

Specs Are All You Need: Leveraging LLMs, Lean, and POSIX/W3C Specs to Find Bugs in Coreutils and WebKit/Safari

Authors: Chad Brewbaker and Suhaas Teja Vijjagiri **Affiliation:** The Secure Program Synthesis Hackathon, May 2026

Abstract

We demonstrate that grounding LLMs with authoritative specification documents and Lean 4 theorem proving produces meaningful software quality improvements on long-standing, widely-deployed codebases. Using only POSIX SUSv4-2018, RFC, WHATWG, and W3C specifications as our grounding material – plus Lean 4 source code for additional LLM grounding – we found 14 specification-level bugs in WebKit/Safari parsers and 4 confirmed implementation bugs in MIT Rust coreutils. Our three-layer methodology (source analysis, spec grounding, formal verification) revealed systemic patterns: unchecked I/O return values across 205+ utilities, underspecified security defaults in RFC 8941 SameSite cookies, and complete absence of parsing rules for FTP LIST output. Two independent formalizations – our I/O-focused approach and the coauthor’s spec-to-theorem empirical testing – converged on complementary bug categories, demonstrating the robustness of the methodology. This work shows that technical specifications, when formally interpreted through LLM-assisted theorem proving, serve as both correctness oracles and specification improvement tools for codebases that have never been formally verified.

1. Introduction

WebKit/Safari and POSIX core utilities are foundational infrastructure: one serves billions of users through every modern browser, the other underpins every Unix and Linux system. Both have been maintained for decades, both are widely trusted, and neither has been formally verified.

We asked: *Can LLMs, guided by authoritative specifications and Lean 4 theorem proving, find real bugs in these mature codebases?* Our answer is yes. Using only publicly available RFCs, WHATWG/W3C specifications, and SUSv4-2018 as grounding material – we produced formal specifications that surfaced 14 specification ambiguities and 4 confirmed implementation bugs.

Our contributions:

1. 15 WebKit parser modules formalized in Lean 4 (5,441 lines, 207 theorems), revealing 14 specification bugs including 2 CRITICAL security issues in SameSite cookies and FTP LIST format.

2. 515 POSIX core utilities surveyed, 316 found with critical I/O gaps. A systemic pattern: 205+ utilities ignore `printf` return values, risking silent data loss.
3. 111 Rust utils formalized (779 Lean files), with wc receiving the most thorough treatment (889 lines, 2 confirmed bugs: splice EINTR undercount, 32-bit overflow).
4. Independent wc verification by coauthor (4 Lean modules, 21 empirical tests) confirming 2 POSIX violations in utils/coreutils: `-c/-m` mutual exclusion silently violated, and a unicode counting anomaly.
5. A convergent methodology: two independent wc formalizations using different approaches found complementary bugs, demonstrating robustness.

2. Related Work

LLM-assisted formal verification. Prior work has explored LLMs for proof generation (LeanDojo, ProofNet) and specification extraction from natural language. Our work differs by using LLMs as a *specification-grounded bug-finding tool* rather than a proof automation engine – the LLM reads the RFC/POSIX spec, writes the Lean formalization, and the process of formalization itself surfaces bugs.

WebKit security analysis. Fuzzing (OSS-Fuzz) and symbolic execution (KLEE) have found WebKit bugs, but these approaches test implementations against expected behavior. Our approach is complementary: we formalize the specification first, then check both spec and implementation. This reveals spec bugs that no fuzzer would find.

POSIX compliance. Existing POSIX checkers verify API usage but do not model I/O semantics. Our Lean specifications model return values, error conditions, and buffering behavior – enabling proofs about program behavior under failure.

What distinguishes our approach: We use LLMs to translate authoritative specifications into executable Lean 4 theorems, then use the formalization process itself to discover that the specifications are incomplete or contradictory. The bugs we found exist in specs that have governed software for decades.

3. Methodology

Three-Layer Verification Framework

1. **Source analysis:** Read the implementation (WebKit C++, POSIX C, Rust utils) to understand actual parsing and I/O logic.
2. **Specification grounding:** Locate the governing standard (RFC, WHATWG, W3C, SUSv4-2018) and extract ABNF grammars, algorithmic descriptions, and normative requirements.

3. **Formal specification:** Write Lean 4 functions mirroring the implementation, grounded in the spec. Each ABNF rule maps to a Lean function; each normative requirement maps to a theorem.

Two Independent wc Formalizations

Our project as a “toy problem”, formalized wc through an **I/O semantics approach** (modeling BufRead, splice, fstat, exit codes – 889 lines across 8 files), and aa **spec-to-theorem approach**: extract POSIX requirements, express as Lean theorems, then empirically test against the utils binary (4 modules, 21 tests).

These different approaches found different bugs – demonstrating that specification-grounded formalization is robust to methodological choices.

What Didn’t Work According to Hermes Agent with Qwen3.6 27B describing difeculties it ran into

- **Automated ABNF-to-Lean:** ABNF repetition counts and grouping don’t map cleanly to Lean functions.
- **String-based parsing:** Lean’s `String` is not a data type, preventing structural induction. We switched to `List Char` operations. (Finding found by Qwen3.6 27B attempting to explain it’s struggles with LEAN4 syntax)
- **Naive decide proofs:** Parser theorems require unfolding multiple function layers. Custom `simp` lemmas were needed.

4. Results

WebKit: 14 Specification Bugs Across 15 Parsers (Note this is preliminary. It likely contains false positives.)

Parser Module	Lines	Theorems	Governing Spec
HTTP/1.1 Messages	558	11	RFC 9112/9110
HTTP Range Header	593	51	RFC 9110 14.2
SameSite Cookies	277	18	RFC 8941
Content-Type	380	14	RFC 9110 8.3
WebSocket Protocol	383	10	RFC 6455
(+ 10 more)	2,450	103	Various

CRITICAL bugs found:

- **BUG-SS-001:** RFC 8941 says unknown SameSite values should be treated as “not present” but never defines what that means. An attacker could craft `SameSite=Srtict` (typo) to silently downgrade a Strict cookie, exploiting browser divergence.

- **BUG-FTP-001:** RFC 959 says FTP LIST output is “considered identical to Unix `ls -l`” but provides no ABNF or parsing rules. Every FTP client reverse-engineers server formats. WebKit supports 8+ because no standard exists.

Additional findings: 4 HIGH bugs (whitespace in SameSite values, error recovery in Server-Timing, duplicate parameter handling contradicting RFC 2045, WebKit silently dropping multi-range requests), 5 MEDIUM, 3 LOW.

Core Utilities: Systemic I/O Bug Pattern

515 utilities surveyed across 8 directories. 316 have critical I/O gaps – they use I/O functions whose return values are not checked:

Uncovered In	Call	Impact
205 utilities	<code>printf</code>	Silent output loss on disk full/pipe break
176 utilities	<code>fprintf</code>	Same
109 utilities	<code>fputs</code>	Same
68 utilities	<code>fclose</code>	Buffer data lost on error

This is a decades-old pattern: when stdout encounters a write error, utilities continue silently producing incomplete output. POSIX requires checking return values, but 205+ utilities do not.

wc: Two Approaches, Four Confirmed Bugs

Our I/O-focused formalization (889 lines, `src/uu/wc/formal/`): - BUG-1: `splice()` EINTR returns partial byte count (signal interrupts counting) - BUG-3: 32-bit overflow on files > 4GB (`usize` wraps on 32-bit systems)

Coauthor’s spec-to-theorem approach (github.com/suhaasteja/wc-formal-spec): - BUG-A: `-c` and `-m` accepted together despite POSIX mutual exclusion (`wc [-c|-m]`). `utils` silently accepts both, produces ambiguous output, exits 0. - BUG-B: On unicode input with `-c -m`, `-c` is silently dropped and a mysterious count (matching neither bytes nor chars) appears in the second field.

The two approaches found different categories of bugs: ours found I/O path issues (`splice`, overflow), the coauthor’s found spec compliance issues (flag handling, counting logic). Together they provide more comprehensive coverage than either alone.

What Grounding Was Most Useful (According to Qwen 3.6)

ABNF grammars (best): Each ABNF rule maps directly to a Lean function. The HTTP Range module (593 lines, 51 theorems) was almost entirely ABNF-driven.

Algorithmic specs (very good): WHATWG’s executable pseudocode reads like code. The URL parser (276 lines) was straightforward.

SUSv4-2018 function specs (good): Clear pre/post conditions for I/O operations.

Natural language prose (problematic): W3C specs and informative RFC text left critical behavior undefined. BUG-ST-001 (error recovery) and BUG-RQ-002 (suffix-length 0 contradicts its own ABNF) are examples.

Lean 4 Challenges

7 standard library gaps in Lean 4.30: no `Nat8`, no `List.findIndex`, no `Char.ofNat!`, no `String.substring`, deprecated `String.trim`, no pipe operator, reserved keywords (`from`, `prefix`).

5 proving challenges: `String` is not a data type (no structural induction), strict match indentation, `Option` pattern matching requires explicit constructors, computation-heavy theorems need layered unfolding, type class inference fails on `UInt32.toNat`.

Lake4 bug: `LEAN_PATH` not auto-populated. Fix: `path="src"` in `lakefile.toml`.

5. Scaling and Conclusions

How This Scales

1. **Template generation:** ~40% of each Lean file is boilerplate (character classes, string helpers). A library could help here. These are due to Hermes Agent and Qwen3.6 writing boilerplate specs then refining them.
2. **Modular theorem organization:** `HTTPRange` (51 theorems) and `wc` (30+ theorems across 4 modules) show large theorem sets are manageable when split by category (character classes, parsing, edge cases, integration). Use of a Makefile also radically cut down on build time for smaller collections of programs.
3. **Automated sorry resolution:** `simp`; `try decide` resolves many placeholders. The `HTTP/1.1` module achieved 0 sorries. (Qwen 3.6 observation)
4. **Empirical test integration:** Following the coauthor’s model, each formal module should include automated tests confirming theorem predictions against the actual implementation.
5. **Priority-driven completion:** 16 remaining P0 security-critical WebKit parsers need ~8-12 more modules (estimated 2-3 weeks). 316 POSIX utilities with I/O gaps need thick specs. (That is Qwen’s way of saying we didn’t have enough time to do long LLM passes on every part of the codebase)

6. **LLM-assisted scaling:** The LLM reads the spec, writes the Lean, finds the bugs – this pipeline scales linearly with specification quality.

Thesis Restated

Grounding LLMs with authoritative technical specifications and Lean 4 theorem proving produces meaningful software quality improvements on codebases that have never been formally verified. The process of formalization itself is a specification improvement tool: we found 14 bugs in specs that have governed software for decades.

The convergence of two independent wc formalizations – using different methodologies but the same grounding principle (specs as ground truth) – demonstrates that this approach is robust. Different methods find different bugs; together they provide comprehensive coverage.

The formal specifications serve dual purposes: as **correctness oracles** (verifying implementations match specs) and as **specification improvement tools** (identifying ambiguities in the governing standards). This dual role makes the investment productive even before all theorems are proven.

Limitations

- 15 of 53 WebKit parser categories formalized; 38 remain (Mac Studio running Qwen 3.6 27B over the course of the weekend)
- 138 of 207 WebKit theorems use **sorry** placeholders
- 515 utilities surveyed but only 11 have thick hand-written specs
- Lean version fragmentation (4.30-rc2 vs 4.14) creates minor friction

Code and Data

- **WebKit formal specs:** WebKit/formal/WebKitFormal/ (13 modules, 5,070 lines)
- **WebKit findings:** WebKit/formal/SPEC_BUGS.md (387 lines, 14 bugs) (We will upload after manual verification)
- **Core utilities:** src/uu/*/formal/ (111 utilities, 779 Lean files)
- **wc findings:** src/uu/wc/formal/BUGS.md (7 issues, 2 confirmed)
- **Coauthor’s wc:** github.com/suhaasteja/wc-formal-spec
- **POSIX/RFC reference:** susv4-2018/functions/, rfc_library/

References

1. RFC 9110: HTTP Semantics. <https://www.rfc-editor.org/rfc/rfc9110.html>
2. RFC 8941: SameSite Cookie Attribute. <https://www.rfc-editor.org/rfc/rfc8941.html>
3. WHATWG URL Standard. <https://url.spec.whatwg.org/>
4. SUSv4-2018. <https://pubs.opengroup.org/onlinepubs/9699919799/>
5. Lean 4. <https://leanprover-community.github.io/>

6. `uutils/coreutils`. <https://github.com/uutils/coreutils>
7. `suhaasteja/wc-formal-spec`. <https://github.com/suhaasteja/wc-formal-spec>

LLM Usage Statement

We used Hermes Agent and Qwen3.6 27B to draft formal specifications, debug Lean 4 code, and write portions of this report. The coauthor used Claude-assisted development.