. A substring detector could mistake either for syntax. The structural detector looks for additional evidence, including operators and literal relations. It remains bounded inspection, not a parser for every application language.

6.2. LDAP filters are built from parentheses, `|`, `&`, `!`, and `=`. The detector needs a byte class for `(` and `)` (new) and can reuse the existing `=` and shell-separator classes for `|` and `&`. It should fire on a **sequence** such as `) (` or `( | (`, not on a single parenthesis, for the same reason the SQL detector requires a quote or a tautology.

11.7 Part VII: Browser Engine

7.1. At the default 16 work bits, depth 13 with 16 openings, the WebAssembly prover on a laptop-class V8 takes on the order of 15 ms, so the JavaScript fallback takes on the order of one second; on a slow phone several seconds. A sensible policy: let the page forward the `fallback` message, log it server-side as a metric, and lower the difficulty for a rule that matches clients known to block WebAssembly rather than for everyone.

11.8 Part VIII: Evaluation

8.2. CPU microseconds per request divided by cores busy gives the wall time the product spent per request; it is far above the sub-microsecond classification cost because the product row also contains the kernel's socket read and write, the parser, the session tag check, the metrics increments, the response formatting, the writer flush, and the scheduler's hand-off between the load generator and the connection thread. The primitive suite times none of those.

11.9 Part IX: Operations

9.1. If the data commits before a separate receipt update, a crash between them leaves the old receipt and a retry can apply the effect again. If the receipt commits first, a crash before the data write can cause the retry to skip data that never existed. One atomic transaction removes both gaps. Losing the transaction's acknowledgement still requires the retained retry.

9.2. `ip_rules`: `{"10.20.0.0/16": "ALLOW"}` (or an `ip_reputation` row with score 100) admits staging; a rule `{"name": "admin", "path": "/admin/*", "action": "CHALLENGE", "challenge": {"difficulty": 20, "algorithm": "posw"}}` demands the work; a rule matching `headers`: `{"X-Partner-Key": ".*"}` with `ALLOW` admits the partner. The 30-per-10-seconds limit is a daemon flag (`--rate-limit 30 --rate-window 10`) and applies to every client, so either accept that or place the partner behind its own Sibuna instance.

11.10 Part X: Reference

10.1. The challenge record carries the keyed fingerprint of the address that fetched it; the solution arrives from a new address, so `verifyAndMint` returns `FingerprintMismatch` before looking at the proof. The smallest recovery is in the interstitial: on a **400** whose title is `CLIENT FINGERPRINT MISMATCH`, fetch a fresh challenge and solve again instead of showing the error, since the work already done cannot be transferred.

11.11 Continue the Investigation

Change one invariant at a time in a disposable test. Remove the snapshot recheck, omit the incident receipt, or round the rate interval downward. Predict a failure schedule before you run the test. A useful regression does more than assert the happy-path output: it demonstrates why the omitted condition is necessary.

Glossary

Admission. The decision to forward a request to the origin. In Sibuna it is made by a valid session, an `ALLOW` rule, a static bypass path, or a reputation `allow` prefix, and it never overrides a WAF denial.

Aho–Corasick automaton. A finite automaton that finds any of a fixed set of patterns in one pass over the input. Sibuna folds failure links into a dense transition table so each byte costs one table load.

Ban table. 4,096 lock-free slots of banned client addresses with expiry, written by the honeypot and by replicated reputation, read on every request.

Campaign. A cluster of security incidents whose payload embeddings lie within cosine distance 0.35 of one another; a heuristic grouping, not a proof of common origin.

Challenge record. The 70-character stateless identifier a client must solve: version, algorithm, difficulty, openings, issue time, client fingerprint, PRF nonce, rule hash, and a 16-byte keyed BLAKE3 tag. Issuing one writes nothing on the server.

Difficulty, work bits. The single integer that scales both tiers: the number of leading zero bits for Hashcash, or three more than the tree depth for sequential work.

Edge. A Sibuna deployment with `--data-dir`, and optionally `-Dcluster` replication, giving persistent policies, replicated reputation, and incident forensics to either surface.

Engine slot. One of two policy-engine instances with a reader count. Workers pin a slot while classifying; the storage thread rebuilds the other and publishes it by pointer swap.

Fingerprint. A keyed hash of client address and User-Agent, bound into challenges and tokens so neither can be replayed from another client.

Forward auth. Deployment mode in which an ingress asks Sibuna whether to admit a request (200) or challenge, deny, or limit it (401, 403, 429), and proxies the origin itself.

Gate. The surface with proof-of-work admission, sessions, rules, reputation, GCRA limits and bans, but no application inspection (`--gate`).

GCRA. The Generic Cell Rate Algorithm: one theoretical arrival time per client that admits a burst of N then one request per emission interval, with no window boundaries.

Hashcash. Tier One: find a nonce such that SHA-256 of the challenge, a colon, and the decimal nonce has b leading zero bits. Expected 2^b trials, verified with one hash.

Honeypot. An invisible link to `/__sibuna/honeypot`; a client that follows it is banned and an incident with reputation `-100` is recorded.

Idle reaper. A thread that closes connections that have been silent longer than `--idle-timeout`, so a slow client cannot pin a connection thread.

Interstitial. The embedded HTML page served in place of a protected page; it fetches a challenge, solves it in a Web Worker, posts the solution, and reloads.

Keyed BLAKE3. The pseudorandom function behind Sibuna's key schedule, challenge tags, session tags, fingerprints, and challenge nonces.

MAC token. The default session cookie: a 40-byte payload (version, work level, timestamp, expiry, rule hash, fingerprint) and a 16-byte keyed BLAKE3 tag, verified in constant time.

Work level. The mechanism and work bits a session's holder actually solved, carried in the token; a route admits a session only when its level reaches the route's requirement.

Opening. In a sequential-work proof, one leaf label plus the sibling labels along its path to the root; the verifier recomputes the path and compares with the commitment.

Proof of Sequential Work (PoSW). Tier Two: the Cohen–Pietrzak hash graph whose labels depend on all earlier labels, forcing $2^{n+1} - 1$ sequential hashes however many cores the prover has. Verified with $t(n + 1)$ hashes.

Reputation trie. The 128-bit radix trie holding `allow`, `deny`, and `challenge` prefixes from the policy file and from the replicated `ip_reputation` table.

Robin Hood spent set. The fixed-capacity open-addressed table of solved challenge tags, which prevents a valid solution from being submitted twice.

Rule hash. A 64-bit hash of the rule name that demanded a challenge, carried in the challenge, the token, and the `X-Sibuna-Rule-Hash` header.

Shield. The default surface: Gate plus the semantic firewall (`--shield`).

SID. A Shibuna Discussion record: a paper-style design document under `docs/sid/records` with the assumptions, proofs, and measurements behind one part of the system.

Surface. One of the two product shapes, Gate or Shield; storage and clustering are options for either.

WEIGH. A rule action that adds a signed weight to a request's score instead of deciding; totals at or above the thresholds challenge with extra work bits or deny.

Zaxonlite. The embedded SQLite replication engine (Multi-Paxos) behind `--data-dir` and cluster mode, fetched as a pinned release in `build.zig.zon`.

Bibliography

- Aho, A. V., and Corasick, M. J. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM* 18(6), 1975.
- ATM Forum. *Traffic Management Specification Version 4.0*, af-tm-0056.000, 1996 (the Generic Cell Rate Algorithm).
- Back, A. "Hashcash: a denial of service counter-measure." Technical report, 2002.
- Blocki, J., Lee, S., and Zhou, S. "On the security of proofs of sequential work in a post-quantum world." *Information-Theoretic Cryptography (ITC)*, 2021.
- Celis, P. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.
- Cohen, B., and Pietrzak, K. "Simple proofs of sequential work." *EUROCRYPT*, 2018.
- Dwork, C., and Naor, M. "Pricing via processing or combatting junk mail." *CRYPTO*, 1992.
- Fiat, A., and Shamir, A. "How to prove yourself: practical solutions to identification and signature problems." *CRYPTO*, 1986.
- Fielding, R., Nottingham, M., and Reschke, J. *HTTP Semantics*, RFC 9110, and *HTTP/1.1*, RFC 9112. IETF, 2022.
- Hanson, N. "libinjection: SQL injection detection by tokenization." Black Hat USA, 2012.
- Juels, A., and Brainard, J. "Client puzzles: a cryptographic countermeasure against connection depletion attacks." *NDSS*, 1999.
- Lamport, L. "The part-time parliament." *ACM Transactions on Computer Systems* 16(2), 1998; and "Paxos made simple." *ACM SIGACT News* 32(4), 2001.
- Mahmoody, M., Moran, T., and Vadhan, S. "Publicly verifiable proofs of sequential work." *Innovations in Theoretical Computer Science (ITCS)*, 2013.
- O'Connor, J., Aumasson, J.-P., Neves, S., and Wilcox-O'Hearn, Z. *BLAKE3: one function, fast everywhere*. Specification, 2020.
- Vyukov, D. "Bounded MPMC queue." 1024cores.net, 2011 (the sequence-stamped ring used for the incident queue).
- Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. "Hyperscan: a fast multi-pattern regex matcher for modern CPUs." *NSDI*, 2019.
- Sibuna Shibuna Discussions 0001–0006, [docs/sid/records](#), 2026: process, foundation architecture, declarative policy engine, semantic inspection, Zaxonlite storage architecture, and mathematical foundations.