

Bugzy: Mining Executable Security Specifications from Audit Findings

A commit-grounded pipeline for turning Code4rena access-control reports into self-validated Halmos property tests

Demiladepy

Independent · SPS Hackathon Submission, May 2026

Abstract

Smart contract audits encode thousands of implicit security specifications in natural language. These properties are fixed once and rarely reused. We study whether modern LLMs can extract formal specs from audit prose, compile them into symbolic tests over real protocol code, and self-validate them against public git history (vulnerable commit vs. fix commit). We introduce **Bugzy**, a six-stage pipeline (fetch → extract → codegen → verify → validate → generalize) evaluated on N=20 access-control findings from Code4rena. Extraction succeeds on 20/20 findings; automated test codegen on 18/18 non-curated entries. Two findings reach full self-validation: Halmos returns *FAIL* on the vulnerable commit and *PASS* on the fix. On the decidable subset where both verification legs complete, validation precision is **100% (2/2)**. End-to-end completion on the full corpus is **10% (2/20)**. We characterize the remaining 18 cases: failures are dominated by build heterogeneity, missing fix commits, and Halmos harness integration—not spec extraction. We argue Bugzy occupies an orthogonal position to zero-shot detectors such as Slither, Mythril, and bare LLMs.

1. Introduction

Each Code4rena contest produces dozens of high-severity findings. A typical report states a violated invariant in prose—for example, “*only the owner may call _authorizeUpgrade*” or “*the owner must be able to call setTopNPools*”. Sponsors patch the code; the report archives. The specification is never lifted into a durable, executable artifact that can guard future code.

As AI-assisted development raises contract throughput, this waste becomes costly. We need mechanisms to bootstrap formal properties from prior human audit work and re-apply them as regression tests. Recent benchmarks (Wu et al., ASE 2025) report that state-of-the-art access-control detection tools achieve low recall on Code4rena findings: Slither 8 true positives across 162 findings (with 389 false positives), Mythril 4 true positives with 26% timeout rate, and zero-shot LLMs 53-75% recall but with hallucination risk.

Bugzy poses a different question. Instead of zero-shot bug detection (“*what unknown bugs exist in this contract?*”) we ask: **given a known audit finding, can we produce a test that provably distinguishes the bug from its fix?** Precision becomes high by construction; recall is bounded by pipeline engineering and corpus coverage. The artifacts produced are reusable: a single validated spec can be re-run against any contract in the same vulnerability class.

Research questions

RQ1 (Elicitation): Can LLMs reliably extract structured formal specs from contest-grade audit markdown? **RQ2 (Validation):** Do generated Halmos tests fail on vulnerable commits and pass on fix commits without manual proof? **RQ3 (Scale):** Where does the pipeline break when applied uniformly to N=20 findings without per-finding tuning? **RQ4 (Reuse):** Can validated specs transfer to unrelated contracts in the same vulnerability class?

2. Method

2.1 Pipeline

Bugzy is a six-stage Python pipeline. **fetch** ingests Code4rena issue URLs, auto-resolves repo, commits, and affected file via the GitHub API. **extract** queries Claude Sonnet to convert the finding's markdown into a structured JSON spec: function signature, precondition, expected revert behavior, and a Halmos property sketch. **codegen** generates a Foundry test contract that imports the real target contract and exposes a single `check_*` function for symbolic exploration. **verify** clones the target repo at a specific commit, builds with Foundry, and runs Halmos 0.3.3 with a 60-second timeout per check. **validate** applies our verdict criterion. **generalize** adapts validated specs to unrelated repos.

2.2 Self-validation criterion

A mined spec is **VALIDATED** if and only if:

```
verify(spec, vulnerable_commit) = FAIL
verify(spec, fix_commit)       = PASS
```

Outcomes BUILD_FAILED, TIMEOUT, and ERROR are inconclusive: they do not falsify the spec, but they block a validation verdict. Additional diagnostic labels capture cases where the test passes on both commits (*spec too weak*) or fails on both (*spec too strict*).

2.3 Corpus and tooling

We curate 20 Code4rena access-control findings of High or Critical severity, including three pilot seeds (Llama, Basin, Blackhole). Extraction and codegen use Claude Sonnet via the Anthropic API. Verification uses Foundry for builds and Halmos 0.3.3 with the Z3 backend for symbolic execution. Curated pilots receive minimal human wiring (proxy address bored by the real factory rather than the raw implementation); the bulk evaluation runs unattended.

3. Results

3.1 Pipeline funnel

Table 1 reports per-stage attempt and success counts on the bulk benchmark (*wall-clock 938s, ~15.6 min*).

Stage	Att.	OK	Failure modes
Ingestion	20	20	17/20 with all 4 fields
Extraction	20	20	18 hi / 1 med / 1 low conf.
Codegen	18	18	2 curated pilots skipped
Verify (vuln)	16	4 FAIL	5 BUILD, 6 ERR, 1 TIMEOUT
Verify (fix)	20	1 PASS	8 BUILD, rest ERR/other
Validation	20	2	Basin + Blackhole

Table 1. Per-stage pipeline outcomes on N=20.

3.2 Validated findings

Finding	Property (abbreviated)	Vuln	Fix
Basin H-01	Non-owner upgradeTo must revert	FAIL	PASS
Blackhole H-10	Owner must call setTopNpools	FAIL	PASS

Table 2. Findings that satisfy the self-validation criterion.

3.3 Near-misses with diagnostic value

Three Ronin findings illustrate the pipeline's failure surface: the NPM-#67 case (missing-authorization-check-in-increaseL) returns FAIL on the vulnerable commit—catching the bug correctly—but ERROR on the fix leg due to build noise, leaving it *near-valid*. A second case (katanagovernance-isauthorized) builds cleanly on both commits but returns FAIL on each, signalling *spec-too-strict*: the symbolic harness over-constrains the precondition. The third (unauthorized-liquidity-manipulation) hits Halmos's 60-second timeout on the vulnerable commit.

These three together—build noise, over-constraint, and timeout—cover the three qualitatively distinct ways the pipeline can fail to produce a clean verdict. None reflect failure of the extraction or codegen stages.

3.4 Headline metrics

Metric	Value	Scope
Extraction success	100% (20/20)	Full corpus
Codegen success	100% (18/18)	Non-curated
Validation precision	100% (2/2)	Decidable subset
End-to-end validation	10% (2/20)	With curation
Unattended end-to-end	5% (1/20)	Bulk run
Halmos timeout rate	1/16	Verify-vuln attempts

Table 3. Use these numbers precisely. We do not claim Bugzy replaces human auditors or competes with zero-shot recall.

4. Comparison to prior work

ACToolBench (Wu et al., ASE 2025) evaluates Slither, Mythril, and zero-shot LLMs on a 162-finding Code4rena access-control corpus. Slither produces 8 true positives with 389 false positives; Mythril 4 true positives with 26% timeout; LLMs 53–75% recall with hallucination risk. These systems answer “*what unknown bugs exist?*” Bugzy answers “*given a known bug, can we mint a reusable test that distinguishes it from its fix?*” The precision/recall comparison is therefore not direct.

The qualitative gain is reusability. Slither emits one finding per (contract, run); LLM scans emit one bug list per contract. A Bugzy-mined spec is a Solidity property test that can be applied to any new contract claiming to implement the same interface,

without further LLM cost. The Basin spec, for example, encodes a general property over any UUPSUpgradeable._authorizeUpgrade override.

5. Limitations

Single bug class. Access control is the cleanest class to formalize because the invariant is local and ownership state is conventionally stored. Arithmetic, reentrancy, and oracle-manipulation bugs are richer and untested here.

Small N. 20 findings is too few for statistical claims about extraction failure modes.

Curation dependence. Two findings reached validation only with curated test wiring; one used a synthetic fix patch where no public sponsor commit is available. Future scaling requires automating fix-commit discovery from mitigation-review PRs.

Symbolic budget. A 60-second Halmos timeout suffices for most access-control tests but excludes contracts with deep deployment graphs (Ronin NFPM).

No statistical claim of latent-bug discovery. Our generalize sweep over a small UUPS pool returned zero potential findings. We report this honestly; it does not falsify the approach.

6. Future work

We propose four directions, deliverable on a four-month timeline:

1. Scale corpus to $N \geq 1000$ across all Code4rena audits 2022–2025. 2. Expand bug-class coverage to arithmetic, reentrancy, and oracle manipulation. 3. Publish an open benchmark—spec JSON, generated test, vulnerable/fix commits, and verdict—such that other tools (Slither, Mythril, future LLMs) can be scored against the same mined specs. 4. Integrate with Slither output to convert its false negatives into high-precision regression tests.

Cross-language equivalence (BabelBench-style proof export to Lean and Dafny) and proof-search targets (Deductive Vericoding) are natural extensions to adjacent SPS research directions.

7. Conclusion

Audit findings encode reusable security specifications. We have shown that with modern LLMs and a symbolic verifier, these specs can be extracted, compiled to executable tests, and self-validated against the public commit history that defines them. The dominant failure modes at small scale are infrastructural, not cognitive: build heterogeneity and missing-commit metadata block more findings than extraction does. A polished pipeline that addresses these bottlenecks is a plausible path from *one-shot audit deliverable* to *continuously re-validated security regression suite*.

References

[1] Wu, D., et al. “Have We Solved Access Control Vulnerability Detection in Smart Contracts?” *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025. daoyuan14.github.io/papers/ASE25_ACToolBench.pdf

[2] a16z crypto. *Halmos: A symbolic testing tool for EVM smart contracts*. github.com/a16z/halmos

[3] Code4rena. *Audit Findings Repository*. github.com/code-423n4

[4] Cyfrin. *Solodit: aggregated security audit findings database*. solodit.cyfrin.io

[5] Anthropic. *Claude Sonnet API*. docs.anthropic.com