The Rust Port: Workspace, Crates, Dependency Policy and Performance Parity

STATE

committed

STATUS

Committed

TYPE

Architectural Specification

AUTHORS

Vikrant Rathore

CREATED

2026-10-15

UPDATED

2026-10-16

SUMMARY

Ports Sibuna from Zig 0.16 to a Rust workspace that keeps every wire format bit for bit, builds the data plane from small in-house crates without external dependencies, and is accepted when it matches or exceeds the Zig daemon in the benchmarks of SID 0002.

DISCUSSION

No external discussion URL assigned

LABELS

architecture, rust, performance

Record status

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

Contents

1. Abstract 1
2. Motivation and scope 2
3. Specification 2
4. Rationale and alternatives 5
5. Security and operational considerations 6
6. Backwards compatibility 6
7. Acceptance criteria and implementation status 6
8. Open questions 7
9. Review history 7
10. References 7

1. Abstract

Sibuna was written in Zig 0.16. We port it to Rust. The Zig source is the specification of behaviour, and SID 0001 to SID 0009 remain the design records; this record says how their mechanisms map onto a Rust workspace. The port keeps every wire format bit for bit, so a Rust node and a Zig node that share a seed accept each other's challenges and session tokens. The data plane depends on no external crate: hashing, signatures, base64, JSON, DEFLATE, Aho-Corasick and the HTTP/1.1 parser are small workspace crates that other projects may use on their own. Storage is the one exception, and SID 0011 records it. The port is

accepted when it passes the ported test suites and matches or exceeds the Zig daemon in the throughput and latency benchmarks of SID 0002.

2. Motivation and scope

Zig 0.16 served the first version well. Its standard library carried SHA-256 with SHA-NI, BLAKE3, Ed25519, base64, JSON and a TLS client, so the daemon needed almost nothing else. Rust brings an ownership model that turns the Zig comments "this slice borrows the connection buffer until the next read" into lifetimes the compiler checks, a mature ecosystem for the one large problem we do not want to own (SQL storage), and a broader pool of maintainers.

The scope is the whole repository: the data plane (SID 0002 to SID 0004, SID 0006, SID 0009), the browser solver, storage (SID 0005, replaced by SID 0011), the console and its WebAssembly interface (SID 0007, SID 0008), the tools, the benchmarks and the documentation. Documentation moves from Typst to reStructuredText managed with guidedog: the design records with `guidedog dsc` and the book with `guidedog build`.

3. Specification

3.1. Behavioural parity

The following are identical to the Zig daemon, and each has tests whose expected values were produced with the Zig 0.16 standard library or taken from the Zig test suite:

- the challenge identifier (36-byte payload, 16-byte keyed BLAKE3 tag, 70 URL-safe base64 characters), the requirement ticket (59 characters), and both session token formats (75-character keyed BLAKE3 MAC token, 139-character Ed25519 token);
- the key schedule (keyed BLAKE3 subkeys `sibuna/token/v1` and the others of SID 0002) and node binding of challenge keys;
- Hashcash (`SHA-256(challenge ":" decimal(nonce))` with bit-level difficulty) and the Cohen-Pietrzak proof layout of SID 0006;
- the rule hash (Zig's `std.hash.Wyhash` with its fixed seed), so rule identities stored in tokens agree across implementations;
- the HTTP/1.1 parser's acceptance and rejection set, including every smuggling and chunked-framing case of SID 0009;
- policy evaluation order, WEIGH scoring, WAF categories and the Elm-style diagnostics of SID 0001 and SID 0003;
- every route, status code, header and response body the end-to-end suite observes.

3.2. Workspace

Table 21 Crates and their responsibilities

Crate	Reusable	Responsibility
`sibuna-crypto`	yes	SHA-1, SHA-256 with SHA-NI and ARMv8 dispatch, BLAKE3 (keyed, derive-key), HMAC, Ed25519, Wyhash, constant-time comparison; `no_std`
`sibuna-encoding`	yes	base64 (standard and URL-safe, with and without padding) and hexadecimal; `no_std`
`sibuna-json`	yes	bounded JSON parser with located errors and an escaping writer
`sibuna-pow`	yes	Hashcash and the Cohen-Pietrzak prover and verifier with a fixed workspace; `no_std`
`sibuna-http`	yes	zero-copy HTTP/1.1 request parser, strict chunked decoder, response writer, RFC 6455 frames and handshake
`sibuna-patterns`	yes	tagged Aho-Corasick automata over bytes
`sibuna-cidr`	yes	IPv4 and IPv6 parsing and a radix trie of prefixes
`sibuna-store`	yes	Vyukov MPSC ring, Robin Hood spent set, GCRA limiter, ban table, striped counters

Crate	Reusable	Responsibility
`sibuna-inflate`	yes	DEFLATE and gzip with explicit output bounds
`sibuna-geoip`	yes	IP to country lookup from open CSV datasets and canonical snapshots
`sibuna-html`	yes	escaped HTML snippet rendering; templates are compiled to Rust at build time
`sibuna-core`	no	configuration, command-line parsing, diagnostics, metrics, incident types
`sibuna-policy`	no	rules, loader, evaluation engine, semantic WAF, normaliser, page templates, embeddings
`sibuna-challenge`	no	keys, tokens, challenge coordinator, tickets, adaptive difficulty
`sibuna-net`	no	streaming reverse proxy, upstream pool, forwarded metadata, WebSocket relay
`sibuna-storage`	no	persistent layer over sqlx (SID 0011)
`sibuna-serve`	yes	bounded HTTP/1.1 listener with joinable workers, deadlines, admission classes and WebSocket upgrades
`sibuna-tls`	no	blocking outbound TLS through rustls for the console (SID 0012)
`sibuna-console*`	no	console protocol, server and WebAssembly interface (SID 0007)

Binaries live under `cmd/`: the daemon `sibuna` and the browser solver `sibuna-pow-wasm`. Repository tools are Rust programs under `tools/`, run through `cargo xtask`. Dependencies point one way: reusable crates, then `sibuna-core`, then the application crates, then the binaries.

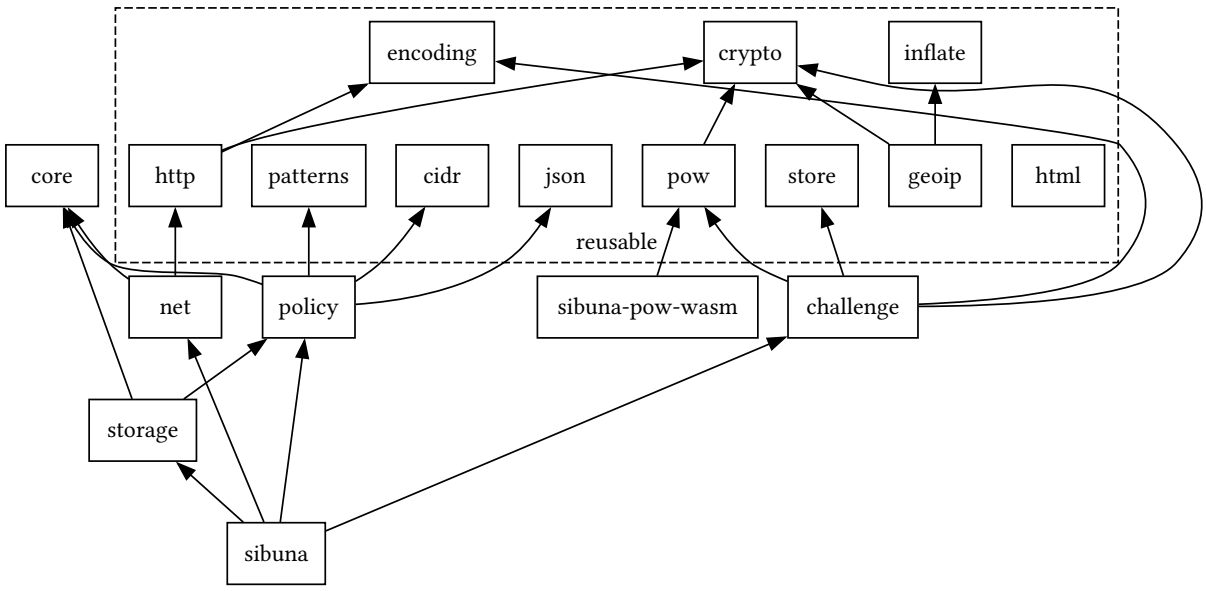


Fig. 4 Crate dependencies. Reusable crates depend on no application crate.

3.3. Mapping of mechanisms

Table 22 Zig idioms and their Rust form

Zig	Rust
slices into the 64 KiB connection buffer	`&'a [u8]` borrowed from the buffer; a parsed `Request<'a>` cannot outlive the bytes it describes, and the compiler rejects a use after the buffer is refilled
error unions and error sets	`Result<T, E>` with one `enum` per module implementing `Display` and `std::error::Error`
`maxInt` and empty-string sentinels	`Option<T>`

Zig	Rust
engine slot with an atomic reader count (read-copy-update)	`Arc<Engine>` published through a slot; the storage thread builds a new engine off the hot path and swaps it in; the old one is freed when its last request finishes
lock-striped fixed tables with spin locks	the same striping with `std::sync::Mutex` per shard (a futex in the uncontended case), unless a benchmark shows a spin lock wins
`threadlocal` sampling state and counter stripe	`thread_local!` with `Cell`
`comptime` tables and the HTML snippet compiler	`const` tables and `const fn`; snippets are compiled by a build script into Rust functions, so template errors (HTML001 to HTML007) stop the build as before
`@embedFile`	`include_bytes!` and `include_str!`
`defer` and `errdefer`	`Drop` and guard values
thread-per-connection with a joinable task pool	the same model on `std::thread`: accept workers hand each connection to a bounded pool of threads, and the slot is reclaimed by joining before reuse

3.4. Concurrency model

The Zig daemon serves each connection on its own thread, bounded by `--max-connections`, with one 64 KiB buffer per connection. We keep this model. It is simple, it isolates a slow client to one thread, and it lets the request path stay synchronous and allocation-free. An asynchronous runtime would add a large dependency to the hot path and change the cost model of every mechanism the earlier records measured. The storage layer (SID 0011) is the only place that runs an asynchronous runtime, on its own thread, and the request path never waits on it.

3.5. Hardware acceleration

Zig's standard library selected SHA-NI and the ARMv8 SHA-2 instructions at build time. We detect them at run time with `is_x86_feature_detected!` and `is_aarch64_feature_detected!` and fall back to a portable implementation, so one binary is fast on every machine it runs on. Hashcash verification, PoSW verification and the challenge proof are SHA-256 bound, so this is where hardware acceleration pays. Other SIMD use (for example a vectorised byte-class pre-scan before WAF inspection) is added only with a benchmark that shows the gain, and lives in a module of its own with `// SAFETY:` comments on every `unsafe` block.

3.6. The browser solver

`sibuna-pow-wasm` is a `no_std` `cdylib` for `wasm32-unknown-unknown`. It exports the same functions as the Zig module (`sibuna_get_buffer_ptr`, `sibuna_get_buffer_len`, `sibuna_solve_step`, `sibuna_solve_sha256`, `sibuna_posw_solve`, `sibuna_posw_proof_ptr`), so the worker script and the JavaScript fallback are unchanged. It compiles the same `sibuna-pow` sources that the server uses to verify, so prover and verifier cannot drift apart. It uses no `wasm-bindgen`; the module is built for size.

3.7. Dependency policy

Every external crate is a liability: it must be reviewed, updated and trusted. The data plane uses none. A dependency is admitted only when it is mature, self-contained and solves a genuinely large problem, and the reason is recorded next to it in `Cargo.toml` and in a design record. Storage is such a case: we use `sqlx` (SID 0011), confined to `sibuna-storage` behind cargo features, so `cargo build --no-default-features` produces a daemon with no external crate at all.

3.8. Engineering standards

The standards of SID 0001 carry over, written for Rust:

- A function body has at most 70 logical lines; a file at most 1,408 lines; a line at most 99 characters (`rustfmt.toml` sets `max_width = 99`). `cargo xtask check` enforces the limits, and `clippy.toml` sets `too-many-lines-threshold = 70`.
- Every public item has a `///` comment saying what it borrows, what it returns and how it fails.
- No `unsafe` outside FFI and measured SIMD modules; every `unsafe` block has a `// SAFETY:` comment.

- No `unwrap()` or `expect()` on input-dependent values in library code. Libraries return structured errors and never print. The binaries render Elm-style diagnostics.
- Plain-data APIs: public-field structs with `Default`, a small verb vocabulary (`load_*`, `begin_*`/`end_*`, `is_*`, `*_count`), no `get_` prefix.
- Bounded work on untrusted input: explicit limits that return errors, never panics or unbounded allocation.
- `cargo build`, `cargo test`, `cargo clippy --all-targets -- -D warnings`, `cargo fmt --check` and `cargo xtask check` pass before a change is merged. GitHub workflows may use shell or Python glue; repository tools are Rust.

3.9. Memory management that Rust makes unnecessary

Much Zig code existed only to manage memory by hand or to avoid footguns: `deinit` and `errdefer` chains, arenas reset per rebuild, double-buffered engine slots reused to avoid reallocating several megabytes, `undefined` initialisation, alignment workarounds for heap fixtures, and comments of the form "this slice must not outlive the next read". The port does not carry this machinery over. Ownership, borrowing, `Drop` and values built once off the hot path give the same guarantees, checked by the compiler. Each crate's documentation names the Zig mechanisms it removed and what replaced them. The observable behaviour (bytes on the wire, decisions, bounds and error texts) does not change, and the request path still does not allocate.

3.10. Tools and documentation

Repository tools are Rust programs under `tools/`, in the pattern guidedog uses: `tools/xtask` dispatches `cargo xtask <task>` (`check`, and later `wasm` and `bench`), and standalone tool crates sit beside it. The Zig repository's Python and shell drivers are not ported as scripts. End-to-end tests become Rust integration tests, and browser checks drive headless Chromium from Rust over the DevTools protocol, reusing the workspace's WebSocket and JSON crates. Python is allowed only in GitHub workflow glue, or in a benchmark driver when no reasonable Rust alternative exists.

Documentation is not code. The design records are a guidedog discussion catalog (`docs/sid`, managed with `guidedog dsc`) and the book and manual are guidedog projects; their PDFs use the Sibuna Typst design in each project's `_templates/`. The repository contains no documentation build code: the Zig `tools/sid.zig` and the `zig build sid*` and `book` steps have no Rust counterpart.

3.11. The OWASP Core Rule Set

Sibuna's WAF is its own structural inspection (SID 0004), not the OWASP Core Rule Set, and the port keeps it that way. We evaluated the external SecLang engines available to Rust (pure-Rust engines a few months old with a single maintainer each, and bindings to libmodsecurity or Coraza) and admit none of them: an inspection engine on an edge server is security-critical code we must own, review and maintain, as the Zig daemon owned its own. If Sibuna adds compatibility with CRS rules, it does so in-house: a new design record specifies the subset of SecLang, the transformations, the operators and their algorithms (for example the existing Aho-Corasick automata for phrase matching and linear-time matching for expressions), and how CRS anomaly scores map onto WEIGH decisions; the implementation follows that record.

4. Rationale and alternatives

We considered porting zaxonlite as well. It is a 48,000-line Zig project of its own with a consensus library; porting it would double the work and keep an unusual storage engine in a firewall whose state is small. SID 0011 explains the replacement.

We considered `tokio` with `hyper` for the data plane. The Zig daemon's measured performance comes from a zero-copy parser, a fixed per-connection buffer and allocation-free decisions, none of which an asynchronous runtime improves; `hyper` would also take the request bytes away from the inspection code. We keep the thread model and measure it against the Zig daemon before considering an event loop. If the comparison shows a gap that the thread model cannot close, an `epoll` loop over the same parser is the next step, recorded in a new SID.

We considered `wasm-bindgen` for the browser modules. The solver and the console already use a small fixed bridge (shared buffers and integer exports); `wasm-bindgen` would add generated glue and a build tool without adding a capability.

5. Security and operational considerations

Bit-identical formats are a security property as well as a convenience: a mixed Zig and Rust cluster during migration must not accept a token one node minted under different rules. The test vectors pin each format. Constant-time comparison is used wherever the Zig code used `timing_safe.eql`. Secrets are zeroed on drop where the Zig code used `secureZero`.

6. Backwards compatibility

Command-line options, routes, the policy file format and wire formats are unchanged, except for the storage options that SID 0011 replaces. A data directory written by the Zig daemon (Zaxonlite files) is not read by the Rust daemon; operators export policies with `sibuna console policies export` before migrating and import them afterwards.

7. Acceptance criteria and implementation status

The port is accepted when:

1. Every Zig unit test has a Rust counterpart that passes, and the end-to-end suite passes against the Rust daemon.
2. Cross-implementation tests show that a Zig node and a Rust node sharing a seed verify each other's session tokens and requirement tickets.
3. The browser solver passes in a headless browser with WebAssembly and with the JavaScript fallback.
4. The benchmarks of SID 0002 (primitives, and whole-product runs under `wrk`) show the Rust daemon at or above the Zig daemon's throughput, with p99 latency no worse, on the same machine; the results are recorded under `benchmarks/results/`.
5. The console passes its golden fixtures and its browser workflows in a headless browser, with screenshots reviewed.

Status: all five criteria are met (evidence below, 2026-10-15). Committed on 2026-10-16 after the maintainer's review; released in 1.0.0-beta.1.

Evidence recorded so far (2026-10-15):

- Criterion 2: Rust and Zig daemons sharing a seed verify each other's MAC and Ed25519 session tokens and requirement tickets (`crates/challenge` cross-implementation tests against the Zig baseline binary).
- Criterion 3: `cargo xtask browser` drives Chromium 155 headless through the interstitial; the challenge passes with WebAssembly (solve 29 to 34 ms) and with the JavaScript fallback (84 to 91 ms, WebAssembly disabled). Detection relies on the default interstitial markup (`#status`, `.progress-bar`).
- Differential behaviour: `cargo xtask parity` replays generated corpora against both daemons in 17 configurations. At commit `dda1162` four seeds (three of 20,000 requests and one of 60,000) gave no unexplained difference; every allowed difference is listed with its reason in `tools/parity/allow-list.json`.
- Criterion 4, met: on an AMD Ryzen 9 5950X with the daemons on CPUs 0 to 3 and the load on CPUs 8 to 15 (`benchmarks/results/`, 2026-10-15), the Rust daemon is at or above the Zig daemon's throughput in all 16 whole-product workloads (ratios 1.00 to 1.09) with p99 no worse (ratios 0.46 to 1.00) and 4 to 18% less CPU per request; peak memory is about half (16 MiB against 31 MiB). At equal offered load from 10,000 to 150,000 requests per second, every percentile from p50 to p99.9 is at or below Zig's, after the connection threads were made reusable (SID 0002 revision note; before that, p99.9 at 100,000 requests per second was 0.3 to 2 ms against Zig's 0.1 ms). Of the primitive rows, all are faster than Zig except `gcra_check` and `gcra_four_rule_scopes` (12.8 against 5.7 ns), whose byte API now includes the keyed SipHash-1-3 of the client (a security fix); the daemon hashes once per request, and the hashed rows measure 6.2 ns.
- Criterion 5, met: the console's 290 interface tests, including the golden fixtures, pass. `cargo xtask browser console` passed all 17 steps against a fresh daemon in Chromium 155 (no WebSocket and no map request before sign-in, the forced password change, live tiles across stats deltas, every page with no exception, console error or CSP violation, the wall display limited to statistics, the WebSocket closed at sign-out), and we reviewed its screenshots; the review led to three interface fixes (sparkline width, the Audit and Tokens scroll position, the heading focus ring). The eight-dashboard smoke run is in `benchmarks/results/console-8-dashboards-2026-10-15.md`.

- Criterion 1, met: every one of the 567 named Zig tests (`test "..."` blocks under `libs/` and `apps/`) is mapped in `docs/sid/assets/0010-zig-test-map.tsv`: 540 have a Rust test that checks the same behaviour, 13 had none and were ported on 2026-10-15 as 11 Rust tests, and 14 do not apply, each with the record that changed or removed the mechanism (Zaxonlite and the consensus log, Zig data migrations, the comptime JSON decoder, SQL quoting replaced by bound values, and the like). One ported test found a deviation, now fixed: a retried incident batch mapped its country again instead of keeping the one it was first mapped with (SID 0013). The workspace suite (about 2,100 tests on SQLite, and 700 more against PostgreSQL) and the daemon and console end-to-end suites pass.
- Differential behaviour, latest: at commit `f5fa5be`, `cargo xtask parity` (seed 20261015, 20,018 requests, 17 configurations) gave no unexplained difference.

8. Open questions

- Closed 2026-10-15: the spent set and the limiter use `std::sync::Mutex` (see the review history for the measurements); the ban table keeps its seqlock readers with a mutex for writers.
- Closed 2026-10-15: no `epoll` event loop is needed for parity. Blocking threads, reused across connections, meet criterion 4.

9. Review history

- 2026-10-15: drafted with the start of the port.
- 2026-10-15 (Rust port): each crate lists its deliberate changes from Zig in its crate documentation, and the design records that those changes touch carry dated revision notes (SID 0002 to SID 0007, SID 0009, SID 0011, SID 0012). Evidence for the first open question: `sibuna-store` measured `std::sync::Mutex` against a spin lock on a Ryzen 9 5950X. The spin lock saves about a nanosecond per uncontended operation but loses under contention (the spent set at 64 threads: 11.2 against 6.9 million operations per second; GCRA on six hot addresses at 64 threads: 37.4 against 6.5 million), so the spent set and the limiter use the mutex; the question is closed in favour of the mutex. The HTML snippet compiler of `sibuna-html` adds the code `HTML008` (a snippet file name must yield Rust item names) to `HTML001` to `HTML007`.
- 2026-10-15 (security review): limiter and ban-table placement is process-local, keyed per table, and not part of any wire format; only the rule hash keeps Zig's `std.hash.Wyhash` with its fixed seed. The Vyukov ring remains the store's only non-FFI `unsafe`: `std::sync::mpsc::sync_channel` was measured to cost 20 to 45% more per push with 16 producers (`tests/timing.rs`, `bounded_queue_producers`). An adversarial review (2026-10-15) found request-path weaknesses inherited from Zig (trusted forwarded address, origin pool reuse, request-target normalisation, WAF canonicalisation bound, keyed tables, honeypot cross-site requests, adaptive input, connection limits); each fix is recorded in the SID that specifies the mechanism.
- 2026-10-15 (console port): `cmd/sibuna` features are `default = [sqlite, console]`; `console` enables storage and the crates `sibuna-console`, `sibuna-console-protocol`, `sibuna-console-storage` and `sibuna-json`. `--no-default-features` stays free of external crates. The console server is the application crate `sibuna-console`; TOTP joins `sibuna-crypto` and the range-to-prefix split joins `sibuna-geoip`, both reusable.
- 2026-10-15 (console port): `sibuna-crypto` gains `totp` (RFC 4226 and 6238, HMAC-SHA1, six digits, 30 s steps, base32 seeds) for the console's second factor; `sibuna-geoip::prefixes` holds the range-to-prefix split used by country actions.
- 2026-10-15 (console port): `sibuna-tls` has a `testing` feature, `testing::Ingress`, a TLS-terminating test proxy with the fixed test certificates; only dev-dependencies enable it.
- 2026-10-15 (console port): `sibuna` has an optional build-dependency on `sibuna-geoip` (console feature) to validate the embedded GeoIP snapshot. The console's end-to-end suite is `cmd/sibuna/tests/console_*.rs`; its PostgreSQL cluster scenarios run when `SIBUNA_TEST_DATABASE_URL` is set. With the console compiled in, disabled, storage only, or enabled and idle, the 16 product workloads stay within about 1% of the daemon built without it, which is within noise.

10. References

- SID 0001: the Sibuna Discussion process and engineering standards.
- SID 0002: foundation architecture and performance contract.

- SID 0006: mathematical foundations.
- SID 0009: chunked request bodies.
- SID 0011: storage through sqlx.
- The Rust Reference and the Rust standard library documentation, `std::arch` (runtime feature detection).

Lifecycle event

> 2026-10-15: prediscussion -> discussion. Specification of the port, drafted
> before the code

Lifecycle event

> 2026-10-16: discussion -> accepted. Implemented and released in Sibuna
> 1.0.0-beta.1; reviewed by the maintainer on 2026-10-16.

Lifecycle event

> 2026-10-16: accepted -> committed. Implemented and released in Sibuna
> 1.0.0-beta.1; reviewed by the maintainer on 2026-10-16. All five acceptance
> criteria met, with the evidence recorded in the record (test map
> docs/sid/assets/0010-zig-test-map.tsv, cross-implementation token tests,
> cargo xtask browser, benchmarks/results); release v1.0.0-beta.1