DEMO: SemCRS: Streamlining Vulnerability Discovery and Patching with LLMs

Yu Nong
University at Buffalo
Buffalo, New York, USA
yunong@buffalo.edu

Haipeng Cai*
University at Buffalo
Buffalo, New York, USA
haipengc@buffalo.edu

Abstract

Finding software vulnerabilities and fixing them without human intervention is a long-standing goal of security research. We present SemCRS, an autonomous *Sem*antics-guided *C*yber *R*easoning *S*ystem that, given a program, its fuzzing harness, and a set of target sanitizer classes, discovers a vulnerable function, generates a proof-of-vulnerability (PoV) input, validates it against a runtime sanitizer, and generates a patch that blocks the validated PoV while the program's available functional tests continue to pass. SemCRS pairs large language model (LLM) reasoning with lightweight program analysis and *closes the loop*: every candidate PoV is executed under the real sanitizer before anything is reported, turning imprecise LLM judgments into verifiable results. The pipeline runs from one command, yet every stage remains inspectable. We demonstrate SemCRS on a compact program and report four runs on programs of up to 6,172 functions. Our artifact including source code and a video walkthrough is at https://github.com/SRestLabUB/SemCRS.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

vulnerability discovery, automated program repair, LLMs

ACM Reference Format:
Yu Nong and Haipeng Cai. 2026. DEMO: SemCRS: Streamlining Vulnerability Discovery and Patching with LLMs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3830454.3846402

1 Introduction

Continuous fuzzing has made *finding* bugs almost routine: infrastructure such as OSS-Fuzz [2] has surfaced tens of thousands of bugs in open-source software. Yet each crash still lands on a human to reproduce, understand, and fix, and that human effort—not the finding—is now the bottleneck, leaving vulnerabilities unpatched for weeks. A system that could take a program from a raw harness all the way to a *validated, patched* fix, with no human in the loop, would change the economics of defense. Concretely, given a target program, a harness that exposes an entry point [2, 7], and target sanitizer classes [6], such a system must (i) generate an input that triggers a sanitizer, i.e., a *proof-of-vulnerability* (PoV); (ii) identify the vulnerability, reporting the analyzed target commit and the triggering sanitizer; and (iii) generate a patch that removes the flaw without breaking functionality.

This is hard for reasons that defeat naive uses of LLMs. The vulnerable function is not known in advance and must be located within a potentially large call graph. PoVs are often binary or highly structured inputs an LLM cannot reliably emit as raw bytes; it must instead reason about the trigger condition and emit a *program* that constructs the input. Also, LLM judgments are imprecise: a model asked whether code is exploitable often over-approximates, and a fix it proposes may be wrong or may silently break the program. Therefore, an unverified model verdict is unsafe to act on.

Existing approaches each cover only part of this task. Automatic exploit generation derives triggering inputs by symbolic execution and constraint solving [1], but is brittle on complex semantics and stops at the exploit, offering no fix. Coverage-guided fuzzing [2, 7] with runtime sanitizers [6] surfaces crashes reliably, yet leaves a human to localize and repair the flaw. Recent work applies LLMs to vulnerability and program repair [3, 5], but assumes the vulnerability is already known and judges a fix in isolation. Autonomous discover-to-patch systems exist, from binary exploit-and-patch pipelines to recent LLM-based cyber reasoning systems, but they largely trust the model's or solver's verdict; the open problem is to let discovery over-approximate while reporting and patching only what execution confirms.

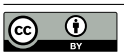
We present SemCRS, which addresses these issues by pairing LLM reasoning with program analysis and, most importantly, by *grounding every reported finding and generated patch in execution.* Detection is allowed to be noisy, as precision comes from a validation loop that rebuilds the program and runs each candidate PoV under the real sanitizer, so only sanitizer-confirmed findings are reported, and each patch is re-checked against the same PoV and the available functional tests.

Our contributions are: (1) a streamlined discover–prove–patch pipeline, run from a single command as six inspectable stages, in which a validated finding flows directly into patch generation and re-validation against the same PoV; (2) a validation-before-report design that converts imprecise LLM outputs into verifiable results; and (3) a public tool that exposes the full trace of every run.

2 Approach

SemCRS pairs LLM reasoning with lightweight program analysis and a sanitizer-based execution oracle. Its central technique is to *decouple recall from precision*: the LLM is used where it is strong—reading code and synthesizing inputs and patches—and is allowed to over-approximate, while precision is recovered separately by

*Corresponding author.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846402

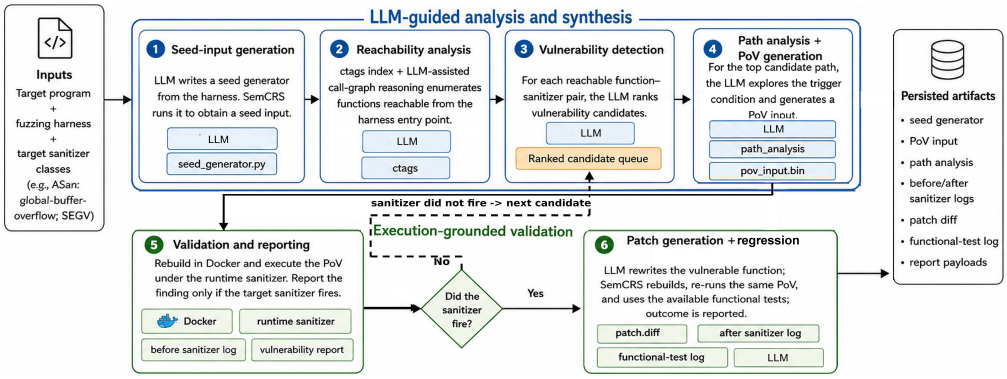


Figure 1: The SemCRS (*Sem*antics-guided *C*yber *R*easoning *S*ystem) pipeline. Stages 1–4 (*LLM-guided analysis and synthesis*) turn a program, its fuzzing harness, and target sanitizer classes into a candidate PoV; Stages 5–6 (*execution-grounded validation*) rebuild the target and run the PoV, report a finding only when the target sanitizer fires (otherwise advancing to the next candidate), then generate and regression-test a patch. SemCRS persists inspectable artifacts across the pipeline.

executing each candidate under the real sanitizer. Two further ideas make this practical. Reachability is over-approximated cheaply by pairing a single concrete seed execution with LLM-inferred call edges, avoiding heavyweight whole-program static analysis. And because a PoV is often a binary or structured blob, the LLM emits a *generator program* rather than the input itself; the same PoV, executed under the sanitizer, provides validation evidence for both discovery and repair. SemCRS treats the LLM as a configurable backend—our prototype uses gpt-5-2025-08-07 [4] (default sampling, one trial per run)—and runs as six stages (Figure 1).

Design rationale. We arrived at an execution-grounded design after a more conventional alternative failed. We first coupled the LLM with a satisfiability modulo theories (SMT) solver, intending the solver to discharge path constraints while the LLM supplied the semantics of library calls. In practice the LLM often produced incorrect or even syntactically invalid constraints, and the coupling introduced failure modes rather than removing them. Prompting the LLM instead to analyze the call path in natural language and emit a *program* that constructs the input, then validating that input by execution, proved far more reliable.

Stage 1: Seed-input generation. From the harness, the LLM writes a small program that generates a seed input, which SemCRS runs to obtain its starting test data. This removes dependence on an external corpus and gives later stages a concrete execution to observe.

Stage 2: Reachability analysis. SemCRS indexes every function and method definition using Universal Ctags [8]. Starting from the functions exercised by the seed run, it iteratively asks the LLM which user-defined functions each function calls and resolves every returned name against the index, expanding the set to a fixpoint. Pairing a concrete seed run with LLM-inferred call edges can recover functions reachable only under specific conditions—which purely dynamic coverage would miss—while still bounding the search to code the harness can reach. These high-volume call-graph queries can use a lighter, cheaper model, while SemCRS reserves the strongest model for the later analysis, generation, and patching.

Stage 3: Vulnerability detection. For each reachable function and each candidate sanitizer, the LLM judges whether the function could trigger that sanitizer, producing a ranked queue of candidates rather than a final verdict. The step is intentionally recall-oriented; its imprecision is corrected later by execution.

Stage 4: PoV generation. For the top candidate, SemCRS gives the LLM the relevant code slice and the harness, asks it to explain how the sanitizer can be triggered (recorded as the *path analysis*), and then to emit a program that writes the (binary) PoV input. Constructing PoVs with a program lets SemCRS handle inputs an LLM could not produce directly.

Stage 5: Validation and reporting. SemCRS rebuilds the program inside a Docker container and runs the PoV. *Only if the target sanitizer fires* is the finding reported with the target commit and the sanitizer that fired; otherwise SemCRS advances to the next ranked candidate, up to a given number of attempts (we used the top-ranked candidate and a single generator synthesis by default). This closed loop reduces the impact of detector imprecision: unsupported candidates from Stage 3 do not survive execution, so every reported finding comes with a concrete input that triggers the sanitizer.

Stage 6: Patch generation and regression. The LLM is prompted to rewrite the vulnerable function while preserving compilability and behavior. SemCRS rebuilds the program, re-runs the *same* PoV to confirm the sanitizer no longer fires, and runs the program's available functional tests to check that they continue to pass. The patch is reported as a unified diff.

Notably, SemCRS persists the artifacts of a run—the seed generator, the PoV input, the path analysis, the before/after sanitizer logs, the patch, functional-test evidence, and the exact report payloads—so every step is inspectable and the entire process is auditable.

3 Demonstration Scenario

Figure 2 traces SemCRS on a compact C program (the "Mock CP") that reads records into a fixed-size global array. Given the program, a fuzzing harness, and two target sanitizer classes, the system synthesizes a seed input, observes the functions the seed exercises, and expands that to an approximate reachable set. Detection over-approximates and flags two functions: `func_a` for a global buffer overflow and `func_b` for a segmentation fault (SEGV). For the top candidate, SemCRS reasons about the trigger—the input carries more records than the array has rows, so the loop writes past the last row—and emits a generator that produces `pov_input.bin`.

Validation resolves the imprecision. SemCRS rebuilds the program in a container and runs the PoV; AddressSanitizer reports a global-buffer-overflow in `func_a`, confirming the finding, which

A live SemCRS run on the Mock CP (one command, GPT-5)

```
$ python demo.py
SemCRS | LLM: LIVE (gpt-5) | target: Mock CP
sanitizers: global-buffer-overflow, SEGV
(3) Vulnerability detection
  func_a global-buffer-overflow : FLAGGED
  func_b SEGV : FLAGGED
(4) PoV generation → pov_input.bin
(5) Validation (rebuild + run PoV)
  ERROR: AddressSanitizer:
    global-buffer-overflow in func_a
  sanitizer fired: True
(6) Patch + regression
  PoV re-run after patch: fired=False ( blocked )
  functional tests: PASS
RESULT: PoV validated; patch blocks it; 15 LLM calls
```

Figure 2: A live SemCRS run. Detection over-approximates (two functions flagged), but only the execution-validated finding in `func_a` is reported; the patch blocks the same PoV and the available functional tests still pass.

is only then formatted for reporting with the target commit; a candidate that failed to reproduce would simply be skipped. The LLM then rewrites `func_a`, and SemCRS rebuilds and re-runs the *same* PoV: the sanitizer stays silent and the program's available functional tests pass, so the patch blocks the validated PoV. The run issues 15 model queries (about 8,900 tokens, $0.04 in API cost), finishes in about five and a half minutes, and records key artifacts—the seed, PoV, before/after sanitizer traces, and patch—so the run can be inspected and independently re-checked. The example is small enough to read end to end; the same pipeline drives the larger programs of Table 1.

4 Using SemCRS

SemCRS is set up the way a fuzzing campaign is, not the way a static analyzer is configured. A user supplies three things: the program under test, a fuzzing harness that defines the entry point, and the sanitizer classes of interest (for example, AddressSanitizer's memory-safety categories for C/C++, or a Jazzer bug detector such as OS command injection for Java). No seed corpus, no annotations, and no hint of where the bug lies are required. In return, for each sanitizer-confirmed finding SemCRS yields a sanitizer-validated PoV, the target commit and sanitizer, and a patch checked against the PoV and available tests, each paired with the evidence.

Because every stage is inspectable, SemCRS is auditable rather than a black box: an analyst can read the path analysis, the exact PoV, the sanitizer trace that confirms it, and the patch diff with the tests it passes—a finding is trusted because a concrete execution produced it, not because a model asserted it.

Three knobs matter in practice: the LLM backend, the set of target sanitizer classes, and the search budget for PoV generation. This makes SemCRS a fit for triaging fuzzing harnesses at scale, for gating regressions in continuous integration, and as a testbed for studying where LLM reasoning suffices and where execution grounding is indispensable.

5 Capabilities and Results

We report four SemCRS runs that complete the full discover–prove–patch loop, spanning C and Java and different vulnerability classes

Table 1: SemCRS on four benchmark programs. PoV = validated by the target sanitizer; Patch (✓) = blocks the validated PoV *and* the program's available functional tests pass. The Mock CP row is reproduced by the released tool; the others are from earlier development runs

Program	Lang	Vulnerability (sanitizer)	PoV	Patch
Mock CP	C	global-buffer-overflow	✓	✓
Jenkins	Java	OS command injection	✓	✓
SQLite3	C	heap-buffer-overflow	✓	✓
nginx	C	global-buffer-overflow	✓	✓

(Table 1). The targets range up to 6,172 functions (nginx), where the vulnerable code lies sixteen calls deep from the harness; reachability approximation narrows such a program to a candidate subset, so detection queries the LLM only on that subset. These results follow from the design rather than from a strong detector: Stage 3 over-approximates, and it is the execution-grounded validation loop that backs *reported* finding with a concrete input that triggers the bugs.

6 Conclusion and Future Work

SemCRS shows that combining LLM reasoning with execution-grounded validation yields an autonomous discover–prove–patch pipeline that is both practical and transparent: complex behavior behind a single command, with every step open to inspection.

SemCRS also has several limitations. A patch is validated only against the discovered PoV and the tests a project already ships, so passing tests show that *tested* behavior survives—not that the root cause is fixed or that no new weakness was introduced. Detection and reachability are recall-oriented approximations whose LLM-inferred call edges may be incomplete or spurious, and each run reports a single vulnerability. Table 1 reports successful runs rather than an exhaustive evaluation, and cost and latency were instrumented only for the demonstration.

Future work includes scaling reachability to very large codebases, handling multiple co-located vulnerabilities in one run, and strengthening patch validation beyond the discovered PoV. The tool, a video walkthrough, and per-run traces are available at https://github.com/SRestLabUB/SemCRS.

Acknowledgment

We thank reviewers for their constructive comments. This work was supported by NSF under Grant 2505223 and Office of Naval Research (ONR) under Grant N000142512252.

References

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. In *NDSS*. 1–18.
[2] Google. 2026. OSS-Fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz.
[3] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {Real-World} software vulnerability patching. In *USENIX Security*. 4481–4500.
[4] OpenAI. 2025. GPT-5. https://openai.com.
[5] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *(S&P)*. 2339–2356.
[6] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *ATC*. 309–318.
[7] The LLVM Project. 2026. libFuzzer: A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[8] Universal Ctags Project. 2026. Universal Ctags. https://ctags.io.