Locus: Agentic Predicate Synthesis for Directed Fuzzing

Jie Zhu
University of Chicago
Chicago, USA

Chihao Shen
University of Maryland
College Park, USA

Ziyang Li
Johns Hopkins University
Baltimore, USA

Jiahao Yu
Northwestern University
Evanston, USA

Yizheng Chen
University of Maryland
College Park, USA

Kexin Pei
University of Chicago
Chicago, USA

Abstract

Directed fuzzing aims to find program inputs that lead to specified target program states. It has broad applications, such as debugging system crashes, confirming reported bugs, and generating exploits for potential vulnerabilities. This task is inherently challenging because target states are often deeply nested in the program, while the search space manifested by numerous possible program inputs is prohibitively large. Existing approaches rely on branch distances or manually-specified constraints to guide the search; however, the branches alone are often insufficient to precisely characterize progress toward reaching the target states, while the manually specified constraints are often tailored for specific bug types and thus difficult to generalize to diverse target states and programs.

We present Locus, a novel framework to improve the efficiency of directed fuzzing. Our key insight is to synthesize predicates to capture fuzzing progress as semantically meaningful intermediate states, serving as milestones towards reaching the target states. When used to instrument the program under fuzzing, they can reject executions unlikely to reach the target states, while providing additional coverage guidance. To automate this task and generalize to diverse programs, Locus features an agentic framework with program analysis tools to synthesize and iteratively refine the candidate predicates, while ensuring the predicates strictly relax the target states to prevent false rejections via symbolic execution. Our evaluation shows that Locus substantially improves the efficiency of eight state-of-the-art fuzzers in discovering real-world vulnerabilities, achieving an average speedup of 41.6×. So far, Locus has found nine previously unpatched bugs, with three already acknowledged with draft patches.

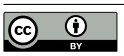
CCS Concepts

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

Keywords

Directed Fuzzing, Test Generation, LLM Agent

ACM Reference Format:
Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. 2026. Locus: Agentic Predicate Synthesis for Directed Fuzzing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773102



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773102

1 Introduction

Directed Grey-box Fuzzing (DGF) aims to search for program inputs leading its execution to reach specific target program states, *e.g.,* an index variable used to access an array is out of the array bounds, thereby uncovering potential bugs or vulnerabilities. It is widely used in software engineering and security applications, including debugging system crashes [85, 92], testing patches [5, 64], verifying bug reports from static analysis [16, 67], and generating Proof-of-Concept (PoC) exploits of vulnerabilities [7, 56]. While the original purpose of directed fuzzing is largely not for discovering new bugs, *i.e.,* it needs a specified target, it has found numerous impactful zero-day vulnerabilities [9, 53, 68, 78, 93].

DGF is challenging, as the target program states are often deeply nested in the program, while the search space introduced by the complexity of real-world software is prohibitively large. To speed up the search and schedule promising inputs, most existing techniques rely on metrics based on control flow proximity, *e.g.,* the distance to target location in the control flow graph, or heuristics based on the semantics of the branch predicates [13, 51, 78], *e.g.,* the target state `if x==42` has the distance metric of $|x-42|$. However, such feedback is sometimes too sparse or indirect to reliably measure the progress, especially when there is a long chain of *implicit* preconditions guarding the target program states [2, 26, 36, 44, 45, 66, 73, 80, 106]. For example, triggering CVE-2018-13785 in `libpng` requires a PNG file to satisfy a precise sequence of preconditions, *i.e.,* valid signature, correct chunk layout, specific IHDR fields (*e.g.,* bit depth, color type), and a magic image width (`0x55555555`), to trigger an integer overflow [44], while the predicates to explicitly check these conditions are largely absent in the code to provide an incremental progress guidance.

To capture the intricate feedback to improve search efficiency, more advanced approaches identify progress-capturing constraints in the program to drive execution towards satisfying specific temporal orders and preconditions [2, 4, 27, 36, 41, 47, 51, 53, 65, 69]. However, these constraints are often manually crafted by experts and tailored to specific target state types, *e.g.,* focusing on a temporal memory safety bug like a use-after-free by enforcing an allocate–free–use sequence. As different programs can have diverse target states and disparate functionalities, the feedback metric effective in one case may not generalize to another [15].

As Machine Learning (ML) and Large Language Models (LLMs) have demonstrated surprising code reasoning capabilities, there

has been a growing interest in extending such capabilities to help guide fuzzing [70]. A common approach is to employ LLMs to directly generate inputs or grammar-aware input generators (*i.e.,* fuzz driver or harness) [11, 40, 56, 57, 62, 81, 102, 107], where the preconditions for reaching target states are expressed as part of the input grammar constraints. However, not all conditions to reach target states can be easily represented as input grammars. For example, Figure 1 shows that an intermediate state (`!found_plte`) necessary to trigger the buffer overflow in `libpng` only emerges in the middle of the execution and cannot be easily checked at the input level. More importantly, such a task formulation is particularly challenging for LLMs, as *reasoning from the target program states all the way to the input* often requires an extremely long context and convoluted analysis chain [21, 42, 48, 49, 52, 74, 75, 75], a common pitfall for LLM hallucination [76, 104], let alone the challenge to verify the correctness of the LLM generated inputs and harnesses, which can significantly impede the fuzzing progress if the generated constraints are incorrect [41, 69].

Our approach We present Locus, a new framework that integrates LLMs' code reasoning capabilities for directed fuzzing by synthesizing semantically meaningful and *verifiable* predicates to guide the search. Unlike existing LLM-based approaches that focus on constraining the search space at the input (harness) level, which poses a high reasoning burden on LLMs and can be hard to verify, Locus generalizes the constraint generation to be at arbitrary program points. Specifically, given a target state to reach, Locus *automates* the analysis about the intermediate program states that verifiably *relax* the target states and synthesizes predicates as a curriculum to capture the gradual progress towards reaching them. Such predicates serve as the preconditions dominating all executions to reach the target states, providing fine-grained progress feedback to guide the fuzzers for input scheduling and early termination [38, 60, 82]. As these predicates are implemented as source-level instrumentation, they are *agnostic* to any fuzzer implementations and can thus be integrated without any customization. Moreover, as the instrumentation is a one-time offline process, the cost of running Locus can be amortized in all the succeeding fuzzing campaigns.

Agentic design Automating the generation of effective progress-capturing predicates for diverse target states involves nontrivial reasoning across the entire codebase, spanning multiple functions and their associated data and control flows. Simple prompting techniques, such as in-context learning, chain-of-thought (CoT) prompting, retrieval-augmented generation, etc., can hardly support sophisticated analysis at the repository level. To this end, Locus features an agentic synthesizer-validator workflow, equipped with diverse program analysis tools to support the traversal of the control flow graph, tracking data dependencies, retrieving function calls, and symbolic execution. They serve as the agent's action space, allowing the LLM to *reason* via CoT and *act* by calling them [103] to iteratively propose, refine, and validate candidate predicates.

Importantly, by constraining the output space of the agent at the *predicate*-level, Locus ensures any (inevitable) LLM errors are corrected before deployment for fuzzing. Specifically, Locus's output is validated by both the compiler and symbolic execution [8]. The former checks the syntactic correctness, while the latter ensures the generated predicates strictly relax the target states, *i.e.,* by checking whether there exists a path that violates the predicates while satisfying the target states. Such a strict relaxation ensures that the fuzzing execution can be safely early-terminated if it violates the predicate. Figure 2 shows the Locus's workflow.

Results We evaluate Locus on the Magma benchmark [35] with eight widely used libraries and ten vulnerability types across eight state-of-the-art fuzzers, covering both directed and undirected ones. Locus achieves a significant 70.3× speedup on average for directed fuzzers, with up to 214.2× speedup when integrated to accelerate SelectFuzz [60], one of the state-of-the-art directed fuzzers. For coverage-guided fuzzers, Locus accelerates them by 13× on average, including 15.3× speedup for the extensively optimized fuzzer like AFL++. So far, Locus has found nine previously unpatched bugs. We have responsibly reported all bugs to the maintainers, and three of which already have pending fixes.

2 Overview

In this section, we first describe the background of directed fuzzing. We then contrast our idea with the existing approaches to demonstrate how Locus complements the existing design (Figure 1).

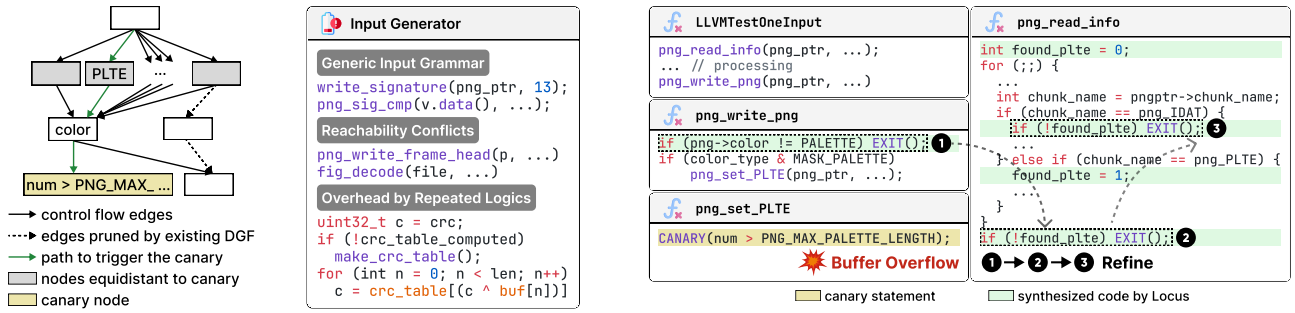
2.1 Directed Fuzzing

Canary Directed fuzzing aims to generate inputs that drive the program execution to reach predefined program states. In this paper, we represent these states with *canaries* [35], which are considered reached when the corresponding canary condition is satisfied. Early works like AFLGo [6] and Beacon [38] treat canaries primarily as the reachability to particular program points, *i.e.,* specific line numbers in a code file. Recent works [35, 95] adopt assertion-based canaries, where predicates are explicitly inserted to check vulnerability-triggering conditions at runtime. Two examples of canaries are highlighted in yellow in Figure 1.

Canaries for directed fuzzing originate from various sources, including static analysis alerts, manually identified vulnerability sites, or runtime sanitizers. For example, address sanitizers [77] can also be approximately viewed as canaries checking memory safety violations, *e.g.,* inserting `canary(index > maxbound)` to detect out-of-bound accesses.

Common strategies to reach canaries Previous research on directed fuzzing can be broadly grouped into four main strategies:

- Distance-guided scheduling. This approach approximates the distance from the currently covered regions of the CFG to the canary, and then prioritizes seeds that are more likely to drive execution along paths that reduce this distance [6, 12, 25]. For paths that are unlikely to reach the canary, the execution can be early terminated [38, 60, 82].
- Specialized progress-capturing state representations. Some approaches introduce domain-specific abstractions tailored for certain classes of vulnerabilities [47, 65]. For example, tracking dataflow from memory allocation to deallocation events for the use-after-free vulnerability.
- LLM-assisted harness generation. A more recent direction leverages LLMs to synthesize input generators to manipulate



(a) A simplified CFG (b) LLM-generated code (c) Locus synthesizes predicates in relevant functions.

Figure 1: A motivating example (CVE-2013-6954) showing how Locus complements existing works. (a) Traditional approaches based on distance to targets in CFG lack fine-grained guidance to distinguish nodes when they have the same distance. (b) LLM-based harness generation is limited to help reach the target. (c) Predicates (as `if` statements) synthesized by Locus provide extra semantic guidance for DGFs, while relaxing the constraint generation from input-level to arbitrary program points.

raw program inputs [58, 97, 102, 108]. These methods attempt to constrain the valid execution at the input level by leveraging the learned input grammar knowledge in the LLMs.

2.2 Motivating Example

We use a real-world vulnerability CVE-2013-6954 in `libpng`, a widely used C library for parsing PNG files, to demonstrate how Locus complements existing approaches.

A simplified CFG of this vulnerability is shown in Figure 1a, where all the nodes highlighted in gray are equidistant to the canary node (highlighted in yellow), but only one node `PLTE` is in the path that can lead to the vulnerability (annotated as the green arrows). Based on this, fuzzers can only perform conservative pruning (annotated as dashed arrows) while omitting a large part of the paths that are irrelevant to the vulnerability, *e.g.,* all solid black arrows. Capturing the progress towards reaching this canary requires specialized knowledge about `libpng` parsing logic. However, developing such specialized progress-capturing state representations requires manual efforts from experts and cannot easily generalize to different vulnerabilities and programs.

Figure 1b shows an example of the input generator synthesized by LLM-based approaches that enforced some specific input constraints. However, this approach is inherently limited as program inputs can have sophisticated structures that cannot be easily enforced at the input level, *e.g.,* the PNG file contains a compressed data chunk `IDAT`. Therefore, the constraints on the input generated by the LLM often reduce to generic input grammars, *e.g.,* `png_sig_cmp` only trivially ensures the input is a valid PNG file. We also note that the synthesized generators cannot be easily checked to determine whether they can effectively help reach the canary. For example, `png_write_frame_head` constrains the generated input to a special PNG type `APNG` that is essentially impossible to reach the canary, but automatically checking this fact is challenging. Moreover, to constrain the input to satisfy certain properties necessary to reach the target, the generator sometimes needs to repeat the input processing logic presented in the execution path. Such repeated execution introduces additional overhead.

Locus synthesizes progress-capturing predicates at arbitrary program points to provide more fine-grained guidance and complement the above approaches. Specifically, Locus generates a predicate in `png_read_info` (shown in Figure 1c), representing the precondition to trigger a real-world buffer overflow vulnerability and terminate the execution if it is not satisfied (`if(!found_plte)`). The target state is highlighted in a `canary` statement. Locus's trajectory reveals that it relies on the semantic reasoning of `libpng`'s parsing behaviors for generating this semantically meaningful predicate.

Specifically, PNG files consist of a series of structured chunks, many of which are optional and can appear in varying orders. One such optional chunk, `PLTE`, stores the palette data used for indexed-color images and must adhere to strict size constraints. In this example, an input PNG file can trigger a buffer overflow in the `png_set_PLTE` function only when the PNG file contains a `PLTE` chunk, which serves as the necessary state before the target state can be reached, *i.e.,* the palette size exceeds the expected bounds.

Existing fuzzers struggle with such a vulnerability due to the complex parsing logic. Since PNG chunks may appear in arbitrary order and many are optional, the primary parsing routine implements a loop in `png_read_info` to iteratively process each chunk. Within this loop, multiple branches exist, and each is responsible for handling a specific chunk type, *e.g.,* `IHDR`, `IDAT`, `PLTE`. This loop makes it possible to identify whether a given PNG file contains a `PLTE` chunk. However, since these branches are syntactically parallel and executed without a fixed order, they all appear equidistant from the vulnerability site (`png_set_PLTE`) in the control flow graph. As a result, the naive path distances cannot distinguish between them to prioritize one path over the other. This explains why, in Table 2, all existing fuzzers take a substantial amount of time to reach this target state in `libpng`.

Locus starts with generating a predicate to check whether the input PNG file contains a PLTE chunk at the caller of the canary function (❶). This implies that if the PNG file does not contain a PLTE chunk, the execution can terminate, as it is impossible to reach the target state. While this predicate is semantically correct, *i.e.,* the symbolic execution confirms there is no feasible path to

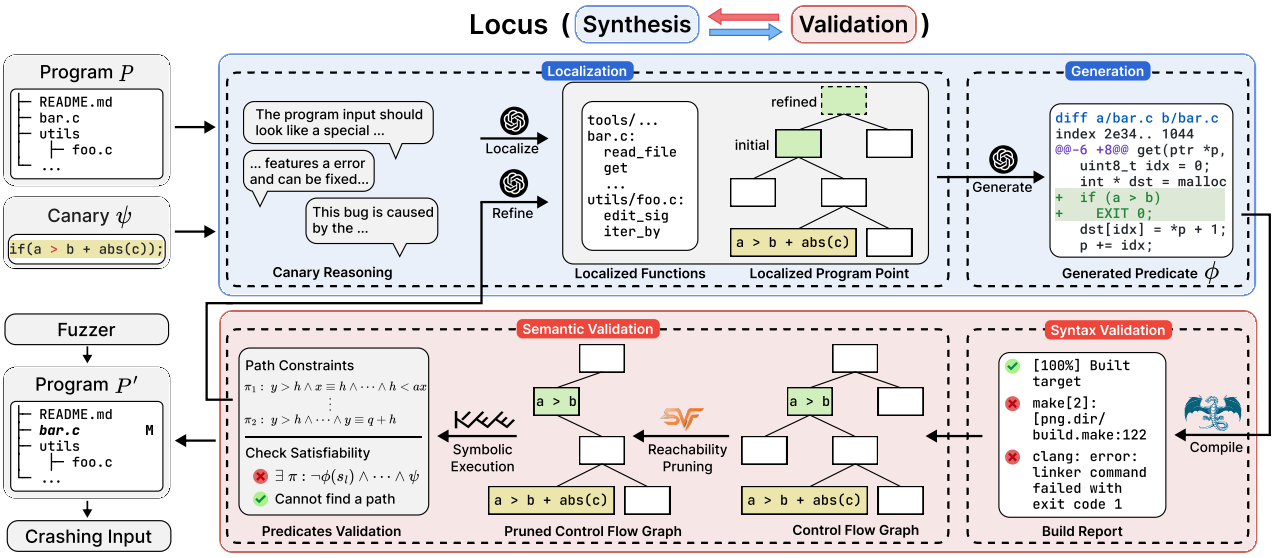


Figure 2: Overview of Locus workflow. Locus takes as inputs the program codebase P and the canary ψ, and produces a program P' instrumented with the progress-capturing predicates. The predicate branches provide extra coverage feedback and guards (via early termination) to guide the fuzzer toward reaching the target state, *i.e.,* canary ψ, more efficiently.

satisfy the target state while violating the generated predicate (Section 3.4), and can help filter out non-palette-based PNG files, it is almost redundant as there is an existing one immediately after the generated check (`if (color_type & MASK_PALETTE)`), and thus the generated predicate could barely help the fuzzer.

To address this issue, Locus includes another refinement iteration (Algorithm 1) to propagate this predicate to a location closer to the program entry such that the infeasible execution can be terminated earlier. By traversing the call graph and retrieving the functions along the call chain, the refinement proposes a new location in function `png_read_info` right after the parsing loop (❷). Such a propagated predicates will also be validated again to ensure it remains a strict precondition to reach the target states. At the next iteration of refinement, Locus proposes to propagate this predicate further inside the branch that parses the `IDAT` chunk (❸) and creates a new helper variable `found_plte`. That is because the specification of the PNG file requires that the optional `PLTE` chunk must appear before the `IDAT` chunk. Therefore, by the time the parsing procedure reaches the `IDAT` chunk, we can already determine whether the input PNG file contains a `PLTE` chunk.

Such a finalized predicate benefits all fuzzers, as it can prioritize inputs with PLTE properties early in the execution and save the fuzzers from considering PLTE-irrelevant PNG images. With such a predicate, AFLGo [6] gained an impressive 8× speedup in triggering this vulnerability with only a three-line change in `png_read_info`.

3 Methodology

Figure 2 illustrates the high-level workflow of Locus. Given a specified canary ψ and the program codebase P, Locus outputs a new codebase P' instrumented by a set of predicates Φ, where each $\phi \in \Phi$ is represented by a branch condition with early exit if the predicate condition is not satisfied. The synthesizer is responsible for generating candidate predicates Φ, while the validator ensures that Φ are both syntactically valid and semantically consistent with the ψ, *i.e.,* relaxing the canary conditions. The fuzzer will run on P' to receive more progress feedback to the canary while enjoying the early termination. In the following, we formalize the task and then elaborate on each design component in Locus.

3.1 Formalization

We use the notation $P \Downarrow_x S$ to indicate that the program P, when executed with input $x \in X$ sampled from P's input space X, can reach a set of program states S. Assume triggering a vulnerability v can be characterized as reaching a set of program states S_v. Given the program under fuzz (P') instrumented by Φ, we need to make sure that the early termination introduced in Φ preserves the same fuzzing behavior on P' as that of P, *i.e.,* Φ do not reject any $x \in X$ that would have reached S_v in P. To this end, we formally define *fuzzing admissibility.*

Definition 1. *For a vulnerability v, the program P' is fuzzing admissible to P, iff* $\forall x \in X, P' \Downarrow_x S_v \implies P \Downarrow_x S_v$.

While it is unlikely to ensure P' is fuzzing admissible to P in general without a pre-defined target vulnerability v, we show such a property is well-defined when v is given and explicitly represented by the canary ψ.

A program predicate $\phi : s \to \{\text{True}, \text{False}\}$ is a boolean mapping over the program state space. Concretely, any conditions inside the branch statements, *e.g.,* `if`, `while`, or `assert`, can be regarded as predicates, as they evaluate a Boolean expression over the program state. We define a special class of predicates, namely *canaries*, to characterize vulnerable states:

Definition 2. *A vulnerability canary ψ is a predicate s.t.* $\forall s \in S_v \Leftrightarrow \psi(s) = \text{True}$.

Table 1: Toolset for synthesizer agent

API	Description	Output Example	Stats
Search			
`class(cls, [file])`	Search for class or structure `cls` in the codebase or `file`	`typedef struct png_XYZ { ... }`	13%
`method(m, [file])`	Search for method or function `m` in the codebase or `file`	`void png_read(png_ptr, ...) {...}`	17%
`symbol(s, [file])`	Search for symbol in the codebase or `file`	`TIFFDataType dtype = TIFF_BYTE;`	10%
`code(c, [file])`	Search for code snippet `c` in the codebase or `file`	`raw2tiff.c:302:memcpy(buf, ...)`	21%
Graph			
`callers(f)`	Get the caller names of function `f` in the codebase	[exif_iif_add_tag, exif_iif_add_fmt, ...]	13%
`callees(f)`	Get the callee names of function `f` in the codebase	`[php_strnlen, estrndup, php_ifd_double,...]`	6%
`references(s)`	Get all references of the symbol `s` in the codebase	`[misc.c:2670:if(sig_num!=SIGALRM){, ...]`	8%
Listing			
`files(dir)`	List all file names under the given path `dir`	`[caf.c, chunk.c, common.h, ...]`	11%
`classes(file)`	list all classes and structures defined in `file`	`[SF_INFO, sf_private_tag, SF_VIRTUAL_IO, ...]`	5%
`methods(file)`	List all methods and functions defined in `file`	`[exif_get_tag_ht, ifd_get32s, ptr_offset, ...]`	6%

The goal of directed fuzzing towards S_v is equivalent to finding inputs that satisfy the canary ψ. To provide semantically meaningful guides to directed fuzzing, we may instrument the program P with an additional predicate ϕ. Such instrumentation is admissible if and only if ϕ is a *relaxation* of the true vulnerability canary ψ:

Definition 3. *A predicate ϕ is the relaxation of canary ψ, if $\forall s \in S_v, \psi(s) = True \implies \phi(s) = True$.*

By definition, P' instrumented by Φ is fuzzing admissible if every predicate ϕ is a relaxation of ψ. This suggests that fuzzing P' is equivalent to fuzzing P while enjoying the additional guidance and early termination introduced by the instrumented predicates.

Theorem 1. *The instrumented program P' is fuzzing admissible to P, if P' is instrumented with Φ, where every $\phi \in \Phi$ is the relaxation of ψ.*

We next illustrate how our agentic synthesizer-validator workflow can produce an *admissible* instrumented program P' that provides *rich semantic feedback* to the directed fuzzer to reach vulnerable states S_v.

Algorithm 1 Scaffold of Locus's agentic workflow

Require: original program P, vulnerability canary ψ
Ensure: a target-conditional equivalent program P'

$C \leftarrow$ CanaryReasoning(P, ψ) ▷ list of reasonings
$\Phi \leftarrow \emptyset$
for all $c \in C$ **do**
  $l \leftarrow$ Localize(c, P) ▷ find the initial program point
  $n \leftarrow 0$
  repeat
    $\phi_l \leftarrow$ Generate(l, c, P)
    while $\neg$validate(ϕ_l, ψ, P) **do** ▷ syntax and semantic
      $\phi_l \leftarrow$ Generate(l, ϕ, P)
    end while
    $l \leftarrow$ Localize(ϕ, l, c, P) ▷ refine, find a better location
    $n \leftarrow n + 1$
  until $l =$ None $\vee$ $n >$ MaxIterations
  $\Phi \leftarrow \Phi \cup \{\phi_l\}$
end for
$P' \leftarrow$ Instrument(P, Φ) ▷ fuzzing admissible program

3.2 Agent Toolset

Existing agentic practices for software analyses, such as bug fixing or fault localization, are often equipped with lightweight command-line tools to perform local reasoning [71, 101, 111]. This is effective because the root cause of a bug is often spatially close to the observable failure, allowing the agent to retrieve relevant context from a narrow portion of the entire codebase. However, since our task formulation permits predicates to be synthesized at arbitrary program points, purely lexical tools cannot capture the necessary semantic relationships, and the agent therefore needs additional tools to understand program behaviors.

To equip the agent with such code reasoning capability at arbitrary program points, we provide a suite of tools that Locus can invoke to navigate the entire codebase and retrieve relevant code snippets. Particularly, in addition to common tools like code search and file listing, Locus integrates specialized tools for traversing program graphs, including call graphs and reference graphs. Through the call graph APIs, the synthesizer agent can retrieve function call relationships and reason the interprocedural control flow, while the reference graph API allows it to identify variable usages, pointer dereferences, and data access patterns across the codebase. Table 1 shows the complete toolset and the ratio of their actual usage in our experiments. Among them, graph traversal accounts for over a quarter of all API invocations.

It is important to note that these graphs are only provided as supplementary references to Locus, as they are all derived statically and may miss program behaviors such as dynamic dispatches and indirect calls. We observe that the LLM used in Locus is capable of leveraging its learned knowledge to bridge this gap. For example, in the case of vulnerability TIF002 (see Table 2), Locus successfully resolved the indirect function pointer `tif->tif_decoderow` to its concrete implementation `PixarLogDecode`.

3.3 Synthesis

Algorithm 1 elaborates on the workflow of Locus. It iteratively localizes candidate program points and generates the predicates (lines 4–13), then refines and validates them for both syntactic and semantic correctness (line 8). Once validated, the predicates are used to instrument the original program so it remains fuzzing-admissible (line 16).

As we demonstrate in Theorem 1, a predicate ϕ should be instrumented at the execution path that can reach the canary ψ. To synthesize ϕ, LLMs must reason about the root cause of the vulnerability and approximate potential execution traces of the program that can trigger it. The execution trace of a program is often long and complex, involving multiple functions and files. Therefore, a naive one-shot synthesis approach may not be sufficient, as the synthesizer only retrieves limited context and proposes primitive predicates, *e.g.,* an initial predicate generated by Locus (❷ in Figure 1c). To address this, Locus employs an iterative localization-generation refinement workflow. In each iteration, the synthesizer generates a predicate that preserves the same semantic meaning while moving its placement closer to the program entry.

Locus first synthesizes an initial set of predicates by analyzing the canary and approximates a list of semantic characteristics of the inputs that are likely related to this canary, *e.g.,* data structures, types, and properties. The synthesizer is then prompted to consider these constraints from multiple dimensions. For example, in Figure 1, besides the predicate, the synthesizer generates multiple constraints, such as requiring that the input PNG file must contain a valid signature. These approximations are progressively concretized and refined as the synthesizer retrieves more relevant code and reasons about the program execution.

For each approximated characteristic, the synthesizer needs to identify an appropriate program point l where the predicate can be expressed in terms of the variables and expressions in scope. However, constraining LLM to directly identify the exact program point is often too challenging and unreliable, so the initial stage only asks the synthesizer to select a candidate function rather than a precise program point. Given the reasoning context generated by the LLM so far, the synthesizer attempts to generate an initial predicate ϕ_l in the candidate function. Once ϕ_l passes the validation stage (Section 3.4), we refine it in the following iterations. The goal of such a refinement is to move the predicate to a closer program point to the program entry, while preserving the same semantic meaning as the original predicate, *e.g.,* checking for the same characteristics. This allows the program to reject invalid inputs sooner in the execution, enabling the fuzzer to explore more valid mutations within the same time budget.

It is worth noting that with the refinement iteration, a predicate can be all the way refined toward the fuzzing harness. In some cases, Locus can indeed effectively synthesize predicates at the program entry, making it similar in spirit to automated harness generation works such as HGFuzzer [97] and InputBlaster [58]. However, in most cases, the input constraints cannot be directly accessed at the harness level. For example, in Section 2.2, verifying the presence and content of a PLTE chunk requires parsing internal structures of the input that are only accessible deeper in the program. This necessitates placing predicates at intermediate program points.

3.4 Validation

The validation step is critical to preserving the correctness of the instrumented program and ensuring that the inserted predicate ϕ_l maintains the instrumented program's fuzzing admissibility. As shown in Algorithm 1 lines 7 to 9, the validator takes as input the candidate predicate ϕ, the vulnerability canary ψ, and the program P, and validates whether the predicate is both syntactically and semantically correct. If the predicate passes both checks, Locus will refine the predicate by exploring potentially better program points closer to the program entries (Section 3.3). If it fails, Locus will self-reflect and attempt to regenerate the predicate, using diagnostic feedback collected from the validator.

Syntax validation The first component of validation is syntactic checking. A predicate that fails to compile cannot be used in a fuzzing campaign, regardless of its intended semantics. To verify this, the program is instrumented by the predicate ϕ at the designated program point l and invokes the project's build system using a predefined command. If the build fails, the associated compiler error messages will be sent to the synthesizer to repair. This diagnostic information typically involves undeclared symbols, type mismatches, or malformed expressions.

Semantic validation The second component is semantic validation, which confirms that the predicate ϕ strictly relaxes ψ. It ensures that the predicate does not reject any execution paths that can reach the vulnerability (Theorem 1). As we demonstrated in Definition 3, such validation requires enumerating all possible program inputs, which cannot be done within an acceptable time budget. Therefore, we utilize symbolic execution to find counterexamples of the relaxation.

Theorem 2. *A predicate ϕ is not a relaxation of ϕ', if there exists $P \Downarrow_x s$, s.t. $\phi(s) = False \wedge \phi'(s) = True$*

Specifically, the predicate is not a strict relaxation of the canary if the symbolic execution can find a path that satisfies the negated predicate $\neg\phi$ while the canary ψ still evaluates to true.

It is natural to employ symbolic execution as a formal checker for Theorem 2 to ensure fuzzing admissibility. However, it often incurs substantial overhead by exploring irrelevant execution paths [8], *e.g.,* branches that are unrelated to either the synthesized predicate or the target canary. This excessive path exploration can lead to prohibitively long validation time, incurring additional overhead of deploying Locus. To mitigate this inefficiency, we adopt a strategy inspired by Chopper [87] to skip unrelated paths and target only the exploration of paths according to our selection. Specifically, we perform a lightweight reachability analysis on the CFG and prune nodes that are not reachable from either the predicate or the canary. Locus then initiates symbolic execution to explore the path between the $\neg\phi$ and ψ. Overall, the semantic validation offers a sound guarantee that the relaxation is harmless, but it remains unable to measure the effectiveness of the generated predicates.

4 Evaluation

We aim to answer the following research questions:

RQ1 Effectiveness: How effective is Locus in accelerating the generation of PoC inputs for given target vulnerabilities?

Table 2: TTE for each vulnerability in the Magma benchmark across different fuzzers. T.O. indicates that the fuzzer cannot find the vulnerability within 24 hours. ∅ indicates that the fuzzer either could not compile the target program or the preprocessing step exceeded 24 hours. Numbers in bold indicate statistical significance ($p \leq 0.05$).

Vul ID	AFLGo		SelectFuzz		Beacon		Titan		AFL++		AFL		MOPT		Fox	
	Origin	Locus	Origin	Locus	Origin	Locus	Origin	Locus	Origin	Locus	Origin	Locus	Origin	Locus	Origin	Locus
PNG003	4	4	3	3	5	5	8	8	12	**8**	3	3	4	4	5	5
PNG006	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	106	63	T.O.	T.O.	T.O.	T.O.	6978	**1582**
PNG007	72536	**9052**	58770	**8537**	7764	**1659**	17561	**11954**	53104	**41101**	55351	**18358**	42811	37522	4651	**4009**
SND001	T.O.	**419**	7764	**5**	432	**13**	115	**8**	451	**19**	75940	**338**	285	**12**	447	**27**
SND005	1015	**205**	102	**4**	95	**16**	30412	**1046**	1233	**193**	796	**9**	53	**9**	82	**15**
SND006	T.O.	**3899**	8519	**12**	T.O.	T.O.	26171	**905**	7026	**24**	T.O.	**2251**	328	**17**	15422	**717**
SND007	T.O.	**3468**	7785	**62**	T.O.	T.O.	310	**28**	810	**34**	T.O.	**3163**	360	**16**	3226	**129**
SND017	4453	**1021**	3235	**1518**	4861	**58**	40955	**256**	841	**196**	3162	**833**	33	**20**	2117	**178**
SND020	3805	**1291**	694	**542**	5683	**41**	70432	**604**	640	204	2916	**1266**	326	**103**	2068	**176**
SND024	T.O.	**2577**	5957	**18**	T.O.	T.O.	1074	**8**	361	**18**	T.O.	**2391**	326	**14**	2642	**32**
TIF002	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	72896	69963	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
TIF005	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	420	**260**	T.O.	T.O.	T.O.	T.O.	4744	**990**
TIF006	80998	**17774**	T.O.	T.O.	35493	**6715**	66692	**65412**	970	**336**	52926	**25366**	39810	45124	22380	**7607**
TIF007	18513	**2107**	1410	**413**	220	**76**	146	**109**	50	**26**	15022	**1172**	155	**56**	8868	**290**
TIF009	32556	**16767**	12067	**4144**	T.O.	T.O.	T.O.	T.O.	17483	**5234**	19302	**14098**	26049	15622	7696	**3406**
TIF012	47424	**18689**	9118	**4787**	18782	**12274**	2922	**2620**	1731	**1122**	28884	16318	3639	**953**	1257	**637**
TIF014	66000	**8134**	58851	**7833**	19035	**9672**	44384	**14360**	2555	**682**	66072	**7303**	1660	**788**	4666	**3275**
LUA004	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	69594	**67442**	20046	**11939**	44954	**22434**	12401	**2098**	35998	**6937**
XML001	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	3720	**1330**	T.O.	T.O.	37726	**16801**	T.O.	T.O.
XML003	T.O.	T.O.	T.O.	T.O.	44056	**13295**	53689	36365	2373	**1394**	T.O.	T.O.	56501	**18636**	12252	**6844**
XML009	T.O.	77513	T.O.	**56**	32585	**6748**	17942	**13879**	2668	**1452**	T.O.	22753	706	670	10827	**8766**
XML017	17	**8**	7	**9**	10	15	23	**13**	52	**15**	26	**7**	6	**16**	77	**35**
SSL001	T.O.	T.O.	72078	**62637**	∅	∅	∅	∅	23533	**1583**	81838	T.O.	43952	**16033**	66778	**17764**
SSL003	222	**231**	73	**69**	∅	∅	∅	∅	193	**178**	90	**63**	96	**79**	225	199
PHP004	165	**269**	11	11	∅	∅	∅	∅	66156	**34974**	82	**89**	760	**185**	∅	∅
PHP009	99	**203**	185	**153**	∅	∅	∅	∅	11890	**5677**	143	179	650	596	∅	∅
PHP011	∅	∅	26	**23**	∅	∅	∅	∅	1626	934	108	76	19	**13**	∅	∅
SQL018	49520	**30855**	53397	**6748**	T.O.	T.O.	78238	**42275**	10355	**4961**	71038	**32073**	22656	28101	15498	**8159**
Speedup	**17.0×**		**214.2×**		**22.1×**		**28.0×**		**15.3×**		**20.4×**		**5.5×**		**10.6×**	

RQ2 **Cost and performance**: What is the time cost and token cost for deploying Locus?

RQ3 **Ablations**: How do the individual components of Locus contribute to its overall performance?

RQ4 **Security impact**: How can Locus assist in real-world vulnerability detection scenarios?

4.1 Setup

Dataset We evaluate Locus on the Magma fuzzing benchmark [35], which includes a diverse set of real-world vulnerabilities selected from nine popular open-source software projects. We also consider popular libraries in the wild, *e.g.,* VLC, to evaluate the capabilities of Locus in finding real-world vulnerabilities (see Section 4.6). Each vulnerability in Magma is represented by a canary statement (Section 2.2), placed within the target function (Figure 1). The canary condition is manually curated to specify the state necessary to trigger the vulnerability and logs the time when it is satisfied. Note that while our evaluation assumes the canary is provided, Locus can be extended to other forms of targets, including vulnerability reports, patches, or static analysis results, by introducing an additional processing step (Section 4.6).

Baselines We select eight fuzzers (See Table 3) as our baseline, covering both the state-of-the-art directed and coverage-guided ones. For each fuzzer, we use the latest stable version available at the time of writing. Note that Locus alone is not a standalone fuzzer, but focuses exclusively on code transformations for the target software. Therefore, it is agnostic to fuzzer implementations and can complement any fuzzers (Figure 2).

Metrics We follow the existing directed fuzzing approaches by adopting the Time To Exposure (TTE) to measure the performance of the baseline fuzzers and the improvement introduced by integrating Locus. In the Magma benchmark, TTE measures the time taken by a fuzzer to find the input that satisfies the canary condition. To mitigate the inherent randomness introduced by fuzzing, we follow Hazimeh et al. [35] by executing 10 independent fuzzing trials per vulnerability sample and report the average TTE. We also employ the Mann-Whitney U Test [63] to demonstrate the statistical significance (p-value) of the results.

Table 3: Overview of selected fuzzers in the evaluation

Fuzzer	Category	Description
AFLGo [6]	directed	Distance-based seeds scheduling
SelectFuzz [60]	directed	Selective path exploration
Beacon [38]	directed	Fuzzer with efficient path pruning
Titan [39]	directed	Targets correlations inference
AFL [105]	coverage-guided	Evolutionary mutation strategies
AFL++ [28]	coverage-guided	Community-enhanced AFL
MOPT [61]	coverage-guided	Fuzzer with Swarm Optimization
Fox [79]	coverage-guided	Online stochastic control

Implementations We run all the fuzzers with the same initial seed inputs provided in Magma to ensure a fair comparison. Each fuzzing trial is capped at 24 hours, so those that fail to find the triggering input are recorded at this maximum duration. This potentially underestimates the actual TTE for baseline fuzzers, but it offers a lower-bound estimation. Therefore, the actual improvement brought by Locus can be even larger. To eliminate hardware and system-related discrepancies, all experiments are conducted on a dedicated cluster, where each server comes with an Intel Xeon Gold 6126 CPU and 128GB of RAM running Ubuntu 20.04.

We implement Locus using the PydanticAI framework [17]. The synthesizer's tooling is primarily built on top of Multiplier [31]. We leverage the SVF static analysis framework [84] to perform a lightweight reachability analysis and use KLEE [8] as the symbolic execution engine to perform semantic validation. We use `o3-mini-2025-01-31` as the default model with `reasoning level` set to `medium`, but we also evaluate other LLMs in Section 4.4.

4.2 RQ1: Vulnerabilities Reproduction

We apply Locus on both directed fuzzers and coverage-guided fuzzers, comparing the performance with and without Locus. The results are shown in Table 2.

In summary, Locus consistently improves all kinds of fuzzers for vulnerability reproduction. When integrating Locus to directed fuzzers, AFLGo [6], SelectFuzz [60], Beacon [38], and Titan [39] achieve 17.0×, 214.2×, 22.1×, and 28.0× faster TTE on average, with five more vulnerabilities found for AFLGo and one more vulnerability found for SelectFuzz than those without Locus within the fixed 24-hour time window.

For coverage-guided fuzzers, integrating Locus also yields substantial gains: AFL++ [28], AFL [105], MOPT [61], and FOX [79] achieve 15.3×, 20.4×, 5.5×, and 10.6× faster TTE on average, with four more vulnerabilities found for AFL than those without Locus within the 24-hour time window.

4.3 RQ2: Cost Analysis

We measure the time cost introduced by Locus and the token cost incurred by LLM inference.

Time cost In Section 4.2, we focus on measuring TTE, *i.e.,* the time required to trigger the target canary during fuzzing. However, evaluating only the fuzzing phase can be misleading when assessing the overall effectiveness of a directed fuzzer. Many directed fuzzers perform expensive static analysis on the target program before fuzzing begins [6, 39, 40]. Likewise, Locus requires additional preprocessing steps, including codebase indexing, symbolic validation of predicates, and agentic predicate synthesis. Although such preprocessing costs are often treated as a one-time effort amortized over fuzzing, we find that some approaches incur excessive analysis time, sometimes even longer than the time required to discover the bug by the fuzzer.

Table 4: Average deploy cost for Locus and directed fuzzers. All times are calculated in seconds. T.O. indicates that the fuzzer failed to instrument the target library within 24 hours.

Target	PNG	SND	TIF	LUA	XML	SSL	PHP	SQL
Size (LoC)	95k	83k	95k	21k	320k	630k	1.6M	387k
Index	11	34	82	9	76	146	244	137
Synthesis	373	331	212	178	215	384	412	349
Validation	261	133	231	280	475	824	353	407
Total	645	498	525	467	766	1354	1009	893
#Tokens (k)	309	303	256	176	653	598	894	467
AFLGo	122	673	2689	85	5608	24799	T.O.	15630
SelectFuzz	84	199	1167	44	807	2597	4554	383
Beacon	64	113	171	35	1656	T.O.	T.O.	3721
Titan	96	186	967	49	2936	T.O.	T.O.	4965

We report the detailed preprocessing time overhead incurred by Locus and baseline directed fuzzers. As shown in Table 4, the overhead of the baselines exponentially grows with the size of the codebase. In contrast, Locus relies only on lightweight analysis tools, *e.g.,* code retrieval, graph traversal, etc., so it remains efficient regardless of the project size. Particularly, Locus outperforms the best baseline fuzzer SelectFuzz by 4.5× when evaluated on the largest program in Magma (PHP).

LLM token cost We analyze the LLM token usage to assess the financial feasibility of deploying Locus. Table 4 shows the token cost for our Locus workflow. Locus takes 457k tokens (equivalent to $0.72 USD) to generate predicates for one sample in the Magma benchmark on average. The monetary token costs show the potential for Locus to act as an affordable step in assisting fuzzing.

4.4 RQ3: Ablations

We ablate the design choice of different modules in Locus and study how they generalize to various LLMs in Table 5. We pick five representative vulnerabilities that fall under different vulnerability categories in Common Weakness Enumerations (CWEs). The evaluation compares two representative mainstream fuzzing tools, AFL++ from the coverage-guided fuzzers and SelectFuzz from the directed fuzzers.

Effectiveness of each design component We ablate the individual design and measure the average TTE obtained by the resulting predicates. We begin with the *Base* setting, where we only keep the initial synthesized predicate (Algorithm 1 line 7). Next, we apply the refinement strategy, allowing the synthesized predicate

Table 5: TTEs by ablating different designs and models

	PNG007	SND001	TIF012	TIF014	SQL018
Ablate different designs					
AFL++					
Origin	53104	451	1731	2555	10355
+Base	54830	389	1904	1226	11573
+Refine	T.O.	23	4417	33931	54268
+Valid	**41101**	**19**	**1122**	**682**	**4961**
SelectFuzz					
Origin	58770	7764	9118	58851	53397
+Base	46207	2640	8938	12630	48059
+Refine	T.O.	6	10805	T.O.	T.O.
+Valid	**8537**	**5**	**4787**	**7833**	**6748**
Ablate different LLMs					
AFL++					
Origin	53104	451	1731	2555	10355
with `o3-mini`	41101	**19**	1122	**682**	**4961**
with `Deepseek R1`	45104	53	**992**	1439	5433
with `Gemini Flash 2.0`	**38215**	130	1517	2454	5274
SelectFuzz					
Origin	72078	7764	9118	58851	53397
with `o3-mini`	8537	**5**	4787	7833	6748
with `Deepseek R1`	23699	6	**3562**	**3518**	7023
with `Gemini Flash 2.0`	**7020**	12	7206	40332	**3921**

to propagate to a better program point without validating its semantic correctness. Finally, we evaluate the full pipeline of Locus including both the syntax and semantic validation.

Table 5 shows that the design choices in Locus consistently improve the performance. Specifically, we observe that while predicates generated under the base setting may occasionally accelerate fuzzing, they are not consistently beneficial. Without validation, Locus sometimes generates false predicates to the program, as evidenced by the cases PNG007, TIF014, and SQL018.

Varying models Besides the default `o3-mini` model, we consider `Deepseek R1` [86] and `Gemini 2.0 Flash` [33] to study the generality of our agentic framework to different LLM architectures. Table 5 shows that Locus generalizes to different LLM architectures, except for one case where Gemini fails to bring significant improvement to TIF014. We investigate this case and find that Gemini failed to elevate the generated predicate to the caller closer to the input, such that the additional overhead introduced by evaluating the predicates outweighs the benefit it brings to the fuzzing.

4.5 RQ4: Detecting New Vulnerabilities

We integrate Locus into a real-world vulnerability detection workflow and apply the pipeline to a set of well-fuzzed targets, such as `VLC` [89], `libming` [55], `libarchive` [54], and `tcpreplay` [88]. To construct meaningful fuzzing targets, we generate new canaries through two primary approaches. First, we leverage alerts produced by the static analysis tool SVF [84], which identify potentially vulnerable program points based on memory access patterns, aliasing behavior, or use-after-free risks. Second, we derive reachability canaries for program points associated with previously patched bugs. The rationale behind this strategy is that many real-world bugs occur in clusters or evolve from incomplete fixes [94].

Table 6: New vulnerabilities detected by fuzzers with Locus. For each newly found vulnerability, we run AFL++ and SelectFuzz with and without Locus.

Bug ID	Type	AFL++		SelectFuzz	
		origin	Locus	origin	Locus
VLC-29163	Memory leak	T.O.	30847	T.O.	26317
VLC-29162	OOB access	T.O.	74835	T.O.	T.O.
VLC-29238	Memory leak	T.O.	21085	53872	24766
VLC-29239	Use-after-free	43680	3946	22983	5405
libming-365	Null deref	T.O.	80241	T.O.	T.O.
libarchive-hvqg	Null deref	83622	34327	62748	18309
libarchive-fm54	OOB access	T.O.	16397	46577	23280
tcpreplay-pmgq	NULL deref	T.O.	14219	T.O.	T.O.
tcpreplay-pg55	OOB access	T.O.	31526	20392	21305

We launch 48 fuzzing samples in total, with each fuzzing campaign lasting for 24 hours. We found nine previously undiscovered bugs, including memory leaks, use-after-free, null pointer dereference, and out-of-bound memory access (Table 6). At the time of writing, we responsibly reported all the new vulnerabilities to the relevant maintainers, and three of the bugs already have drafted fixes. To further evaluate the efficiency improvement, we also repeat the fuzzing campaign by fuzzers without Locus. The results show that without Locus, AFL++ alone can only detect 2/9 vulnerabilities, and SelectFuzz alone can only detect 5/9 vulnerabilities. Section 4.6 elaborates on one newly detected vulnerability.

4.6 Case Study

Timeout cases Section 4.2 demonstrates that the synthesized predicates can occasionally yield negligible improvement. For example, the predicates associated with vulnerability `TIF009` significantly improve the performance of all evaluated fuzzers except for Beacon and Titan. On average, these predicates bring 2× speedup to all other fuzzers, while Beacon and Titan always timeout with or without Locus synthesized predicates.

Upon further investigation, we find that their employed termination mechanism can sometimes conflict with the canaries. Specifically, Beacon and Titan perform static analysis to identify functions potentially reachable from the vulnerability canary, and primarily prioritize program inputs that can reach the statically identified functions. The predicates synthesized by Locus thus offer only limited guidance when they are put in the functions that are not statically identified as close to the canary.

Canary generation by Locus Section 4.5 discusses our attempts to generate canaries for arbitrary target states without relying on existing ones. This shows that Locus need not rely on a pre-defined canary to be applicable. Instead, it can be extended to automatically generate canaries based on diverse representations of the target states, such as patch and bug descriptions.

To apply Locus in such settings, we also consider letting Locus generate canary conditions based on available security patches.

Fixing Patch

```
--- a/src/expr.c
+++ b/src/expr.c
@@ -73,7 +73,7 @@ Expr *AddCollateToken(
   if( pCollName->n>0 ){
-    Expr *pNew = sqlite3ExprAlloc(pParse,
-      TK_COLLATE, pCollName, 1);
+    Expr *pNew = sqlite3ExprAlloc(pParse,
+      TK_COLLATE, pCollName, dequote);
     if( pNew ){
```

Canary by LOCUS

```
if( pCollName->n > 1 &&
 (pCollName->z[0]=='"' ||
  pCollName->z[0]=='`' ||
  pCollName->z[0]=='[') &&
 !dequote ){ ... }
```

Ground Truth

```
if(!dequote ){...}
```

Figure 3: Locus generates a more precise canary using only the security patch. Previously, an incorrectly set dequoting flag enabled access to uninitialized memory.

Specifically, we iterate over all security patches in Magma and prompt Locus to automate the generation of a canary condition (Magma relies on manual analysis), such that the unpatched version satisfies the generated canary, while the patched version violates it. By manually checking all the canaries generated by the LLMs, we confirm that 27 out of 28 security patches can be correctly translated into the same condition.

We find the remaining case to be intriguing: the LLM produces a more precise vulnerability canary than the original one in the Magma benchmark, as shown in Figure 3). Specifically, this vulnerability is found in `SQLite3` and classified as CWE-908, *i.e.,* use of the uninitialized resource. When building a collate token, a constant value 1 is incorrectly passed in instead of the `dequote` flag. This allows attackers to access uninitialized memory via forced dequoting, which would lead to memory corruption. The original vulnerability canary only checks whether dequoting is disabled, and invalidates cases for unquoted names since they would not be dequoted even if the `dequote` flag were set. Instead, the canary generated by Locus targets mishandled quoted strings, precisely capturing the underlying vulnerability and potentially reducing false positives from imprecise ground-truth canaries.

Real-world vulnerability In this section, we elaborate on the vulnerability `libarchive-fm54` in Table 6. An out-of-bounds error can occur when the program processes a specially crafted RAR file. This vulnerability stems from insufficient bound checking in the delta filter logic used during decompression. In the RAR format, filters are lightweight data transformation routines applied to improve compression efficiency. One common example is the delta filter, which is often used on binary data, such as audio or image streams, to convert absolute values into relative differences, thus making patterns more compressible. During decompression, the filter operates on two memory regions: the source buffer `src`, which holds the raw decompressed data, and the destination buffer `dst`, which stores the filtered output. This vulnerability will be triggered when `src` inadvertently points into the `dst` buffer. This overlap can lead to undefined behavior, including memory corruption, out-of-bounds access, or program crashes, as the filter may read from or write to memory locations outside the valid range.

Figure 4 shows the predicates synthesized by Locus that check the input is a valid RAR format file and that the delta filter processing logic will be triggered during decompression. A predicate comparing the addresses of `src` and `dst` seems more helpful, but these pointers are set and computed inside the vulnerable function.

Check the RAR format

```
int code = format_code & FORMAT_BASE_MASK
if (code == FORMAT_RAR_V5 || code == FORMAT_RAR){
```

Check the delta filter

```
uint64_t fp= filter->prog->fingerprint;
if (fp == 0x1D0E06077D) { ...
```

Fixing Patch

```
diff --git a/read_format_rar.c b/read_format_rar.c
index 2dd0ea34..7f0ad199 100644
@@ -3708,6 +3708,8 @@ filter_delta(filter *f, ...
     uint8_t lastbyte = 0;
     for (idx = i; idx < length; idx += channels) {
+      if (src >= dst)
+        return 0;
       lastbyte = dst[idx] = lastbyte - *src++;
     }
```

Figure 4: A previously unknown vulnerability in `libarchive`

Table 7: TTE for applying Locus to other languages

Vulnerability	Language	Baseline	Locus
idna#108	Python	14	**5**
CVE-2020-35655	Python	T.O.	**36509**
hjson#17	Go	18354	**2298**
gojay#32	Go	T.O.	**74732**
CVE-2021-44228	Java	T.O.	T.O.
CVE-2021-37714	Java	T.O.	T.O.

By the time execution reaches it, the predicate is too late to guide the fuzzer effectively.

Apply Locus's design to other languages Locus analyzes program behaviors using LLMs, which have been shown to have strong performance on code analysis tasks across various programming languages [22, 34, 43]. In this section, we include preliminary results on extending Locus to broader programming languages. We replace the synthesizer agent with Claude Code [1] to ensure greater compatibility. Since KLEE offers limited support for non-C languages, we employ a self-reflection strategy that enables the agent to validate the semantic correctness of generated predicates.

Due to the limited availability of directed fuzzers for other languages, we apply Locus to three coverage-guided fuzzers in other ecosystems: Go-fuzz [90] for Go, Jazzer [29] for Java, and Atheris [32] for Python. For each language, we select two publicly reported vulnerabilities and use LLMs to generate canaries from the corresponding fixing patches (consistent with Section 4.6). We use Claude Sonnet 4.5 as the agent model.

The preliminary results (shown in Table 7) demonstrate that Locus is able to generalize across different languages. For Python and Go, Locus achieves an average TTE speedup of 2.6× and 4.5×, respectively. For the two Java vulnerabilities, while Locus does not trigger the target vulnerability in the given time budget, we observe that the vulnerable function is covered, whereas it is not even reached using Jazzer solely without Locus.

5 Related Work

Directed grey-box fuzzing Directed grey-box fuzzing is challenging, primarily due to the prohibitively large search space with sparse rewards [50, 60], *i.e.,* unlike coverage-guided fuzzing where any new coverage indicates progress. Most prior works develop heuristics through static analysis by computing distances to target states to narrow down the search space [6, 13, 39, 47, 50, 60, 78, 79, 83]. For example, AFLGo [6] uses distance metrics between test inputs and target basic blocks to prioritize seeds that are closer to the target, while CAFL [47] further improves this metric by introducing specialized progress-capturing state representation and calculating the distance from the testing inputs to the nearest state.

Similar to Locus, some recent approaches [3, 65, 72] also explore rewriting the program to direct execution towards hard-to-reach states. These approaches focus on checking sophisticated conditions uncovered by human experts, *e.g.,* temporal properties in network protocols, and may not generalize to broader types of target states and can also incur expensive manual effort. In contrast, Locus offers a generalizable framework for diverse types of programs and target states with minimal human intervention.

LLM-based fuzzing LLMs have demonstrated promising code understanding and analysis capabilities [23, 24, 30, 34, 91, 99]. There has been growing interest in applying them to support fuzzing [18]. Most existing approaches utilize LLMs to directly generate test inputs for the target program [19, 20, 37, 96, 98, 100, 109, 110] or build grammar-aware input generators (*i.e.,* fuzzing harness) to constrain the search space [40, 56, 57, 81, 107]. While these methods demonstrate the potential of LLMs to accelerate fuzzing, they often require LLMs to reason about the target program states *all the way* to the inputs, making them susceptible to fundamental challenges in program analysis, such as path explosions and alias dependencies [21, 42, 52, 74], while also pose challenges to LLMs in terms of the bloated context length and unbounded errors.

As opposed to solely generating inputs or fuzzing harnesses, Locus advances LLM-based fuzzing by constraining its generation at the level of progress-capturing predicates. Such a task formulation enables the LLM to operate within the (relatively) local context, allowing Locus to detect certain code behaviors that only emerge midway through execution to inform subsequent input searches, and also facilitates the rigorous validation of potential LLM errors. As a result, this strategy can effectively sidestep reasoning across lengthy function call chains, enabling a greater focus on more local, intermediate function contexts.

LLM-driven proof synthesis Locus shares a similar spirit with recent research that focuses on reasoning about loop invariants and program specifications to synthesize proofs and automate program verification [10, 14, 46, 59]. Locus resembles these approaches in the sense that they also integrate LLMs with rule-based verifiers, such as theorem provers or SMT solvers, to check the validity of synthesized properties. However, Locus relaxes the requirement that the synthesized specifications must facilitate rigorous proof steps, but treats the predicates as best-effort guidance for input search. It bridges the gap between formal verification and dynamic testing, leveraging the reasoning capabilities of LLMs to identify meaningful program states while prioritizing efficiency and generality over formal guarantees.

6 Threats to Validity

Threats to predicate validation While Locus leverages symbolic execution to bound the error of the synthesizer, it inherits all of its limitations. For example, we still face the path explosion problem, especially when there are loops between the generated predicates and the canary. While we adopt the common strategies to address these limitations, *e.g.,* loop unrolling, we cannot formally guarantee that the relaxation brought by the generated predicates is always valid and can thus effectively guide fuzzing. This leads to orthogonal but interesting future directions on loop invariant reasoning and function summary (using LLMs) for symbolic execution.

Threats to LLM types Second, our evaluation covers only a limited set of LLMs and benchmark programs. The generality of our results to other models, especially open-weight LLMs and broader software systems, remains to be validated. The performance of our approach may depend on the underlying LLM capability and training data. Additionally, our evaluated benchmark Magma, though built on popular software and real-world vulnerabilities, may not capture the comprehensive challenges present in diverse software.

Threats to data leakage While the task formulation, *i.e.,* predicate synthesis for directed fuzzing, is arguably hard to suffer from the data leakage problem, the software project in our evaluation is largely included in the training data of any major LLMs. As the ultimate goal is to detect and confirm vulnerabilities, we believe that such a threat is minor compared to the practical security impact, as we have shown in Section 4.6.

7 Conclusion

This paper presented Locus, a novel framework for enhancing directed fuzzing through synthetic progress-capturing predicates. With an agentic synthesizer-validator architecture, Locus effectively guides fuzzing via the predicates while also ensuring any errors in the synthesized predicates are fixed by symbolic execution. Our evaluation demonstrated that Locus substantially improves the state-of-the-art fuzzers. So far, it has uncovered nine previously unpatched vulnerabilities across three software projects, with three already acknowledged with draft fixes.

Acknowledgments

We are grateful to Jun Yang, Weichen Li, Chenghao Yang, Chenyuan Yang, Zheng Yu, Heqing Huang, Penghui Li, Weiteng Chen, and Miltos Allamanis for sharing their constructive and insightful feedback, which significantly helped improve this paper. This research is supported in part by Open Philanthropy, NSF CAREER Award CNS-2442719, and generous gifts from OpenAI. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. Language model assistants were used to help polish the paper.

References

[1] Anthropic. 2024. Claude Code. https://claude.com/product/claude-code

[2] Cornelius Aschermann, Sergej Schumilo, et al. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1597–1612.

[3] Jinsheng Ba, Marcel Böhme, et al. 2022. Stateful Greybox Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.

[4] Davide Balzarotti. 2021. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*.

[5] Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*.

[6] Marcel Böhme, Van-Thuan Pham, et al. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM Sigsac Conference on Computer and Communications Security*. 2329–2344.

[7] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*.

[8] Cristian Cadar, Daniel Dunbar, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). 209–224.

[9] Sicong Cao, Biao He, et al. 2023. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[10] Saikat Chakraborty, Shuvendu K Lahiri, et al. 2023. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342* (2023).

[11] Chuyang Chen, Brendan Dolan-Gavitt, and Zhiqiang Lin. 2025. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. *arXiv preprint arXiv:2506.10323* (2025).

[12] Hongxu Chen, Yinxing Xue, et al. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[13] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.

[14] Tianyu Chen, Shuai Lu, et al. 2024. Automated proof generation for rust code via self-evolution. *arXiv preprint arXiv:2410.15756* (2024).

[15] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[16] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*.

[17] Samuel Colvin. 2025. *PydanticAI*. https://ai.pydantic.dev/ Version 0.4.3.

[18] DARPA. 2024. DARPA AI Cyber Challenge. https://aicyberchallenge.com/

[19] Yinlin Deng, Chunqiu Steven Xia, et al. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM Sigsoft International Symposium on Software Testing and Analysis*. 423–435.

[20] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*.

[21] 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? *arXiv preprint arXiv:2403.18624* (2024).

[22] 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? *arXiv preprint arXiv:2403.18624* (2024).

[23] Yangruibo Ding, Jinjun Peng, et al. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems* 37 (2024), 60275–60308.

[24] Yangruibo Ding, Benjamin Steenhoek, et al. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[25] Zhengjie Du, Yuekang Li, et al. 2022. WindRanger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451.

[26] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2023. Formatfuzzer: Effective fuzzing of binary file formats. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–29.

[27] Andrea Fioraldi, Daniele Cono D'Elia, et al. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2829–2846.

[28] Andrea Fioraldi, Dominik Maier, et al. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th Usenix Workshop on Offensive Technologies (Woot 20)*.

[29] Code Intelligence GmbH and Contributors. 2020. *Jazzer*. https://github.com/CodeIntelligenceTesting/jazzer

[30] Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. *arXiv preprint arXiv:2403.04814* (2024).

[31] Peter Goodman. 2025. *Multiplier*. https://github.com/trailofbits/multiplier Version 1705339.

[32] Google and Contributors. 2020. *Atheris*. https://github.com/google/atheris

[33] Google DeepMind. 2024. Gemini 2.0 Flash. https://deepmind.google/technologies/gemini/flash/. Accessed: 2025-03-29.

[34] Alex Gu, Baptiste Rozière, et al. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *Proceedings of the 41st International Conference on Machine Learning*. 16568–16621.

[35] Ahmad Hazimeh, Adrian Herrera, et al. 2020. Magma: A Ground-Truth Fuzzing Benchmark. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Vol. 4. 1–29.

[36] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*.

[37] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).

[38] Heqing Huang et al. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. 36–50.

[39] Heqing Huang, Peisen Yao, et al. 2024. Titan : Efficient Multi-target Directed Greybox Fuzzing. In *2024 IEEE Symposium on Security and Privacy (Sp)*. 1849–1864.

[40] Heqing Huang, Anshunkang Zhou, et al. 2024. Everything Is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (Sp)*. 1956–1973.

[41] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*.

[42] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.

[43] Carlos E Jimenez, John Yang, et al. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.

[44] Tae Eun Kim, Jaeseung Choi, et al. 2023. DAFL: Directed Grey-Box Fuzzing Guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

[45] George Klees, Andrew Ruef, et al. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[46] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.

[47] Gwangmu Lee, Woochul Shim, et al. 2021. Constraint-Guided Directed Greybox Fuzzing. In *30th Usenix Security Symposium (Usenix Security 21)*. 3559–3576.

[48] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[49] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. The Hitchhiker's Guide to Program Analysis, Part II: Deep Thoughts by LLMs. *arXiv preprint arXiv:2504.11711* (2025).

[50] Penghui Li, Wei Meng, et al. 2024. SDFuzz: Target States Driven Directed Fuzzing. In *33rd Usenix Security Symposium (Usenix Security 24)*. 2441–2457.

[51] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*.

[52] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[53] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence directed hybrid fuzzing. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[54] libarchive contributors. 2025. *libarchive: Multi-format archive and compression library*. https://www.libarchive.org/

[55] libming contributors. 2025. *libming: SWF (Flash) file creation library*. https://www.libming.org/

[56] Dongge Liu, Oliver Chang, et al. 2024. OSS-fuzz-gen: Automated Fuzz Target Generation.

[57] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450* (2024).

[58] Zhe Liu et al. 2024. Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[59] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.

[60] Changhua Luo, Wei Meng, et al. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (Sp)*. 2693–2707.

[61] Chenyang Lyu, Shouling Ji, et al. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[62] Yunlong Lyu, Yuxuan Xie, et al. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3793–3807.

[63] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18, 1 (March 1947), 50–60.

[64] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*.

[65] Ruijie Meng, Zhen Dong, et al. 2022. Linear-Time Temporal Logic Guided Greybox Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1343–1355.

[66] Charalambos Mitropoulos et al. 2023. Syntax-aware mutation for testing the solidity compiler. In *European Symposium on Research in Computer Security*. Springer.

[67] Aniruddhan Murali, Noble Mathews, et al. 2024. Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.

[68] Manh-Dung Nguyen, Sébastien Bardin, et al. 2020. Binary-level directed fuzzing for {use-after-free} vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*.

[69] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[70] Jibesh Patra and Michael Pradel. 2016. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

[71] Paul Gauthier. 2024. Aider, AI pair programming in your terminal. https://aider.chat.

[72] Hui Peng, Yan Shoshitaishvili, et al. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (Sp)*. 697–710.

[73] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* (2020).

[74] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*.

[75] Niklas Risse, Jing Liu, et al. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. In *Proceedings of the ACM on Software Engineering*, Vol. 2. 388–410.

[76] Pranab Sahoo, Prabhash Meharia, et al. 2024. A comprehensive survey of hallucination in large language, image, video and audio foundation models. *arXiv preprint arXiv:2405.09589* (2024).

[77] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[78] Abhishek Shah, Dongdong She, et al. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2595–2609.

[79] Dongdong She, Adam Storek, et al. 2024. FOX: Coverage-guided Fuzzing as Online Stochastic Control. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 765–779.

[80] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-Generating C API Fuzzing Harnesses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.

[81] Wenxuan Shi, Yunhang Zhang, et al. 2024. Harnessing Large Language Models for Seed Generation in Greybox Fuzzing. *arXiv preprint arXiv:2411.18143* (2024).

[82] Prashast Srivastava, Stefan Nagy, et al. 2022. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 388–399.

[83] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. 2022. One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction. In *Proceedings of the 38th Annual Computer Security Applications Conference*.

[84] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. 265–266.

[85] Xin Tan, Yuan Zhang, et al. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1630–1644.

[86] {DeepSeek-AI}, Daya Guo, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs]

[87] David Trabish, Andrea Mattavelli, et al. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360.

[88] Aaron Turner and Contributors. 2001. *Tcpreplay*. https://tcpreplay.app/ Replay captured network traffic. Accessed: 2025-11-26.

[89] VideoLAN. 2025. *VLC media player*. https://www.videolan.org/vlc/

[90] Dmitry Vyukov. 2015. *go-fuzz*. https://github.com/dvyukov/go-fuzz

[91] Chengpeng Wang, Wuqi Zhang, et al. 2024. LLMDFA: Analyzing Dataflow in Code with Large Language Models. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. 131545–131574.

[92] Daimeng Wang, Zheng Zhang, et al. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 2741–2758.

[93] Junjie Wang, Yuhan Ma, et al. 2025. PatchFuzz: Patch Fuzzing for JavaScript Engines. *arXiv preprint arXiv:2505.00289* (2025).

[94] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

[95] Felix Weissberg, Jonas Möller, et al. 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*. 1539–1553.

[96] Chunqiu Steven Xia, Matteo Paltenghi, et al. 2024. Fuzz4ALL: Universal Fuzzing with Large Language Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1547–1559.

[97] Hanxiang Xu, Yanjie Zhao, et al. 2025. Directed Greybox Fuzzing via Large Language Model. arXiv:2505.03425 [cs]

[98] Chenyuan Yang, Yinlin Deng, et al. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. In *Object-Oriented Programming, Systems, Languages, and Applications*, Vol. 8. 709–735.

[99] Chenyuan Yang, Zijie Zhao, et al. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. arXiv:2503.09002 [cs]

[100] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563* (2023).

[101] John Yang, Carlos E. Jimenez, et al. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. 50528–50652.

[102] 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)*.

[103] Shunyu Yao et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

[104] Zijun Yao, Yantao Liu, et al. 2025. Are Reasoning Models More Prone to Hallucination? *arXiv preprint arXiv:2505.23646* (2025).

[105] Michal Zalewski. 2020. American Fuzzy Lop.

[106] Cen Zhang, Yuekang Li, et al. 2023. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation.. In *USENIX Security Symposium*.

[107] Cen Zhang, Yaowen Zheng, et al. 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*.

[108] Cen Zhang, Yaowen Zheng, et al. 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*. 1223–1235.

[109] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. Llamafuzz: Large language model enhanced greybox fuzzing. *arXiv preprint arXiv:2406.07714* (2024).

[110] Qiang Zhang et al. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 4238–4249.

[111] Yuntong Zhang, Haifeng Ruan, et al. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.