Scaling Security Testing by Addressing the Reachability Gap

Gaetano Sapia
Max Planck Institute for Security and Privacy (MPI-SP)
Germany
gaetano.sapia@mpi-sp.org

Marcel Böhme
Max Planck Institute for Security and Privacy (MPI-SP)
Germany
marcel.boehme@mpi-sp.org

Abstract

In order to scale automated security testing, we must first solve the reachability gap. Existing approaches to test specific features of a software system always assume some way of *interacting* with the system. For instance, the most popular approach, fuzzing, either assumes command line access, network access, an execution to amplify, or so-called fuzz drivers to send generated inputs to the system's process or its components. Yet, scaling security testing requires so much more than sending inputs. To test a specific feature, we might need to enable specific configuration options in specific files, to set up a specific runtime environment, to write some source code to exchange messages with the system over the network, or to issue system calls to the OS kernel (e.g., to test a device driver). We call the challenge of producing both the environment and input required to trigger specific internal functionality in a system as the reachability gap.

In this paper, we investigate the use of Large Language Model (LLM) agents to address the reachability gap in automated software testing. We introduce a novel end-to-end methodology that combines LLM-driven execution with invivo fuzzing, requiring only that the target software is installed and runnable—no manual harnesses or configuration. First, we evaluate whether an LLM agent can autonomously drive real-world programs into deep internal states. Then, we study the effectiveness of our full methodology: using invivo fuzzing to amplify executions produced by the agent. This approach results in increased code coverage and leads to the discovery of a previously unknown vulnerability in a widely used open source project.

CCS Concepts

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

Keywords

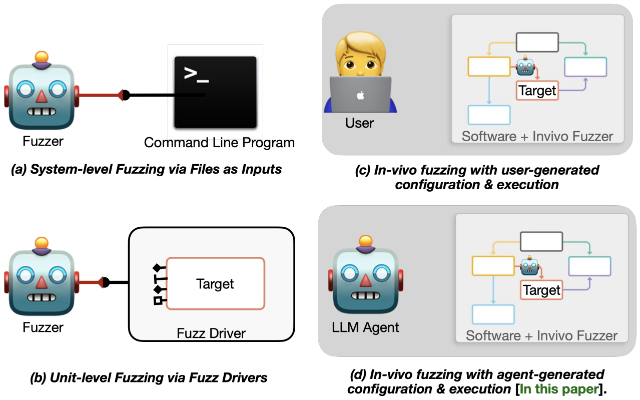
Security analysis, large language model, agent, agentic software engineering, AI4Sec, AI4SE, in-vivo fuzzing, reachability bottleneck

ACM Reference Format:
Gaetano Sapia and Marcel Böhme. 2026. Scaling Security Testing by Addressing the Reachability Gap. 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, 12 pages. https://doi.org/10.1145/3744916.3773116

.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773116



(a) System-level Fuzzing via Files as Inputs

(c) In-vivo fuzzing with user-generated configuration & execution

(b) Unit-level Fuzzing via Fuzz Drivers

(d) In-vivo fuzzing with agent-generated configuration & execution [In this paper].

Figure 1: Approaches to automated security testing.

1 Introduction

Fuzzing is one of the most successful techniques for discovering security vulnerabilities in complex software systems. By executing a program with randomly mutated inputs and monitoring for crashes or anomalies, fuzzers can uncover unexpected behaviors that might lead to exploitable bugs. Over the past decade, the fuzzing ecosystem has matured significantly, with techniques like coverage-guided fuzzing demonstrating their effectiveness in discovering deep bugs across a wide range of programs [20]. Building on this foundation, large-scale fuzzing infrastructures have emerged to continuously test software at scale. For example, Google's ClusterFuzz system, through its OSS-Fuzz instance [5], continuously tests hundreds of popular projects, and has reported tens of thousands of bugs.

Figure 1.(a-c) shows the three most prevalent means of security testing. The first fuzzer that popularized greybox fuzzing was AFL, a *system-level greybox fuzzer* for command line utilities (Fig. 1.a). The command line provides a standardized way of interacting with command line utilities via files or pipes (e.g., `stdin`). For instance, when testing a PNG-image processor, AFL would start with seed corpus of PNG-files, mutate those to generate new files, execute them, and add those that increase code coverage to the seed corpus.

Today, the most widely used approach is *unit-level fuzzing* (Fig. 1.b), where a target component or library is isolated from its host program(s) and connected to a fuzzer via a fuzz driver that exposes the component to the fuzzer, e.g., via a command line interface or a fuzzer-generic function. Writing a good fuzz driver often requires a solid understanding of the codebase in order to determine how the library project or software component is meant to be used, and what environmental conditions must be satisfied for the code to execute meaningfully. Usually, writing these fuzz drivers is manual and error-prone. To automate the manual effort, it is possible to generate fuzz drivers [12]. The existing automated methods typically assemble sequences of calls to the library API based on their observed usage in real-world programs [2, 12]. However, these approaches are primarily *syntactic*: they do not always infer meaningful interactions with the system and reports many false positives [8]. Moreover, while fuzz driver generation is designed to automate unit-level fuzzing, it does not work well for system-level fuzzing which requires first configuring and reproducing the broader execution context in which specific target features are triggered.

An emerging direction that sidesteps the need for fuzz drivers is *invivo fuzzing* [8] (Fig. 1.c) which amplifies actual, externally generated system executions at so-called amplifier points. Whenever an execution reaches a designated software feature to test, the current system state is forked and coverage-guided, function-level fuzzing is run on those shadow executions. Crucially, invivo fuzzing reuses a *contextually valid program state* that would be difficult to recreate manually through a traditional fuzz driver. Although originally applied to libraries in command-line applications, invivo fuzzing naturally extends to programs that process input through other channels, such as network services—where internal logic can be difficult to access due to the lack of clean APIs and reliance on external configuration and runtime conditions.

Yet, the invivo approach surfaces a key challenge: how do we obtain the system executions to amplify? To interact with a specific feature of a software, we might need to enable specific configuration options, to set up a specific runtime environment (e.g. creating a file that the program expects), and to set up a specific means to interact with the system. Moreover, different systems require different modes of interaction. For example, interacting with a kernel filesystem driver may involve mounting a volume and issuing system calls via a C program that needs to be written and compiled, while testing an HTTP server might require placing specific files in the document root folder and a testing client that sends and receives appropriate network requests.

We call *reachability gap* the challenge of generating an initial setup to interact with an arbitrary system and generating an execution to exercise a specific feature. Addressing the reachability gap requires a higher-level, semantic view of the program. That is, an understanding of how to configure the system to potentially enable the target feature and how to interact with the system as an end user or another system would.

Even in settings like the 2025 AIxCC competition [6], which is explicitly designed to advance fully automated vulnerability discovery, participants are provided with manually written fuzz drivers [7]. These drivers effectively bypass the key challenge of figuring out how to configure and interact with a system to reach interesting components in the first place. This design choice underscores a central limitation in current automated approaches: despite advances in automated software testing, the reachability gap remains difficult enough that it is manually abstracted away, even in competitions focused on automation. Ideally, for an automatic security testing tool to scale to the vast heterogeneity of software systems, the only assumption should be that the system is installed and runnable, and perhaps that the source code is available. Nothing more.

In this paper, we propose a methodology for fully automated, end-to-end security testing that integrates (i) a custom LLM-agent to setup the interaction with a given software system and to generate an execution to exercise a given feature in that system with (ii) an invivo fuzzer to amplify the LLM-generated execution once the feature is reached.

Our evaluation demonstrates the effectiveness of this approach. In the first experiment, we assess the agent's capability to reach a specific feature for five (5) real-world software projects, spanning user-space servers and kernel drivers. We compare two modes of interaction of the agent with the target software: full debugger access, which offers flexibility but limited guidance, and code coverage feedback, which constrains the agent's view but provides a clearer signal of progress. Across both modes, the agent is able to successfully trigger the target function in the majority of the cases (56%). In the second experiment, we assess the effectiveness of the end-to-end methodology for four (4) programs: for 20 target functions in each, we task the LLM agent with reaching them and, for the cases where the agent succeeds, apply invivo fuzzing from those runtime states. This results in increased code coverage on all programs with respect to the OSS-fuzz baseline, and lead to the discovery of a previously unknown vulnerability.

In summary, the contributions of this paper are the following:

- We characterize the **reachability gap** as a core obstacle preventing the fully automated security testing of an arbitrary software system, by highlighting the difficulty of triggering arbitrary points in real-world codebases.
- We investigate the use of a **Large Language Model (LLM) agent** to address this challenge, evaluating its ability to setup the necessary environment and to generate concrete executions that drive programs into target internal states.
- We propose a **novel, end-to-end methodology** that combines LLM-generated executions with invivo fuzzing. By allowing the LLM to generate interactions and then amplifying successful executions, we demonstrate a practical path toward scaling automated software testing. Our results show promising outcomes: the agent achieves target reachability in the majority of cases, coverage increases across tested programs, and our automated approach discovers a previously unknown vulnerability in a popular open source project.

Data availability. We publish our tool and data at: https://github.com/GPSapia/ReachabilityAgent_ICSE

2 Illustrating the Reachability Gap

In many cases, a target component to be tested is buried inside a larger application. The component is often not cleanly exposed through an API, at all. The program may not be structured as a library, and the only way to trigger the target code is through a chain of runtime interactions: configuring the software in a certain way, preparing auxiliary files, starting background services, and sending inputs over a network or IPC channel. In the general case (cf. Figure 1.d), it is not even clear how to begin testing, because the usual assumption of direct access to the target functionality no longer holds.

Our motivating example shown in Figure 2 may help to illustrate the complexity that needs to be solved to overcome this reachability challenge. Suppose, we want to test the Server Side Includes (SSI) module in `nginx`. The `nginx` ("engine x") server is an HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server. SSI is a feature that allows HTML files to include dynamic content by inserting special directives such as `<!--#include file="..."-->`. When a request for such a file is received, `nginx` parses the HTML, recognizes the directive, and executes the corresponding logic to include or evaluate the requested content.

How do we test the SSI feature in `nginx`*?*

Existing approaches. The `nginx` server does not facilitate testing the SSI feature via the command line and it is difficult to isolate the SSI component and write a fuzz driver for it. The SSI handling code is not exposed as a library or as a clean API. It is embedded within the larger `nginx` binary. Extracting the relevant code and wrapping it in a standalone fuzz driver would require recreating large parts of `nginx`'s internal execution context, including request parsing, configuration resolution, and file handling. This perfectly illustrates the reachability gap in practice: the target functionality is hidden behind environmental prerequisites that are difficult to discover and fulfill without human insight.

Configuring and Interacting with `nginx`. Reaching the code implementing the SSI feature in practice via the system-level interfaces provided by `nginx` requires several setup steps (cf. Fig. 2). First, SSI is not enabled by default. It must be explicitly turned on in the `nginx` configuration file. One must locate the configuration file in the corresponding folder, localize the specific line in which to add four very specific lines. Second, one must place a valid HTML file containing the correct SSI syntax in the correct server web root directory. Third, one must execute the correct command to launch `nginx` with the updated configuration and serving the target file path. Finally, one must issue an HTTP request to access that file and trigger the SSI parsing logic. While it is possible to use the `wget` utility to request that website, our LLM agent would write, compile, and execute a C file that would open a connection, construct a valid HTTP message, and issue the HTTP request to the correct IP.

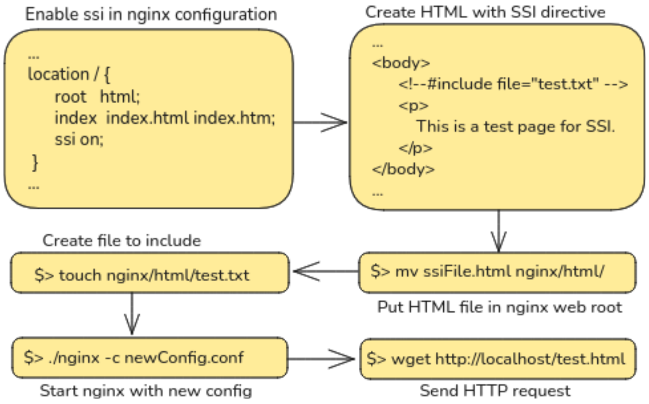
While these tasks might be more or less easy for a human who understands the goal, they are practically impossible to tackle for a fully automated system that lacks not only prior knowledge of the specific functionality to exercise, but also an understanding of the tested software and how to interact with it. In the remainder of the paper, we explore the use of LLM agents as a potential means to address the reachability gap.

3 Background

In this section, we review key concepts that underpin our work. We first provide an overview of the LLM-based software engineering agents, that our approach builds upon. Then, we describe the invivo fuzzing technique, which is the second component of our proposed automated software testing methodology.

3.1 LLM Agents

LLM agents are systems that place a large language model in a feedback-driven loop, allowing it to use tools, observe outcomes, and iteratively adapt its behavior. At each step, the model generates an action, which is executed in the environment. The result of the action is then fed back into the next model prompt, enabling multi-step interaction. Recent work has applied such agents to software engineering tasks, such as debugging, code navigation, and automated patching, and extended them to security domains like capture-the-flag (CTF) solving and vulnerability discovery through interactive system-level tools.


Enable ssi in nginx configuration
Create HTML with SSI directive
Create file to include
Put HTML file in nginx web root
Start nginx with new config
Send HTTP request

```
...
location / {
    root  html;
    index  index.html index.htm;
    ssi on;
}
...
```

```
...
<body>
    <!--#include file="test.txt" -->
    <p>
      This is a test page for SSI.
    </p>
</body>
...
```

```
$> mv ssiFile.html nginx/html/
```

```
$> touch nginx/html/test.txt
```

```
$> ./nginx -c newConfig.conf
```

```
$> wget http://localhost/test.html
```

Figure 2: Sequence of actions required to unlock SSI parsing code in nginx, starting from the only assumption that nginx is installed with the default configuration

A notable example is SWE-agent [26], which explores how to design better interfaces to enable LLMs to perform software engineering tasks. The authors introduce the notion of Agent-Computer Interface (ACI), treating LLMs as users that require structured, machine-facing interfaces to interact effectively with codebases. SWE-agent implements this concept by exposing a small set of commands for file navigation (find_file, search_file, search_dir), file viewing (open, goto, scroll_up, scroll_down), and editing (edit, create). The system is then evaluated on the SWE-bench benchmark [14], which consists of real software engineering tasks extracted from GitHub issues across 12 popular Python repositories.

Building on SWE-agent, EnIGMA [1] extends the agent framework to tackle cybersecurity tasks, by equipping it with access to interactive tools like gdb and remote server connections (via pwntools), enabling dynamic analysis and exploitation workflows. These tools are exposed through high-level commands that allow the agent to, for example, disassemble or decompile functions, start or manage debugging sessions, and interact with remote services. To address the verbose outputs typical of security tooling, EnIGMA also includes an automatic summarization layer that condenses overly long command results. The system is evaluated on a dataset of 390 CTF challenges, demonstrating that integrating domain-specific tools significantly enhances agent performance in security-relevant tasks.

While these agents have demonstrated relatively good performance on software engineering and CTF-style tasks, it remains unclear how well they generalize to the type of deep, system-level interaction required to overcome the reachability gap. In particular, they have not been evaluated in settings where the agent must autonomously configure, interact with, and trigger specific functionality in complex real-world software. This motivates our investigation.

3.2 Invivo Fuzzing

Invivo fuzzing [8] is a technique designed to enable fuzzing without the need for dedicated fuzz drivers. Rather than synthesizing inputs from scratch, it amplifies real program executions by fuzzing inputs passed to selected amplifier points (APs)—functions where user input is parsed.

APs must be specified by the user, either manually or through automated discovery techniques. In their prototype, the authors use CodeQL to identify likely parsing functions, but they note that APs can also be chosen based on expert knowledge of the codebase under test.

The invivo fuzzer embeds a forkserver into the program ahead of time and activates it whenever execution reaches a designated AP. From there, it mutates the input parameters passed to that function and spawns child processes via the forkserver to explore variant executions. This allows efficient in-place fuzzing from a fully initialized program state.

To reduce false positives and maintain valid program behavior, amplification constraints can also be associated with each AP. These constraints are logical preconditions on the AP's arguments that must be preserved while fuzzing (e.g. "the size of the input buffer `buf` must be less than or equal 10").

In the methodology described next, we propose to use invivo fuzzing to amplify executions generated by the LLM agent.

4 An LLM-based Approach to Addressing the Reachability Challenge

In this section, we present our end-to-end methodology for automated software testing, which combines LLM-driven execution discovery with invivo fuzzing to explore deep internal program behavior. The approach is designed to operate from minimal assumptions (i.e., the target software is installed and runnable and the source code is available), without requiring human-written harnesses or configuration.

We begin by outlining the overall methodology (Section 4.1), then describe the extensions we introduce to existing LLM agents to support this workflow (Section 4.2), as well as the criteria we use to select amplification point for invivo fuzzing (Section 4.3).

4.1 Automated Testing Methodology

As discussed above, the reachability bottleneck remains a critical and often overlooked obstacle towards automated software testing: without the ability to drive programs into meaningful internal states, no amount of fuzzing will surface deep or complex bugs.

At the same time, while Large Language Models (LLMs) have recently been proposed as tools for automated vulnerability discovery, early results suggest that a naïve "here's the code, find the bug" approach is ineffective [19, 21, 22]. Vulnerability discovery is not only technically demanding, but also open-ended in a way that often overwhelms even strong language models.

Instead, we hypothesize that the reachability problem presents a more tractable and productive application for LLMs. It occupies a middle ground: on one hand, it requires a higher-level understanding of program behavior, including how to manipulate inputs, configurations, and system state. On the other, it is constrained and concrete, like triggering a specific function, making it a more bounded and achievable task. If an agent reaches the desired point in the code, it can then hand off to a fuzzing engine to explore behavior from that point forward. The methodology we propose, described in Algorithm 1, is built on this hypothesis.

Algorithm 1 End-to-End Testing Methodology

```
Require: Runnable program $P$; target functions $\mathcal{F}$
 1: $globalCovFuncs \leftarrow \emptyset$
 2: for $f \in \mathcal{F}$ do
 3:     if $f \in globalCovFuncs$ then
 4:         continue
 5:     end if
 6:     $a \leftarrow \emptyset$
 7:     $coveredFuncs \leftarrow \emptyset$
 8:     $stackTrace \leftarrow$ callGraphAnalysis($f$)
 9:     $agent \leftarrow$ LLMAgent(goal=$f$, trace=$stackTrace$, budget=\$10)
10:     while $f \notin coveredFuncs$ and budget not exhausted do
11:         $a_t \leftarrow$ agent.nextAction
12:         $a \leftarrow a \cup \{a_t\}$
13:         if interactWithTarget($a_t$) then
14:             $coveredFuncs \leftarrow \emptyset$
15:             Execute $a_t$ in $P$'s environment
16:             $coveredFuncs \leftarrow$ GetNewlyCoveredFuncs($P$)
17:         else
18:             Execute $a_t$ in $P$'s environment
19:         end if
20:     end while
21:     if $f \in coveredFuncs$ then
22:         $globalCovFuncs \leftarrow globalCovFuncs \cup coveredFuncs$
23:         $trace \leftarrow$ ReproduceExecutionTrajectory($P$, $a_{1:t}$)
24:         $ampPoint \leftarrow$ IdentifyInputSyscall($trace$)
25:         InVivoFuzz($P$, $ampPoint$)
26:     end if
27: end for
```

Algorithm 1 requires as input a program P and a set $\mathcal{F}$ of target functions in P that we want to trigger at runtime. While many criteria can be used to do so, in our experiments we select functions based on their complexity: the intuition is that the more complex functions are more likely to contain bugs [16], while the functions with the highest cumulative complexity (i.e. the accumulated complexity of all the functions that can be reached starting from the target function) allow to cover a larger area of code with one execution [11].

For each function, we statically compute the stack traces leading to it: this is done with an LLVM pass at compile time. This information, plus the target function and the path of the target codebase on the machine, are given to the LLM in the initial prompt (Lines 8–9). The possible actions the agent can perform at each step are described in Section 4.2.

If the agent action is a direct interaction with the target program (Lines 13–16), we save the list of functions that were executed. Notice that, at Line 14, before actually executing the agent action in the environment, we clean the coverage information collected so far. The idea here is that we want to show the agent only the effects of the last interaction, in order to avoid overwhelming it with every function that has been executed since the beginning of the trajectory. The agent execution continues until either the target was reached, or the LLM API budget of $10 was exhausted (Lines 10–20).

If the function was reached, we reproduce the trajectory, identify the amplification points for invivo fuzzing (using the criteria described in Section 4.3), and start the fuzzing campaign.

Finally, we keep track of the functions that the agent was able to reach across all successful trajectories (Line 1), so that, in case one trajectory reached more than one target function (e.g. due to overlapping execution flows), we can consider all of them as solved.

4.2 Agent Design

Our agent is built on top of SWE-agent, described in Section 3.1, which we adapt for our software testing use case by introducing extensions described below. First, the agent operates in a fully isolated environment: a QEMU virtual machine that hosts the target program, previously compiled with coverage instrumentation enabled. This setting provides a realistic and flexible testbed, enabling interaction with userspace programs, services, and kernel-level components. The agent's task is to write a C program that, when compiled and executed in the guest machine, drives the target program into a state where a designated function is triggered.

Table 1: ACI commands added on top of SWE-agent

Category	Command	Documentation
Coverage	`getCoverageByFunc`	coverage data for one function, with comparison operators
	`getWholeCoverage`	code coverage from latest interaction
Code Browser	`search_func_def`	return code location where a function is defined
	`search_struct_def`	return code location where a struct is defined
	`search_macro_def`	return code location where a macro is defined
Editing	`ins_lines_in_poc`	Inserts code into C PoC file.
	`ins_lines_in_non_poc`	Inserts code into non-PoC file
	`del_lines_from_poc`	Deletes lines from PoC file.
	`del_lines_from_non_poc`	Deletes lines from non-PoC file.
QEMU	`push_non_poc`	Pushes non-PoC files to QEMU target.
	`compile_PoC`	Compiles PoC code for execution.
	`execOnTargetMachine`	Executes PoC on QEMU-based target.

4.2.1 *Code Browsing Support.*

We equip the agent with a code browsing capability via a custom libTooling pass that statically analyzes the target codebase at build time. This pass generates an index of relevant source-level constructs—function definitions, structure definitions, and macro definitions—which the agent can query during execution using the commands search_function_definition, search_structure_definition, and search_macro_definition.

4.2.2 *Execution Modes.*

The agent can be launched both in coverage mode and in gdb mode. In *coverage mode*, after each attempt at interacting with the target, the agent receives a list of the functions that were executed. If the agent wants to inspect a specific function more closely, it can request fine-grained coverage information, which includes the values of comparison operators in branch condition. The commands available to the agent to obtain coverage information are `getWholeCoverage` and `getCoverageByFunc`.

In *GDB mode*, the agent has access to the debugger and is free to monitor, explore, and interfere with an ongoing program execution however it chooses. This setup removes structured feedback and instead offers a lower-level, open-ended interface to the running system. For this functionality, we extend the existing implementation of EnIGMA to support asynchronous breakpoint usage. In contrast to EnIGMA which, to the best of our knowledge, only allowed breakpoints to be set before program execution began, our interface allows the agent to interrupt a running binary, insert or remove breakpoints, and resume execution. This supports more realistic workflows, such as monitoring the behavior of long-running processes or reacting to runtime conditions discovered during execution.

4.2.3 *Improved Interfaces.*

Finally, we introduced several improvements to the agent-computer interface that address limitations we observed during our experiments. First, we refined the code editing interface by splitting the original `edit_file` command into two separate actions: insert new content and delete previous contents. This change significantly reduced the chance of incorrect edits and aligns with findings in prior works that a carefully designed ACI is critical to improving LLM performance. Second, we relaxed the viewing constraints for the agent-generated C program that is used to interact with the target software. Unlike standard file visualization—by default limited to 50 lines—files authored by the agent are returned in full. This decision was motivated by the observation that partial views often led to subtle mistakes such as mismatched braces or incomplete function definitions, resulting in frequent compilation failures.

4.3 Execution Amplification

Once the agent succeeds in reaching a target function, we apply invivo fuzzing to amplify the resulting execution. This allows us to explore the local behavioral space around the discovered state by mutating input in-place, under realistic program conditions.

In the original invivo fuzzing framework, amplifier points—the locations where fuzzing begins—are typically selected by the developer. For example, one might choose a parser entry point or a function known to process external input, which is a reasonable approach in settings where expert knowledge is available.

But what if we want to automate this step as much as possible?

The invivo paper itself proposes a heuristic solution: select functions whose names contain strings like "parse", assuming they are likely to process user input. While effective in controlled scenarios, this method is clearly brittle and not generalizable across arbitrary targets.

Our key observation is that, regardless of naming conventions or code structure, all external input must ultimately enter the program through system calls. Whether data arrives from a file, socket, or device, it is funneled into the application via standard interfaces such as read(), recv(), or similar. This makes system calls a natural choice for amplifier points when applying invivo fuzzing.

In practice, a single interaction initiated by the agent can trigger multiple system calls, many of which may not directly process the input relevant to the functionality under test. For instance, in response to receiving a network packet, a program might also read configuration files or perform unrelated I/O as part of its normal operation. While all of these system calls handle data originating outside the program, not all of them correspond to the specific input that the agent was attempting to provide in order to reach a target function.

One possible strategy would be to selectively amplify all system calls handling external input. Provided that invivo fuzzing constraints on the input are chosen correctly, any discovered vulnerability would still reflect a valid issue, since the input originated from outside the program and flowed through a real execution path. However, to maintain focus and interpretability in our experiments, we manually identify the system call most semantically tied to the agent's task—for example, the recv() that processes the protocol packet intended to exercise a particular code path. To do so, we set breakpoints to input-receiving syscalls, (e.g, read(), recv()) and run the agent-generated execution. If any of the triggered syscall receives agent-generated input, it's chosen as amplifier point. We also define the corresponding amplifier constraints, which are typically straightforward from the syscall signature (e.g., in read(fd, buffer, len), buffer's size must be smaller than len).

5 Experiments: LLM Agent Effectiveness on Reachability Bottleneck

Since LLM agents have proven to be powerful tools for software engineering tasks that are difficult to describe systematically or algorithmically [27], we first evaluate the capability of our agent in reaching certain target features in software systems that span from Linux kernel drivers to network servers. Specifically, we seek to answer the following research questions:

- **RQ.1** *(Effectiveness).* Can the LLM agent autonomously reach designated target functions in complex software systems?
- **RQ.2** *(Directed Feedback vs. Free Exploration).* What is the impact of giving the agent more freedom but less structured information (via an interactive debugger) versus a constrained set of actions while providing rich feedback (via code coverage information), on its ability to reach target functionality?

5.1 Experimental Setup

Selecting target systems. To evaluate the agent's ability to autonomously reach specific functionality across diverse software systems, we designed a benchmark consisting of five real-world targets. Our objective was to select these software systems to reflect the heterogeneity that makes automated software security testing particularly challenging. These include:

- Nginx, an HTTP server
- Janus, a general-purpose WebRTC server
- Btrfs, a Linux kernel filesystem driver
- Dnsmasq, a lightweight DNS and DHCP server
- ProFTPD, an FTP server

This selection spans user-space applications, kernel-space drivers, networked daemons, and multimedia backends, emphasizing differences not only in interface type (e.g., HTTP, system call, socket-based), but also in the configuration and environment required to meaningfully interact with them.

Selecting target functionalities. For each target, we selected five functions as goals for the agent to reach. In the end-to-end testing methodology described in the previous section, we select target functions based on their cumulative complexity. While this metric enables fuzzer access to a broader code region, it also implies that the chosen functions, being high in the call stack, might be easier for the agent to reach. However, in RQ1/RQ2, our focus is to better stress-test the agent reachability capabilities: therefore, we opted for a metric that did not rule out functions that are potentially harder to reach. To ensure a fair and principled selection, we chose the five (5) most complex functions, as measured by cyclomatic complexity. This is based on the assumption that more complex code is more likely to contain bugs [16]; hence, reaching such functions from a clean system state represents a meaningful and realistic testing goal. As a practical constraint, we limited the selection to functions reachable through a static call stack no deeper than 10 frames, to ensure that the task remained feasible within a single run of the agent—each capped to a cost of $10. Finally, a task is considered successful if, by the end of its interaction loop, the agent triggered the execution of the specific function designated as the goal—either in coverage mode or GDB mode.

Coverage and debugger feedback. In order to provide our LLM Agent with information about the progress towards reaching a target function, in the default *coverage mode*, we use the `clang` coverage instrumentation option (`-fsanitize-coverage=trace-pc-guard,trace-cmp`) and `llvm-cov` coverage profiler. In *GDB mode*, we provide access to `gdb (12.1)` as interactive debugger. We evaluate both versions in RQ-2.

Experiment infrastructure. All experiments were conducted using OpenAI GPT-4 language model [17] with a temperature $T = 0$, chosen to reduce - though not entirely remove [18] - the influence of stochasticity. The agent was executed inside a Docker container equipped with a set of pre-installed tools, including `gdb (12.1)`. It interacted with a virtualized target environment via command execution on a QEMU virtual machine [3] running Linux kernel version 6.13.0-rc7, emulated using QEMU version 8.2.2. All experiments were run on a local workstation equipped with 128 cores and 252 GB of RAM.

5.2 RQ-1. Effectiveness of LLM Agents in Reaching Target Functions

Figure 3 provides an overview of the agent's performance for all 25 evaluation task while Table 2 presents a per-task evaluation of agent effectiveness. For each (target, function) pair, the table indicates whether the agent successfully reached the target in each mode, along with the number of calls to LLM API and the corresponding cost.

Results. In 14 out of the 25 tasks evaluated across the five target systems, the agent was able to successfully reach the target functionality. In 10 out of the 14 successful tasks, the agent had to perform actions beyond mere input generation—such as editing configuration files, preparing runtime environments, or launching dependent services—underscoring that triggering internal functionality often requires navigating complex system-level setup steps. We report further results for RQ-1 in Section 6.2.

Our agent was able to reach the target functionality in the majority of cases (56%). Out of the successful tasks, again the large majority (71%) required actions beyond input generation, such as changing configuration or creating files.

As we can see in Figure 3, while the agent performs particularly well in reaching the targets for `nginx`, `dsnmasq`, and `proftpd`, it struggles to perform well for `janus` and `btrfs`. By analyzing the corresponding trajectories, we believe that failures in Janus were mainly due to the interaction modality. The server exposes an HTTP-based interface, but the agent was instructed to write a C program to trigger the target functionality. We observed that in many of the resulting trajectories, the agent spent a disproportionate amount of time iteratively refining a low-level HTTP client in C, often struggling with implementation details such as parsing the response received by the server in order to prepare the next request. For the 4 failures in the Linux kernel driver task, we believe the challenge is largely due to the inherent complexity of the target code.

The agent succeeded on user-space targets like nginx and dnsmasq, but struggled with janus and btrfs—primarily due to suboptimal interaction modality in the former and the inherent complexity of kernel-level code in the latter.

Reasoning loops. In some tasks, the agent entered what we refer to as *reasoning loops*: repeated iterations of nearly identical reasoning and code generation, often reusing the same actions and explanations verbatim. These loops indicate a failure to integrate negative feedback from the environment and a lack of exploration of alternative strategies. Once in a loop, the agent tended to exhaust its interaction budget without making meaningful progress toward the task objective. This highlights a key limitation in current agent designs: the inability to reflect on failure and revise plans in a structured manner.

When the first strategies fail, the LLM agent often lacks the ability to step back and seek alternative paths.

5.3 RQ-2. Coverage Feedback or Debugger

Effectiveness. Table 2 presents a per-task comparison between the coverage and gdb execution modes. With access to an interactive debugger, our agent can successfully reach about the same number of target functions compared to when given access only to coverage feedback. However, in three cases it does not recognize the target function as reached in gdb mode, because it misses the corresponding coverage feedback. Together they reach four functions more than individually (14 together vs 10 each).

Qualitatively, we observe that the agent's use of gdb is generally superficial—typically limited to placing breakpoints on functions and checking whether they are hit. Notably, the agent never uses gdb to inspect variable values or reason about control-flow decisions. Both of these type of insights (executed functions and variable values in comparison operations) are provided by coverage feedback, with much less overhead for the agent. As it seems, while access to an interactive debugger provides the agent with lower-level control over the target program, it also introduces greater freedom and complexity in the decision space. As the results show, this additional capability does not consistently lead to better outcomes.

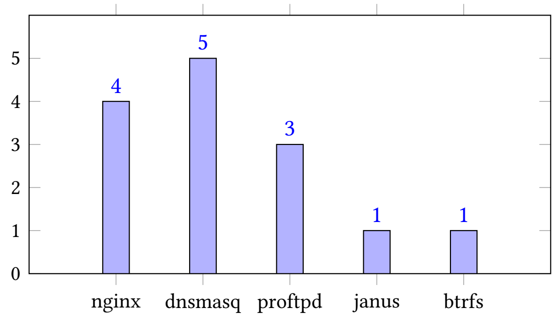


Figure 3: Number of successfully solved tasks per target.

Overhead. In the great majority of the cases, running the agent in gdb mode leads to a higher number of LLM actions than those needed in coverage mode. Specifically, gdb mode needs in average 19 more steps than the ones in the corresponding task executed in coverage mode. This reflects the additional overhead introduced by gdb: the agent must proactively issue commands to extract information, whereas in coverage mode, feedback is automatically provided after each interaction—reducing the need for exploratory actions.

Coverage feedback not only substitutes for—but often improves upon—the information gained through debugger interaction.

6 End-to-End: Integration with Invivo Fuzzing

We now evaluate the effectiveness of our end-to-end methodology, which combines our LLM agent with invivo fuzzing. The goal of this evaluation is to assess whether executions generated by the agent can serve as viable starting points for invivo fuzzing, and whether amplifying these executions leads to improved code coverage and bug discovery. To this end, we apply our approach to four real-world software targets and measure both the coverage gains achieved through amplification and the ability to uncover new, previously unknown vulnerabilities. Therefore, here we want to answer the following research question:

- **RQ3.** Can LLM-generated executions, when combined with invivo fuzzing, increase coverage and discover previously unknown vulnerabilities in real-world software?

6.1 Experimental Setup

To evaluate our full methodology, we apply it to four real-world open-source programs from OSS-Fuzz:

- nginx (HTTP server)
- dnsmasq (DNS/DHCP server)
- lighttpd (HTTP server)
- mosquitto (MQTT broker [publish/subscribe messaging])

Some of the programs used for evaluation of RQ1/RQ2 could not be used to test the full methodology, mainly due to limitations in the current implementation of invivo fuzzing. In particular:

- **btrfs**, a kernel driver, excluded here because afllive is currently not implemented in kernel mode. Nonetheless, we kept btrfs in RQ1/RQ2 to test the agent also on a kernel component.
- **proftpd**, a FTP server. It creates a chroot for incoming connections, hindering the communication between fork server and fuzzer. While this can be manually solved [15], we believe that a more general solution at a fuzzer engineering level would be more interesting to study (but out of scope for this work).
- **janus**, a WebRTC server, excluded from RQ3 because experimental results for RQ1/RQ2 already showed that the agent performed poorly on it.

Programs and targets. For each program, we select the 20 functions with the highest cumulative complexity. We chose a large number of functions (20) to cover the functionality of the system broadly. The *cumulative complexity* of a function f is computed as the sum of the cyclomatic complexity of f and all functions transitively called by f. This metric is explicitly recommended by OSS-Fuzz as a way to identify promising targets for fuzz harnesses, under the intuition that exercising these functions is likely to transitively cover a significant portion of program logic [11]. The agent is tasked with reaching these functions autonomously. Whenever a function is successfully triggered, we replay the execution and start an invivo fuzzing campaign of 2 hours: we chose this duration to be long enough for fuzzing to begin demonstrating its effectiveness, and short enough to allow us to amplify all selected amplifier points. The amplification points are selected as explained in Section 4.3. As in the rest of our methodology, the only assumption at the beginning of the experiments is that the target software is installed on the system with the default configuration, and instrumented for coverage collection.

6.2 Addressing the Reachability Challenge

Table 3 [Target Functions (reached/total)] augments the results for RQ-1. As we can see, on average three quarters (75%) of the target functions can be reached. In Table 4 we list, for each target program, a set of representative functionalities that were exercised by the LLM agent, along with the setup actions required to reach them (e.g., enabling modules, preparing configuration files, launching dependent services). These functionalities are not exhaustive but roughly correspond to the features associated with the 20 target functions selected for amplification, and are intended to help the reader contextualize the scope and complexity of the agent's interactions.

The LLM agent was able to reach a significant fraction of the selected functions in all targets. These functionalities span a range of complex behaviors, such as QUIC support in nginx, DHCPv6 in dnsmasq, and MQTT subscription handling in mosquitto. Importantly, they required substantial configuration effort that would traditionally be handled manually.

6.3 Effectiveness: Integrating Invivo Fuzzing

The results are shown in Table 3. In three of four cases, our LLM agent together with invivo fuzzing achieves significantly more coverage than the manually generated fuzz drivers available in OSS-Fuzz. In two cases, the increase in coverage is by an order of magnitude. Only for `mosquitto`, which has 23 manually written fuzz drivers, our agent achieves a lower coverage (7.9k vs 4.5k).

This supports our central intuition: while fuzzing is highly effective at exploring complex logic once a rich program state is reached, the LLM agent plays a crucial complementary role by unlocking those states—interacting with the system and satisfying preconditions that traditional fuzzers alone fail to handle.

The amplified executions consistently achieve substantially higher code coverage than the highly curated, manually written set of fuzz drivers in OSS-fuzz.

The contribution of invivo fuzzing over and above the LLM-generated execution is demonstrated by the number of coverage-increasing inputs added to the queue of the invivo fuzzer [Table 3 (#Cov.-incr. Inputs)]. For `nginx` and `dnsmasq`, we see several thousand coverage increasing inputs added to the queue (1.6k and 1.5k). Even for `mosquitto`, 348 coverage-increasing inputs are discovered. We note that an invivo-generate execution continues even long after the amplified function returns. Our results validate the usefulness of our amplification point identification heuristic, and demonstrates that fuzzing continues to drive exploration well beyond the agent's initial trajectory.

Finally, the invivo amplification phase successfully uncovered a previously unknown vulnerability in dnsmasq. The out of bounds read (OOB; information disclosure) was confirmed, reported, and patched by the maintainers, and CVE number CVE-2025-54318 was assigned to it.

Our methodology uncovered a previously unknown, real-world vulnerability in dnsmasq, demonstrating its practical effectiveness.

7 Related Work

LLMs for Vulnerability Discovery. Recent work has explored the use of LLM-based agents for vulnerability discovery across a variety of settings. Project Naptime [10] equips an LLM with a rich interface—including a code browser, debugger, and the ability to run Python scripts—to interact with a target codebase and search for memory corruption vulnerabilities. In contrast, BigSleep [9] focuses on variant analysis: the agent is given a natural language description of a previously discovered vulnerability and tasked with finding semantically similar issues in the same codebase. Notably, this approach led to the discovery of a new stack buffer underflow in SQLite. Another example is Fuzz4All [23], which uses LLMs to fuzz programs that take structured languages as input, such as compilers.

Other works found that LLMs have a limited capability to reason about security-related bugs [21]. For this reason, in our work we propose using LLMs for a more constrained, yet impactful task: driving the program into specific internal states where traditional fuzzing can take over.

Table 2: Comparison of effectiveness and number of LLM calls in coverage versus GDB modes for userspace targets, with corresponding API cost. The dagger (✓†) shows where the agent is successful but does not stop the campaign.

Target	Function	Solved (Cov)	Solved (GDB)	LLM calls (Cov)	LLM calls (GDB)
nginx	ngx_http_parse_request_line	✓	✓	**11** — ($0.52)	**14** – ($0.84)
nginx	ngx_http_ssi_parse	✓	✓	**24** – ($1.63)	**26** – ($1.94)
nginx	ngx_ssl_connection_error	✓	×	**26** – ($1.81)	**88** – ($10.00)
nginx	ngx_resolver_process_a	×	×	**78** – ($10.00)	**81** – ($10.00)
nginx	ngx_http_write_filter	✓	✓	**12** – ($0.66)	**13** – ($0.80)
dnsmasq	dhcp_reply	✓	×	**24** – ($1.76)	**78** – ($10.00)
dnsmasq	answer_auth	×	✓	**90** – ($10.00)	**78** – ($9.37)
dnsmasq	answer_request	✓	✓	**21** – ($1.49)	**26** – ($2.09)
dnsmasq	dhcp6_no_relay	×	✓	**84** – ($10.00)	**77** – ($8.22)
dnsmasq	one_opt	✓	✓†	**19** – ($1.21)	**92** – ($10.00)
proftpd	setup_env	✓	×	**40** – ($3.13)	**98** – ($10.00)
proftpd	resolve_logfmt_id	×	✓†	**90** – ($10.00)	**63** – ($10.00)
proftpd	tpl_map_va	×	×	**80** – ($10.00)	**87** – ($10.00)
proftpd	listfile	×	×	**84** – ($10.00)	**85** – ($10.00)
proftpd	dolist	✓	✓†	**29** – ($2.49)	**98** – ($10.00)
janus	janus_videoroom_process_synchronous_request	×	×	**63** – ($10.00)	**82** – ($10.00)
janus	janus_audiobridge_process_synchronous_request	×	×	**68** – ($10.00)	**82** – ($10.00)
janus	janus_streaming_process_synchronous_request	×	×	**67** – ($10.00)	**78** – ($10.00)
janus	janus_process_incoming_admin_request	✓	✓	**18** – ($2.05)	**66** – ($8.37)
janus	janus_textroom_handle_incoming_request	×	×	**61** – ($10.00)	**57** – ($10.00)

Table 3: Comparison of coverage, input discovery, and bug detection across four OSS-Fuzz targets.

	Metric	nginx	dnsmasq	lighttpd	mosquitto
OSS-Fuzz	#Lines Covered	12,619	539	453	7,930
	#Fuzz Drivers	1	5	1	23
	Driver Size (LoC)	331	1,165	77	n/a
Our agent	Target Functions (reached/total)	14/20	16/20	17/20	14/20
	#Lines Covered	23,644	8,048	5540	4,530
	#Cov.-incr. inputs	1685	1,530	861	348
	#Bugs Found	0	1	0	0

Our system is built on top of SWE-agent [26] and EnIGMA [1], which demonstrated how equipping LLMs with structured interfaces and tool access can improve performance in complex software tasks. While those systems focused on software engineering tasks and CTF-style vulnerability discovery, we adapt and extend the agent paradigm to the domain of software reachability—evaluating whether LLMs can autonomously discover how to interact with real-world systems in ways that make security analysis possible in the first place.

Directed Greybox Fuzzing. Directed fuzzing techniques [4, 24] extend traditional greybox fuzzing by guiding input generation toward specific target code locations. These approaches augment coverage feedback with a notion of distance—typically computed using static analysis over the control-flow or call graph—to prioritize inputs that move the generation of new inputs closer to the desired target. This guidance is particularly useful in scenarios such as patch testing or exploit reproduction, where the location of interest is known in advance.

These techniques are effective once the program is already under test and a valid interaction path has been established. However, they assume that the target functionality—understood here as the coarse-grained region of code relevant to a specific feature or behavior—is already reachable via standard input channels such as command-line arguments or file inputs. There is a substantial dependence on the initial seed inputs which are mutated to create new inputs guided to be closer to the targets. In practice, however, many real-world systems require prior configuration, environment setup, and engagement with the correct high-level interface or protocol before the desired functionality becomes accessible. As a result, these approaches often begin one step later in the testing pipeline. In contrast, our work focuses on the earlier and more general problem of semantic reachability: discovering how to enter the broad neighborhood of a functionality so that subsequent testing—directed or otherwise—can be applied effectively.

Automated Fuzz Driver Generation. One possible response to the limitations of directed fuzzing is to automate the step that precedes it—namely, the generation of fuzz drivers that interface with the target code. Several recent approaches have tackled this challenge, using static or dynamic analysis, sometimes combined with language models, to synthesize harnesses that invoke target functions or APIs [2, 12, 13, 25].

Fudge [2] automatically synthesizes fuzz drivers by leveraging a library consumer to extract valid API usage of a library API. FuzzGen [12] implements a whole system analysis to infer the library's interface and synthesizes fuzz drivers accordingly. KernelGPT [25]

Table 4: Examples of functionalities reached by the LLM agent and the required setup actions.

Program	Functionality	Required Setup
nginx	QUIC handling	Enable Hypertext Transfer Protocol version 3 (HTTP/3) and Quick UDP Internet Connections (QUIC) in configuration files; serve Transport Layer Security (TLS) certificates; issue a QUIC-compatible request
	SSL setup	Generate certificates; update configuration; initiate Hypertext Transfer Protocol Secure (HTTPS) request
	Upstream connection	Define upstream servers; configure proxying; send backend-directed request
	UWSGI dispatch	Configure (Useful Web Server Gateway Interface) UWSGI module; set up ‘.uwsgi‘ handler; send request to corresponding endpoint
dnsmasq	DHCPv6 parsing	Enable Dynamic Host Configuration Protocol version 6 (DHCPv6); configure IPv6 interfaces; simulate valid DHCPv6 client request
	DHCP	Enable DHCP service; configure address pool; send crafted DHCP discover packet
	DNS query parsing	Configure Domain Name Server (DNS) domain rules; simulate query via User Datagram Protocol (UDP)
lighttpd	HTTP request parsing	Enable HTTP module; send custom request with crafted headers and paths
	URL normalization	Activate Uniform Resource Locator (URL) rewriting; issue requests with symbolic path components (e.g., ‘../‘)
mosquitto	In-memory DB	publish message to broker; terminate broker to trigger write to database (DB); restart broker
	Subscribe packet	Create valid configuration; send SUBSCRIBE command and wait for ACK

focuses on the Linux Kernel, and leverages Large Language Models to synthesize syscall specifications to enhance kernel fuzzing. UTopia [13] automatically synthesizes fuzz drivers for open source libraries from existing unit tests.

However, automated fuzz driver generation faces fundamental limitations. Most approaches are designed for libraries, assuming that functionality can be exercised in isolation. This assumption breaks down when the target code is embedded in a larger execution context—as is the case for several of our targets, like `dnsmasq` and `proftpd`, where even shallow functionalities show 0% OSS-Fuzz coverage. Isolating such code risks breaking the semantics needed to meaningfully trigger it. Moreover, many systems depend on the availability of existing library clients to inform driver synthesis, which limits applicability to less-used or more complex components.

In-vivo fuzzing [8] offers a complementary approach by sidestepping the need for explicit fuzz driver generation. Instead of isolating the target code, it instruments the program at runtime and amplifies natural executions at selected points, known as amplification points. This allows to test code in its native execution context, preserving real-world data flows and environmental state. This technique still assumes that such execution points can be reached in the first place—typically through existing user interactions or test workloads. The core challenge of how to initially drive execution toward these points remains unaddressed. This is precisely the problem we target: discovering how to reach meaningful functionality in the first place, especially when it requires non-trivial configuration, protocol engagement, or system-level setup. Our work can thus be seen as a prerequisite step that enables fuzzing techniques to operate more effectively.

8 Conclusion

This work recognizes reachability as a central challenge in scaling automated software security testing to general software systems and investigates the effectiveness of Large Language Model (LLM) agents in addressing this issue. It also introduces an end-to-end methodology that integrates the complementary strengths of LLM agents and in-vivo fuzzing. While in-vivo fuzzing amplifies real executions to expose deep program behaviors, solving the reachability challenge is key to unlocking its effectiveness. Our approach addresses this challenge by using an LLM agent to interact with the system, configure the environment, and steer execution toward specific internal functionality, thereby exposing complex program states from which fuzzing can operate.

We demonstrate the feasibility and effectiveness of this methodology through an extensive evaluation across a wide range of real-world software targets. Our findings indicate that the LLM agent can reach the target functionality in most cases, a process that often involves preparatory steps such as adjusting configurations and setting up the environment. Once these functionalities are reached, invivo fuzzing significantly increases coverage and even leads to the discovery of a previously unknown vulnerability.

Beyond empirical results, this work offers a broader contribution: a shift in how LLMs are positioned within software security. Rather than acting as direct vulnerability finders, LLMs can be the enablers that prepare systems for deeper automated testing.

Looking forward, at least two research directions appear worth exploring. First, automating the discovery of invivo fuzzing amplifier points and the corresponding constraints: for example, would it be beneficial to go beyond amplifying system calls, and be even more targeted? And in cases where we are not amplifying system calls, how would we choose the buffer to fuzz? And how would we automatically infer the corresponding constraints? Second, since the success in reaching a target is directly verifiable through coverage feedback after each interaction, the problem provides a natural reward signal. Therefore, reinforcement learning approaches might be used to train agents and specialize them in navigating specific codebases.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and for helping us improve this paper. This research is partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant (Project AT_SCALE, 101179366). It is also partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberly Milner, Sofija Jancheska, John Yang, Carlos E. Jimenez, Farshad Khorrami, Prashanth Krishnamurthy, Brendan Dolan-Gavitt, Muhammad Shafique, Karthik Narasimhan, Ramesh Karri, and Ofir Press. 2025. Interactive Tools Substantially Assist LM Agents in Finding Security Vulnerabilities. arXiv:2409.16165 [cs.AI] https://arxiv.org/abs/2409.16165

[2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. doi:10.1145/3338906.3340456

[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020

[5] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2025-03-04.

[6] Darpa. 2024. AIxCC. https://aicyberchallenge.com/. Accessed: 2025-03-04.

[7] Darpa. 2024. AIxCC. https://aicyberchallenge.com/most-popular-bug-semfinal-competition/. Accessed: 2025-03-04.

[8] Octavio Galland and Marcel Böhme. 2025. Invivo Fuzzing by Amplifying Actual Executions. In *Proceedings of the 47th International Conference on Software Engineering (ICSE'25)*. 13 pages.

[9] Google. 2024. bigsleep. https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html. Accessed: 2025-03-04.

[10] Google. 2024. Naptime. https://googleprojectzero.blogspot.com/2024/06/project-naptime.html. Accessed: 2025-03-04.

[11] Google. 2024. ossFuzzTargets. https://fuzz-introspector.readthedocs.io/en/latest/user-guides/get-ideas-for-new-targets.html. Accessed: 2025-03-04.

[12] 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

[13] 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

[14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[15] Antonio Morales. 2020. Fuzzing sockets, part 1: FTP servers. https://securitylab.github.com/resources/fuzzing-sockets-FTP/.

[16] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. 2013. Using complexity metrics to improve software security. *Computer Fraud & Security* 2013, 5 (2013), 8–17. doi:10.1016/S1361-3723(13)70045-9

[17] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[18] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 42 (Jan. 2025), 28 pages. doi:10.1145/3697010

[19] Niklas Risse and Marcel Böhme. 2024. Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4247–4264. https://www.usenix.org/conference/usenixsecurity24/presentation/risse

[20] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[21] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 862–880. doi:10.1109/SP54263.2024.00210

[22] Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, Vlad Ionescu, Yue Li, and Joshua Saxe. 2024. CYBERSECEVAL 3: Advancing the Evaluation of Cybersecurity Risks and Capabilities in Large Language Models. arXiv:2408.01605 [cs.CR] https://arxiv.org/abs/2408.01605

[23] 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

[24] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Xiao Xiao, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical code guided directed greybox fuzzing for commits. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 138, 16 pages.

[25] 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 560–573. doi:10.1145/3676641.3716022

[26] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE]

[27] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427