`hadlink`: A Minimal, High-Assurance URL Redirection Service

Jacob Seman

Abstract

URL shorteners are deceptively small systems with disproportionate security consequences. Most implementations conflate policy and mechanism, trust input too broadly, and are difficult to reason about. `hadlink` is a URL redirection service designed for infrastructure use cases (CI/CD pipelines, QR codes, SMS notifications) where deterministic behavior, security, and auditability matter more than feature breadth. Its core validation and encoding logic is formally verified using SPARK Ada, while the service layer is composed in Haskell with conservative dependencies. This paper describes the system's design philosophy, assurance model, and deployment approach.

1 Problem Statement

Any service that accepts arbitrary URLs and redirects traffic to them becomes, in effect, an open relay. URL shorteners are frequent abuse targets for spam campaigns, phishing attacks, and malware distribution. The opacity of the short code, hiding the destination until the redirect occurs, is precisely what makes them useful and precisely what makes them dangerous. Despite this, most shorteners are deployed casually, with minimal validation and no formal reasoning about their behavior.

Most URL shortener implementations share common weaknesses. They conflate policy and mechanism, mixing validation logic with routing logic with storage logic in ways that make reasoning about behavior difficult. They trust input too broadly, accepting URLs that point to private network addresses, contain embedded credentials, or use non-HTTP schemes. They offer no formal specification of what the system guarantees, leaving behavior to vary unpredictably with input.

The problem is small enough to be tractable for formal methods. A URL shortener's core logic (validating input, generating short codes, storing mappings, and performing redirects) can be specified precisely and verified completely. The effort is justified because the consequences of getting it wrong extend beyond the service itself to every system that trusts its output.

2 Design Goals and Non-Goals

2.1 Goals

`hadlink` targets infrastructure use cases where predictability matters more than features. Four goals guide the design. First, deterministic behavior: the same input URL always produces the same short code, enabling idempotent operations and predictable caching. Second, a minimal trusted computing base: the verified core contains no I/O, networking, storage, or concurrency, limiting the attack surface to well-understood components. Third, explicit failure modes: all errors are enumerated, and the system never fails silently or returns ambiguous results. Fourth, constrained deployability: a single binary with SQLite storage runs on minimal hardware with no external service dependencies.

2.2 Non-Goals

Equally important are the things `hadlink` will not do. It provides no marketing analytics or click tracking. It has no user accounts, dashboards, or authentication layer. It does not support custom aliases, vanity URLs, or dynamic policy engines. It performs no JavaScript redirects, link previews, or interstitial pages. It does not aim for feature parity with SaaS shorteners like Bitly or TinyURL.

These are conscious trade-offs, not omissions. Each excluded feature would add complexity, expand the attack surface, or conflict with the determinism requirement. Non-goals are documented in the repository and enforced in the contribution policy: pull requests that conflict with stated non-goals are closed with explanation, not negotiated.

3 Architecture

`hadlink` uses a two-layer design with intentional separation between verified core logic and service composition.

3.1 SPARK Core

The formally verified core handles URL canonicalization and short code generation. Canonicalization validates the URL scheme (HTTP or HTTPS only), rejects URLs pointing to private addresses (RFC 1918, RFC 4193, link-local), and detects embedded credentials. Short code generation uses HMAC-SHA256 via SPARKNaCl, producing deterministic 8-character Base62 codes.

The SPARK core contains no I/O, networking, storage, concurrency, or configuration parsing. It is pure computation with explicit preconditions and postconditions [1]. All 137 proof obligations are verified automatically by the CVC5 SMT solver.

3.2 Haskell Service Layer

The service layer composes the system from conservative dependencies. HTTP handling uses Warp, chosen for its minimal API surface and proven reliability. Storage relies on SQLite in WAL mode for its simplicity and reliability. Rate limiting, structured logging, and proof-of-work validation are implemented in Haskell and tested via Hedgehog property tests.

The service layer trusts the SPARK core's postconditions: if canonicalization succeeds, the URL satisfies all security properties. This trust boundary is explicit and documented.

3.3 Foreign Function Interface

The Foreign Function Interface (FFI) between layers is minimal and frozen at API version 1. Three C functions are exported: `hadlink_canonicalize`, `hadlink_make_short_code`, and `hadlink_api_version`. The FFI marshaling code is explicitly outside SPARK verification (`pragma SPARK_Mode Off`) and is tested separately.

A freeze test in the test suite ensures the interface does not change without a corresponding version bump. The FFI is thin by design: data crosses the boundary as C strings with explicit length parameters, and all memory is caller-allocated.

3.4 Split Binaries

Two separate executables serve different trust profiles. `hadlink-redirect` handles the read path, looking up short codes and returning redirects. It is fast and stateless, linking no SPARK code and carrying no dependency on `libHadlink_Core.so`. This binary is designed to be exposed to the public network.

`hadlink-shorten` handles the write path: it validates URLs, generates short codes, and stores mappings. It includes the SPARK FFI, rate limiting, and proof-of-work validation. It is designed to be restricted to internal networks or authenticated access.

This separation is least-privilege at the binary level. The component exposed to untrusted traffic has the smallest possible attack surface.

3.5 Storage

SQLite in WAL mode provides the storage layer. The schema contains three columns: the short code serving as primary key, the canonical URL, and a creation timestamp. The write path is append-only, with no update or delete operations supported.

`INSERT OR IGNORE` provides idempotent creation. Because short codes are deterministic (same URL always produces same code), duplicate requests are harmless: they either insert a new row or silently succeed when the row already exists. The redirect service opens the database read-only.

4 Formal Assurance

The SPARK core provides machine-checked guarantees about the system's behavior. Every stored URL is proven to have a valid HTTP or HTTPS scheme, contain no embedded credentials, and point to no private network address. Short codes are guaranteed to be exactly 8 Base62 characters. All array accesses are proven in-bounds, and the encoding logic is verified free of integer overflow.

Check type	Count
Range checks	31
Overflow checks	30
Preconditions	29
Loop invariants	18
Index checks	11
Postconditions	9
Other (assert, length, division, init)	9
Total	137

Table 1: Proof obligation breakdown by check type.

The verification boundary is explicit about what is not proven. FFI marshaling is outside SPARK verification and tested separately. The Haskell service layer is property-tested via Hedgehog but not formally verified. SQLite is a trusted dependency. Network-level properties (TLS termination, DNS resolution) are outside the system's scope.

Assumptions are confined and documented. The codebase contains exactly two `pragma Assume` statements, both in a ghost lemma that generates no runtime code. Both document pure function determinism: if two inputs are equal, their outputs are equal. There are zero assumes in business logic.

The project references DO-278A (Software Integrity Level 3) as an architectural guide for separation of concerns and evidence requirements. It is not certified; this is a single-developer project without independent verification resources. The formal methods provide confidence, not certification.

5 Deployment

`hadlink` supports three deployment methods. Docker images use pre-built binaries from GitHub Releases or can be built from source. Systemd unit files provide direct installation with security hardening. An Arch Linux AUR package (`hadlink-bin`) offers distribution-native installation.

The systemd configuration applies defense-in-depth hardening: `NoNewPrivileges`, `ProtectSystem=strict`, `ProtectHome`, `PrivateTmp`, and `MemoryDenyWriteExecute`. Resource limits are conservative: 128MB for the shorten service, 64MB for redirect. Both services restart on failure with a 5-second delay.

Configuration follows a minimalist approach, using environment variables only with no configuration file parser to introduce complexity or parsing vulnerabilities. Secret management uses systemd's `EnvironmentFile` directive, keeping secrets out of unit files and command lines.

The redirect service is stateless and recovers instantly from restarts. The shorten service holds no persistent in-memory state; its STM-based rate limiter rebuilds on startup. Pre-built binaries are available from GitHub Releases, so deployment requires no build toolchain. Operators can run `hadlink` without installing GHC, GNAT, or any development dependencies.

6 Limitations and Future Work

`hadlink` has intentional limitations that preserve its design properties. SQLite's scaling envelope is sufficient for infrastructure use cases but not for high-volume marketing workloads processing millions of redirects per second. There is no built-in abuse detection; rate limiting and proof-of-work mitigate automated abuse but do not detect malicious destination URLs. The system is single-node only, with no replication or distributed deployment. URLs cannot expire or be deleted; the append-only design ensures that short codes are permanent and deterministic.

These are conscious trade-offs. SQLite's simplicity eliminates an entire class of distributed systems failures. The absence of abuse detection keeps the system policy-neutral, allowing operators to layer their own detection upstream. Single-node deployment means no consensus protocol, no split-brain scenarios, no coordination overhead. Permanent URLs mean short codes can be printed on physical media without expiration concerns.

Potential future work includes an LMDB backend for higher read throughput, native TLS termination (currently the system relies on a reverse proxy), and a Prometheus metrics endpoint for operational visibility. None of these would change the core assurance model.

7 Conclusion

`hadlink` demonstrates that formal methods are tractable for small infrastructure services. The system prioritizes correctness over features and treats simplicity as a security primitive. By confining verified logic to a pure computational core and composing the service layer from conservative dependencies, it achieves high assurance without the overhead typically associated with formal verification.

The approach (SPARK core with explicit contracts, thin FFI boundary, Haskell composition layer) is applicable to other services with similar trust requirements: authentication tokens, configuration validators, cryptographic utilities, and anywhere a small component has disproportionate security impact. The question for such systems is not whether formal methods are worth the effort, but whether the alternative, trusting unverified code in critical paths, is acceptable.

Availability

`hadlink` is open source and available at `https://github.com/Jbsco/hadlink`.

References

[1] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*, Release 1.2. AdaCore, 2022. `https://www.adacore.com/uploads/books/Spark-Guidance-1.2-web.pdf`