Beyond the Bitmap: LLM-Guided Semantic Mutation Fuzzing

Mohammadreza Ashouri, PhD

ByteScan Security Research Lab audit@bytescan.net https://bytescan.net

ACM CODASPY 2026, Frankfurt

Abstract

Grey-box fuzzers such as AFL [1], AFL++ [2], and libFuzzer [3] instrument the target to collect edge coverage feedback and mutate inputs toward unexplored paths. They are fast but semantically blind because the coverage bitmap cannot distinguish a safe integer operation from one feeding into a memory allocation or an unchecked string copy. They also perform poorly against production binaries using encrypted or packed code. **White-box approaches** such as KLEE [4] suffer from exponential path explosion and are defeated by code obfuscation. **LLM-based fuzzing** (ChatAFL [5]) operates only at the message level with no visibility into the C implementation. We introduce **VULCAN** (Vulnerability-Uncovering LLM-guided Code Fuzzer), addressing all three limitations. Our early prototype confirmed **6 zero-day vulnerabilities** in widely deployed open-source software.

Background

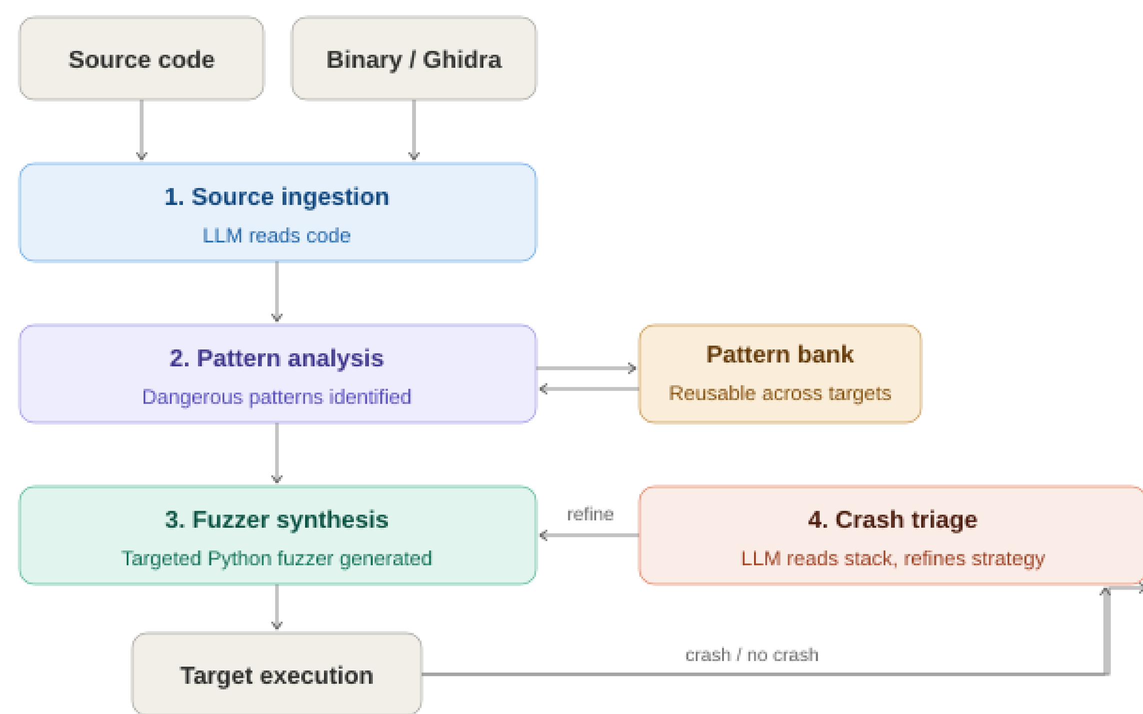
AFL++ / libFuzzer (Grey-box): Coverage-guided via edge bitmaps. These fuzzers are fast but semantically blind, as they cannot distinguish dangerous integer operations from safe ones and require instrumentation or QEMU overhead [2].

KLEE (White-box): SMT-based path analysis. Exponential path explosion on real codebases and are defeated by code obfuscation [4].

ChatAFL (LLM + State-machine): Protocol-state-aware but no visibility into code implementation (for example, C-level implementations). As a result, these approaches cannot identify uncaught exceptions or unguarded `malloc()` calls [5].

Architecture & Pipeline

VULCAN operates in four phases. The **Pattern Bank** accumulates findings across targets sharing common libraries, reducing setup time from hours to minutes.



Phase 1 : Source Ingestion: Software source or Ghidra decompiled Pseudocode is provided to the LLM, which supports multiple programming languages but primarily focuses on C code.

Phase 2 : Pattern Analysis: LLM identifies bugs such as unchecked `atoi()/strtol()`, unsafe `strcpy()`, uncaught exceptions, integer fields flowing into `malloc()`.

Phase 3 : Fuzzer Synthesis: LLM generates targeted Python fuzzer By seeding at dangerous field offsets, this approach avoids generating random bytes.

Phase 4 : Crash Triage: Crashes classified by signal. LLM reads stacks and iteratively refines the mutation strategy.

Comparative Analysis

Criterion	AFL++	libFuzzer	KLEE	ChatAFL	VULCAN
Setup & Requirements					
Source required	Optional	Required	Required	Required	**Optional**
Setup complexity	Medium	Medium	Very High	High	**Low**
Coverage & Path Analysis					
Code coverage	Edge bitmap	Edge map	Path/SMT	None	**Semantic**
Path explosion	✓	✓	× Severe	N/A	**✓ None**
Dead code pruning	×	×	Partial	×	**✓ LLM**
Intelligence & Semantics					
Protocol aware	×	×	×	State machine	**✓ Full**
Uncaught except.	Signal only	Signal only	×	×	**✓ Targeted**
Loop detection	Timeout	Timeout	×	×	**✓ LLM**
Binary-only	✓ QEMU	×	×	×	**✓ Decompile**
Effectiveness (8 targets tested)					
Time to crash	Hours–Days	Hours–Days	Days–∞	Hours	**Minutes–Hours**
Confirmed CVEs	0/8	0/8	N/A	N/A	**6 / 8 targets**

Key Differentiators

- **vs. AFL++:** Semantic pattern targeting compared to blind edge coverage. LLM identifies dangerous field offsets before generating a single input. This eliminates wasted iterations on safe code paths.
- **vs. KLEE:** No path explosion occurs. The LLM reasons semantically with no exponential SMT blowup on 25K+ line codebases. Code obfuscation does not break the analysis.
- **vs. ChatAFL:** Analyzes actual C source code, not only protocol specifications. ChatAFL generates only protocol-valid messages and cannot determine that a password string flows into `strcpy()` without a length check, or that a username field feeds into `malloc()`. VULCAN targets these dangerous operations directly. For this reason, VULCAN identified the `oggenc` vulnerability via a malformed WAV header that violates the format specification, an input that a state-machine approach could never produce.

Evaluation — Confirmed Findings

Applied VULCAN to 8 open-source targets. Six confirmed crashes with CVE submissions (**75% success rate**).

Target	Vulnerability	Signal	CWE	Status
oggenc 1.4.3	NULL ptr via WAVE_FORMAT_EXTENSIBLE	SIGSEGV	476	**CVE**
OpenSCAD 2021	Uncaught `char const*` from ClipperLib	SIGABRT	248	**CVE**
OpenSCAD 2021	Uncaught `boost::filesystem_error`	SIGABRT	248	**CVE**
Beanstalkd 1.13	CRLF injection in TCP parser	Protocol	93	**CVE**
Privoxy 4.1.0	Null byte DoS in HTTP headers	DoS	476	**CVE**
Icecast 2.4.4	Integer overflow in stream params	Overflow	190	**CVE**
ngircd 27	No crash — recently audited	—	—	*None*
FreeRADIUS 3.2.8	No crash — hardened codebase	—	—	*None*

Observation: Targets with no findings had both undergone recent security rewrites, which validates the heuristic that older, unaudited C codebases yield results significantly faster than recently patched codebases.

Conclusion & Future Work

VULCAN is our prototype LLM-based Fuzzer, and so far it has achieved a **75% success rate** (6 out of 8 targets) with minimal manual effort. The Pattern Bank reduces setup time from hours to minutes across targets sharing common libraries. VULCAN can be considered a smart fuzzer that understands the code and generates specific inputs based on code semantics and unexpected trigger conditions to cause unhandled errors.

Future work: We are working on the integration of LLM-assisted deobfuscation [6] to handle packed binaries, and also using VULCAN in a larger benchmark of real-world OS services. We will also explore hybrid pipelines that combine runtime coverage feedback with LLM-guided mutation strategy generation in order to find more bugs and more critical bugs.

References

[1] M. Zalewski. *American Fuzzy Lop (AFL)*. Google Security, 2014.

[2] A. Fioraldi et al. *AFL++: Combining Incremental Steps*. USENIX WOOT, 2020.

[3] K. Serebryany. *libFuzzer: Coverage-Guided Fuzz Testing*. LLVM, 2015.

[4] C. Cadar et al. *KLEE: High-Coverage Tests for Complex Programs*. OSDI, 2008.

[5] Z. Yao et al. *ChatAFL: Enriching Fuzzing with LLMs*. NDSS, 2024.

[6] T. Blazytko et al. *SYNTIA: Semantics of Obfuscated Code*. USENIX Security, 2017.