# Orion: Fuzzing Workflow Automation

Max Bazalii*
NVIDIA
Santa Clara, USA
mbazalii@nvidia.com

Marius Fleischer*
NVIDIA
Santa Clara, USA
mfleischer@nvidia.com



## Abstract

Fuzz testing is one of the most effective techniques for finding software vulnerabilities. While modern fuzzers can generate inputs and monitor executions automatically, the overall workflow, from analyzing a codebase, to configuring harnesses, to triaging results, still requires substantial manual effort. Prior attempts focused on single stages such as harness synthesis or input minimization, leaving researchers to manually connect the pieces into a complete fuzzing campaign.

We introduce Orion, a framework that automates the the manual bottlenecks of fuzzing by integrating LLM reasoning with traditional tools, allowing campaigns to scale to settings where human effort alone was impractical. Orion uses LLMs for code reasoning and semantic guidance, while relying on deterministic tools for verification, iterative refinement, and tasks that require precision. Across our benchmark suite, Orion reduces human effort by 46–204× depending on the workflow stage, and we demonstrate its effectiveness through the discovery of two previously unknown vulnerabilities in the widely used open-source `clib` library.

## CCS Concepts

• **Security and privacy** → *Software security engineering*; *Operating systems security*; • **Software and its engineering** → *Software testing and debugging*.

## Keywords

Fuzzing, Workflow automation, Large language models, Security testing, Vulnerability discovery

## 1 Introduction

Fuzzing is one of the most effective techniques for vulnerability discovery. By executing programs with large numbers of random, malformed, or unexpected inputs, fuzzing has exposed tens of thousands of real-world bugs. Google's OSS-Fuzz [17] service alone has reported more than 50,000 vulnerabilities across over 1,000 open-source projects, while syzkaller [18] has uncovered more than 8,000 kernel bugs. These results highlight fuzzing's importance both in research and in production environments.

However, scaling fuzzing to large production codebases remains a major challenge. Modern fuzzers automate input generation and coverage feedback once a campaign is running, but the broader workflow surrounding a fuzzing campaign is still heavily manual.

Analysts must first examine the codebase to identify promising fuzz targets. They then implement *fuzz harnesses*, small programs that accept fuzzer inputs and invoke target functions with the correct arguments. Additionally, harness construction is often coupled with *seed generation*, the preparation of example inputs that illustrate valid input structures and allow the fuzzer to exercise deeper execution paths.

Post-execution, the analyst must analyze results: triaging crashing inputs, interpreting stack traces, and identifying root causes. Finally, the analyst must write patches for all discovered bugs. These crash analysis and patch generation tasks are particularly time-consuming, requiring a deep understanding of the target program and codebase.

This dependency on manual effort creates a major bottleneck for scaling fuzzing to large production codebases, representing the primary focus of this work.

Prior research has explored automation for individual workflow components, but each approach comes with limitations. Traditional automated patching systems like GenProg [26] and SemFix [38] face fundamental scaling concerns with large codebases. Existing fuzzing automation tools impose restrictive dependencies: Utopia [22] requires unit test availability, Winnie [25] depends on existing exemplar programs for harness generation, and Skyfire [48] relies on pre-existing seed corpora. Critically, these approaches focus exclusively on single workflow steps rather than end-to-end processes. Substantial manual effort is still required to connect these pieces into an end-to-end fuzzing campaign.

Recent advances in large language models (LLMs) suggest a new direction. LLMs demonstrate strong capabilities in code understanding, reasoning, and generation. When combined with agentic frameworks, they can perform long-horizon planning, decompose multi-step tasks, and iteratively refine their own outputs. This makes them natural candidates for automating complex engineering workflows such as fuzzing. Several works have begun exploring this direction—for example, harness generation [28, 31, 56], seed creation [29, 44], or crash analysis [64]—but all focus on isolated stages. No existing system provides an integrated, end-to-end solution.

We present Orion, a framework that automates the fuzzing workflow from start to finish by combining LLM agents with deterministic analysis tools. The core design principle is to leverage each component for what it does best: LLMs provide semantic reasoning about codebases and assist in creative tasks such as harness design or seed inference, while traditional tools supply reliable checks through compilation, execution, and static analysis. Outputs from LLM agents are validated wherever possible, and errors are fed back into iterative refinement loops. This mitigates the unreliability of probabilistic models while exploiting their strengths in code reasoning and generation.

Orion mirrors the workflow of human fuzzing experts. It begins by transforming the target codebase into a structured knowledge base, enabling precise retrieval of relevant context for subsequent agents. Specialized subsystems then carry out each stage

*Both authors contributed equally to this research.

of the workflow: target identification, seed generation, harness creation, crash triage, and patch suggestion. Agents are equipped with the ability to invoke external tools, run code, and collect feedback, and employ prompting strategies such as self-consistency and self-reflection to improve reliability. Through this design, Orion replaces the manual bottlenecks of fuzzing with automated, tool-supported agents, scaling fuzzing to settings where human effort previously made it impractical.

We evaluate Orion using a custom benchmark across two open-source and one proprietary library. Across these targets, Orion reduces required human effort by 46–204× depending on the workflow stage. During evaluation, it also discovered two previously unknown vulnerabilities in the widely used `clib` library [9], both responsibly disclosed to developers. These results demonstrate that combining LLM agents with deterministic verification can make end-to-end fuzzing automation practical.

In summary, we make the following contributions:

(1) We introduce Orion, the first end-to-end fuzzing workflow automation framework that integrates LLM agents with traditional tools.
(2) We design techniques to improve LLM reliability in this setting, including tool-assisted verification, feedback-driven refinement, and prompting strategies such as self-consistency and self-reflection.
(3) We implement and evaluate Orion on real-world codebases, showing substantial reductions in required human effort and the discovery of two zero-day vulnerabilities.

## 2 Background

### 2.1 Fuzzing 101

Fuzzing is a widely used dynamic analysis technique for vulnerability discovery. It executes target programs with automatically generated inputs while monitoring for abnormal behavior or crashes. When a crash occurs, the fuzzer records the crashing input and stack trace for later analysis.

Modern fuzzers employ coverage guidance to increase exploration efficiency [14]. Inputs that trigger new execution paths are retained in the fuzzer's corpus and subsequently mutated, allowing the fuzzer to incrementally learn the structure of valid inputs rather than starting from scratch. To detect bugs beyond crashes, fuzzers rely on runtime instrumentation, or *sanitizers*, that identify invalid program states (e.g., out-of-bounds accesses, use-after-free) and deliberately terminate execution with diagnostic messages. Sanitizers extend fuzzing's reach to memory safety, thread safety, undefined behavior, and memory leak detection.

Effective fuzzing requires careful interface selection. Fuzzing every interface in a large codebase is infeasible, and many interfaces are irrelevant from a security perspective. Analysts typically focus on attacker-accessible interfaces or those with high security relevance (e.g., authentication logic), guided by existing threat models. Since fuzzing identifies issues through crashes, the chosen targets must include code where invalid states can manifest as observable failures (e.g., invalid pointer dereferences).

Harnesses are required to connect fuzzers to target programs. A fuzz harness receives byte buffer inputs from the fuzzer, transforms them into the program's expected format, and invokes the target interface. Harness quality has a direct impact on effectiveness: poor harnesses can cause spurious crashes, misuse interfaces, or exhaust resources, thereby obscuring real vulnerabilities. High-quality harnesses perform proper initialization and dependency setup, execute the target interface correctly, and include teardown logic to restore program state. This ensures reproducibility across iterations and minimizes runtime overhead, enabling fuzzers to operate at maximum speed.

Seed corpora further improve fuzzing effectiveness. While fuzzers can start with an empty corpus, providing example inputs accelerates progress by illustrating valid input structures [43]. High-quality seed corpora contain diverse inputs that exercise a wide range of program behaviors and edge cases, enabling the fuzzer to focus on deeper exploration rather than format inference.

Fuzzers nevertheless struggle with input constraints that are difficult to satisfy through random mutation, such as checksums or constant value comparisons. Although fuzzers favor common boundary values (e.g., -1 for integers), complex input validation logic often requires crafted seeds or harness logic that encodes expected values. Such guidance is critical for enabling the fuzzer to reach code behind hard-to-satisfy conditions.

At the end of a campaign, fuzzers report coverage statistics and discovered crashes, each accompanied by the triggering input and stack trace. However, the crash site may differ from the actual defect location, since corrupted state can propagate before triggering a visible failure. A single defect can produce multiple crashes (e.g., a missing NULL assignment after freeing memory may appear as both a use-after-free and a double-free). Consequently, crash triaging and root cause analysis remain essential steps before fixing bugs.

### 2.2 Human Fuzzing Workflow

Human analysts typically follow a structured workflow when fuzzing target programs.

The process begins with **target selection**. Analysts leverage threat models and source code analysis to identify interfaces that balance high bug likelihood with attacker accessibility. These interfaces are prioritized for fuzzing campaigns.

Next, analysts construct **fuzz harnesses** for selected targets. This requires detailed knowledge of input formats and interface dependencies. Harnesses must provide appropriate initialization, interface invocation, and cleanup logic. Lightweight and reliable harnesses are critical for campaign success.

Before execution, analysts design **seed corpora** considering target interface behavior and expected harness input formats. The goal is to create diverse corpora that demonstrate expected input formats and edge cases.

Analysts then execute **fuzzing campaigns**, typically the only phase that runs autonomously once configured. Fuzzers operate until they uncover crashing inputs, which are logged with execution details.

When crashes occur, analysts conduct **bug triage and patching**. Triaging involves identifying root causes, correlating multiple manifestations of the same bug, and assessing severity. This requires interpreting stack traces, analyzing source code, and replaying crashing inputs with debugging tools. Once the underlying issue is

identified, analysts develop and test patches, often requiring several iterations before resolution.

This workflow illustrates that only the execution phase is substantially automated today; the surrounding tasks still depend heavily on human expertise.

## 2.3 LLMs and LLM Agents

Large language models (LLMs) are transformer-based neural networks trained on massive text corpora to predict token sequences, thereby capturing statistical patterns of natural language and source code [15, 47]. After pretraining, instruction tuning and alignment (e.g., RLHF) enable them to follow prompts for tasks such as question answering, code generation, and multi-step reasoning. Benchmarks such as SWE-bench [24] and HumanEval [7] demonstrate LLMs' ability to understand program semantics and generate functional code. These capabilities have enabled widespread adoption of LLMs in development workflows (e.g., IDE assistants [4, 52]) and in agent-based frameworks for software engineering [3, 10, 40]. Applications to security-specific tasks are discussed further in Section 6.

However, LLMs exhibit significant limitations requiring careful system design. Knowledge retrieval errors frequently occur, particularly for fine details crucial in code understanding, often leading to hallucinations where LLMs generate incorrect information [15]. Context window constraints restrict the amount of text they can process, and large contexts often suffer from the *needle-in-the-haystack* problem, where critical details are overlooked [37]. LLMs also struggle with counting and arithmetic, and their probabilistic nature produces inconsistent outputs even for identical inputs. These weaknesses complicate reliable use in security workflows, despite mitigation techniques such as self-consistency [49].

LLM agents extend base models with tool usage, external memory, and planning capabilities [51]. In this paradigm, the LLM serves as the agent's reasoning core while tools enable concrete actions beyond text generation (e.g., invoking command-line utilities). Such approaches have been applied to software development [3, 10, 40] and security analysis [54]. However, agents amplify underlying LLM weaknesses: tool misuse can corrupt environments (e.g., accidental file deletion), and iterative reasoning can propagate early mistakes across multiple steps. These error cascades make it difficult to rely on LLM agents for tasks that demand precision, such as crash triage or automated patching in security testing.

# 3 Orion Design

We present Orion, a framework for automating the fuzzing workflow by combining LLM-based agents with deterministic tools. The design mirrors the workflow of a human analyst (Figure 1) and enables fuzzing campaigns to scale with minimal manual effort. Orion supports pre-fuzzing tasks including target identification (Section 3.3), seed generation (Section 3.4), and harness construction (Section 3.5), as well as post-fuzzing tasks such as crash triage (Section 3.7.1) and patch generation (Section 3.7.2). Each stage is handled by a dedicated agent, which decomposes the workflow into simpler subtasks and allows independent optimization of components. A codebase indexer (Section 3.2) underpins all stages by enabling targeted information retrieval.

## 3.1 Overview of the Orion Workflow

Orion takes as input the target project's source code and build configuration (e.g., compilation commands, compiler flags, agent parameters). It begins by constructing a codebase index (Section 3.2) to support targeted queries. The target identification stage (Section 3.3) then selects candidate interfaces for fuzzing by ranking non-static functions likely to contain fuzzer-triggerable bugs.

For each selected function, Orion generates seed inputs (Section 3.4). Seeds are crafted to address input constructs that fuzzers typically struggle with (e.g., checksums) and to provide structural specifications used during harness generation (Section 3.5).

Harness generation (Section 3.5) involves two agents: a dependency analysis agent (Section 3.5.1) and a harness synthesis agent (Section 3.5.2). The dependency agent identifies required setup and teardown procedures, as well as header file dependencies, while the harness agent constructs compilable fuzz drivers compatible with generated seeds. Compiler feedback ensures harness validity and enables iterative refinement when errors are detected.

The resulting harnesses and seeds are passed to the fuzzing execution infrastructure (Section 3.6), which runs the fuzzer, monitors coverage, and records crashes. Coverage feedback can trigger harness or seed adjustments, while crashes are analyzed by downstream agents.

When crashes occur, the triage agent (Section 3.7.1) filters out harness-related issues, determines root causes, and produces minimal reproducers. These reproducers support bug understanding and serve as validation inputs for the patching agent (Section 3.7.2), which proposes candidate patches by analyzing the faulty code. Proposed patches are validated through compilation and replay against reproducers before producing a final patch diff for application.

**Challenges.** The design of Orion highlights several challenges that must be addressed to achieve reliable automation.

First, as discussed in Section 2.3, LLMs possess broad world knowledge but cannot retrieve it reliably at fine granularity. This is problematic when reasoning about source code, where small details are often critical. Moreover, proprietary codebases and documentation are not part of LLM pretraining data, so Orion cannot rely on pretrained knowledge alone and must supply relevant context. The key challenge is determining which information to provide: excessive context can confuse the model, while insufficient context can lead to fabricated outputs. Orion must therefore balance between overwhelming the model and omitting important details.

Second, LLM agents operate with broad action spaces, which increases flexibility but also risk. Incorrect reasoning traces may yield invalid outputs or corrupt the runtime environment [39]. While agents offer benefits through semantic reasoning beyond what deterministic tools can capture, it is critical to assign them tasks that match their strengths and to enforce oversight mechanisms that limit harmful behavior.

Finally, LLMs are inherently stochastic (Section 2.3), producing different outputs even for identical inputs. Without strong oracles to verify correctness, these random variations can propagate through the workflow, invalidating subsequent analyses and undermining reproducibility. This motivates the design mitigations described below.

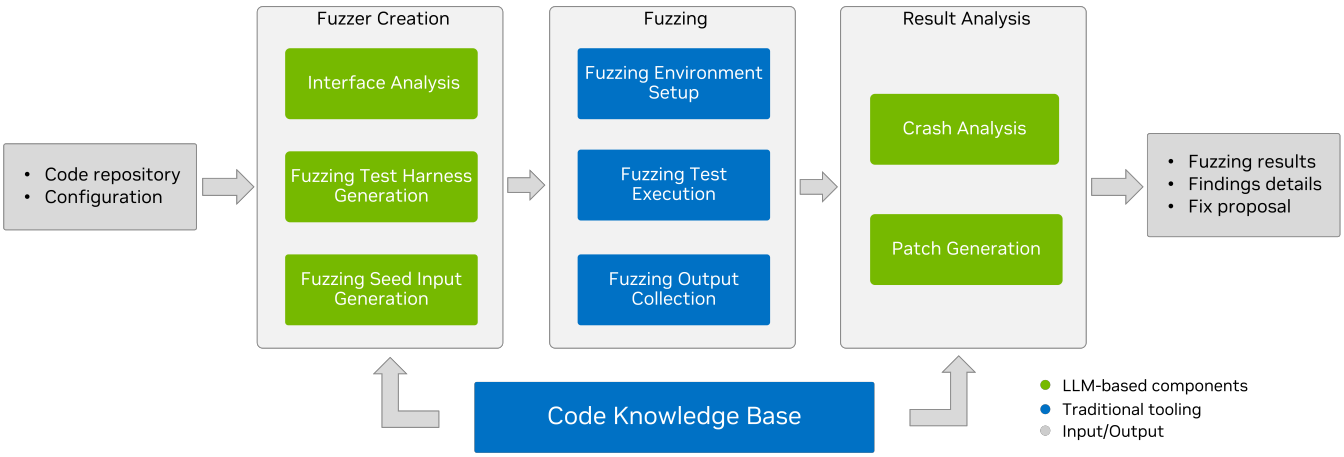


**Figure 1: Overview of the Orion fuzzing workflow automation pipeline.**

**Techniques.** We now describe how Orion addresses the above challenges.

To overcome context retrieval limitations, Orion employs a codebase indexer built on compiler tooling. The indexer supports fine-grained queries for specific program elements (e.g., function or type definitions, header declarations), enabling the system to supply the LLM with precisely the context required for each task. This reduces both information overload and the risk of missing crucial details.

To balance LLM strengths with reliability, Orion integrates them with deterministic tools. Tasks that can be handled precisely by existing tools (e.g., code metric calculation) are delegated entirely to those tools. For tasks requiring semantic reasoning, LLMs operate in conjunction with traditional tools that both provide structured context and validate outputs. Tool feedback loops prevent erroneous results from propagating further in the workflow.

Finally, Orion improves reproducibility through consensus-based prompting techniques. Self-consistency [49] issues multiple queries with identical input and aggregates results, selecting the majority outcome to mitigate stochastic variation. Self-reflection [45] allows the LLM to critique its own reasoning process and revise outputs based on detected errors. These mechanisms, applied across all workflow stages, enhance reliability and reduce the impact of individual model errors.

## 3.2 Code Base Indexing

The codebase indexer is the first component executed by Orion and underpins all subsequent workflow stages by addressing two key challenges in context retrieval for LLMs.

First, LLMs have limited context windows, which are too small to accommodate most production codebases (Section 2.3). A common mitigation is to split input into smaller chunks and select those most relevant to the current query. For code analysis, however, chunking must respect semantic boundaries to preserve meaning. Orion therefore uses function boundaries as natural chunking units, since functions are self-contained logical entities and common fuzzing targets are function entry points. This design enables precise retrieval and supports reasoning at the granularity of individual interfaces.

Second, because proprietary codebases and project-specific libraries are not part of LLM pretraining data (Section 3.1), Orion must supply all necessary context explicitly. This includes function signatures, definitions, type declarations, header file contents, and caller–callee relationships. Accurate retrieval of this information is critical, as irrelevant or missing context directly reduces the reliability of LLM reasoning.

The indexer addresses these requirements by parsing source code with the compiler toolchain. It extracts function metadata (declarations, definitions, signatures, scope), constructs a global call graph, and builds a type index associated with header files. These artifacts enable targeted queries such as retrieving functions defined in specific headers (target identification, Section 3.3), resolving types by header (harness generation, Section 3.5.2), or recursively enumerating callees (dependency analysis, Section 3.5.1). By grounding LLM prompts in compiler-derived information, the indexer ensures that Orion can provide the model with precise and relevant context without exceeding its limits.

## 3.3 Target Identification

As described in Section 2.2, human analysts begin fuzzing by selecting interfaces that balance attacker relevance and expected bug density. This typically requires external threat models, which limits applicability to well-studied systems. To support broader deployment, Orion instead focuses on identifying interfaces that maximize fuzzer effectiveness, ensuring practical results even when threat models are unavailable.

Because fuzzers primarily expose sanitizer-detectable bugs, the goal is to prioritize interfaces with high likelihood of containing such vulnerabilities. Orion estimates this likelihood using a set of independent metrics (Section 3.3.1) that evaluate functions individually. Metric outputs are then combined into a composite ranking from which Orion selects the top candidates within the available fuzzing budget.

The metric set captures diverse code properties correlated with vulnerability risk, though it is not exhaustive and no single metric can definitively identify vulnerable interfaces. Orion's design allows new metrics to be introduced or existing subsets to be selected, enabling customization for different codebases.

Several metrics rely on LLM analysis, raising concerns about reliability. To address this, Orion applies self-consistency and self-reflection (Section 2.3): multiple evaluations are aggregated to mitigate stochastic variation, and outputs are iteratively reviewed and refined to reduce errors.

Finally, since metrics produce heterogeneous outputs (e.g., numerical scores, binary indicators, text patterns), direct aggregation is infeasible. Orion therefore prompts the LLM to synthesize results into a ranked list of fuzzing targets.

*3.3.1 Target Identification Metrics.* Orion applies a set of metrics to prioritize candidate fuzzing targets. Each captures a distinct property correlated with vulnerability risk.

► *Cyclomatic Complexity* measures the number of linearly independent paths through a function. Higher complexity correlates with increased likelihood of defects. This metric is computed directly using the codebase indexer and static analysis tooling.

► *Internal Function Calls* counts how frequently a function is invoked by others. Bugs in frequently used functions can have greater security impact than those in rarely used code. This metric is derived from the call graph.

► *Lines of Code (LOC)* serves as a simple proxy for complexity: longer functions are harder to reason about and more likely to contain errors. LOC is measured from function bodies retrieved by the indexer.

► *Callgraph Size* measures the number of functions reachable from the analyzed function. Large reachable sets indicate higher structural complexity and broader potential impact of vulnerabilities. This is computed from the global call graph.

► *Dangerous Expressions* identifies low-level constructs such as pointer arithmetic, manual memory management, and bit-level operations, which are historically associated with memory safety issues. These are detected using LLMs, which can leverage surrounding context to distinguish benign from potentially dangerous uses.

► *Sink Functions* captures functions commonly linked to security vulnerabilities (e.g., `memcpy`, `strcpy`, `malloc`). Orion builds on Fuzz-Introspector's CVE-based function set [41, 42], extending it with LLM support to detect wrapper functions or project-specific variants, reducing manual configuration effort.

► *Parsing Functions* identifies functions that parse structured inputs, which are prone to errors due to format complexity, recursion, and edge cases. LLMs are used here as well, since names alone are insufficient to recognize project-specific parsers or unconventional naming schemes.

## 3.4 Seed Generation

The seed generation phase receives target interfaces from the previous step and produces high-quality seeds that improve initial coverage, demonstrate valid input formats, and address constraints that fuzzers typically struggle to satisfy (e.g., input validation, checksums; Section 2.1). Unlike human analysts, who create seeds after writing harnesses, Orion generates seeds first. Both tasks require understanding the function's input space, but harness construction additionally involves dependency resolution and correct invocation. By generating seeds first, Orion reduces harness complexity by providing parsing specifications and concrete examples, which improves LLM reliability in both stages (Section 2.3).

The seed generation agent analyzes each target function to characterize its inputs and behavior, then crafts diverse seed inputs covering expected paths, edge cases, and error conditions. Seeds are produced as scripts that generate input files rather than as raw binary data, since LLMs are better at code generation than binary output. Execution of these scripts provides natural feedback: errors are detected and used to refine outputs through iterative self-reflection. The final outputs consist of (1) concrete seed files, (2) textual descriptions of seed formats, and (3) code analysis results.

Preliminary code analysis is performed before seed construction to capture relevant input and behavioral properties. This combines information from the codebase indexer with reasoning from LLM agents. Each analysis produces written findings that include semantic details (e.g., from comments or developer conventions) often missed by static tools.

► *Function Signature Analysis* extracts function prototypes and parameter details, including expected value ranges, implicit relations, ownership semantics, and default values.

► *Input Surface Analysis* examines input channels beyond function arguments (e.g., files, globals, network, IPC) and identifies constraints for each.

► *Control Flow Analysis* summarizes behavior by identifying branches, loops, and recursive calls, distinguishing normal paths from error-handling paths.

► *Memory Management Analysis* identifies heap/stack interactions, allocation sizes, and buffer operations, highlighting potential risks such as out-of-bounds accesses or use-after-free.

► *Error Handling Analysis* inspects how the function detects errors, propagates faulty state, and performs recovery or graceful failure.

► *Call Dependency Analysis* explores direct, indirect, and function-pointer-based calls to understand dependencies beyond static call graphs.

► *Coverage Goal Analysis* highlights code regions most important to exercise (e.g., critical conditionals, memory operations, complex branches) and produces a prioritized coverage list.

► *Vulnerability Analysis* provides heuristic estimates of potential fuzzer-detectable bugs (e.g., integer overflows, double frees, TOCTTOU issues). The goal is not precise detection [12], but to provide starting points that guide seed generation toward high-value paths.

## 3.5 Harness Generation

Given the selected target interfaces (Section 3.3) and seeds with analysis results (Section 3.4), the harness generation phase constructs fuzzing harnesses compatible with both the seeds and the chosen fuzzing engine. High-quality harnesses must (i) correctly initialize the environment required by the interface (e.g., global state, files, sockets), (ii) parse fuzzer input and populate corresponding data structures, (iii) perform teardown operations after invocation to prevent leaks and ensure repeatability, and (iv) minimize runtime

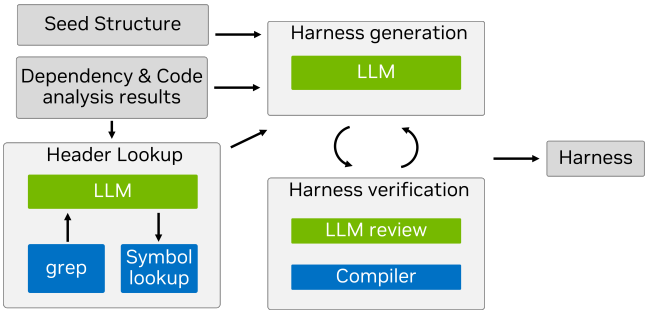


**Figure 2: Workflow of the harness generation agent.**

overhead to maximize the number of fuzzing iterations achievable within a time budget.

Orion employs two agents to meet these requirements. The *dependency analysis agent* examines how interfaces interact with their environment (other functions, filesystem, global variables) and outputs setup and teardown requirements for each interface. The *harness generation agent* then produces harnesses for individual interfaces, guided by the dependency information and seed analysis. Compiler feedback loops ensure harnesses compile and execute correctly, while self-reflection is used to refine them into compact implementations that reduce per-iteration cost.

*3.5.1 Dependency Analysis Agent.* The dependency analysis agent identifies relationships between interfaces by examining their interactions with the runtime environment, including accesses to global variables, files, environment variables, devices, and pointer-based inputs. Dependencies occur when one interface requires specific state that another creates (e.g., a file must exist before it can be opened). Teardown relationships occur when interfaces revert state changes (e.g., freeing memory, resetting variables). Understanding these relations requires reasoning about program semantics that traditional static analysis tools cannot fully capture; thus the dependency agent relies primarily on LLM reasoning, supported by compiler-derived code retrieval.

The agent operates in three passes:

(1) **State extraction:** For each interface, collect expected environment state (read accesses), modifications (writes, allocations), and pointer-based parameter expectations.
(2) **Dependency matching:** Compare expectations against modifications across interfaces. Matches are recorded as dependencies, each accompanied by a brief textual rationale.
(3) **Teardown identification:** Identify interfaces that revert prior modifications (e.g., deallocation of memory, removal of files) and record them as teardown relationships with supporting explanations.

The output is a structured list of setup and teardown requirements. Each requirement includes a description of the state to be established or reverted and the set of interfaces capable of fulfilling it.

*3.5.2 Harness Generation Agent.* The harness generation agent produces compilable harnesses using the dependency analysis outputs, seed files, and seed analysis results (Figure 2). Orion targets

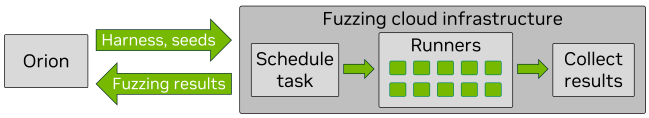


**Figure 3: FuzzerHub execution model.**

userspace fuzzing and adopts the `LLVMFuzzerTestOneInput` interface, compatible with both libFuzzer and AFL++.

The agent begins by retrieving relevant header files through the codebase indexer and lightweight text search, as harness compilation requires correct type and interface declarations. It then assembles a structured prompt containing setup and teardown requirements, header files, available types and interfaces, seed structures, and input surface analyses. From this context, the LLM generates an initial harness candidate.

Harnesses are refined through iterative loops: self-reflection is used to review and adjust the generated code, followed by compiler checks to validate correctness. Compilation failures trigger additional self-reflection to ensure corrections do not reintroduce earlier errors. The process repeats until a valid harness is produced.

The final output is a compilable harness aligned with the seed corpus and compatible with the selected fuzzing engine.

## 3.6 Fuzzing Execution

With harnesses and seed corpora prepared, the next step is to execute fuzzing campaigns against the selected interfaces. Although fuzzing engines run autonomously once launched, they require appropriate host setup and, in some cases, specialized hardware. Because infrastructure design is outside Orion's core contributions, this section focuses on the execution platform capabilities needed to support the framework.

The fuzzing platform must:

- accept automated submissions of harnesses and seed corpora;
- execute fuzzing runs in isolated environments to avoid interference and ensure reproducibility;
- collect results including coverage information, crash reports, and crashing inputs; and
- return these results to Orion for downstream processing (Figure 3).

Beyond executing campaigns, the collected statistics enable feedback loops: coverage data is compared against expectations to refine harnesses and seeds, while unexpected harness crashes can be diagnosed and corrected. Crashes attributable to the target interface, rather than the harness, transition Orion to the crash analysis and patching stages discussed next.

## 3.7 Crash Analysis

Scaling fuzzing campaigns produces large volumes of crashes, but these results are only useful if they can be triaged and acted upon. Raw crash reports often overwhelm developers, as many crashes are duplicates, false positives from harness issues, or lack sufficient detail for debugging. Without effective triage, the value of fuzzing campaigns is limited.

To address this, Orion integrates an automated crash analysis stage. The crash analysis agent, acting in place of a human analyst,

filters out harness-related faults, clusters crashes that stem from the same underlying bug, and produces detailed reports. Each report includes minimal reproducers, stack traces, and contextual explanations of the suspected root cause. These outputs reduce the effort required by developers and provide a clean handoff to the patching stage, ensuring that fuzzing results can be remediated efficiently.

*3.7.1 Crash Triaging Agent.* The crash triaging agent automates two key objectives after crash discovery: (i) distinguishing harness-induced faults (false positives) from genuine interface crashes, and (ii) identifying likely root causes with minimal reproducers for the latter.

The agent receives crashing inputs, stack traces, and access to codebase exploration tools derived from the indexer. It analyzes these artifacts to generate minimal reproducers that exercise the same failure. Executing the reproducers against the target interface provides a direct feedback loop, confirming both root cause identification and the validity of the reproducer.

Reproducers serve a dual purpose: they give developers concise artifacts to validate and understand vulnerabilities, and they provide inputs for the patching agent. The triage stage outputs a structured report containing the suspected root cause, supporting stack trace information, and validated minimal reproducers.

*3.7.2 Patching Agent.* The patching agent is the final stage of Orion's workflow. Its goal is to propose candidate patches for vulnerabilities identified during crash triage. Effective patches should (i) eliminate the root cause without altering intended interface behavior, (ii) avoid introducing new vulnerabilities or regressions, and (iii) conform to project coding style and apply cleanly.

To approach these goals, the patching agent is equipped with tools for direct file inspection and modification, symbol lookup via the codebase indexer, compiler invocation, and reproducer execution. These capabilities allow the agent to interleave code exploration, code modification, and validation. Candidate patches are compiled and tested against the minimal reproducers; only those that build successfully and prevent the original crash are returned.

The output of this stage is a patch proposal that passes automated validation. Developers may then review and apply the patch, modify it as needed, or reject it, ensuring that final acceptance remains under human control.

# 4 Evaluation

We evaluate Orion with respect to the following research questions:

**RQ1:** How effective is Orion at ranking interfaces using the identification metrics? (Section 4.3)

**RQ2:** What is the quality of the fuzzing harnesses generated by Orion? (Section 4.4)

**RQ3:** How much time does Orion save compared to human analysts? (Section 4.5)

**RQ4:** Can Orion be applied to discover previously unknown vulnerabilities? (Section 4.6)

## 4.1 Benchmark Setup

To study RQ1 and RQ2, we developed a benchmark spanning both open-source and proprietary projects. The open-source set includes `clib` [9] (commit `6d96e533`) and `H3` [46] (commit `d5af2344`), while the proprietary set consists of an NVIDIA QNX GPIO driver. This diversity allows us to assess Orion across multiple domains, code sizes, and coding styles, and to identify both strengths and limitations.

All experiments use GPT-4.1 as Orion's primary reasoning model. For evaluation and judging tasks, we employ multiple LLMs as independent reviewers: Llama 3.1 405B, Llama 3.3 70B, and Llama 3.3 Nemotron Super 49B v1.

## 4.2 Benchmark Design

The benchmark is designed to provide systematic evaluation across multiple metrics, each capturing a distinct aspect of output quality. By measuring both per-metric performance and overall success rates, it enables fine-grained analysis of Orion's behavior while reducing reliance on repeated manual assessment.

Evaluation relies on human-annotated ground truth, with different strategies depending on output type. For numerical outputs, we compare system results directly to ground truth values, applying a tolerance of 0.5 to account for floating-point precision and off-by-one variation. For textual outputs, we assess semantic equivalence against ground truth using LLM judges, which compare candidate outputs to reference statements and determine correctness.

We note that LLM judges are only reliable when reference ground truth is available; without such standards, judgments can be inconsistent. Finally, because interface identification and harness generation present distinct challenges, we define custom evaluation metrics for each task, as detailed in the following subsections.

### Interface Identification Metrics

The evaluation metrics for interface identification align with those used by Orion's analysis agent. They fall into three categories:

- *Numerical metrics:* Lines of Code, Cyclomatic Complexity, Internal Calls, and Callgraph Size. These are compared directly against ground truth values, with minor tolerance applied for floating-point precision.
- *Qualitative metrics:* Sink Functions, Dangerous Expressions, and Parsing Functions. These produce natural-language outputs that are evaluated against annotated ground truth using LLM judges for semantic equivalence.
- *Auxiliary classification:* Identification of functions with no parameters, reported as a binary accuracy metric.

### Fuzz Harness Quality Metrics

The harness generation benchmark evaluates Orion's outputs against eight criteria that reflect the requirements for functional and effective fuzz harnesses (Section 2.1). All metrics are assessed using LLM judges against human-annotated ground truth.

- *Verification*: Checks whether harnesses pass self-reflection and compiler validation. Only compilable and semantically valid harnesses proceed to fuzzing.
- *Input Channels*: Ensures harnesses accept input through all relevant channels (e.g., parameters, globals, files).
- *Input Structure*: Evaluates correct transformation of fuzzer input into the target interface format, including appropriate type usage and population.

| Metric | clib | H3 | GPIO | Average |
|---|---|---|---|---|
| Lines of Code | 100.00% | 100.00% | 100.00% | 100.00% |
| Cyclomatic Complexity | 100.00% | 100.00% | 100.00% | 100.00% |
| Internal Calls | 100.00% | 100.00% | 100.00% | 100.00% |
| Callgraph Size | 100.00% | 100.00% | 100.00% | 100.00% |
| Sink Functions | 86.21% | 96.97% | 93.33% | 94.30% |
| Dangerous Expressions | 89.66% | 75.76% | 66.67% | 76.58% |
| Parsing Functions | - | 100.00% | 100.00% | 100.00% |
| No Arguments | 100.00% | 100.00% | 60.00% | 85.71% |
| Overall | 95.42% | 96.11% | 93.49% | 95.48% |

**Table 1: Benchmark Results For Interface Identification Metrics**

| Metric | clib | H3 | GPIO | Average |
|---|---|---|---|---|
| Verification | 100.00% | 100.00% | 42.42% | 88.20% |
| Input Channels | 93.10% | 97.98% | 100.00% | 97.52% |
| Input Structure | 82.76% | 90.91% | 60.61% | 83.22% |
| Dependencies | 48.28% | 86.87% | 27.27% | 67.70% |
| Teardown | 89.66% | 100.00% | 33.33% | 84.47% |
| Unnecessary Functions | 96.55% | 97.98% | 90.91% | 96.27% |
| Proper Use of Fuzzed Data | 82.76% | 94.95% | 100.00% | 93.79% |
| Target Function Invocation | 100.00% | 100.00% | 100.00% | 100.00% |
| Overall | 86.64% | 96.09% | 69.32% | 88.90% |

**Table 2: Benchmark Results For Fuzz Harness Quality Metrics**

- *Dependencies*: Confirms correct program setup, including initialization routines and dependency handling. Missing dependencies are counted as failures, as they prevent successful execution.
- *Teardown*: Validates proper resource cleanup and program state reset after each iteration to avoid contamination and performance degradation.
- *Unnecessary Functions*: Flags operations outside the target interface, setup, teardown, or dependencies that reduce fuzzing throughput.
- *Proper Use of Fuzzed Data*: Checks that input is derived correctly from fuzzer data, with required constants or checksums incorporated appropriately.
- *Target Function Invocation*: Ensures the harness ultimately calls the intended target interface, even after iterative refinement and error correction.

## 4.3 RQ1: Interface Identification Metrics

Table 1 reports Orion's performance on the interface identification task. Across all metrics and projects, Orion achieves an average success rate of 95.5%. The system attains perfect scores on the four numerical metrics (`Lines of Code`, `Cyclomatic Complexity`, `Internal Calls`, and `Callgraph Size`), indicating reliable integration of tool-based measures with LLM reasoning. High success rates on the remaining metrics reflect the benefits of reliability techniques such as self-consistency, self-reflection, reasoning-focused prompts, and precise code retrieval.

We identify three primary sources of error in the more challenging metrics (`Sink Functions` and `Dangerous Expressions`): (i) **Insufficient context specificity**, particularly in H3 where heavy macro usage makes code harder to resolve. Providing too much context risks the needle-in-the-haystack effect (Section 2.3), while too little context omits critical information. (ii) **Counting and dereference errors**, including miscounts of sink functions and missed pointer dereferences. These stem from known LLM weaknesses with arithmetic and limited contextual resolution. (iii) **Self-reflection pitfalls**, where incorrect initial analyses occasionally persisted because the review stage accepted, rather than corrected, flawed reasoning.

Although carefully crafted prompts and feedback loops mitigate most failures, these results highlight areas where LLM-based analysis remains vulnerable, underscoring the importance of validation and tool support.

| Task | Human Effort | Orion Effort | Speedup |
|---|---|---|---|
| Interface Identification | ~1 week | 62 min | 92× |
| Harness Generation | ~2 weeks | 49 min | 204× |
| Patching | 1 hr | 1 min 20 sec | 46× |

**Table 3: Time savings achieved by Orion compared to human effort.**

## 4.4 RQ2: Fuzz Harness Quality

Orion achieves an average success rate of 88.9% across harness quality metrics, making this task more challenging than interface identification. The lowest scores appear in `Dependencies` (67.7%) and `Teardown` (84.5%), reflecting the inherent difficulty of reasoning about initialization and cleanup requirements from source code - a task that is error-prone even for human experts. In contrast, Orion performs strongly on `Verification`, `Input Channels`, and `Target Function Invocation`, showing that iterative refinement and compiler feedback reliably enforce basic correctness.

Error analysis reveals three main challenges. First, dependency handling remains brittle: missing initialization or teardown steps frequently led to invalid harnesses. Second, harnesses often failed compilation on first attempt, making iterative refinement through compiler feedback and self-reflection essential. Finally, the GPIO driver results illustrate the difficulty of scaling to large proprietary codebases with unfamiliar build systems. Here, Orion frequently produced incorrect header assumptions and misinterpreted compiler diagnostics, reducing verification accuracy.

Overall, these results show that while Orion can reliably generate functional harnesses in many cases, dependency analysis and complex build environments remain key limitations that warrant further research.

## 4.5 RQ3: Time Savings

To quantify Orion's practical benefit, we measured the time required for interface identification, harness creation, and vulnerability patching, using NVIDIA internal data for tasks comparable to the clib zero-days.

As shown in Table 3, Orion reduces workflows that typically take days or weeks of analyst effort to less than two hours of automated execution. Harness generation yields the largest speedup, reflecting the high manual burden of crafting harnesses through detailed program analysis and iterative refinement. Interface identification achieves a smaller (yet still substantial) gain, as it must

process significantly more candidate interfaces. Here, the use of self-consistency increases runtime by requiring multiple parallel LLM queries, introducing a deliberate speed–accuracy tradeoff. Finally, patching also provides large acceleration, but still requires human verification of candidate patches, limiting potential speedup relative to earlier workflow stages.

### 4.6 RQ4: Zero-day Vulnerabilities

The ultimate measure of Orion's effectiveness is whether its outputs can lead to the discovery of new, previously unknown vulnerabilities. To evaluate this, we applied Orion to the top 12 candidate functions identified in the H3 and clib codebases, executing each harness for up to 24 hours or until a crash was detected.

This evaluation yielded two previously unknown vulnerabilities in `clib`: a controlled stack buffer overflow and a null pointer dereference. Both issues were discovered automatically through Orion's generated harnesses and seeds, demonstrating that the system can not only reproduce known bug-finding workflows but also surface new security-critical defects.

At the time of writing, both vulnerabilities remain under coordinated disclosure. Full technical details are therefore withheld, but will be released once responsible disclosure has concluded.

## 5 Discussion

The code knowledge base serves as Orion's crucial foundation, enabling precise context retrieval for LLM tasks such as symbol definitions and function callers. While these tasks resemble those of Language Server Protocols (LSPs) [35] used by modern IDEs, existing LSPs require specialized wrappers to support AI workflow requirements. Creating LSP wrappers targeted at AI systems could provide valuable community building blocks and could unlock the full potential of LLMs for code analysis.

Build system interactions present significant challenges due to diverse implementations across projects, even within identical build systems. This variety complicates harness integration into the codebase and tasks requiring build configuration knowledge, such as header resolution for symbols. Orion currently addresses this incompletely, as build system understanding represents an orthogonal problem to fuzzing automation. Solutions involve either requiring detailed user inputs (reducing usability) or developing specialized build system analysis components (increasing complexity but lowering barriers), where LLM agents show promise.

Orion's modular design intentionally decouples components to enable isolated usage and flexible adaptation for special cases. This modularity facilitates experimentation with different approaches across individual workflow steps and multi-step combinations, with Section 6 identifying promising research directions.

## 6 Related Work

In this section we describe prior efforts in the area of scaling fuzzing as well as the use of LLMs for security tasks.

Prior work relevant to Orion falls into three areas: (i) applying LLMs to fuzzing, (ii) scaling fuzzing workflows without generative AI, and (iii) broader AI applications in software security.

**LLMs for Fuzzing.** Recent research has applied LLMs to different components of the fuzzing pipeline. *End-to-end systems:* The closest effort is DARPA's AIxCC challenge [11], where finalist teams built systems for vulnerability discovery and patching across diverse codebases. Unlike Orion, these systems focused on general vulnerability detection rather than workflow automation, and relied on organizer-provided harnesses. *Single-step automation:* Harness generation was first demonstrated by Liu et al. [28] through OSS-Fuzz integration, followed by iterative refinement [6, 31] and context-aware prompting [2, 56, 59, 62]. Seed generation has been explored via refinement [44], targeted context [29], and constraint solving [57]. For crash triage and patching, CodeRoverS [64] showed that minimally customized agents can generate fixes for fuzzing-induced crashes. All of these focus on isolated workflow steps, whereas Orion automates the complete pipeline from target identification through patching. *Closed-source fuzzing:* LibLMFuzz [19] applies agentic workflows with disassemblers, compilers, and fuzzers to fuzz binary-only targets. *LLM-enhanced fuzzing engines:* A separate line of work uses LLMs to augment fuzzing itself, including input generation and mutation [34, 60, 63], constraint solving [33, 61], and engine replacement (e.g., Fuzz4All [55]). These efforts complement Orion but focus on fuzzing mechanics rather than workflow automation.

**Scaling Fuzzing without Generative AI.** Before LLMs, researchers explored traditional automation for individual workflow steps. Target identification relied on static metrics [50]. Harness generation used unit tests [22], static analysis, or dynamic tracing [21, 25, 53]. Seed generation applied structural learning, grammars, or ML-based feature extraction [8, 30, 48]. Crash analysis employed dynamic tracing for invariant learning [5] and grouping methods for root cause identification [23]. Automated patching leveraged genetic programming [26] or symbolic execution with program synthesis [20, 32, 38]. While valuable, these methods addressed only single workflow phases, often without semantic understanding, and produced outputs that were difficult to maintain in practice.

**AI for Security.** Beyond fuzzing, generative AI has enabled a range of security applications. Big Sleep [1] (formerly Project Naptime [16]) performs variant analysis on large codebases. XBOW [54] applies agentic AI to web application penetration testing. VulRAG [13] leverages vulnerability history for LLM-powered vulnerability detection. Other work augments static analysis, such as reducing false positives [27, 36] and automatically generating checkers to improve scalability [58]. These advances highlight the growing role of AI in software security and motivate Orion's focus on end-to-end fuzzing workflows.

## 7 Conclusion

We introduced Orion, an end-to-end framework for fuzzing workflow automation that enables scalable campaigns on production-size codebases. Orion combines LLM-based agents with deterministic tools to automate the complete pipeline from target identification through patch generation. The hybrid design exploits LLM strengths in semantic reasoning and code understanding while relying on traditional tooling for verification, iterative refinement, and precise context retrieval.

Evaluation results demonstrate that Orion delivers both effectiveness and efficiency: it achieved 95.5% accuracy in interface

identification, 88.9% in harness generation, reduced manual effort by up to 204×, and uncovered two previously unknown vulnerabilities. These findings highlight the feasibility of integrating LLM agents with conventional techniques for security-critical software testing.

Important challenges remain. Improved dependency analysis, support for complex build environments, and more rigorous patch validation represent promising directions for future work. Addressing these challenges can further increase the reliability and adoption of automated fuzzing workflows in large-scale software development.

## Acknowledgments

We thank Tom McReynolds for supporting this project.

## References

[1] Miltos Allamanis, Martin Arjovsky, Charles Blundell, Lars Buesing, Mark Brand, Sergei Glazunov, Dominik Maier, Petros Maniatis, Guilherme Marinho, Henryk Michalewski, Koushik Sen, Charles Sutton, Vaibhav Tulsyan, Marco Vanotti, Theophane Weber, and Dan Zheng. 2024. From Naptime to Big Sleep: Using Large Language Models To Catch Vulnerabilities In Real-World Code. https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html#bigsleepteam

[2] Georgios Androutsopoulos and Antonio Bianchi. 2025. deepSURF: Detecting Memory Safety Vulnerabilities in Rust Through Fuzzing LLM-Augmented Harnesses. https://arxiv.org/abs/2506.15648

[3] Anthropic. [n. d.]. Claude Code. https://www.anthropic.com/claude-code

[4] Anysphere. [n. d.]. Cursor. https://www.cursor.com/

[5] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko

[6] Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. 2025. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *34th USENIX Security Symposium (USENIX Security 25*. USENIX Association. https://www.usenix.org/conference/usenixsecurity25/presentation/chen-chuyang

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374

[8] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. 2019. Optimizing Seed Inputs in Fuzzing with Machine Learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 244–245. doi:10.1109/ICSE-Companion.2019.00096

[9] clib. 2017. clib. https://github.com/clibs/clib

[10] Cognition. [n. d.]. Devin. https://devin.ai/

[11] DARPA. 2023. AI Cyber Challenge. https://aicyberchallenge.com/

[12] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? https://arxiv.org/abs/2403.18624

[13] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2025. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. https://arxiv.org/abs/2406.11147

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[15] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. https://arxiv.org/abs/2312.10997

[16] Sergei Glazunov and Mark Brand. 2024. Project Naptime: Evaluating Offensive Security Capabilities of Large Language Models. https://googleprojectzero.blogspot.com/2024/06/project-naptime.html

[17] Google. 2016. OSS-Fuzz. https://github.com/google/oss-fuzz

[18] Google. 2016. Syzbot. https://syzkaller.appspot.com/upstream

[19] Ian Hardgrove and John D. Hastings. 2025. LibLMFuzz: LLM-Augmented Fuzz Target Generation for Black-box Libraries. https://arxiv.org/abs/2507.15058

[20] Zunchen Huang and Chao Wang. 2024. Constraint Based Program Repair for Persistent Memory Bugs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 91, 12 pages. doi:10.1145/3597503.3639204

[21] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2271–2287. https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[22] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2676–2692. doi:10.1109/SP46215.2023.10179394

[23] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash Deduplication Through Root-Cause Clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3460120.3485364

[24] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[25] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*.

[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[27] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. doi:10.1145/3649828

[28] GDongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. 2023. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html

[29] Zhengxiong Luo, Qingpeng Du, Yujue Wang, Abhik Roychoudhury, and Yu Jiang. 2025. Enhancing Protocol Fuzzing via Diverse Seed Corpus Generation. *IEEE Transactions on Software Engineering* (2025), 1–16. doi:10.1109/TSE.2025.3595396

[30] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2019. SmartSeed: Smart Seed Generation for Efficient Fuzzing. https://arxiv.org/abs/1807.02606

[31] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. https://arxiv.org/abs/2312.17677

[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. doi:10.1145/2884781.2884807

[33] Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. 2024. Large Language Model assisted Hybrid Fuzzing. https://arxiv.org/abs/2412.15931

[34] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[35] Microsoft. 2016. Language Server Protocol. https://microsoft.github.io/language-server-protocol/

[36] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3664646.3664777

[37] Elliot Nelson, Georgios Kollias, Payel Das, Subhajit Chaudhury, and Soham Dan. 2024. Needle in the Haystack for Memory Based Large Language Models. https://arxiv.org/abs/2407.01437

[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. doi:10.1109/ICSE.2013.

6606623
[39] Beatrice Nolan. 2025. An AI-powered coding tool wiped out a software company's database, then apologized for a 'catastrophic failure on my part'. https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure/.
[40] OpenAI. [n. d.]. Codex. https://openai.com/codex/
[41] OpenSSF. 2022. Fuzz-Introspector. https://github.com/ossf/fuzz-introspector
[42] OpenSSF. 2022. Fuzz-Introspector CWE data. https://github.com/ossf/fuzz-introspector/blob/main/src/fuzz_introspector/analyses/data/cwe_data.py
[43] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1974–1993. doi:10.1109/SP54263.2024.00137
[44] Wenxuan Shi, Yunhang Zhang, Xinyu Xing, and Jun Xu. 2024. Harnessing Large Language Models for Seed Generation in Greybox Fuzzing. https://arxiv.org/abs/2411.18143
[45] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. https://arxiv.org/abs/2303.11366
[46] Uber. 2015. H3. https://github.com/uber/h3
[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. https://arxiv.org/abs/1706.03762
[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. doi:10.1109/SP.2017.23
[49] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. https://arxiv.org/abs/2203.11171
[50] Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (Singapore, Singapore) *(ASIA CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1539–1553. doi:10.1145/3634737.3661141
[51] Lilian Weng. 2023. LLM Powered Autonomous Agents. https://lilianweng.github.io/posts/2023-06-23-agent/
[52] Windsurf. [n. d.]. Windsurf IDE. https://windsurf.com/
[53] Wei-Cheng Wu, Stefan Nagy, and Christophe Hauser. 2025. WildSync: Automated Fuzzing Harness Synthesis via Wild API Usage Recovery. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA043 (June 2025), 22 pages. doi:10.1145/3728918
[54] XBOW. [n. d.]. XBOW. https://xbow.com/
[55] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121
[56] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 243–254. doi:10.1109/ICSE-Companion66252.2025.00079
[57] Hanxiang Xu, Yanjie Zhao, and Haoyu Wang. 2025. Directed Greybox Fuzzing via Large Language Model. https://arxiv.org/abs/2505.03425
[58] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. https://arxiv.org/abs/2503.09002
[59] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 560–573. http://dx.doi.org/10.1145/3676641.3716022
[60] Shaoyu Yang, Chunrong Fang, Haifeng Lin, Xiang Chen, and Zhenyu Chen. 2025. May the Feedback Be with You! Unlocking the Power of Feedback-Driven Deep Learning Framework Fuzzing via LLMs. https://arxiv.org/abs/2506.17642
[61] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. 2025. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity25/presentation/yang-yupeng
[62] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1223–1235. doi:10.1145/3650212.3680355
[63] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing. https://arxiv.org/abs/2406.07714
[64] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, Dongge Liu, Abhishek Arya, Oliver Chang, and Abhik Roychoudhury. 2024. Fixing Security Vulnerabilities with AI in OSS-Fuzz. https://arxiv.org/abs/2411.03346