

Review

On the Challenges and Opportunities of Fuzzing via Large Language Models: A Review

Yiqing Sun, Villads H. Andersen , Gaurav Choudhary  and Nicola Dragoni *

Section of Cybersecurity Engineering, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), 2800 Kongens Lyngby, Denmark; s252599@student.dtu.dk (Y.S.); s205349@student.dtu.dk (V.H.A.); gauch@dtu.dk (G.C.)

* Correspondence: ndra@dtu.dk

Abstract

Fuzzing is an important automated testing technique for discovering vulnerabilities and abnormal software behavior, but conventional and pre-LLM learning-based approaches often struggle with strict validity constraints, limited semantic understanding, and poor adaptability across targets. Recent advances in large language models (LLMs) create new opportunities for fuzzing by supplying semantic guidance from source code, documentation, traces, and execution feedback. As the literature on LLM-based fuzzing has grown rapidly, a systematic synthesis is needed. This paper presents a literature review on the integration of LLMs into fuzzing workflows. We review how LLMs are applied across the fuzzing workflow, compare these approaches with earlier deep learning-based fuzzing, and summarize the main patterns in task coverage, application strategies, targets, testing modes, and model choices. The survey shows that current work is concentrated in test case generation, while defect detection and post-processing remain less represented, and broader workflow integration is uneven. Based on these findings, we identify four main directions for future research as follows: broader coverage of the fuzzing workflow, stronger context construction and workflow engineering, richer reasoning for fuzzing control, and shared evaluation standards and benchmarks.

Keywords: large language models; fuzzing; software testing

1. Introduction

Fuzzing, or fuzz testing, was first proposed by Miller et al. [1] in 1990. The core idea of fuzzing involves generating a series of unexpected or random inputs to evaluate the reliability and security of software systems. With the continuous evolution of the modern software industry, fuzzing has become an important method of software testing. Although fuzzing has achieved a considerable degree of automation, designing effective fuzzing strategies for software with complex functionality and structure remains challenging, even with auxiliary methods such as constraint-based [2], random-based [3], and learning-based [4] approaches.

In recent years, progress in artificial intelligence has led to the emergence of large language models (LLMs) with massive parameter scales, such as DeepSeek-R1 [5] and the GPT-5 series [6]. Pretrained on large-scale datasets, these models initially demonstrated strong performance in natural language processing (NLP) tasks and have since shown broader potential across general tasks, presenting new opportunities for fuzzing methodology research. Moreover, code-oriented LLMs tailored for programming-language



Academic Editor: Douglas O'Shaughnessy

Received: 17 April 2026

Revised: 18 May 2026

Accepted: 18 May 2026

Published: 21 May 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

tasks have further expanded this potential, including CodeGen2 [7], DeepSeek Coder [8], Codex [9], and Claude Code [10]. Trained on extensive amounts of program source code, these models are better suited to tasks such as code generation, program repair, and semantic analysis, making them natural fits for test generation and defect analysis in fuzzing.

Applying large language models (LLMs) to fuzzing can increase automation, help analyze abnormal behavior in the tested system, and support efficient vulnerability detection [11]. On the one hand, LLMs can interpret target programs and generate test cases with limited human intervention. On the other hand, their reasoning and representation capabilities may support the analysis of vulnerabilities involving complex program logic and calling relationships. Meanwhile, because fuzzing benefits from input diversity, LLM-generated test cases may also help increase coverage and reveal unexpected vulnerabilities [11].

As a growing body of work has proposed LLM-based fuzzing methods, a systematic and comprehensive survey of this area is still lacking. To address this gap, this paper presents a thorough, systematic literature review of current research on the application of LLMs to fuzzing.

1.1. Research Questions

To systematically investigate this topic, this paper proposes the following research questions:

1. What advantages do fuzzing techniques incorporating LLMs provide compared to traditional deep learning-based fuzzing approaches?
2. How have large language models been integrated into different stages of fuzzing workflows?
3. What are the potential opportunities and challenges for future fuzzing techniques that incorporate LLMs?

1.2. Contributions

This paper makes the following contributions:

1. We provide an overview of the fuzzing workflow and summarize common techniques for incorporating LLMs into fuzzing.
2. We examine the limitations of pre-LLM deep learning-based fuzzing approaches and analyze how LLMs address these shortcomings.
3. We systematically analyze how LLMs are integrated into different stages of the fuzzing workflow.
4. We discuss the challenges and outline future research directions for applying LLMs to fuzzing.

1.3. Outline

The paper is structured as a systematic review of the literature, with primary emphasis on the rapid body of LLM-based fuzzing research that emerged after the release of ChatGPT-3.5 in November 2022 [12].

Section 1 provides a general overview of the topic and presents the research questions addressed in this paper. Section 2 describes the research methodology adopted in this study. Section 3 reviews existing surveys and the related literature. Section 4 introduces the background knowledge of fuzzing and relevant LLM techniques applied in fuzzing. Section 5 reviews deep learning-based fuzzing approaches prior to the LLM era and analyzes the advantages of LLM-based fuzzing techniques compared to traditional deep learning methods. Section 6 categorizes and summarizes existing work on LLM-assisted fuzzing and introduces their main ideas. Section 7 analyzes the limitations of current

LLM-based fuzzing methods across test case generation, defect detection, and post-fuzz processing. Section 8 discusses future research directions, including broader workflow coverage, context and reasoning support, and shared evaluation standards. Finally, Section 9 concludes the paper.

2. Research Methodology

This section describes the methodology used to identify, select, and analyze relevant papers for this literature review. Figure 1 illustrates this process.

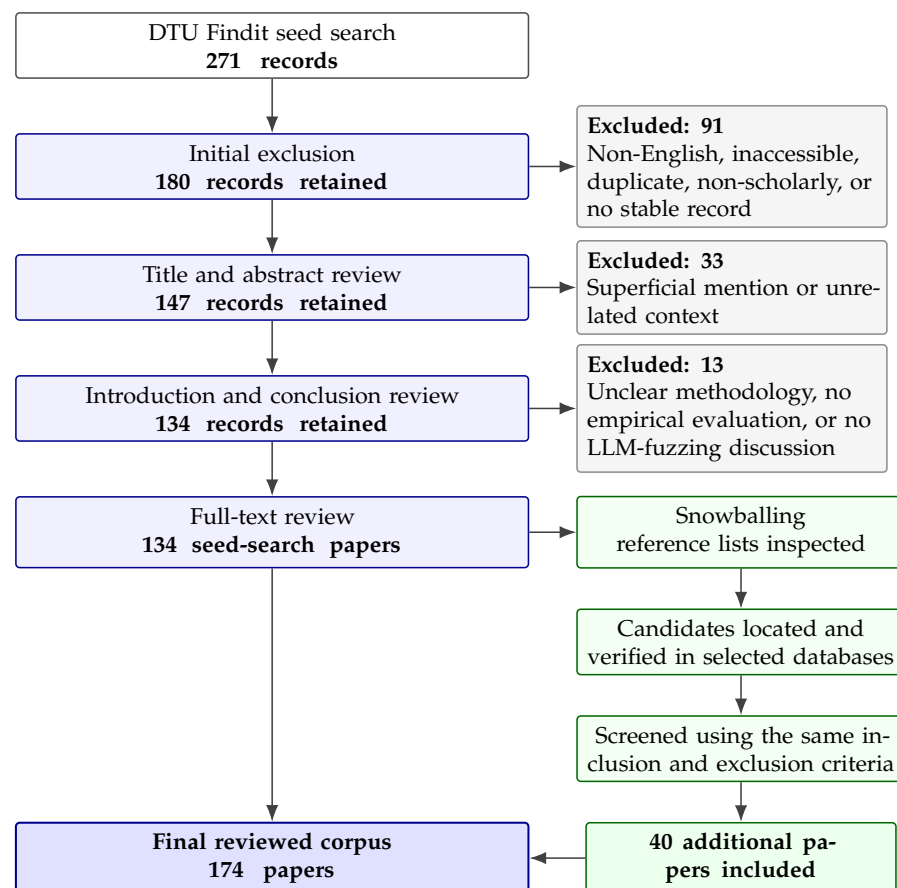


Figure 1. Paper selection and screening process for the systematic review of LLM-based fuzzing research.

2.1. Search Strategy

To identify keywords and what search query to perform, we defined the PICOC criteria [13] based on the research questions.

- **Population:** Research papers that propose, improve, evaluate, or analyze fuzzing techniques that incorporate large language models.
- **Intervention:** The integration of large language models into fuzzing workflows.
- **Comparison:** A comparison between different approaches of integrating LLMs into fuzzing workflows.
- **Outcomes:** An overview of how LLMs have been incorporated into fuzzing workflows and an analysis of their reported performance.
- **Context:** The context of this review includes large language models, fuzzing, software testing, software security, fuzz testing, and the incorporation of large language models in fuzzing workflows.

Based on the PICOC criteria, the search used the following two core concepts: large language models and fuzzing. DTU Findit was used as the initial discovery portal because

it indexes publications from multiple publishers and bibliographic databases. The seed query was applied to title and keyword fields to retrieve papers with an explicit focus on both concepts: “(title:(LLM OR “large language model”) AND fuzzing) OR (keyword:(LLM OR “large language model”) AND fuzzing)”. This query produced 271 results, which formed the initial candidate set for screening. After the initial search, the results underwent several stages of exclusion and screening at different levels of granularity to determine their relevance and suitability for inclusion in the final synthesis. The details of these stages are described in the following subsections.

To reduce the risk of missing relevant studies that used different terminology or fell outside the seed query, the initial search was later expanded through snowballing. In this step, the reference lists of papers that passed full-text review were inspected, and relevant candidate papers were located and verified through data sources in addition to DTU Findit, including ACM Digital Library, IEEE Xplore, SpringerLink, USENIX, Scopus/Web of Science where available through institutional access, Google Scholar, and arXiv. Additional variants, such as “fuzz driver”, “harness generation”, and “vulnerability discovery”, were considered during snowballing. Candidate papers identified through snowballing were screened using the same inclusion and exclusion criteria as the original DTU Findit results.

We distinguish between the reviewed corpus and contextual references. The reviewed corpus consists of research papers that were analyzed and classified as relevant to the integration of LLMs into fuzzing workflows. It includes peer-reviewed journal and conference papers, as well as scholarly preprints, when they directly propose, evaluate, or analyze an LLM-based fuzzing technique and provide sufficient methodological detail for classification. Non-scholarly sources, such as tool pages, project websites, product documentation, and technical blog posts, served only as contextual references when needed to describe background, tools, or model releases. Contextual references may also include peer-reviewed papers or scholarly preprints that are relevant to the discussion and fall outside the final set of 174 reviewed papers described below because they came from contextual reading outside the search process. Among these 174 papers, 156 are summarized in Table A1, and 18 are cited as related works in Table 1. Descriptive corpus counts in this paper report coded-study frequencies without weighting evidence of technical superiority. Appendix Table A1 marks scholarly preprints in the coded corpus; technical reports and vendor or product material remain contextual references and are excluded from corpus counts.

Since LLMs have primarily emerged since OpenAI released its initial public version of ChatGPT-3.5 [12] in November 2022, there were no relevant papers from before the year of 2022.

Table 1. Comparison of representative work on LLM-assisted fuzzing.

Paper	Scope	Study Type			LLM Application Stage					
		Surv.	Emp.	DL	Seed	Input	Drv.	Det.	Anal.	Flow
Aldysty et al. [14]	ICS/INP	✓							✓	
Asmita et al. [15]	Firmware	✓							✓	
Black et al. [16]	Python fuzzing		✓		✓					
Jiang et al. [17]	Directed testing		✓			✓				
Jiang et al. [18]	DBMS fuzzing		✓						✓	
Jiang et al. [19]	Code LLMs	✓								
Miao et al. [20]	DL fuzzing	✓		✓						
Wang et al. [11]	LLM testing	✓				✓				
Huang et al. [21]	Software security	✓			✓	✓	✓	✓	✓	
Nakanishi et al. [22]	IoT seeds		✓		✓					
Panebianco et al. [23]	Defect detection		✓					✓		

Table 1. Cont.

Paper	Scope	Study Type			LLM Application Stage					
		Surv.	Emp.	DL	Seed	Input	Drv.	Det.	Anal.	Flow
Qiu et al. [24]	Greybox fuzzing	✓		✓	✓				✓	
Zhang et al. [25]	C API fuzzing		✓				✓			
Feng et al. [26]	Fuzz drivers	✓					✓			
Zhu et al. [27]	Software security	✓						✓	✓	
Gubbi et al. [28]	Hardware security	✓			✓	✓	✓		✓	
Cheng et al. [29] ^P	Software security	✓			✓		✓			
Hardgrove and Xu [30]	Software security	✓			✓			✓	✓	
This paper	General *	✓		✓	✓	✓	✓	✓	✓	✓

* This paper focuses primarily on software security; however, the scope of the reviewed literature covers both software and hardware security. Legend: Surv. = survey; Emp. = empirical study; DL = discussion of deep learning-based fuzzing prior to LLMs; Seed = seed generation; Input = directed input generation; Drv. = fuzz driver generation; Det. = vulnerability detection; Anal. = post-fuzz analysis; Flow = full fuzzing workflow coverage; ^P = scholarly preprint for which no peer-reviewed venue was identified at the time of screening; ✓ indicates that the paper explicitly covers the category; blank cells indicate that the category is not a primary focus or is not discussed.

2.2. Paper Selection

Several exclusion steps and one additional identification step were executed to identify which papers to include in this paper. This process is diagrammed in Figure 1.

2.2.1. Initial Exclusion

To determine the relevant papers of the 271 results, the following criteria were selected for the initial exclusion. Any papers that did not fulfill them were excluded.

- English language.
- Accessible full text.
- Published research paper or scholarly preprint.
- Has a Digital Object Identifier (DOI), arXiv identifier, or stable publisher record.
- Not a duplicate.

After the initial exclusion, there were 180 papers remaining.

2.2.2. Screening Procedure

The screening was conducted by the authors in multiple stages. The first author performed the initial search, deduplication, and first-pass exclusion. The remaining candidate papers were then checked against the inclusion and exclusion criteria during title/abstract screening, introduction/conclusion review, and full-text review. Borderline cases and classification decisions were discussed with the co-authors and resolved by consensus. When a paper's publication type or relevance was unclear, the authors inspected the publisher page, DOI or arXiv record, abstract, and full text before making a final decision. The protocol omitted a formal inter-rater agreement measure, such as Cohen's kappa, because screening was organized as a staged review with consensus checks. Disagreements were resolved through discussion using the criteria described above.

2.2.3. Title and Abstract Review

The titles and abstracts of the 180 papers were systematically screened to determine their relevance and if they could answer the research questions. A paper was included if it directly proposed, improved, or evaluated fuzzing techniques incorporating large language models. Papers that mentioned fuzzing or large language models only superficially, or addressed them in unrelated contexts, were excluded at this stage. The number of papers left after this review were 147.

2.2.4. Introduction and Conclusion Review

The introduction and conclusion of the 147 remaining papers were reviewed to further assess their relevance and suitability for inclusion. Papers were included if these sections clearly described fuzzing techniques that integrate large language models and reported on their evaluation or performance. Studies that lacked a clear methodology, empirical evaluation, or discussion of LLM-based fuzzing were excluded at this stage. After this review, 134 papers remained.

2.2.5. Full Text Review

A full-text review was performed on the remaining 134 papers. The purpose of this review was to determine if and how each paper could contribute to the research questions. This review was also used to classify the different papers based on how they integrate LLMs and what the fuzzing targets were.

To make the descriptive synthesis reproducible, each paper in the reviewed corpus was coded using the classification fields reported in Table A1: task, LLM application strategy, testing type, target system, target language, and model family. During full-text review, the first author assigned initial labels from each paper's method and evaluation sections. Ambiguous cases, including papers spanning several fuzzing stages or using both prompting and traditional analysis, were discussed by the co-authors and resolved by consensus. Multiple labels were allowed when a paper clearly supported more than one category; otherwise, the most specific category supported by the paper was used.

The task labels distinguish test-case or seed generation (TCG), driver or harness generation (DG), defect detection (DD), post-processing (PP), and rare workflow-level tasks outside these four stages. Strategy labels distinguish fine-tuning (FT), prompt engineering (PE), hybrid systems that combine an LLM with conventional fuzzing, static analysis, retrieval, execution feedback, or validation components, and agentic workflows where an LLM controls multi-step tool use or planning. Prompt-engineering sublabels record zero-shot, few-shot, chain-of-thought, tree-of-thought, and automatic prompt construction when a paper reports them. Testing mode was coded as black-box, gray-box, or white-box based on the information available to the fuzzing loop. Target and target-language labels were copied from the study when explicit; otherwise, they were inferred from the evaluated systems and marked conservatively. Undetermined fields were assigned the label "not reported". Sections 6–8 give a more detailed description of this process and the resulting patterns in the reviewed corpus.

2.2.6. Snowballing

After the full-text review of the 134 papers returned from the initial DTU Findit pipeline, snowballing was performed before the final synthesis of the corpus. In this step, the reference lists of papers that passed full-text review were inspected to identify relevant studies missed by the initial DTU Findit query or that used different terminology for LLM-assisted fuzzing. In addition to DTU Findit, ACM Digital Library, IEEE Xplore, SpringerLink, USENIX, Scopus/Web of Science where available through institutional access, Google Scholar, and arXiv were used to locate, verify, and retrieve candidate papers identified during this process. Each candidate identified through snowballing was screened using the same criteria as the original search results: English language, accessible full text, published research paper or scholarly preprint, DOI/arXiv identifier/stable publisher record, no duplication, direct relevance to LLM-based fuzzing, clear methodology, and sufficient evaluation or performance discussion. The same inclusion criteria applied to the seed-search and snowballing papers. This step added 40 papers, making the total number of papers related to LLM-based fuzzing 174.

3. Related Work

During paper selection, several surveys related to this review were identified. Table 1 succinctly presents the related studies from our final reviewed corpus and compares them with this paper. In addition, we provide a more detailed comparison in Appendix Table A2.

In Table 1, we categorize prior work according to two main aspects: study type and LLM application stage. The study type indicates whether a work is a survey or an empirical study, and whether it discusses deep learning-based fuzzing approaches prior to the emergence of LLMs (denoted as DL). The LLM application stage captures the specific stages of the fuzzing workflow where LLMs are applied, including seed generation, directed input generation, fuzz driver generation, vulnerability detection, and post-fuzz analysis. The column Flow indicates whether a work provides coverage of the overall fuzzing workflow rather than focusing on a single stage.

As shown in Table 1, most existing studies either focus on a specific fuzzing stage (e.g., seed generation [16] or driver generation [25]) or present empirical evaluations within a particular domain [16–18,22,23,25]. Existing surveys also occupy adjacent but narrower positions. General fuzzing and learning-based fuzzing surveys cover classical or pre-LLM techniques [20,24,31], while broader LLM-for-testing, code-generation, and software-security surveys discuss fuzzing as one task among many [11,19,27]. More direct LLM-fuzzing surveys and roadmaps focus on LLM-generated fuzz tests, reliability issues, or specific stages such as fuzz-driver generation, including Huang et al. [21], Cheng et al. [29], and Feng et al. [26]. Domain-specific surveys additionally focus on ICS, firmware, and hardware settings [14,15,28]. In contrast, this paper offers a more holistic view by (i) reviewing the fuzzing workflow and relevant LLM techniques, (ii) analyzing the limitations of pre-LLM deep learning approaches, and (iii) systematically examining how LLMs can be integrated across multiple stages of the fuzzing process. This allows our work to present a unified perspective on the research landscape.

4. Background

4.1. Fuzzing

The term fuzzing was first coined by Miller et al. in 1990 to refer to a program that “generates a stream of random characters to be consumed by a target program” [1]. Since then, fuzzing has grown to be a widely used technique for software testing. Prior to 2005, fuzzing was dominated by black-box approaches based primarily on random input mutation.

Between 2006 and 2010, fuzzing began to incorporate program analysis techniques such as taint analysis and symbolic execution, giving rise to more sophisticated methods, some of which were later incorporated into commercial tools. From 2011 to 2015, the field advanced substantially with the emergence of coverage-guided fuzzing, the adoption of diverse scheduling strategies, and the integration of broader techniques to improve efficiency [31].

During 2016 and 2017, many fuzzing techniques were developed based on AFL [32], such as AFL++ [33] and AFLGo [34]. Since then, fuzzing has also increasingly combined traditional program analysis with techniques such as machine learning.

Fuzzing can be classified along multiple dimensions. From the perspective of test case generation, it is commonly divided into mutation-based and generation-based fuzzing. Mutation-based fuzzing derives new inputs by applying systematic or random modifications to existing seed inputs, typically without requiring a formal specification of the input format. In contrast, generation-based fuzzing constructs inputs from scratch based on a model, grammar, or specification of the expected input structure, thereby enabling the production of syntactically valid test cases. From the perspective of target visibility,

it is typically categorized as black-box, white-box, or gray-box fuzzing [31]. Black-box fuzzing assumes no prior knowledge of the target and generates test cases by mutating seed inputs according to predefined rules. White-box fuzzing, in contrast, leverages detailed program information to guide input generation and, in principle, can achieve complete path coverage. Gray-box fuzzing occupies an intermediate position by using partial program information, most commonly runtime code coverage, to guide and refine mutation strategies [31].

4.1.1. Terminology

Intuitively, fuzzing refers to executing a Program Under Test (PUT) or System Under Test (SUT) with fuzz inputs, i.e., inputs that the program does not anticipate and may process incorrectly, potentially triggering unintended behavior. Following Miller et al. [1] and Manès et al. [31], we formalize fuzzing as follows.

Definition 1 (Fuzzing). *Fuzzing is the execution of a PUT using input(s) sampled from a fuzz input space that extends beyond the expected input space of the PUT.*

Definition 2 (Fuzz Testing). *Fuzz testing is the application of fuzzing to determine whether a PUT violates a security policy.*

Definition 3 (Fuzzer). *A fuzzer is a program that performs fuzz testing on a PUT.*

Definition 4 (Fuzz Campaign). *A fuzz campaign is a specific execution of a fuzzer on a PUT under a defined security policy.*

The goal of a fuzz campaign is to uncover bugs that violate the specified security policy. While early fuzzers primarily checked for program crashes, fuzz testing can be applied to any execution-observable (EM-enforceable) security policy. The component responsible for detecting such violations is referred to as the bug oracle.

Definition 5 (Bug Oracle). *A bug oracle (or simply oracle) is a program, possibly integrated into a fuzzer, that determines whether a PUT execution violates a given security policy.*

Each fuzzer implements a fuzz algorithm, whose behavior is governed by a set of parameters known as a fuzz configuration.

Definition 6 (Fuzz Configuration). *A fuzz configuration consists of the parameter values that control a fuzz algorithm.*

The configuration space varies with the fuzzing approach. Simple fuzzers may require only the PUT as configuration, whereas more advanced fuzzers dynamically adjust parameters such as mutation ratios and input seeds during execution. A seed is a typically well-structured input used to generate test cases through mutation, and fuzzers commonly maintain a collection of seeds, known as a seed pool. Some fuzzers also associate additional data with each configuration, such as code coverage information in coverage-guided fuzzing.

4.1.2. Working Process of Fuzzing

Fuzz testing is currently one of the most successful techniques for detecting security flaws in information systems. Its principle involves generating random or targeted inputs to the system under test and monitoring abnormal results to uncover vulnerabilities [31].

As shown in Figure 2, the general workflow of fuzzing can be divided into five stages as follows: pre-processing, test input generation, test execution, defect detection, and post-fuzz processing.

1. The pre-processing stage comprises the preparatory steps performed on the target system before fuzzing begins. Its purpose is to establish the necessary environment and conditions for an effective fuzzing campaign. Typical tasks in this stage include seed selection, seed pruning, and, where necessary, program instrumentation.
2. The test input generation stage aims to produce a large number of inputs that may trigger abnormal program behaviors. In mutation-based fuzzing, new inputs are derived from existing samples through a sequence of decisions, such as seed scheduling, mutation scheduling, and byte scheduling, which determine the seed input, mutation operator, and target byte positions, respectively. In generation-based fuzzing, inputs are constructed directly from protocol descriptions, grammars, or interface specifications.
3. The execution phase runs the target program with the generated inputs. Depending on the extent of program information available during execution, fuzzing techniques can be categorized as black-box, gray-box, or white-box.
4. The defect detection stage monitors execution outcomes to identify new program states, crashes, or other anomalous behaviors captured by an oracle.
5. The post-fuzz processing stage organizes execution results, generates reports, and updates fuzzing configurations based on the detected defects. More broadly, tasks such as automated program repair may also be regarded as part of post-fuzz processing, as they contribute to improving the overall testing framework.

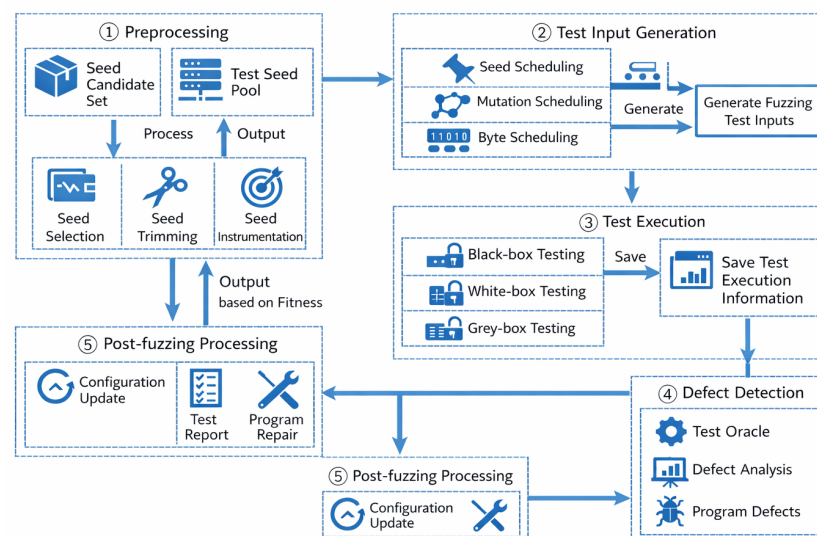


Figure 2. The fuzz testing procedure in general. The arrows indicate the flow of seeds, generated test inputs, execution information, detected defects, and feedback used for configuration updates across the fuzzing stages.

In addition, we give a formal description of the fuzzing process in Algorithm 1 [31]. This abstraction is sufficiently broad to accommodate existing black-box, gray-box, and white-box fuzzing techniques. Algorithm 1 takes as input a set of fuzzing configurations $\mathbb{C}$ and a timeout t_{limit} , and it outputs a set of discovered bugs $\mathbb{B}$. It consists of two components. The first is the PREPROCESS function, which is executed once at the start of a fuzzing campaign. The second is an iterative loop comprising five functions: SCHEDULE, INPUTGEN, INPUTEVAL, CONFUPDATE, and CONTINUE. Each execution of this loop constitutes a fuzz iteration, and each invocation of INPUTEVAL on the program under test (PUT) corresponds to a fuzz run. Not all fuzzers implement all five functions.

Algorithm 1: Fuzz testing general procedure

Input: C, t_{limit}
Output: B // a finite set of bugs

```

1  $B \leftarrow \emptyset$ ;
2  $C \leftarrow \text{PREPROCESS}(C)$ ;
3 while  $t_{\text{elapsed}} < t_{\text{limit}}$  and  $\text{CONTINUE}(C)$  do
4    $conf \leftarrow \text{SCHEDULE}(C, t_{\text{elapsed}}, t_{\text{limit}})$ ;
5    $tcs \leftarrow \text{INPUTGEN}(conf)$ ;
6   //  $O_{\text{bug}}$  is embedded in a fuzzer;
7    $B', execinfos \leftarrow \text{INPUTEVAL}(conf, tcs, O_{\text{bug}})$ ;
8    $C \leftarrow \text{CONFUPDATE}(C, conf, execinfos)$ ;
9    $B \leftarrow B \cup B'$ ;

```

4.2. LLM-Related Techniques in Fuzzing

Current LLM-based fuzzing techniques mainly include fine-tuning, prompt engineering, hybrid methods that combine LLMs with traditional algorithms, and increasingly agentic workflows that let an LLM plan or coordinate multi-step tool use. While fine-tuning can substantially improve LLM performance on a specific task domain, it generally requires access to open-source models and considerable computational resources. Prompt engineering, by contrast, improves task adaptability without retraining the model and is therefore more flexible in practice. However, because prompt effectiveness is highly sensitive to formulation, iterative adjustments and experiments are often required to achieve optimal prompt design. In addition to these methods, hybrid approaches integrate LLMs with traditional program analysis techniques, compensating for inherent limitations of a standalone LLM. Because such integration typically requires specifically designed auxiliary modules, it is often more complex and target-specific.

4.2.1. Fine-Tuning

Fine-tuning refers to making minor adjustments to the hyper-parameters of a pre-trained model to adapt it to the requirements of a specific downstream task, usually through supervised learning and reinforcement learning. By leveraging knowledge acquired during pretraining, fine-tuning can better align the model with task-specific requirements and therefore improve performance. Since fine-tuning adapts knowledge acquired during pretraining, it can be regarded as a form of transfer learning for LLMs.

Supervised learning is a common approach in fine-tuning tasks. Given a dataset of task inputs X and corresponding labels Y , the objective of supervised fine-tuning is to optimize the model hyper-parameters θ by minimizing the training loss, which can be expressed as

$$\theta^* = \arg \min_{\theta} \mathcal{L}(Y, f_{\theta}(X)). \quad (1)$$

Here, $f_{\theta}(X)$ denotes the model output and $\mathcal{L}$ denotes the loss function [35]. In practice, LLM fine-tuning generally involves three steps: selecting an open-source base model, preparing and pre-processing a task-specific dataset, and training the model on that dataset. Some representative fine-tuning methods include Freeze [36], P-tuning [37], LoRA [38], and QLoRA [39].

Compared with training from scratch, fine-tuning substantially reduces training time, training data, and computation requirements. It is particularly attractive for software

testing tasks, where domain-specific adaptation is often needed and newly collected data can be incorporated incrementally to improve performance over time.

4.2.2. Prompt Engineering

Although fine-tuning can raise the performance of the LLM on specific tasks, it is often limited by the availability of open-source base models. On the one hand, many advanced LLMs remain proprietary because of their commercial value; on the other hand, it is still computationally expensive and time-consuming to fine-tune a model because of the scale and complexity of current LLM designs. In this context, prompt engineering has become a widely adopted strategy for its simplicity and low cost. Prior work [35] reports that a carefully designed prompt can perform better than a fine-tuned model in some cases. Some representative categories of prompt engineering strategies include zero-shot prompting [40], few-shot prompting [41], LLM reasoning based on chain-of-thought (CoT) [42] or tree-of-thought prompting [43], and automatic prompt construction [44].

Zero-shot prompting queries an LLM without providing explicit examples. Under code generation settings, the model is given only a task description and is expected to produce the required output directly. Guided approaches, such as ChatUniTest discussed later in Section 6, employ explicit instructions such as “Please help me generate JUnit tests for a specific Java method...” to support test generation. By contrast, unguided approaches [40] provide only minimal context, such as a unit test header (e.g., “class className Test”), leaving the model to infer the remaining task autonomously. Guided prompts tend to yield more accurate outputs in general, whereas unguided prompts may be useful in more specialized settings.

Few-shot prompting tries to improve the quality of output by including a small number of high-quality examples in the prompt input. In code generation settings, these examples typically consist of input–output pairs that illustrate the desired behavior and formatting, therefore helping the model to infer task requirements more accurately. For instance, LiBro [45] guides test case generation by providing error reports together with corresponding test cases that reproduce those errors.

Chain-of-thought (CoT) prompting is a gradient-free method that encourages an LLM to generate explicit intermediate reasoning steps at inference time. Wei et al. [42] showed that such explicit intermediate reasoning can be elicited either through zero-shot CoT prompts or through manually designed few-shot demonstrations, referred to as Manual-CoT. In zero-shot CoT, simple instructions such as “let’s think step by step” encourage step-by-step reasoning without providing examples. Manual-CoT, on the other hand, supplies exemplars that contain both intermediate reasoning steps and final answers, thereby demonstrating the desired reasoning process explicitly.

Tree-of-thought prompting extends this idea by exploring multiple candidate reasoning paths before selecting or validating a final answer. In fuzzing-related workflows, this pattern is most relevant when the model must decompose a complex task such as vulnerability repair, fault localization, or exploit reasoning into several checked intermediate decisions.

Auto-prompting automatically generates prompt templates, reducing the need for manual prompt design. In fuzzing applications, this often involves two models: one analyzes the task context and generates candidate prompts, while the other produces fuzzing inputs based on these prompts. Kojima et al. [46] report that auto-prompting can perform better than manually designed zero-shot prompting by adapting prompt structures to specific tasks and generating more effective test inputs.

4.2.3. LLM Agents and Agentic Workflows

Agentic workflows extend prompt-based use of LLMs from a single response to an iterative process. The model receives task context, takes an action, observes feedback, and uses that feedback to guide the next step. ReAct shows that interleaving reasoning and action improves question answering and interactive decision-making, while Toolformer shows that language models can learn to call useful external tools [47,48]. Reflexion further demonstrates that feedback stored as verbal memory can improve later trials without updating model weights [49]. Broader evaluations such as AgentBench and SWE-agent show that agent performance depends strongly on the interaction environment and the interface through which the model acts [50,51].

For fuzzing, the implication is methodological. An LLM can help control a feedback loop around generation and validation: generated artifacts are checked by external tools, and the checked result guides subsequent decisions. Agentic workflows are therefore most relevant when fuzzing requires repeated observation, validation, and adaptation across a campaign. Because the result depends on this sequence of interactions, reproducible evaluation requires recording the run trajectory.

4.2.4. Combined with Traditional Algorithms

Although LLMs have shown strong performance across a wide range of natural language processing tasks, their effectiveness in generation and reasoning varies across model architectures, scales, and training data distributions. For instance, general-purpose models such as GPT-4 [52] exhibit strong capabilities in contextual understanding and abstraction, whereas more recent code-oriented models (e.g., Qwen3-Coder-Next [53]) are optimized for code generation and program reasoning. Consequently, no single LLM consistently performs optimally across the diverse tasks in fuzzing workflows, making it challenging to rely on a standalone model to address all stages of the fuzzing process.

To mitigate these limitations, researchers have explored integrating LLMs with traditional program analysis and fuzzing techniques in order to combine their complementary strengths. A common strategy is to use feedback-driven validation mechanisms to assess and iteratively refine LLM-generated test cases. In addition, program analysis techniques can provide structural and behavioral information about the SUT, enabling LLMs to generate more targeted inputs and thereby improve coverage. Conventional mutation operators can further diversify generated inputs for a broader range of execution paths, while statistical analysis methods support systematic evaluation of fuzzing outcomes.

5. Deep Learning-Based Fuzzing Before LLMs

Before the emergence of LLMs, deep learning techniques had been widely applied in fuzzing. This section introduces those deep learning-based fuzzing techniques before the LLM era and analyzes their deficiencies.

5.1. Traditional Deep Learning-Based Fuzzing Techniques

As a powerful machine learning paradigm, deep learning has achieved notable success. In computer vision, for example, deep learning models have demonstrated outstanding performance in benchmarks such as ImageNet [54], where they can capture fine-grained feature differences across massive image collections and thereby support highly effective classification and detection of objects. In natural language processing, architectures such as Long Short-Term Memory (LSTM) [55] and its variants can model long-range dependencies in sequential data, enabling more accurate sentiment analysis and related applications, including social media monitoring and market trend analysis [56]. The Transformer architecture [57], through parallel computation and self-attention mechanisms, further improved

the quality and efficiency of tasks such as machine translation, achieving lower error rates on multiple benchmarks [58]. These advances suggest that deep learning can capture high-level patterns in complex data and thus provides promising opportunities for improving the effectiveness and intelligence of software testing techniques such as fuzzing. In fuzzing research, deep learning has been applied primarily to three stages of the workflow: test case generation, defect detection, and post-fuzz processing. A summary of this line of work is provided in Table 2.

Table 2. Comprehensive summary of deep learning-based fuzzing techniques.

Fuzzing Stage	Category	Specific Task	Model/Tech.	Description
Test Case Generation	General Concept	Input pattern learning	CNNs [59], RNNs [60]	Leverages neural networks to capture inherent patterns and complexity within program inputs to produce effective test cases.
	Example	PDF format fuzzing	Char-RNN	Learn&Fuzz [61]: Treats PDF objects as character sequences to fuzz file parsing tools.
	Example	Directed gray-box fuzzing	3-layer CNN	FuzzGuard [62]: Predicts target reachability of seeds based on data flow structure to filter unreachable inputs.
Defect Detection	General Concept	Oracle generation and defect analysis	Deep Neural Networks	Extracts semantic information from source code to identify code defects. Often represents program structure as vectors.
	Example	Test oracle generation	Deep Neural Networks	Monsefi et al. [63]: Encodes software inputs and expected outputs to train networks for oracle generation.
	Example	Vulnerability identification	CNN [59]	Li et al. [64]: Converts abstract syntax trees (ASTs) to numerical vectors via word embeddings to learn semantic and structural features.
Post Processing	General Concept	Post-fuzz tasks overview	Various	Encompasses diverse tasks including test report generation [33,65] and fuzz test update [66].
	General Concept	Automatic program repair	Various	Current deep learning systems predominantly focus on program repair by learning code fragments before and after the repair process.
	Example	Fault location and auto repair	LSTM [55] and data flow analysis	DEAR [67]: Trains on ASTs of source code before and after repair to automatically locate and fix faults.

5.1.1. Deep Learning-Based Test Case Generation

Deep learning-based test case generation methods use neural networks to learn patterns in program inputs and produce effective test cases. Common neural network architectures include convolutional neural networks (CNNs) [59], recurrent neural networks (RNNs) [60], etc.

For example, Learn&Fuzz [61] models PDF objects as character sequences and trains a Char-RNN to fuzz PDF parsers. FuzzGuard [62] uses a three-layer CNN to predict whether a seed can reach a target based on program data flow before execution. This design supports directed gray-box fuzzing by filtering out inputs unlikely to reach the target, thereby improving efficiency.

5.1.2. Deep Learning-Based Defect Detection

In a general fuzzing procedure, defect detection mainly involves test oracle generation and defect analysis.

For test oracle generation, deep learning-based methods often use neural networks to learn oracle behavior from historical test data. For example, Monsefi et al. [63] encoded

software inputs and their expected outputs as training data and trained deep neural networks to generate test oracles.

In addition, for defect detection tasks, neural networks are used to learn semantic and structural features from source code. To better represent program structure, some studies transform code into vectorized forms before training. For example, Li et al. [64] converted programs into abstract syntax trees, mapped them into numerical vectors using embeddings, and trained a CNN [59] on these vectors to identify defects and vulnerabilities.

5.1.3. Deep Learning-Based Post-Processing in Fuzzing

In a general fuzzing workflow, post-fuzz processing includes tasks such as test report generation [33,65] and fuzzing configuration updates [66]. Existing deep learning-based methods applied in this phase, however, have focused mainly on program repair. These methods learn from code fragments before and after repair to support automated fixing. For instance, Li et al. [67] proposed DEAR, which combines deep learning with data flow analysis. By training an LSTM [55] on abstract syntax trees, the model can locate faults and generate repairs automatically.

5.2. Deficiencies of Deep Learning-Based Fuzzing

Although deep learning-based fuzzing has improved testing efficiency and effectiveness, it still has obvious shortcomings. Compared to LLM-based fuzzing, traditional deep learning-based fuzzing primarily suffers from two types of deficiencies: inherent limitations of neural network architectures and application limitations of these methods. A summary of this line of work is provided in Table 3.

Table 3. Comprehensive summary of deficiencies of deep learning-based fuzzing techniques.

Limitation Category	Specific Area	Key Challenge	Example/Ref.	Description and Impact
Inherent Limitations	Training Data	Scarcity and Imbalance	[68]	Lacks sufficient samples (positive and negative). Imbalanced public datasets and the resulting statistical bias in predicting.
	Feature Learning	Complex Call Relationships	[20,69],	Performs poorly on deeply nested functions or multi-point vulnerabilities. Its black-box nature limits interpretability, and guided mutations do not always improve results.
Application Limitations	Test Case Generation	Invalid/ Inefficient Cases	DeepXplore [70]	Generates invalid inputs that waste execution time or cause compilation issues. It depends heavily on initial seed quality and tends to remain in originally reachable regions.
	Defect Detection (Oracles)	Costly Data and Interpretability	-	Struggles to capture complex software behaviors. It requires substantial labeled data, which is expensive to obtain, and its decision process is difficult to interpret.
	Defect Detection (Vulnerabilities)	Poor Generalization and Simplification	CWE [71,72]	Fails to cover all defect types or novel defects. It often reduces complex vulnerabilities, such as those in CWE, to simple classification problems, and remains limited in industrial deployment.
	Post Processing	Noise, Overfitting, and Low Automation	CoCoNuT [72,73]	Training datasets often contain noise, such as duplicate samples. It also lacks intelligent test-suite selection, leading to high execution cost, while overfitting still requires manual patch inspection.

5.2.1. Inherent Limitations of Deep Learning-Based Fuzzing

Limitations of Training Data

The performance of deep learning-based fuzzing depends heavily on both model design and training data quality. In practice, effective training requires large and well-balanced datasets with sufficient positive and negative samples. In fuzzing scenarios, this is challenging, since tasks vary across programming languages, input formats, and testing goals. As a result, many neural network models were trained without adequate task-specific data. Existing methods mainly rely on web-crawled data or samples produced by other fuzzing tools, leading to limited diversity, weak feature coverage, and severe class imbalance. Prior work shows that, under such imbalance, models may simply predict branches as “undetected” according to the dominant class distribution [68].

By contrast, LLMs are pretrained on broad and diverse corpora, and can often adapt to testing tasks with only few-shot or zero-shot prompting, reducing reliance on task-specific training data.

Limitations in Feature Learning and Reasoning

Although deep learning models have shown strong performance in fields such as speech recognition and computer vision, they are less effective when applied to programs with complex control flow and call relationships. This limitation becomes more evident when vulnerabilities span multiple execution points or appear in deeply nested functions [20]. While such models can learn useful features through multilayer architectures, their black-box nature limits interpretability. Their practical benefit in fuzzing is also uncertain. For instance, Li et al. [69] found that deep learning-based fuzzing does not consistently outperform traditional fuzzing tools, and that neural guidance for input mutation did not produce statistically significant gains on several projects.

5.2.2. Application Limitations of Deep Learning-Based Fuzzing

Limitations in Test Case Generation

Deep learning-based test input generation has improved fuzzing under some settings. However, there are limitations. These methods may generate invalid or low-value test cases, leading to compilation failures or wasted execution time. For instance, DeepXplore [70] generates inputs only by modifying existing ones, so its performance depends strongly on the quality of the initial seeds. Because the method applies only small perturbations to valid inputs, the resulting test cases tend to remain close to the original seeds, leaving much of the valid input space unexplored.

Limitations in Defect Detection

In test oracle generation, traditional neural networks often lack sufficient semantic understanding of program behavior. This limitation is tied to their model capacity, and single-model approaches in particular may struggle to capture complex software semantics. The issue of lacking sufficient labeled training data, as previously mentioned, might also contribute to this problem.

In addition, traditional neural network-based methods are limited in generalization and categorization. Since their performance depends strongly on the coverage and diversity of the training data, when certain defect categories are underrepresented, detection performance for those categories is likely to be poor. This causes a serious issue in vulnerability analysis since defect types are diverse. For instance, the Common Weakness Enumeration (CWE) [71] defines ten broad categories and hundreds of fine-grained subcategories. Traditional methods often design rules for specific vulnerability types, whereas many deep

learning approaches reduce the task to binary or coarse multi-class classification and do not capture these distinctions well.

At present, research on deep learning for defect (vulnerability) detection remains focused largely on benchmark performance. Since vulnerability detection is a critical component of real software development, future work should be evaluated in large-scale engineering settings and with practical deployment constraints in mind. Such work remains limited [72].

Limitations in Post-Processing

Deep learning has also been used for automated program repair, but there are still limitations. One issue is limited automation. Many approaches cannot perform intelligent test selection, and instead execute the full functional test suite for every candidate patch, which is costly [72]. Overfitting is another common problem, so plausible patches often still require manual inspection. This reduces the degree of automation and leaves room for human error.

In summary, deep learning-based fuzzing supports automated test generation and defect detection by learning patterns from data, but its limitations remain substantial. These constraints affect the generation, detection, and post-processing stages alike. In response, recent work has increasingly turned to large language models to support fuzzing, opening a new line of research. The next section reviews these LLM-based fuzzing methods in detail.

6. LLM-Based Fuzzing Techniques

As large language models (LLMs) increasingly demonstrate strong performance in tasks like code generation and reasoning, researchers have gained deeper insights into LLM-based fuzzing methodologies.

This section reviews LLM-based fuzzing through three analytical questions. First, what artifact or claim does the model produce: an input, a driver, an oracle, a vulnerability hypothesis, a crash explanation, or a patch? Second, what validation signal makes that artifact useful: compilation, parser acceptance, coverage, replayable crashes, differential behavior, symbolic constraints, CI tests, or human review? Third, what cost and evaluation assumptions support the reported result? This framing follows Wang et al.'s observation that LLMs address some limitations of earlier deep learning-based fuzzing through broad pretraining, code generation, and reasoning ability [11], while also making clear that LLM output still needs grounding in fuzzing evidence.

Most surveyed systems still concentrate on three stages of the fuzzing pipeline: test case generation, defect detection, and post-fuzz processing. Recent work also touches seed preparation, context construction, execution feedback, driver and harness generation, and workflow automation. Table A1 in the Appendix A provides the complete reference map used in this survey. The following subsections review the main stages and emphasize the evidence standard each stage requires.

6.1. LLM-Driven Test Case Generation

In the coded corpus, test case generation is the most frequently represented application of large language models in fuzzing. Compared to traditional deep learning models, LLMs benefit from more diverse pretraining data, which can help them interpret test input constraints. This mitigates some limitations of prior deep learning-based generation methods, although the magnitude of any improvement remains target- and evaluation-dependent. As shown in Table A1, prompt engineering-based methods are the most frequently reported in this category, with smaller but still important research streams on task-specific fine-tuning and hybrid pipelines. This distribution should be read as

a research-attention pattern rather than a performance ranking; LLM outputs typically require a combination of execution feedback and traditional fuzzing heuristics to become useful fuzzing artifacts.

In the following sections, we summarize the three mentioned approaches for test case generation in LLM-based fuzzing. Table 4 shows selected papers that used LLM for test case generation.

6.1.1. Fine-Tuning-Based Generation

Fine-tuning is typically adopted when the target domain imposes constraints that are difficult to express reliably in prompts alone, such as structured input formats, solver options, or library-specific API semantics. Based on the general pretrained base model, these approaches further adapt LLMs with code specific data (or fuzzing setting specific data) such as execution traces, historical bug-triggering inputs, or paired examples of source context and valid tests. Earlier unit-test-oriented studies, such as A3Test and the CodeT5-based framework of Hashtroudi et al., follow this pattern by tuning on assertion or method-test corpora to improve the generation of contextually appropriate test cases [74,75].

Reported fine-tuning benefits are easiest to interpret in domains with strong structural constraints. LasT fine-tunes language models for SMT-formula generation and reports 65 genuine bugs in Z3, cvc5, and Bitwuzla, 45 of which were fixed by developers [76]. FuzzGPT applies fine-tuning to bias generation toward edge-case deep-learning programs rather than ordinary, high-frequency inputs [77]. LLAMAFUZZ fine-tunes Llama on paired mutation seeds for structured-data fuzzing and reports 27.19% more branches than AFL++ on real-world programs, with 41 more bugs on average than an evaluated baseline in that study [78]. Similar domain adaptation appears in FuzzCoder, which learns mutation patterns from successful IoT fuzzing histories and reports more than 20% higher code coverage than AFL-family baselines in its evaluation [79], and in CovRL, which combines an LLM mutator with reinforcement learning from coverage feedback and reports 58 security-related bugs in JavaScript interpreters, including 50 previously unknown bugs and 15 CVEs [80].

Despite these results, fine-tuning remains less common than prompting in the broader LLM-based fuzzing literature, mostly because of the cost of fine-tuning. The effectiveness of fine-tuning depends on the availability of target-specific training data and on the cost of maintaining specialized models as formats, APIs, or software versions evolve. For this reason, fine-tuning is mainly used in domains where validity constraints are strict and reusable training traces are available.

Table 4. LLM-based fuzzing techniques by test case generation (TCG). In “Strategy”, FT = fine-tuning, PE = prompt engineering (Z = zero-shot learning, F = few-shot learning, C = chain-of-thought, ToT = tree-of-thought, A = auto-prompting), and Hybrid = LLM combined with traditional analysis, fuzzing, retrieval, or verification components. Multiple strategies are listed when a paper combines them; rare variants are written in full. In “Test Types”, B = black-box testing, W = white-box testing, and G = gray-box testing; rare cross-stage cases are written in full. In “Target”, R means the target is regular software. Otherwise, the type of target system is marked. In “Target Language”, General means the proposed fuzzer applies to multiple languages.

Literature	Task	Strategy	Test Types	Target	Target Language	LLM
[81] Fuzz4All	TCG	PE (A)	B	R	General	GPT-4, StarCoder
[45] LiBro	TCG	PE (F)	B	R	Java	Codex
[74] A3Test	TCG	FT	G	R	Java	PLBART
[75] Hashtroudi et al.	TCG	FT	G	R	Java	CodeT5

Table 4. Cont.

Literature	Task	Strategy	Test Types	Target	Target Language	LLM
[82] AceCoder	TCG	PE (Z)	B	R	General	GPT-3.5
[83] Asmita et al.	TCG	PE (Z)	B	Embedded firmware	C	GPT-4
[84] Ackerman et al.	TCG	PE (C)	G	Parsers	General	GPT-4
[85] ChatUniTest	TCG	Hybrid	W	R	Java	ChatGPT
[86] ChatTester	TCG	Hybrid	W	R	Java	ChatGPT
[87] LLM4Fuzz	TCG	Hybrid	G	Smart contracts	Solidity	GPT-4
[88] ChatFuzz	TCG	Hybrid	G	R	C/C++	ChatGPT
[89] CCTEST	TCG	Hybrid	G	Code completion models	Python	–
[90] SearchGEM5	TCG	Hybrid	G	Simulators	C/C++	GPT-3.5-Turbo
[91] CodaMosa	TCG	Hybrid	G	R	Python	Codex
[92] Xia et al.	TCG	Hybrid	G	R	General	GPT-3.5-Turbo
... (full table in Appendix A)						

6.1.2. Prompt Engineering and Seed Mutation

In the coded test-case-generation entries, prompt engineering is the most frequently reported strategy; this likely reflects the flexibility of off-the-shelf models and the lower cost of avoiding retraining. In this setting, the prompt becomes the main interface through which researchers encode API signatures, documentation, source-code context, protocol traces, or seed examples. The central challenge is not only to generate valid inputs, but also to steer LLM generation toward states, combinations, and corner cases that ordinary code generation would be unlikely to cover. Current prompting methods applied with LLMs include zero-shot learning, few-shot learning, chain-of-thought, and auto-prompting techniques.

Zero-Shot Learning Based Methods

Zero-shot learning refers to prompting methods that do not provide explicit learning examples in the prompts. It enables LLMs to infer and associate based on pretrained knowledge, yielding results by simply inputting task text into the model. To achieve useful generation outcomes, this approach requires designing clear and specific prompts to stimulate the model's accurate understanding of the task.

Zero-shot prompting is most clearly applicable when the target already exposes informative specifications or when the model is likely to have encountered similar code snippets in pretraining datasets. TitanFuzz illustrates this pattern by applying prompting directly to deep-learning-library fuzzing [93]. A similar observation appears in embedded software, where Asmita et al. apply zero-shot prompting to BusyBox in an embedded-firmware setting [83]. More recent work uses zero-shot prompting not only to emit inputs directly but also to recover hidden relations from documentation. OptionFuzz, for example, asks an LLM to infer dependencies among SMT solver options from official documentation, extracting option relationships with 95.23% accuracy and 90.10% recall, reducing the option search space by 70.11%, and detecting 34 unique bugs [94].

Few-Shot Learning Based Methods

Unlike zero-shot learning, few-shot learning provides an LLM with a small number of training examples in the prompts. This lets the model infer task conventions from limited examples with little task-specific training.

Few-shot prompting is especially relevant when input validity depends on protocol state, message layout, or task-specific conventions that are difficult to convey in a single instruction. In protocol fuzzing, ChatAFL uses a small set of example interactions and expected outcomes to expand seed diversity when only natural-language RFCs are available [95]. ChatHTTPFuzz similarly combines example-oriented prompting with protocol-field labeling and code-aware enrichment for IoT HTTP services; it reports 116 vulnerabilities across 16 devices, 70 of which were unique and 23 assigned CVEs [96].

Few-shot prompting has also been used to supply representative bug patterns. Kang et al. report that small sets of bug-reproduction examples can help LLMs construct more reliable test cases than unguided prompting [45], while MSFuzz uses LLM-extracted message syntax to guide seed mutation and reports branch-coverage improvements of 29.30% over AFLNet and 23.13% over ChatAFL [97]. In practice, few-shot examples are useful because they anchor generation in the expected structure of valid inputs.

Chain-of-Thought Based Methods

The Chain-of-Thought (CoT) approach is a prompting method that mimics human reasoning steps. It emphasizes progressively building context within prompts to ensure that generated content maintains logical coherence and contextual relevance. This stepwise guide helps ensure that generated output remains consistent and structurally sound.

CoT is used when generation must be decomposed into intermediate reasoning steps, such as interpreting format specifications, inferring preconditions, or constructing multi-stage test scenarios. As LLM reasoning attracts increasing research interest, CoT-like reasoning has become a standard step in LLM inference. Ackerman and Cybenko apply a staged prompting process to parser fuzzing: the model first interprets a natural-language format description and then derives seed instances from that description, reporting preliminary improvements over a basic mutation fuzzer [84]. The same idea appears in later studies that ask the model to reason explicitly about dependencies or generation stages before generating test cases. SeedMind, for example, uses iterative, progress-aware reasoning to generate test case generators rather than direct input, and reports improvements over earlier LLM-based seed generation approaches [98]. In this line of work, the value of chain-of-thought mainly lies in separating analysis from emission, which can reduce invalid outputs and support coverage-oriented generation.

Automatic Prompt Construction

Another strategy is to automate prompt construction itself. This reduces the manual effort required to tailor prompts to each target and helps compress large amounts of program context into a form that fits within the model's context window. These methods typically employ multiple LLMs. For example, in Fuzz4All, one model distills relevant information into fuzzing prompts for another model, and these prompts are updated throughout the fuzzing loop, based on the feedback from fuzzing campaign and pre-defined prompt templates. Across nine systems and six input languages, Fuzz4All reports higher coverage than language-specific baselines and reports 98 bugs, 64 confirmed as previously unknown [81]. In another example, WhiteFox extends the same idea to white-box compiler fuzzing by separating analysis and generation into two cooperating agents; on three deep-learning compilers, it reports exercising up to eight times more optimizations than prior fuzzers and reports 101 bugs, including 92 confirmed previously unknown

defects [99]. Automatic prompt construction is therefore a practical way to combine broad model knowledge with target-specific context without resorting to model training.

6.1.3. Hybrid Frameworks Combining LLMs with Traditional Fuzzing

As discussed in the background section, a single LLM is not always capable of correctly handling all tasks within the fuzzing workflow. Additionally, practitioners often aim to limit reliance on LLMs or prefer smaller, local models to reduce costs. Consequently, many reported fuzzing systems integrate LLMs within a conventional fuzzing loop rather than positioning them at its boundaries. These hybrid approaches typically combine LLM-based input generation with validation and repair mechanisms, program analysis, coverage-guided mutation, and seed scheduling.

Feedback Validation

Since LLM-generated test cases may contain compilation issues such as syntax errors and semantic errors, verification and remediation of LLM outputs are essential to obtain test inputs suitable for software testing tasks. ChatUniTest adopts a generate-validate-repair workflow in which failing tests are repaired either by rules or by further LLM interaction [85]. ChatTester makes this loop explicit and reports quantitative benefits: compared with default ChatGPT, it produces 34.3% more compilable tests and 18.7% more tests with correct assertions [86]. Similar feedback loops also appear in broader fuzzing systems. Fuzz4All updates prompts using execution outcomes [81], and PILLM uses coverage and source-level execution information to optimize subsequent generations, reporting coverage improvements of 45.7% over Fuzz4All and 14.9% over Fuzzilli on JavaScript engines [100]. These results indicate that feedback is not merely a repair mechanism; it is a way to turn one-shot generation into an iterative search process.

Program Analysis

Hybrid frameworks also rely on program analysis to filter and compress the context sent to the model. This is particularly important when the target contains rich control-flow or data-flow constraints that are too large to expose directly in a prompt. LLM4Fuzz uses source-level information and user-defined invariants to steer smart-contract fuzzing toward high-value regions; its evaluation reports improvements in efficiency, coverage, and vulnerability discovery, and identifies five critical vulnerabilities in real DeFi projects [87]. WhiteFox uses static analysis to infer optimization-triggering requirements before test generation [99], while StatePre employs LLM-guided refinement of protocol state annotations to support stateful protocol fuzzing [101].

In Linux kernel fuzzing, even with abundant documentation and examples, efficiently applying LLMs for direct test case generation is still challenging [102–104]. The primary reason is the kernel's massive and highly complex code base, involving numerous modules, making comprehensive coverage of all functionality difficult. Additionally, the kernel's vast state space makes it difficult for fuzzing tools to effectively explore deep state interactions. Kernel-level fuzzing requires simulating environments close to actual runtime conditions, and constructing and maintaining these high-overhead environments significantly increases testing complexity and time costs. At the same time, the kernel enforces strict input validation logic, often filtering randomly generated inputs and preventing thorough testing of core functionalities. Particularly, many kernel vulnerabilities reside within specific boundary conditions, where conventional fuzzing tools struggle to generate sufficiently precise inputs to trigger them. Therefore, KernelGPT [105] performs static analysis on source code to extract kernel-related operators, kernel functions, parameter definitions, and other information. It leverages the reasoning capabilities of LLMs to generate specific test specifications, thereby supporting kernel fuzzing.

Mutation Guidance

Another established design is to let the LLM complement, rather than replace, conventional mutation. ChatFuzz follows this pattern by generating format-conforming seed variations from existing corpus entries and reports a 12.77% edge-coverage improvement over AFL++ on structured-input programs [88]. SearchGEM5 similarly combines ChatGPT-generated parameterized programs with AFL++-style mutation, increasing gem5 test coverage by more than 1000 lines and exposing 244 mismatches between simulated and expected behaviour [90]. CodaMosa uses pretrained LLMs to break coverage plateaus by supplying new test cases for uncovered regions when conventional search stalls [91]. In these systems, the LLM contribution is easiest to justify where mutation alone struggles: producing inputs that remain valid while still deviating enough to exercise new behavior.

Seed Selection and Scheduling

Once LLMs can generate many plausible test case candidates, prioritizing which ones to execute becomes a design problem. Researchers therefore sometimes choose to use conventional search or statistical scheduling to decide where LLM effort should be spent. PILLM scores candidate test cases using execution feedback [100], ChatHTTPFuzz schedules seed templates with an enhanced Thompson-sampling policy [96], and SearchGEM5 filters LLM outputs through compilation and search-based mutation before execution [90]. These studies suggest that LLMs can support test case generation, but selection pressure still comes from standard fuzzing signals such as validity, coverage growth, state exploration, and crash yield.

6.2. LLM-Driven Defect Detection

Compared with LLM-driven test case generation, LLM-driven defect detection remains a smaller and more heterogeneous line of work. In this stage, the model is rarely used as a standalone bug classifier. More often, it helps derive executable oracles from natural-language requirements, interpret outputs or execution traces, explain suspicious behavior, or validate whether a candidate alarm is exploitable. Table A1 shows that hybrid pipelines are the most frequently reported design pattern in this category, whereas pure fine-tuning approaches are relatively uncommon. Read descriptively, this distribution is consistent with the nature of the task: defect detection depends less on open-ended generation than on combining LLM reasoning with program analysis, retrieval, and runtime feedback.

Conceptually, LLM-driven defect detection can be divided into two parts. The first is oracle generation, where the LLM helps determine what behavior should count as erroneous or security-relevant. The second is defect analysis and verification, where the LLM interprets crashes, responses, traces, or source context to decide whether an observed behavior corresponds to a real vulnerability.

Table 5 lists representative fuzzing methods using LLM-driven defect detection.

6.2.1. Oracle Generation

Test oracles remain a bottleneck in fuzzing because expected behavior is often expressed in documentation, security rules, or domain conventions rather than in formal specifications. LLMs are useful in this setting because they can translate such informal artifacts into executable checks or vulnerability predicates. Among the surveyed defect-detection literature, oracle construction is therefore less about producing ordinary assertions and more about operationalizing requirements that are otherwise difficult to encode.

Table 5. LLM-based fuzzing techniques by defect detection (DD). In “Strategy”, FT = fine-tuning, PE = prompt engineering (Z = zero-shot learning, F = few-shot learning, C = chain-of-thought, ToT = tree-of-thought, A = auto-prompting), and Hybrid = LLM combined with traditional analysis, fuzzing, retrieval, or verification components. Multiple strategies are listed when a paper combines them; rare variants are written in full. In “Test Types”, B = black-box testing, W = white-box testing, and G = gray-box testing; rare cross-stage cases are written in full. In “Target”, R means the target is regular software. Otherwise, the type of target system is marked. In “Target Language”, General means the proposed fuzzer applies to multiple languages.

Literature	Task	Strategy	Test Types	Target	Target Language	LLM
[106] AVD System	DD	FT, PE (C)	G	R	C/C++	DeepSeek-R1, Llama 3
[107] ACBreaker	DD	PE (C)	W	IoT firmware	C	Qwen-2.5-Coder-32B, GPT-4o
[108] ADFEmu	DD	Hybrid	G	Firmware	C	GPT-4
[109] CrawlMLLM	DD	PE (F)	W	Web applications	PHP	GPT-4o
[110] PROMFUZZ	DD	Hybrid	G	Smart contracts	Solidity	GPT
[111] GLEIPNIR	TCG, DD	Hybrid	G	Windows remote IPC clients	IPC return values	GPT-4
[112] EAGLEYE	TCG, DD	Hybrid	B	IoT web interfaces	HTTP requests	GPT-3.5-Turbo
[113] AgentFuzz	DD	Hybrid	G	LLM-based agents	General	GPT-4o
[114] OBsmith	DD	Hybrid	–	JavaScript obfuscators	JavaScript	GPT-5, Gemini 2.5 Pro, Claude 4, Qwen3, Grok 4
[115] ORFuzz	DD	Hybrid	B	LLM safety/over-refusal testing	Natural language prompts	Qwen2.5-72B-Instruct
[116] HFFuzzer	DD	PE	B	Code LLMs	Natural language prompts	GPT-4o
[117] Cohen	TCG, DD	PE (A), Hybrid	B	Agentic AI browsers	Web pages/prompts	–
... (full table in Appendix A)						

Specification-Derived Oracles

When high-quality natural-language specifications are available, LLMs can help convert them into machine-checkable conditions. Jiang et al. use Javadocs to generate executable oracles for checking whether client code conforms to the requirements of Java library APIs; their study reports that 98.8% of the generated oracles compile and 96.4% accurately reflect the intended properties [118]. DIVAS applies the same general principle in a security setting, using LLM reasoning within an end-to-end framework for SoC security analysis and policy-based protection [119]. These studies indicate that LLM assistance is easiest to justify for oracle construction when the expected behavior already exists in descriptive form but has not yet been translated into an executable check.

Differential, Dynamic, and Formal Oracles

When explicit specifications are incomplete, recent systems construct oracles indirectly. ACBreaker extracts device-specific firmware semantics and combines LLM reasoning with differential analysis to identify broken access control, reporting 39 previously unknown vulnerabilities across 11 devices [107]. In this design, the oracle is not a single generated assertion; it is a validation procedure that checks whether different interfaces, privilege levels, or execution contexts behave in a security-relevant way.

Formal and constraint-based methods provide another grounding mechanism. AD-FEmu combines concolic execution with LLM-assisted semantic emulation for firmware fuzzing [108]; SyzForge combines static analysis, symbolic execution, Syzkaller fuzzing, and LLM-driven refinement to generate Linux syscall specifications [120]; and HLPFUZZ uses

LLM-based constraint solving inside a hybrid fuzzing framework with explicit constraint prioritization [121]. These systems suggest that formal methods are not merely alternatives to LLMs. They can turn model-generated hypotheses into path conditions, constraints, or specifications that fuzzing can test. Their limitation is also clear: formal abstractions are costly and incomplete, so they strengthen but do not replace executable validation.

6.2.2. Defect Analysis and Verification

Once fuzzing produces crashes, anomalous outputs, or suspicious paths, the central problem shifts from exploration to interpretation. LLMs can summarize context, explain possible causes, infer missing preconditions, and prioritize findings, but these are hypothesis-forming activities. The decisive evidence still comes from replay, instrumentation, differential checks, scanners, symbolic or concolic evidence, or expert review.

Fine-Tuning and Domain Adaptation

Fine-tuning appears mainly in narrow domains where domain-specific code patterns and vulnerability labels are available. The AVD System combines adapted reasoning models with static analysis, fuzz-target extraction, iterative compilation, and log summarization for vulnerability-oriented analysis in C codebases [106]. PLC LLM Security follows a more explicitly supervised path for programmable-logic-controller code, fine-tuning CodeLlama, Qwen2.5-Coder, and StarCoder2 on Structured Text vulnerability data and reporting gains in binary and multi-class classification [122]. These studies suggest that domain adaptation may strengthen vulnerability-oriented analysis, but its applicability depends on the availability of curated bug histories and target-specific training data.

Prompted Analysis

Prompting is useful when the model is asked to interpret bounded evidence rather than decide vulnerability truth alone. AutoSD treats debugging as an iterative process using debugger feedback [123]; PrediQL interprets GraphQL responses, error messages, and status codes as part of vulnerability detection [124]; and CrawlMLLM combines page-state reasoning with audit tools, reporting a higher vulnerability count than its baselines in benchmark and real-world web applications [109]. CrossProbe shows a related transfer pattern in deep-learning frameworks, where LLM guidance reduces generation iterations and helps detect mirrored bugs [125]. These examples are easier to check than generic prompting because the model output is tied to observable program behavior.

Hybrid Detection Pipelines

Hybrid systems can provide more directly checkable evidence when they separate semantic guidance from validation. UniFuzz combines static and semantic analysis, audit-report retrieval, and agent coordination for smart contracts, reaching up to 90.99% code coverage and reporting branch-coverage gains of up to 6.6 percentage points [126]. CrossGuard reports improvements in exploration of path-sensitive cross-contract flaws, yet it also shows a useful failure case: numeric edge cases remain difficult for LLM-guided reasoning [127]. In firmware and systems settings, ADFEmu, GLEIPNIR, EAGLEYE, and browser-agent fuzzing all follow the same pattern: the LLM constructs or interprets high-value context, while execution or differential behavior provides the vulnerability signal [108,111,112,117]. HyFuzz reports up to 10× faster detection and a 35% improvement in crash discovery by combining CVE correlation with adaptive fuzzing [128].

Taken together, these studies show that LLMs contribute to defect detection primarily as semantic reasoning layers between raw fuzzing outputs and final vulnerability judgments. The evidence is easiest to assess when the final claim is grounded in replayable behavior, differential validation, symbolic or concolic constraints, or independent tools. It

is harder to assess when the model is asked to infer vulnerability truth from source text or logs without such grounding.

6.3. LLM-Driven Post Processing

Post-processing is the least populated stage in the current LLM-based fuzzing literature. In contrast to test case generation and defect detection, the model is not primarily used to steer exploration of the search space. Instead, it is used after fuzzing has already produced crashes, anomalous outputs, or candidate vulnerabilities. The main tasks in this stage are result interpretation, crash triage, patch generation, and remediation planning. Table A1 also shows that many entries in this category overlap with defect detection or test case generation, which suggests that post-processing is rarely implemented as a fully independent module. Most existing work therefore combines LLMs with conventional analysis, validation, or repair pipelines rather than relying on the model alone.

Three patterns recur. First, LLMs interpret outputs and help distinguish meaningful anomalies from benign deviations. Second, fine-tuned or prompted models generate candidate patches. Third, recent systems embed the model into remediation workflows that include exploit replay, fuzzing feedback, static analysis, crash deduplication, runtime monitoring, or deployment support. The analytical issue is that post-processing outputs can be made plausible before they are trustworthy: a cluster may group different bugs, an explanation may identify the wrong cause, and a patch may resemble a correct fix while leaving the exploit intact.

Table 6 lists representative fuzzing techniques using LLMs in post-processing.

Table 6. LLM-based fuzzing techniques by post-processing (PP). In “Strategy”, FT = fine-tuning, PE = prompt engineering (Z = zero-shot learning, F = few-shot learning, C = chain-of-thought, ToT = tree-of-thought, A = auto-prompting), and Hybrid = LLM combined with traditional analysis, fuzzing, retrieval, or verification components. Multiple strategies are listed when a paper combines them; rare variants are written in full. In “Test Types”, B = black-box testing, W = white-box testing, and G = gray-box testing; rare cross-stage cases are written in full. In “Target”, R means the target is regular software. Otherwise, the type of target system is marked. In “Target Language”, General means the proposed fuzzer applies to multiple languages.

Literature	Task	Strategy	Test Types	Target	Target Language	LLM
[129] Abtahi et al.	DD, PP	Hybrid	–	Firmware	C/C++	GPT-4
[130] AePPollo	PP	Hybrid	–	Web applications/services	JavaScript	Qwen-Max-2025-01-25
[131] CHEMFUZZ	PP	PE (Z)	G	R	C++, Fortran	GPT-3.5, etc.
[132] Mashhadi et al.	PP	FT	G	R	Java	CodeBERT
[133] CodeRover-S	PP	PE	G	R	General	GPT-4o
[134] InferFix	PP	FT	G	R	General	Codex
[135] Pearce et al.	PP	PE (F)	G	R	C/C++	Codex etc.
[136] SAN2PATCH	PP	PE (ToT)	–	R	C/C++	GPT, Claude, Gemini
[137] GPTrace	PP	Hybrid	–	R	–	Text-embedding-3-large
[138] Chen et al.	PP	Hybrid	–	R	General	–
[139] LAC2R	PP	Hybrid	–	C-to-Rust translation	C, Rust	GPT-4o, DeepSeek V3
[140] RustAssure	PP	PE	–	C-to-Rust translation	C, Rust	–
... (full table in Appendix A)						

6.3.1. Fine-Tuning-Based Program Repair

Fine-tuning in post-processing is used mainly for automated program repair. Instead of learning how to produce test cases, the model learns how to map defective

code to candidate fixes. This setup is most defensible when paired bug-fix corpora are available, the repair objective is localized, and the generated patch is checked by tests or static-analysis evidence.

Mashhadi and Hemmati illustrate this line of work by fine-tuning CodeBERT for automated repair of simple Java bugs [132]. InferFix extends the same idea to security and performance bugs by combining retrieval of semantically similar bugs with a fine-tuned generator. Its evaluation reports a top-1 repair accuracy of 65.6% for C# and 76.8% for Java, and the system is described as part of an end-to-end workflow that includes bug detection, classification, and validation in a continuous-integration setting [134]. These studies suggest that fine-tuning is better supported when repair targets are well represented in historical repositories and when the repair pipeline can rely on metadata such as bug type or retrieved analogues.

However, repair accuracy is not the same as security correctness. Fine-tuned repair models may learn recurring edit patterns rather than the underlying exploit condition, and datasets are expensive to curate, language-dependent, and biased toward common bug shapes. As a result, post-fuzzing repair has moved more quickly toward prompting and hybrid workflows than toward maintaining task-specific repair models.

6.3.2. Prompt-Based Post Processing

In the surveyed post-processing entries, prompting is reported more often than pure fine-tuning. This likely reflects that prompting allows the model to reason over crash reports, exploit inputs, code snippets, or natural-language vulnerability descriptions without retraining. In this setting, the LLM is typically used to synthesize candidate patches, explain why a failure occurred, or produce more actionable summaries for developers.

Pearce et al. provide an early security-oriented example by examining zero-shot vulnerability repair with LLMs [135]. Their work shows that prompt-only repair is plausible, but it also illustrates why validation is essential. Xia and Zhang report that ChatGPT fixes 162 out of 337 bugs at low monetary cost [141]; this kind of result is useful, but it should be interpreted through the scope of the bug set and validation protocol. For fuzzing-derived security patches, Zhang et al. argue that patch correctness must be assessed through exploit execution rather than code-similarity metrics alone [133]. SAN2PATCH makes the validation problem explicit by decomposing sanitizer-log repair into vulnerability comprehension, fault localization, fix strategy, and patch generation [136]. The recurring lesson is that textual plausibility is a weak signal: a patch may look like a known fix and still fail on the triggering input.

Prompting is also used for interpretation rather than direct repair. CHEMFUZZ treats the LLM as part of the result-analysis stage, using it as a test oracle to analyze anomalous computation outputs in quantum chemistry software; the system reports 40 unique bugs with 17.4% code coverage [131]. In this setting, the value of the LLM lies less in generating code and more in translating complex domain outputs into decisions that can guide reporting and follow-up analysis.

6.3.3. Hybrid Post-Fuzz Analysis and Remediation

A visible trend in post-processing is to place the LLM inside a larger remediation workflow. These systems combine model reasoning with exploit validation, dynamic execution, crash clustering, static analysis, or defense deployment. The LLM is therefore one component in a loop that determines which findings are distinct, which are actionable, and how they should be fixed or mitigated.

One recurring task is triage. GPTrace addresses crash deduplication by using LLM embeddings to compare crash-related artifacts and cluster failures that stem from the

same underlying bug. Evaluated on more than 300,000 crashing inputs from 14 targets, it reports improvements over hand-crafted stack-trace comparison and other deduplication methods [137]. This is practically important because raw crash volume is a poor measure of progress: the developer cost depends on distinct root causes, not the number of crashing inputs. The risk is that embedding similarity may capture symptoms rather than fault identity, so triage systems still need labels, replay evidence, or downstream confirmation.

Another recurring task is end-to-end remediation. FuzzingBrain CRS combines vulnerability detection and patching in an integrated cyber-reasoning system; in the AIxCC final round, it autonomously discovered 28 vulnerabilities, including six zero-days, and patched 14 of them in real-world C and Java projects [142]. A related firmware workflow validates LLM-generated code with fuzzing, static analysis, and runtime monitoring before iterative patching, reporting a 92.4% vulnerability-remediation rate and improved compliance and runtime-security metrics [129]. These results are easier to interpret than isolated patch-generation metrics because they include validation steps, but they also depend on the coverage of the tests, monitors, and threat models used in the loop.

Post-processing is also beginning to expand beyond patch generation into mitigation and response. ExpFuzz links vulnerability identification with dynamic defense deployment for modern web applications [143], while Cyber Xipher uses an LLM to produce contextual mitigation reports alongside adaptive web fuzzing [144]. Although these systems are early-stage systems compared with patch-generation work, they suggest that LLMs may become useful not only for fixing individual bugs but also for helping operational teams prioritize and deploy defensive actions after fuzzing has already identified a weakness.

Overall, current LLM-based post-processing work is narrower than test case generation and defect detection, but it addresses a practical bottleneck in fuzzing workflows: turning findings into developer-actionable outcomes. The literature so far suggests that LLM contributions are easiest to justify here when they are coupled with validation signals such as exploit replay, fuzzing outputs, static analysis, or runtime monitoring. Their contribution is less about replacing classical post-processing and more about reducing the manual effort needed for triage, repair, and remediation.

6.4. Summary of Current LLM-Based Fuzzing Techniques

This subsection summarizes the surveyed literature across four dimensions: task and application strategy, target system and language, testing type, and model family. It also presents a consolidated view of reported performance and explains the difficulty of direct comparison caused by heterogeneous metrics and experimental setups. The observations are derived from Table A1 using the same coding scheme, and the resulting counts provide a structured map of the literature without constituting a statistical meta-analysis of effectiveness. Because several studies span more than one task or strategy, category counts are multi-label and may sum to more than the yearly total.

Figure 3 summarizes the coded corpus by year for 2022–2025 and shows rapid growth after 2023, together with uneven research attention across stages. We use 2022 as the lower cutoff because no relevant LLM-based fuzzing records were found before that point in our reviewed data, and we exclude 2026 because the 2026 corpus window was incomplete at the time of collection. Current methods remain concentrated on test case generation, where the LLM is often used as a one-shot content generator. More integrated uses within fuzzing workflows, including agentic workflows and reasoning-based designs, remain limited. Applications to defect detection, post-processing, driver generation, and workflow-level automation become more visible in recent work. A similar pattern appears in integration strategies: prompt engineering is common throughout, and hybrid and agentic systems

become more prominent as studies move from one-shot generation toward feedback-guided fuzzing loops.

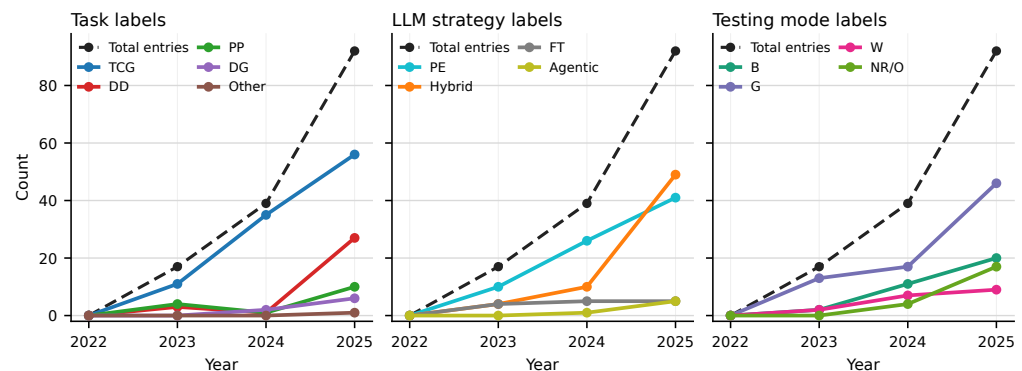


Figure 3. Yearly distribution of the coded LLM-based fuzzing corpus for 2022–2025. Counts are derived from the coded corpus underlying Table A1; year denotes the first scholarly record available during screening. Task and strategy labels are multi-label, so their yearly sums may exceed the total number of coded entries for that year. The dashed line in each panel shows the yearly total. The trend starts at 2022 because no relevant LLM-based fuzzing records were found before that cutoff in the reviewed data, and it ends at 2025 because 2026 was incomplete at the time of collection. In “Testing mode labels”, NR/O denotes entries with unreported black-box, gray-box, or white-box mode, as well as workflow-level or other entries.

6.4.1. By Tasks and Application Strategy

The clearest pattern in the current literature is the concentration of work on test case generation. In Table A1, 107 entries carry a test case generation label, alongside 32 for defect detection, 16 for post-processing, and 9 for fuzz-driver or harness generation. One additional row is tagged as workflow automation. Because some studies span multiple stages, these counts overlap. Interpreted descriptively, they show that current LLM use is concentrated in front-end test case generation, with fewer studies on in-process or post-process tasks like program diagnosis or repair. This interpretation remains descriptive, since the count alone does not establish that generation methods are technically more mature in every target domain. Even so, the pattern aligns with current LLM strengths: they can convert partial specifications, source context, and examples into structured inputs, and they still require external validation for bug triage, vulnerability confirmation, or program repair.

The strategy labels reinforce this picture. Counting the appendix labels yields 82 prompt-related entries, 65 hybrid entries, 16 entries involving fine-tuning, and 6 agentic entries, again with overlap because several systems combine multiple strategies. In this corpus, prompting is the most frequently reported choice; this likely reflects its low deployment cost and ease of adaptation to new targets. Hybrid systems are also frequently reported when LLM generation is combined with compilation checks, coverage feedback, static analysis, retrieval, or exploit validation. Fine-tuning appears less frequently and is often reported in domains with strong validity constraints and reusable historical corpora, such as deep-learning libraries, SMT solvers, JavaScript interpreters, and IoT inputs.

Within prompt-based work, few-shot prompting appears more often than pure zero-shot generation, and chain-of-thought prompting and automatic prompt construction also recur in settings with complex structural constraints. Tree-of-thought prompting appears mainly as a specialized program repair strategy in post processing. Zero-shot prompting remains useful when the target exposes a clear specification, API description, or documentation trail; richer prompt designs are more common when the system must recover semantics from limited or fragmented context.

6.4.2. By Target System and Target Language

LLM-based fuzzing has been applied to a broad set of targets, including conventional software, network protocols, deep-learning frameworks, libraries, IoT devices, compilers, Linux kernels, smart contracts, and firmware. In Table A1 from Appendix A, regular software remains the largest single category with 40 entries, followed by visible concentrations in network protocols (11), deep-learning frameworks and IoT devices (9 each), other library targets (8), compilers (6), and Linux kernels and smart contracts (5 each). Together, these concentrations suggest that LLMs are often studied where input spaces are structurally rich, documentation is available, and manual generator design would require substantial domain expertise.

The target-language distribution follows the same pattern. General multi-language approaches account for 30 entries, and the most common concrete language families are C/C++ and C, which together form the largest group in the table. Binary or protocol-message inputs also form a substantial group, followed by Java, Python, JavaScript, and Solidity. These categories show that LLM-based fuzzing extends beyond source-code synthesis and often operates over message formats, configuration spaces, API sequences, or mixed source-and-binary contexts, especially in protocol, firmware, and embedded domains.

The predominance of C-family targets reflects common fuzzing practice. Many fuzzing benchmarks, systems programs, compilers, kernels, and embedded applications are implemented in C or C++, and these targets often expose low-level constraints that benefit from semantic guidance. The presence of Java, Python, JavaScript, and Solidity extends this picture to higher-level software ecosystems where APIs, stateful behaviors, and documentation are especially important.

6.4.3. By Testing Types

Among the 134 entries in Table A1 from Appendix A, that explicitly report a black-box, gray-box, or white-box mode, gray-box fuzzing is the most frequently reported setting: 82 entries (61.2 %) are gray-box, alongside 33 (24.6 %) black-box and 19 (14.2%) white-box entries. Read descriptively, this distribution may reflect the practical fit between gray-box feedback and LLM-based systems, since gray-box fuzzing allows the model to use partial structural cues, execution feedback, or selected program context without requiring full symbolic reasoning over large codebases.

Black-box designs are more common when source code is unavailable or when documentation, protocol traces, or observed interface behavior provide enough information to guide generation. White-box designs appear mainly when source analysis can be distilled into compact prompts or auxiliary representations, as in compiler fuzzing, unit-test generation, or documentation-guided oracle construction. Taken together, these modes suggest that many LLM-based fuzzing systems benefit from targeted context that is sufficient for semantic guidance and compact enough for reliable use. Gray-box workflows often provide this balance.

This balance may also help explain why hybrid designs are frequently reported in the literature. Once execution feedback, static analysis, or retrieved context is introduced, the LLM operates within a feedback-guided testing loop. This architecture allows systems to use LLM reasoning while retaining the efficiency and validation mechanisms of conventional fuzzing.

6.4.4. By LLMs Applied in Fuzzing

Among the 156 entries in Table A1, 136 specify at least one model and 20 are coded as null because they report no concrete model or are not designed around a specific model. Counting papers by model family, with overlap allowed when a study lists multiple models,

OpenAI-associated GPT/ChatGPT/Codex labels are the most frequently reported model family in 83 papers. Within this group, GPT-4 non-4o variants appear in 32 papers and GPT-4o appears in 17 papers; when these labels are aggregated, GPT-4-family models appear in 49 papers, and GPT-3.5-family models appear in 15 papers. These figures indicate that GPT-style and OpenAI-associated models remain the most frequently reported models in the surveyed literature.

Non-OpenAI families are also widely represented. Llama/CodeLlama/TinyLlama appear in 16 papers, DeepSeek in 11, Qwen in 10, and Claude in 9. StarCoder/StarCode, Mistral/Mixtral, and GLM/ChatGLM/Zhipu each appear in 5 papers; Gemini appears in 4. The same table also shows that 34 of the 136 papers that specify a model (25.0 %) list more than one model, indicating that a substantial minority of studies combine multiple models within a single workflow. Overall, the distribution suggests that LLM-based fuzzing relies on a mixed model ecosystem in which GPT-style systems are the most common choice, and open-weight and code-oriented families are also used across a broad range of tasks.

Figure 4 summarizes the same model column by normalized model label. The plot shows the twenty most frequent reported labels individually, groups less frequent reported labels into “Other reported”, and keeps null-coded entries as a separate bar.

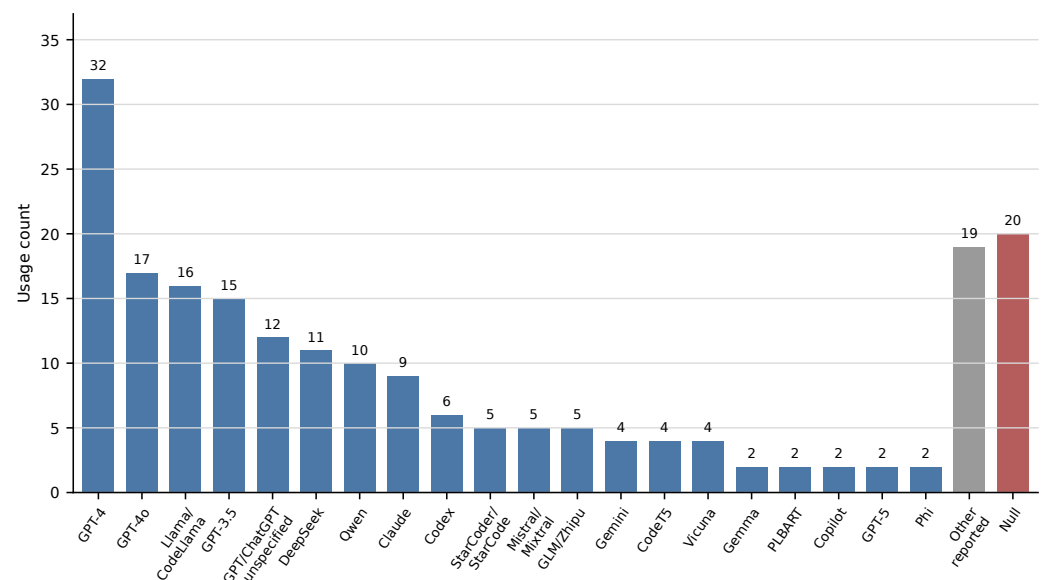


Figure 4. Distribution of reported LLM usage in the coded corpus. Counts are derived from the LLM column in Table A1. A paper may contribute to more than one reported model label when multiple models are listed. “Other reported” aggregates lower-frequency reported model labels, and “Null” denotes entries with no concrete model or no design around a specific model.

6.4.5. Representative Reported Performance

Corpus-level counts describe where LLMs are being applied; they do not directly measure technical effectiveness. For this reason, Table 7 summarizes representative quantitative results reported by individual studies. Throughout the table, reported metrics should be read within each study’s own targets, budgets, baselines, and validation protocol. Because these studies use different targets, budgets, baselines, models, and metrics, the table is unsuitable as a global ranking of methods. Its purpose is to show the kinds of evidence that currently support claims about LLM-based fuzzing, including validity improvements, coverage gains, bug discovery, vulnerability confirmation, crash triage, and repair outcomes.

Table 7. Representative numerical results reported by LLM-based fuzzing studies. The figures are headline metrics from heterogeneous targets, baselines, and evaluation protocols, and should not be read as a method ranking.

Study	Metric Type	Reported Metric(s)	Target System	Baseline/Scope
LLAMAFUZZ [78]	Coverage/bugs	+27.19% branch coverage; higher bug yield than an evaluated baseline, as reported in that study	Structured-input programs	vs. AFL++ and other fuzzing baselines
ChatTester [86]	Validity	+34.3% compilable tests; +18.7% correct assertions	Java unit tests	vs. default ChatGPT generation
OptionFuzz [94]	Constraints/bugs	95.23% accuracy; 90.10% recall; 70.11% search-space reduction; 34 unique bugs	SMT solvers	documentation-guided option exploration
MSFuzz [97]	Coverage	+29.30% branch coverage over AFLNet; +23.13% over ChatAFL	Network protocols	vs. AFLNet and ChatAFL
PILLM [100]	Coverage	+45.7% coverage over Fuzz4All; +14.9% over Fuzzilli	JavaScript engines	source-level and coverage-feedback setting
ChatHTTPFuzz [96]	Vulnerabilities	116 vulnerabilities; 70 unique; 23 CVEs	IoT HTTP services	across 16 devices
ACBreaker [107]	Vulnerabilities	39 previously unknown vulnerabilities	IoT firmware	across 11 devices
UniFuzz [126]	Coverage	up to 90.99% code coverage; up to +6.6 percentage-point branch coverage	Smart contracts	vs. evaluated smart-contract tools
HyFuzz [128]	Detection/crashes	up to 10× faster detection; +35% crash discovery	Network services	vs. baseline fuzzers
GPTrace [137]	Crash triage	more than 300,000 crashing inputs; reported deduplication-quality improvement	Crash deduplication	across 14 targets
FuzzingBrain CRS [142]	Detection/repair	28 vulnerabilities; 6 zero-days; 14 patched	Real-world C and Java projects	AIxCC final-round setting

The table shows why direct ranking is misleading. Test-generation studies often report validity, coverage, or bug yield; defect-detection studies report vulnerability counts, detection speed, coverage, or CVEs; post-processing studies report deduplication quality or patch outcomes. These metrics sit at different distances from the final security claim. Coverage growth is useful but does not guarantee vulnerability discovery; a vulnerability count is meaningful only when reports are validated and deduplicated; and a patch metric is credible only when exploit replay, regression tests, or equivalent checks are included. The numerical evidence therefore supports a cautious conclusion: LLM contributions are easiest to interpret when their outputs are embedded in a validation loop and harder to interpret when the reported metric mainly measures textual plausibility or unverified model judgment.

7. Limitations of Current LLM-Based Fuzzing Techniques

The preceding section shows that LLMs can assist test generation, defect detection, and post-fuzz analysis. However, challenges of current LLM-based fuzzing still remain. In this section, we analyze those limitations following the same stage-wise structure as the review: we evaluate limitations in input and harness generation, defect detection, and

post-processing. We then discuss cross-cutting issues surrounding computational cost, validation, model turnover, and reproducibility.

7.1. Limitations in Input and Harness Generation

The input-and-harness generation stage is the largest part of the current coded literature, but it also exposes the basic gap in LLM-based fuzzing: models often produce plausible artifacts that miss meaningful target behavior. This creates a validity hierarchy. Syntax, compilation, coverage growth, and confirmed bugs measure distinct levels of evidence. Reported gains therefore depend on which level is measured. Seed-generation work reports strong target and model dependence [16,22]. Structured-input, driver, and harness studies show this issue through build and execution feedback: shallow checks catch syntax failures, then target-specific validation is needed for semantic fit [29,88,145,146]. The recurring limitation is the absence of reliable semantic filters that connect generated artifacts to target behavior.

Fine-tuning addresses semantic fit by training on domain-specific fuzzing traces. The evidence shows benefits in constrained domains: LasT reports 65 genuine SMT-solver bugs, LLAMAFUZZ reports branch-coverage and bug-finding gains on structured-data targets, and CovRL reports 58 security-related JavaScript-engine bugs [76,78,80]. Related domain adaptation appears in FuzzGPT and FuzzCoder for deep-learning-library and IoT fuzzing [77,79]. The limitation follows from this design choice. The model depends on target-specific corpora, which are costly to curate and often tied to particular APIs, formats, versions, or bug distributions. A fine-tuned model can learn repeated mutation patterns and still miss the full constraints of a new target. As targets evolve, training data and models need maintenance. This limits transfer and complicates cross-study comparison.

Prompting and auto-prompting avoid retraining, so the main risk moves to prompt construction, context selection, and model stability. Zero-shot and few-shot methods require useful target context, such as documentation, source snippets, examples, or protocol traces [83,93,95,96]. Chain-of-thought and auto-prompting separate analysis from emission, which can improve validity and coverage [81,84,98,99]. Their reliability depends on the retrieved or summarized context. Context compression can remove rare preconditions and boundary cases, which are often central to fuzzing. Closed or rapidly changing model interfaces add instability: a prompt that works for one snapshot can change validity or diversity on another.

LLMs are useful when they recover relations that are hard to encode manually, such as API usage, option dependencies, state transitions, or syscall constraints. These hints remain underconstrained until checked through execution feedback, retrieval, or constraint-based evidence [17,94,147]. Stateful and system targets make this dependency especially visible. Network protocols, firmware, kernels, and embedded systems require valid state transitions, environment models, peripheral behavior, or syscall specifications before generated inputs can reach meaningful code paths [105,108,120]. Well-supported studies combine model-generated semantic hints with deterministic checks for executability, scheduling, coverage, and crash yield. Future evaluations should report this evidence ladder explicitly: syntactic validity, semantic validity, coverage, confirmed bugs, and budget.

7.2. Limitations in Defect Detection

Defect detection has a high evidentiary bar because it makes a claim about faulty or vulnerable behavior. The surveyed evidence gives limited support for standalone LLM vulnerability detection: zero-shot detection can be weak, and fine-tuning may overfit [23]. A recurring pattern in Table A1 is role separation. LLMs help interpret context, rank paths, or produce candidate PoCs. Final vulnerability judgments require executable evidence. Detec-

tion systems should therefore be evaluated by the evidence chain they create: reachability, trigger conditions, and confirmation.

Oracle construction remains the main bottleneck. Documentation can support executable oracle generation. Jiang et al. report that Javadoc-derived oracles compile in 98.8% of cases and accurately reflect intended properties in 96.4% of cases [118]. The limitation appears when expected behavior remains implicit in developer knowledge, security policy, or domain convention. In those cases, plausible explanations leave reachability, exploitability, and policy violation unresolved. The missing object is a checkable behavioral claim. LLMs can translate requirements or access-control rules into candidate checks. Ambiguous target behavior still requires a policy, specification, or expert decision.

Evidence assembly is also underdeveloped. Fuzzing may expose crashes and other anomalies. Defect detection requires a link from each observation to root cause and impact. Prompted analysis is useful when it operates over bounded evidence, such as source slices, logs, page states, GraphQL responses, firmware-derived access-control facts, or differential traces [107,109,124,125]. Ungrounded output often summarizes symptoms. Fault identity, exploitability, and security relevance remain unresolved.

Hybrid detection pipelines address this gap by placing LLMs around evidence-producing components. Dynamic analysis, differential testing, symbolic methods, concolic execution, and static analysis supply replayable or checkable evidence [107,108,120,121,127,148]. Their value lies in turning model suggestions into claims that can fail under replay or analysis. Formal abstractions can be incomplete, and symbolic execution can be expensive. These components still address the key limitation of prompt-only detection.

Defect-detection results are also difficult to generalize because the target domains are heterogeneous. Table A1 includes smart contracts, firmware, web applications, GraphQL APIs, network services, LLM agents, code models, obfuscators, and over-refusal testing. Each domain defines its own oracles, threat models, and confirmation standards. Current results therefore support a cautious conclusion. LLMs are useful semantic reasoning layers for detection. Vulnerability truth remains a property of independently validated program behavior. CrossGuard illustrates a remaining gap: LLM-guided exploration improves path-sensitive smart-contract testing, but numeric edge cases remain difficult [127].

7.3. Limitations in Post Processing

Post-fuzz processing is valuable because it reduces human triage and repair effort. It is also sparsely represented in the current literature. Table A1 shows that post-processing entries occupy a small part of the coded corpus. This stage carries a high risk of false confidence: explanations, clusters, exploit sketches, mitigation reports, and patches can look actionable before validation. This failure mode is specific to post-processing. Here, the output is often readable and developer-facing, so textual plausibility can be mistaken for evidence.

Crash clustering and result interpretation require validation beyond similarity. Trace, log, stack, and embedding similarity can reduce report volume. They may also group crashes by surface symptoms and miss root cause. Crash clustering should therefore demonstrate shared fault behavior or downstream confirmation; trace or embedding similarity gives only a limited proxy [137]. Domain-specific interpretation has a related limitation. An LLM can help analyze scientific outputs, sanitizer logs, or exploit traces. The resulting explanation remains a hypothesis until checked against replay, differential evidence, or expert review [131,136].

Repair is a security-sensitive post-processing task. Fine-tuned repair models benefit from historical bug-fix data. Such data are language-dependent, expensive to curate, and biased toward common localized fixes [134]. Prompt-based repair supports target-specific context. Surface success can be misleading: a patch may compile, resemble a reference fix, or satisfy a benchmark metric and still leave the triggering exploit intact. Repair evaluation should therefore show exploit failure after the patch and acceptable regression behavior [133,135]. Textual plausibility and code similarity only provide supporting signals.

Reliable remediation therefore needs an evidence loop: reproduce the failure, localize the fault, validate the patch, run regressions, and retain audit evidence. LLMs can coordinate parts of this loop. Assurance comes from independent checks such as replay, sanitizers, static analysis, runtime monitoring, and CI [129,142]. The analytical risk is attribution. A generated patch may remove a crash symptom, leave the root cause, introduce a regression, or weaken a security invariant. Patch suggestions become security claims only after behavioral validation. Existing post-processing studies also rarely measure maintainability outcomes such as developer acceptance, time to fix, false deduplication cost, or long-term regression risk.

7.4. Cost, Evaluation, and Reproducibility

7.4.1. Computational Cost

Computational cost is a methodological limitation as well as an implementation expense. LLM-based fuzzing adds a priced inference loop to a testing technique whose strength normally comes from cheap, high-frequency execution. Zhang et al.'s fuzz-driver generation study evaluates 736,430 generated drivers, reports 0.85 billion tokens and more than \$8000 in token charges, and conducts 3.75 CPU-years of fuzzing experiments [25]. These numbers indicate that reported benefits may depend on a large sampling and filtering budget. Many candidates must be generated and filtered before useful artifacts are found. Cost therefore affects the interpretation of effectiveness. When studies omit token use, inference latency, hardware, filtering cost, and fuzzing time, the contribution of guidance and sampling budget becomes unclear.

Cost also creates an architectural mismatch. Semantic-Aware Fuzzing reports integration problems caused by mismatched GPU-intensive LLM workloads and CPU-intensive fuzzing workloads, including latency and throughput bottlenecks [149]. Conventional fuzzers can consume feedback at high CPU speed. LLM-assisted workflows add prompt construction, model scheduling, context-window limits, batching decisions, and model-response variability to the loop. This weakens tight feedback cycles and makes real-time guidance difficult to scale. The limitation is especially important for reproducibility: model prices, rate limits, hardware availability, and vendor-side model changes can alter the practical feasibility of a technique even when the algorithmic description remains unchanged. Evaluations should therefore report cost normalized by outcome, such as valid artifact, coverage gain, confirmed bug, or triage decision. Computing should be treated as part of the method.

7.4.2. Validation and Grounding

Across all stages, LLM outputs should be treated as hypotheses until checked by executable or independently verifiable evidence. For generation, this means compilation, parser acceptance, coverage feedback, and semantic checks. For defect detection and post-fuzz processing, it means replayable crashes, PoCs, differential evidence, regression tests, or expert review. This is also where hallucination risk becomes operationally important: a model may invent constraints, misread logs, overstate exploitability, or explain a crash without evidence. Formal and constraint-based methods can reduce the gap by supply-

ing invariants, path conditions, or checking obligations [17,108,120,121]. The main trust problem is the absence of a reliable path from model output to reproducible claims about program behavior.

7.4.3. Model Turnover

Rapid model release cycles make single-model conclusions fragile because the evaluated object may disappear or change before the result can be reproduced. A method that depends on a particular model's instruction following, context length, refusal behavior, code knowledge, or tool-use policy may lose effectiveness when the provider updates the model or when another laboratory uses a different snapshot. Vendor reports on newer OpenAI and Anthropic systems describe gains in coding, tool use, and defensive cybersecurity workflows [150–152]. These reports provide useful context. Task-level fuzzing evaluation remains necessary. The limitation is that model capability, prompt design, retrieval context, validator design, and budget are often entangled. Evaluations should therefore separate the reusable workflow from the particular model snapshot and report enough interface details to attribute results to method design, model snapshot, and budget.

7.4.4. Agentic Reproducibility

Agentic fuzzing makes the run trajectory part of the experimental object. Recent agentic and cyber-reasoning systems show why the model may choose context, call tools, revise hypotheses, vary budgets, and stop under different evidence [113,117,142]. Two runs with the same target and nominal model may therefore diverge for reasons unrelated to the underlying fuzzing idea.

This raises reporting requirements. Final coverage or bug counts are insufficient because they hide the sequence of choices that produced the result. Agentic studies should log calls and prompts, generated artifacts, tool outputs, budgets, stopping rules, and environment state. They should also distinguish decisions made by the model from those imposed by the surrounding harness, scheduler, validator, or human operator. Without these logs, improved results are hard to attribute. They may reflect model capability, fuzzing logic, tool orchestration, or simply run length.

Overall, the durable limitation is the lack of stable interfaces between probabilistic model outputs and checkable fuzzing evidence. Progress should therefore be judged as an evidence-construction problem: the surrounding system must turn model outputs into valid inputs, reproducible defects, trustworthy repairs, and comparable experimental results.

8. Future Directions

As large language models develop rapidly, their integration into fuzzing workflows appears promising. Current work already includes multi-step workflows, yet significant challenges remain. Section 7 summarizes the main technical and evaluation gaps.

Building on these observations, in this section we propose four main future directions: broader fuzzing workflow coverage, stronger LLM context and workflow engineering, more verifiable LLM reasoning inside the fuzzing loop, and more comparable benchmarking and evaluation. Figure 5 maps the main limitations to these directions.

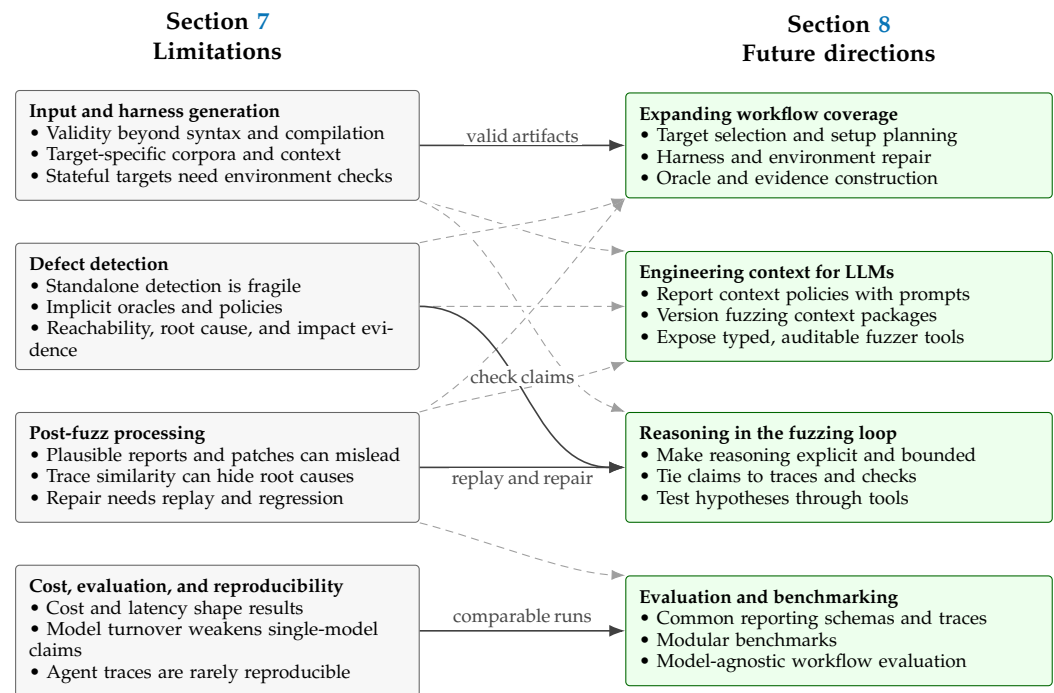


Figure 5. Direct map from limitations of current LLM-based fuzzing methods to future research directions. Solid arrows show the primary response to each limitation cluster, with short labels indicating the relationship. Dashed arrows show secondary dependencies.

8.1. Expanding Workflow Coverage

As previously mentioned, test case generation is still the most frequent use of LLMs. Future work should therefore define concrete LLM-facing decision points across other fuzzing stages.

A first direction is pre-campaign planning. LLMs can help choose targets and prepare the initial fuzzing setup before execution begins. ELFUZZ and PromeFuzz provide starting points because they use program and API context before fuzzing becomes effective [146,153]. This direction is especially relevant for large codebases, where target selection and setup often dominate the early cost. Planning should therefore be evaluated by early coverage and security-relevant interface coverage.

A second direction is harness and environment maintenance during execution. HarnessAgent shows that LLMs can synthesize fuzzing harnesses from code context [145]. Long campaigns also need repairs when compilation fails, coverage stalls, or environment assumptions prove wrong. Such systems should log each proposed change, replay the old and new harness on the same corpus, and measure whether the repair increases valid executions or masks failures.

A third direction is oracle and evidence construction. LLMs can help translate specifications, sanitizer traces, and differential outputs into checks that can be replayed. Systems that discover vulnerable interfaces, construct execution context, or combine generation with defect detection point in this direction [111,112,124,154]. The common requirement is grounding. A useful model output should become executable evidence. Evaluation should report false positives, missed bugs, and remaining human review.

8.2. Engineering Context for LLM

Prompt engineering remains the most visible adaptation technique in LLM-based fuzzing. Prompt wording alone is too narrow a design target because model performance depends on how target context, corpus state, and execution feedback are selected

and presented [101,112,145,146,155]. Future systems should therefore report the context-construction policy together with the prompt template.

Recent LLM research describes this broader layer as context engineering: the design of retrieval, context assembly, compression, memory, and tool-facing representations at inference time [156]. A recent study of AI context files in open-source software further shows that projects are beginning to encode build steps, project structure, coding rules, and workflows into version-controlled files for agentic tools, although no stable structure has yet emerged [157]. For fuzzing, an analogous artifact could be a versioned fuzzing context package: target maps, build and execution recipes, API or state models, seed and dictionary summaries, coverage frontiers, known invalid inputs, crash ledgers, oracle assumptions, and responsible-disclosure constraints. Such packages would make context reusable across campaigns and model versions, while allowing researchers to ablate individual context sources.

Harness-engineering reports and software-engineering agent research add another implication. Effective agents depend on repository-local knowledge, structured handoff, and clear tool interfaces [51,158,159]. Fuzzing systems should similarly expose core fuzzer operations as typed and auditable tools. Tool outputs should be summarized for the model, with full logs kept for replay and review.

A concrete research direction is context-adaptive fuzzing. During a campaign, the LLM would revise its context package using execution evidence. The system would then form a closed context-verification loop. Evaluation should compare fixed-prompt baselines with context-adaptive systems under equal compute and token budgets, and should report trajectories so that improvements can be attributed to specific context decisions.

8.3. Reasoning in the Fuzzing Loop

LLM reasoning should be made more explicit and more checkable inside fuzzing loops. Existing systems already use semantic reasoning in bounded settings, including API fuzzing, stateful protocol fuzzing, and hybrid analysis [99,101,120,121,154,160–162]. These systems show value and leave open how reasoning steps should be represented, checked, and bounded.

Code reasoning is needed when models infer program behavior. The same need appears after fuzzing, where SAN2PATCH converts sanitizer logs into fault localization, repair intent, and candidate patches that still require validation [136]. Broader code-reasoning evidence gives a cautionary note. CRUXEval shows that code generation scores do not fully measure code understanding or execution reasoning [163]. Chain-of-execution supervision suggests that execution-grounded traces can improve reasoning [164]. For fuzzing, this means reasoning claims should be tied to executable traces or checks.

Agentic reasoning can turn code-reasoning hypotheses into executable steps. Orion illustrates this direction by combining LLM guidance with deterministic verification and iterative refinement [165]. Agentic Code Reasoning is also relevant because its semi-formal method makes intermediate reasoning explicit [166]. In fuzzing, an agent should test hypotheses through tools and then revise its next action from the observed result. The loop should log these steps and tie them back to deterministic evidence.

Formal and mathematical reasoning can provide adjacent evidence. Recent theorem-proving and tool-augmented mathematical systems, including Goedel-Prover and Agent-Math, show that LLMs can work with formal languages and verifiable intermediate steps when correctness matters [167,168]. Similar mechanisms could strengthen fuzzing oracles when linked to executable evidence. ReAct, Toolformer, and TUMIX further show that reasoning can be interleaved with tool use and specialized agent roles [47,48,169]. A comparable fuzzing design should keep each role auditable and testable.

8.4. Evaluation and Benchmarking

The main evaluation problem is the lack of comparable evidence. Existing studies measure different artifacts on different targets, so their results answer different questions. Kernel studies, protocol/API fuzzers, and CRS-style systems each use valid, domain-specific metrics [101,120,124,142,160,161,170–172]. These results support within-domain conclusions; direct cross-paper rankings need controlled benchmarks.

A first direction is a common reporting schema. Each paper should define its metrics and record the conditions under which they were measured. Agentic systems should also report their action traces. This makes reported gains easier to reproduce, compare, and attribute.

A second direction is benchmark modularity. ORFuzzSet and the AIXCC leaderboard show the value of reusable evaluation assets [115,142]. The next step is to package targets so that models, context strategies, and fuzzing infrastructure can be varied independently. This keeps domain-specific benchmarks useful and still improves comparison.

Finally, evaluation should be model-agnostic under model turnover. Recent vendor reports from OpenAI and Anthropic claim gains in agentic coding and vulnerability work [150–152]. The specific model names will change, and the claims need independent validation. The durable research asset is a reusable fuzzing environment with stable execution, validation, and disclosure support. In such designs, the LLM becomes a replaceable component whose behavior can be compared across model generations.

9. Conclusions

This paper presents a systematic review of research on fuzzing with large language models (LLMs). We first outlined the fuzzing workflow and the main LLM adaptation strategies relevant to this area, then contrasted LLM-based methods with earlier deep learning-based fuzzing approaches. Across the surveyed literature, the main advantage of LLMs is not only generation quality, but also their ability to contribute semantic prior knowledge derived from source code, documentation, traces, bug histories, and execution feedback. At the same time, the evidence remains uneven. Test case generation is still the most frequent application in the coded corpus, while defect detection, fuzz-driver or harness generation, and post-processing are growing but less represented. Prompting remains the most visible integration strategy, yet many validation-oriented systems already rely on hybrid designs that combine LLMs with validation, retrieval, static analysis, coverage feedback, and iterative repair.

The review also shows that LLM-based fuzzing now spans a broad set of targets, from conventional software and compilers to kernels, protocols, firmware, smart contracts, and deep-learning frameworks, with gray-box settings appearing most often. The model ecosystem has likewise broadened from early dependence on GPT-style systems toward a mix of proprietary and open-source code-oriented models. These patterns suggest that the central challenge is no longer merely whether LLMs can assist fuzzing, but how they should be grounded, orchestrated, and evaluated across the full workflow. Accordingly, this paper highlights four directions for future work: broader coverage of the fuzzing lifecycle, stronger context construction and workflow engineering, richer code, formal and agentic reasoning for fuzzing control, and stronger evaluation standards with shared benchmarks. Progress along these directions will be important for turning promising task-specific prototypes into reliable, comparable, and more autonomous fuzzing systems.

Author Contributions: Conceptualization, Y.S.; methodology, Y.S., V.H.A., and G.C.; validation, Y.S. and G.C. and N.D.; formal analysis, Y.S.; investigation, Y.S. and V.H.A.; resources, Y.S. and V.H.A.; data curation, Y.S. and V.H.A.; writing—original draft preparation, Y.S.; writing—review and editing, Y.S.; visualization, Y.S.; supervision, G.C. and N.D.; project administration, N.D. and G.C.; funding acquisition, N.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Coding Form and Corpus Map

Table A1. Coding form and corpus map for LLM-based fuzzing techniques. Each row records the classification fields used in the synthesis: task, application strategy, testing type, target, target language, and LLM. Unmarked entries are peer-reviewed publications; ^P marks scholarly preprints in the current reference list. Technical reports and vendor or product material are not included as coded studies. In “Task”, TCG = test case/seed/mutation generation, DG = fuzz driver/harness/specification generation, DD = defect detection, and PP = post-processing (including crash triage, patching, exploit generation, and mitigation). Multiple task labels are used when a paper spans several stages; rare auxiliary tasks are written in full. In “Strategy”, FT = fine-tuning, PE = prompt engineering (Z = zero-shot learning, F = few-shot learning, C = chain-of-thought, ToT = tree-of-thought, A = auto-prompting), and Hybrid = LLM combined with traditional analysis, fuzzing, retrieval, or verification components. Multiple strategies are listed when a paper combines them; rare variants are written in full. In “Mode”, B = black-box testing, W = white-box testing, and G = gray-box testing; rare cross-stage cases are written in full. In “Target System”, R means the target is regular software. Otherwise, the type of target system is marked. In “Target Language”, General means the proposed fuzzer applies to multiple languages.

Literature	Task	Strategy	Mode	Target System	Target Language	LLM
Fuzz4All [81]	TCG	PE (A)	B	R	General	GPT-4, StarCoder
LiBro [45]	TCG	PE (F)	B	R	Java	Codex
A3Test [74]	TCG	FT	G	R	Java	PLBART
Hashtroudi et al. [75]	TCG	FT	G	R	Java	CodeT5
AceCoder [82] ^P	TCG	PE (Z)	B	R	General	GPT-3.5
Ackerman et al. [84]	TCG	PE (C)	G	Parsers	General	GPT-4
ChatUniTest [85]	TCG	Hybrid	W	R	Java	ChatGPT
ChatTester [86]	TCG	Hybrid	W	R	Java	ChatGPT
LLM4Fuzz [87] ^P	TCG	Hybrid	G	Smart contracts	Solidity	GPT-4
ChatFuzz [88] ^P	TCG	Hybrid	G	R	C/C++	ChatGPT
CCTEST [89]	TCG	Hybrid	G	Code completion models	Python	–
SearchGEM5 [90]	TCG	Hybrid	G	Simulators	C/C++	GPT-3.5-Turbo
CodaMosa [91]	TCG	Hybrid	G	R	Python	Codex
Xia et al. [92]	TCG	Hybrid	G	R	General	GPT-3.5-Turbo
AGM [173]	TCG	PE (A)	G	Smart contracts	Solidity	GPT-4
NetworkFuzzer [174]	TCG	PE (F)	B	Network protocols	Python	GPT-4
ARCANE [175]	TCG	Hybrid	B	5G O-RAN	C	Claude 3 Opus
CGIFuzz [176]	TCG	PE (F)	G	IoT devices	C	GPT-4
CHAT4SEED [177]	TCG	PE (F)	G	R	C	GLM-4
PWFuzz [178]	TCG	PE (F)	G	PLC web applications	General	GPT-4o

Table A1. Cont.

Literature	Task	Strategy	Mode	Target System	Target Language	LLM
DynER [161]	TCG	PE	G	RESTful APIs	PHP, Ruby	–
PSG [179]	TCG	PE (F)	G	Network protocols	General	GPT-4o
SynFuzz [180]	TCG	PE (F)	G	Firmware	C/C++	GPT-4
JailFuzzer [181]	TCG	PE (A)	B	R	Python	LLaVA, Vicuna, ShareGPT4V
FUZZCODER [79]	TCG	FT	G	IoT programs	C/C++	CodeLlama, DeepSeek-Coder, CodeQwen
LegoFuzz [182]	TCG	Hybrid	B	Compilers	C	GPT-4o-mini
LLAMAFUZZ [78]	TCG	FT	G	R	C/C++	llama2-7b-chat-hf
G2FUZZ [183]	TCG	PE (A)	G	R	C/C++	llama-3-8b, llama-3-70b
FuzzGPT [77]	TCG	FT, PE (F, Z)	G	DL frameworks	Python	Codex, CodeGen
MirrorFuzz [184]	TCG	PE (F, C)	G	DL frameworks	C/C++	CodeLlama, CodeGemma, Qwen
LLMFuzz [185]	TCG	PE (F)	G	Network protocols	C	GPT-4
RESTLess [160]	TCG	PE (A)	B	Cloud services	General	ChatGPT
LAST [76]	TCG	FT	W	SMT solvers	C/C++	GPT-2
TensorJSFuzz [186]	TCG	Hybrid	G	DL frameworks	JavaScript	GPT-4
SyzGPT [147]	TCG	PE (F)	G	Linux kernel	C	GPT, Claude, CodeLlama
ChatHTTPFuzz [96]	TCG	PE (F)	B	IoT HTTP services	General	GPT-4o
SyzAgent [170]	TCG	PE (A), Agentic	G	Linux kernel	C	GPT-4o
LLM-RLFuzz [187]	TCG	PE (F)	G	IoT protocols	General	Transformer, Bi-RNN
LLMUZZ [188]	TCG	PE (F)	B	IoT devices	General	Zhipu AI
VistaFuzz [189]	TCG	PE (F)	B	OpenCV	Python	GPT-4
LLMIF [190]	TCG	PE (F)	B	Zigbee protocol	Binary/protocol messages	ChatGPT-3.5
WhiteFox [99]	TCG	PE (F)	W	DL compilers	Python, C++	GPT-4, StarCoder
SeedMind [98] ^P	TCG	PE (C)	G	R	C/C++	GPT-4o, Claude-3.5-Sonnet
CovRL [80]	TCG	FT	W	JavaScript interpreters	JavaScript	Code-LLM, CodeT5+
StatePre [101]	TCG	PE (A)	G	Network protocols	General	SOTA models
SyzMutateX [191]	TCG	PE (F)	W	Linux kernel	C	–
MSFuzz [97]	TCG	PE (F)	G	Network protocols	C/C++	Qwen-plus
TitanFuzz [93]	TCG	PE (Z)	W	DL frameworks	General	Codex
Asmita et al. [83]	TCG	PE (Z)	B	Embedded firmware	C	GPT-4
ISC4DGF [192]	TCG	PE	G	R	C/C++	GPT-4
DFUZZ [193]	TCG	PE (F)	W	DL frameworks	C/C++	GPT-3.5
ProphetFuzz [194]	TCG	PE (F, C)	W	Open source software	General	GPT-4-Turbo
LMFuzz [195]	TCG	Hybrid	W	Compilers	General	StarCoder-1B, DeepSeek-Coder-6.7B
ScenarioFuzz [196]	TCG	PE (C)	W	Autoware.ai	C++	GPT-4-Turbo
NUWA [197]	TCG	PE (F)	W	MLIR	C++	GPT-4o
GPTCombFuzz [198]	TCG	PE (F)	G	R	–	GPT-3.5
LLMgSSA [199]	TCG	PE	G	Network protocols	Binary/protocol messages	ChatGPT-3.5
UI-based fuzzing [200]	TCG	PE (C)	B	IoT devices	–	Mistral-7B-Instruct-v0.2

Table A1. Cont.

Literature	Task	Strategy	Mode	Target System	Target Language	LLM
ECG [201]	TCG	PE (A)	W	Embedded OSs	C/C++	Mixtral-8x7b
OptionFuzz [94]	TCG	PE (Z)	W	SMT solvers	C/C++	GPT-4
LIPFuzzer [202]	TCG	PE (F)	G	IoT protocols	Binary/protocol messages	GLM-4
Mhiri et al. [203]	TCG	FT	G	XML programs	XML	Llama2
Nakanishi et al. [22]	TCG	PE (C)	G	IoT devices	General	Vicuna, Mistral, Phi, Llama
PILLM [100]	TCG	PE (Z)	W	JavaScript engines	JavaScript	GPT-4
SaFliTe [204]	TCG	PE (C)	G	Autonomous systems	General	GPT, Mistral, Llama
FUTURE [69]	TCG	FT	W	DL frameworks	C++	CodeLlama
WebFuzzAuto [205]	TCG	Agentic	B	Web applications	–	–
ChatAFL [95]	TCG	PE (F)	G	Network protocols	Binary/protocol messages	GPT-3.5-Turbo
MALF [172] ^P	TCG	FT, Hybrid, Agentic	G	Industrial control protocols	Binary/protocol messages	–
PrediQL [124] ^P	TCG, DD	Hybrid	B	GraphQL APIs	GraphQL	–
Gu et al. [206]	TCG	FT, PE	G	Compilers	Go	CodeT5
AutoSafeCoder [207]	TCG	PE, Hybrid	–	R	–	GPT-4o
Wang et al. [208]	TCG	PE, Hybrid	B	Compilers	C/C++	Microsoft Copilot, ChatGLM3-6B
EvilPromptFuzzer [209]	TCG	PE	–	Text-to-image models	Natural language prompts	DALL-E 3
mGPTFuzz [210]	TCG	PE	B	IoT devices	Binary/protocol messages	GPT-4-Turbo
HyLLfuzz [211] ^P	TCG	Hybrid	G	R	General	–
Magneto [212]	TCG	Hybrid	G	Libraries	–	GPT-4
Wang et al. [213]	TCG	PE	B	Medical software	General	MobileSAM
MetaMut [214]	TCG	PE	G	Compilers	C	GPT-4
Hasegawa et al. [215]	TCG	PE	B	IoT devices	Web inputs	Mistral-7B-Instruct-v0.2, Vicuna-7B-v1.5
Liao et al. [216]	TCG	Hybrid	G	DL frameworks	Python	DeepSeek-V3, DeepSeek-R1-Distill-Qwen-14B
Zhang et al. [217] ^P	TCG	Hybrid	G	R	–	Claude-Opus-4.520251101
ClozeMaster [218]	TCG	PE	G	Compilers	Rust	InCoder, StarCode, CodeShell, GPT-4o
Cyber Xipher [144]	TCG, PP	Hybrid	B	Web applications/services	–	Gemini
HGFuzzer [219] ^P	TCG	Hybrid	G	R	C/C++	Claude-3.5-Sonnet
ELFUZZ [153]	TCG	Hybrid	G	R	General	CodeLlama
PILOT [220] ^P	TCG	Hybrid	B	CLI applications	CLI options + file inputs	Claude Sonnet 4.0, GPT-4.1
SearchSYS [221]	TCG	Hybrid	G	Simulators	C/C++	–
GLMFuzz [171]	TCG	PE (A)	G	Network protocols	Binary/protocol messages	CHATGLM-4-PLUS
Lin, Shiyin [222]	TCG	Hybrid	G	R	C/C++	GPT-4
HLPUZZ [121]	TCG	Hybrid	G	Compilers/interpreters	General	GPT-4o-mini

Table A1. *Cont.*

Literature	Task	Strategy	Mode	Target System	Target Language	LLM
Mcshane et al. [223]	TCG	PE	B	Automotive diagnostic protocols	Binary/protocol messages	GPT-4
MOJOFuzzer [224]	TCG	Hybrid	G	R	Mojo	GPT, LLAMA2
Maklad et al. [225]	TCG	PE (C)	G	Network protocols	Binary/protocol messages	Gemma-2-9B
Rug [226]	TCG	Hybrid	G	Rust software	Rust	Gpt-3.5-turbo-16k-0613, Gpt-4-1106
SSFuzz [227]	TCG	Hybrid	B	IoT devices	Binary/protocol messages	–
SearchSYS [228]	TCG	Hybrid	G	ARM simulators	C/C++	TinyLlama, Phi2, Llama2, Magicoder, CodeBooga, GPT-3.5-turbo
Lu et al. [149] ^P	TCG	PE (F)	G	Network protocols	Binary/protocol messages	DeepSeek-R1, Llama
LLM1dFuzz [229]	TCG	PE (A)	G	Known vulnerabilities	Program inputs	GPT
GLEIPNIR [111]	TCG, DD	Hybrid	G	Windows remote IPC clients	IPC return values	GPT-4
EAGLEYE [112]	TCG, DD	Hybrid	B	IoT web interfaces	HTTP requests	GPT-3.5-Turbo
GenHuzz [230]	TCG	Hybrid	W	Hardware/RISC-V cores	RISC-V assembly	–
FIAL [155]	TCG	Hybrid	B	IoT devices	Android app interfaces	–
Robofuzz [231]	TCG	Hybrid	G	Robotic software	Robot messages	Marcoroni
Avihay Cohen [117] ^P	TCG, DD	PE (A), Hybrid	B	Agentic AI browsers	Web pages/prompts	–
Nakanishi et al. [232]	TCG	PE (Z)	B	IoT devices	Initial seeds	Vicuna-13B-v1.3
DSM-SST-LLMT [162]	TCG	Hybrid	G	R	General	–
Orion (workflow automation) [165] ^P	Workflow automation	Hybrid	Workflow-level	R	C/C++	GPT-4.1
Jiang et al. [118] ^P	DD	PE (F, C)	–	Java libraries	Java	GPT-4
AVD System [106]	DD	FT, PE (C)	G	R	C/C++	DeepSeek-R1, Llama 3
ACBreaker [107]	DD	PE (C)	W	IoT firmware	C	Qwen-2.5-Coder-32B, GPT-4o
ADFEmu [108]	DD	Hybrid	G	Firmware	C	GPT-4
CrawlMLLM [109]	DD	PE (F)	W	Web applications	PHP	GPT-4o
DIVAS [119] ^P	DD	PE (Z)	G	SoC	General	ChatGPT, BART
AutoSD [123]	DD	PE (F)	G	R	Java	ChatGPT
AutoFL [233] ^P	DD	PE (C)	G	R	General	PLBART, CodeT5, etc.
SkipAnalyzer [234] ^P	DD, PP	PE (F)	G	R	Java	GPT-4, GPT-3.5-Turbo
UniFuzz [126]	DD	Hybrid	G	Smart contracts	Solidity	Gemini-1.5
CrossGuard [127]	DD	Hybrid	G	Smart contracts	Solidity	Llama
DCGAN-MNFuzz [235]	DD	Hybrid	B	IoT networks	Binary/protocol messages	–
HyFuzz [128]	DD	Hybrid	B	Network services	Binary/protocol messages	DeepSeek-R1
DKFuzz [236]	DD	Hybrid	G	DL frameworks	Python	–
ExpFuzz [143]	DD, PP	Hybrid	B	Web applications/services	JavaScript	–
FuzzingBrain CRS [142] ^P	DD, PP	Hybrid	–	R	C, Java	–

Table A1. Cont.

Literature	Task	Strategy	Mode	Target System	Target Language	LLM
CrossProbe [125]	DD	PE	–	DL frameworks	Python	Gpt-4o-2024-08-06
PROMFUZZ [110]	DD	Hybrid	G	Smart contracts	Solidity	GPT
Wang et al. [237]	DD	Hybrid	G	Network protocols	Binary/protocol messages	GPT-4
Abtahi et al. [129]	DD, PP	Hybrid	–	Firmware	C/C++	GPT-4
Chen et al. [122]	DD	FT	–	PLC code	Structured Text	CodeLlama, Qwen2.5-Coder, StarCoder2
AgentFuzz [113]	DD	Hybrid, Agentic	G	LLM-based agents	General	GPT-4o
OBsmith [114]	DD	Hybrid	–	JavaScript obfuscators	JavaScript	GPT-5, Gemini 2.5 Pro, Claude 4, Qwen3, Grok 4
ORFuzz [115]	DD	Hybrid	B	LLM safety/over-refusal testing	Natural language prompts	Qwen2.5-14B-Instruct
TransferFuzz-Pro [148]	DD	Hybrid	–	R	General	Claude3.5-Sonnet
deepSURF [154]	DD	Hybrid	–	Rust software	Rust	DeepSeek-R1
Lyu et al. [238]	DD	Hybrid	–	Database connectors	Java	Qwen
HFuzzer [116]	DD	PE	B	Code LLMs	Natural language prompts	GPT-4o
HarnessAgent [145] ^P	DG	Hybrid, Agentic	–	Libraries	C/C++	GPT5-Mini
Kim et al. [239]	DG	FT, PE	–	Libraries	C	GPT-3.5-Turbo, GPT-3.5
AutoGen [240]	DG	PE	–	Libraries	C	–
PromptFuzz [241]	DG	PE (Z)	G	Libraries	General	GPT-3.5
PromeFuzz [146]	DG	Hybrid	G	Libraries	General	DeepSeek-V3
KernelGPT [105]	DG	Hybrid	G	Linux kernel	C	GPT-4
SyzForge [120]	DG	Hybrid	G	Linux kernel	C	DeepSeek-R1
CKGFuzzer [242]	DG	Hybrid, Agentic	G	Libraries	C	DeepSeek-V2-Coder
Tran et al. [243]	DG	PE	–	Libraries	C	GitHub Copilot
AePPollo [130]	PP	Hybrid	–	Web applications/services	JavaScript	Qwen-Max-2025-01-25
CHEMFUZZ [131]	PP	PE (Z)	G	R	C++, Fortran	GPT-3.5, etc.
Mashhadi et al. [132]	PP	FT	G	R	Java	CodeBERT
CodeRover-S [133]	PP	PE	G	R	General	GPT-4o
InferFix [134]	PP	FT	G	R	General	Codex
Pearce et al. [135]	PP	PE (F)	G	R	C/C++	Codex etc.
GPTTrace [137]	PP	Hybrid	–	R	–	Text-embedding-3-large
Chen et al. [138]	PP	Hybrid	–	R	General	–
LAC2R [139] ^P	PP	Hybrid	–	C-to-Rust translation	C, Rust	GPT-4o, DeepSeek V3
RustAssure [140]	PP	PE	–	C-to-Rust translation	C, Rust	–
SAN2PATCH [136]	PP	PE (ToT)	–	R	C/C++	GPT, Claude, Gemini

Appendix B. Comparison with Related Work

Table A2. A detailed comparison between this literature review and representative related works on LLM-based fuzzing and adjacent areas. ^P marks scholarly preprints for which no peer-reviewed venue was identified at the time of screening.

Reference	Target Domain	Evaluation Methodology/Metrics	Evaluated Performance and Key Findings	Limitation Compared with This Paper
Aldysty et al. [14]	Industrial Network Protocols (INPs)	Qualitative assessment of INP fuzzing combined with a mini case study evaluating LLMs on the Modbus protocol.	LLMs' contextual understanding and knowledge consolidation can overcome traditional challenges like message integrity and log analysis in ICS.	Restricted to industrial network protocols, with only a small LLM case study on Modbus rather than a broad, stage-wise survey of LLM-assisted fuzzing.
Asmita et al. [15]	Bare-Metal and Monolithic Firmware Fuzzing	Qualitative review of emulation strategies, rehosting, back-end instrumentation, and the projection of generic software LLM fuzzing onto firmware.	Found that most current research focuses heavily on basic emulation and direct integration with existing fuzzers. Identifies LLMs as a highly promising future direction for overcoming hardware complexities.	Concentrates on the firmware domain and infrastructure challenges; treatment of LLMs is largely prospective and does not provide a comprehensive survey of LLM-based fuzzing across the full workflow.
Black et al. [16]	Seed Generation (Python)	Generated over 38,000 seeds via LLMs for 50 Python functions using the Atheris framework. Validated via linear regression analysis.	Reports the following result in its setting: fewer than 100 LLM-generated seeds achieved higher coverage and instruction counts than over 100,000 conventionally produced inputs.	Limited to Python seed generation under Atheris; does not cover other fuzzing stages, target domains, or provide a survey-level synthesis.
Jiang et al. [17]	Directed Test Input Generation	Systematic comparative evaluation of mainstream LLMs (e.g., ChatGPT) against traditional constraint-based tools (symbolic execution/concolic testing) for reaching uncovered branches.	Reports 43.40–58.57% LLM success rates, comparable to or higher than constraint-based tools in the evaluated setting. A proposed hybrid LLM + constraint-solver prototype reported 1.4× to 2.3× gains over baselines.	Focuses narrowly on directed branch-reaching input generation rather than the broader fuzzing lifecycle, and is an empirical study rather than a dedicated survey.
Jiang et al. [18]	LLM-Assisted Fuzzing (DBMS)	Revisited recent top-tier conference papers to identify challenges; conducted preliminary empirical evaluations on DBMS fuzzing.	Identified 5 major challenges in LLM-assisted fuzzing. Demonstrated that actionable recommendations effectively address these challenges.	Evaluations are preliminary and heavily restricted to the Database Management System (DBMS) domain.
Jiang et al. [19]	Code Generation via LLMs	Empirical comparison of Code LLMs using HumanEval, MBPP, and BigCodeBench across various difficulties and task types.	Documented progressive enhancements in code generation capabilities; introduced a taxonomy covering data curation, ethics, and performance.	Broadly targets code generation rather than fuzzing; lacks a fuzzing-oriented taxonomy, stage-wise workflow analysis, and discussion of fuzzing-specific challenges.
Miao et al. [20]	Deep Learning in Fuzzing	Literature survey of deep learning-based fuzzing techniques, including neural input modeling, feature learning, seed generation, mutation guidance, and defect-detection support.	Shows that deep learning can improve parts of fuzzing by learning input structures and program-relevant features, but its effectiveness depends heavily on task-specific data, model design, and training quality.	Focuses on pre-LLM deep learning-based fuzzing; it does not analyze modern LLM capabilities, prompt/retrieval workflows, or LLM integration across the fuzzing lifecycle.
Wang et al. [11]	Software Testing with LLMs	Survey of 102 studies on LLMs for software testing, analyzed from both software-testing tasks and LLM usage perspectives.	Finds LLMs are widely used for test case preparation, program repair, prompt engineering, and other testing tasks, while also identifying challenges and future opportunities.	Covers software testing broadly rather than fuzzing specifically; fuzzing is not organized by workflow stages, and pre-LLM deep learning-based fuzzing is not treated as a central comparison baseline.
Huang et al. [21]	LLM-Based Fuzzing Techniques	Statistical analysis and systematic literature review of state-of-the-art methods fusing LLMs and fuzzing tests up to submission.	Highlights that LLMs demonstrate remarkable performance in fuzzing and show high promise for advancing automated software testing practices.	Primarily emphasizes existing LLM-based fuzzing methods, but gives less explicit comparison against pre-LLM deep learning-based fuzzing and less end-to-end workflow-oriented analysis.

Table A2. Cont.

Reference	Target Domain	Evaluation Methodology/Metrics	Evaluated Performance and Key Findings	Limitation Compared with This Paper
Nakanishi et al. [22]	Seed Generation (IoT Devices)	Compared Vicuna v1.5, Mistral, Phi-3, and Llama 3 on seed format conformance and vulnerability detection rates in IoT targets.	Mistral, Phi-3, and Llama 3 detected numerous vulnerabilities and maintained high conformance rates for required seed formats.	Evaluation is limited to only four LLM models and is strictly confined to the Internet of Things (IoT) hardware/software domain.
Panebianco et al. [23]	Zero-Shot Vulnerability Detection	Evaluated quantized (Mistral v0.3), specialized (CodeQwen 1.5), and fine-tuned (PDBERT) models for zero-shot detection.	Zero-shot models performed poorly (Precision < 0.46). Fine-tuned models like PDBERT showed high metrics but suffered from heavy overfitting.	Addresses vulnerability detection rather than fuzzing itself, and focuses on model evaluation without covering the broader fuzzing workflow or integration stages.
Qiu et al. [24]	Coverage-Guided Greybox Fuzzing	Taxonomy and capability analysis of Deep Neural Models (DNNs/LLMs) across seed generation, selection, and mutation phases.	Identifies hybrid models (LLMs paired with neurosymbolic reasoning) as a promising future path for tackling path explosion.	Centers on coverage-guided greybox fuzzing with deep neural models in general, rather than offering a dedicated, comprehensive survey of LLM-assisted fuzzing across diverse stages and settings.
Zhang et al. [25]	Fuzz Driver Generation (C APIs)	Massive scale evaluation: 736,430 drivers generated, 5 LLMs, 6 prompt strategies, 86 questions from 30 C projects, 3.75 CPU-years of fuzzing.	LLM drivers perform on par with industry standards. Iterative querying and providing examples significantly improved generation quality.	Restricted to fuzz driver generation for C library APIs; despite its scale, it does not survey or analyze LLM integration across the full fuzzing pipeline.
Feng et al. [26]	Fuzz Driver Generation	Survey of 30 fuzz-driver-generation studies from 2019–2025, organized by data sources such as usage artifacts, source-code constraints, binary semantics, and heterogeneous data fusion.	Highlights validity, reachability-related evidence, reproducibility, and cost as key comparison dimensions, and shows that LLMs are increasingly used for driver synthesis, constraint alignment, and repair.	Dedicated to automated fuzz driver generation, so it covers only one stage of the fuzzing workflow and does not provide a full LLM-assisted fuzzing survey across generation, detection, analysis, and pre-LLM baselines.
Zhu et al. [27]	Broad Software Security	Systematic deconstruction of security techniques (fuzzing, repair, bug triage) into specific stages to map LLM utility.	Shows deep learning and LLMs provide significant enhancements. Proposes future directions for integrating conventional deep learning with LLMs.	Broad security scope dilutes depth on fuzzing; it is not a dedicated survey of LLM-based fuzzing and does not focus on fuzzing-specific workflow, taxonomy, and challenges in comparable detail.
Gubbi et al. [28]	Hardware Fuzzing/Hardware Security	Survey of hardware fuzzing methods plus a synthesis of evaluation metrics such as total bugs found, time-to-first bug, coverage, execution speed, and scalability; also discusses ML/LLM use cases including testbench generation, constraint optimization, and debugging.	Finds AI-assisted hardware fuzzing can improve test generation, coverage exploration, and debugging, while identifying high computational cost, limited training data, and explainability as key barriers to broader adoption.	Focused on hardware verification and partly forward-looking on LLMs; it is not a dedicated survey of implemented LLM-assisted fuzzing techniques across software-oriented stages.
Cheng et al. [29] ^P	Reliable LLM-driven Fuzz Driver and Seed Generation	Vision/review paper synthesizing existing LLM4Fuzz work on driver and seed generation; identifies four challenges, four research directions, and proposes reliability metrics for driver validity and seed quality.	Highlights complex API dependencies, insufficient driver quality assurance, seed quality trade-offs, and semantic consistency as core bottlenecks. Proposes metrics such as compilation/runtime success, API usage correctness, validity rate, path coverage, and vulnerability discovery rate.	Concentrates on reliability of drivers and seeds rather than surveying the full fuzzing workflow or contrasting LLM-assisted fuzzing with earlier deep-learning approaches.

Table A2. Cont.

Reference	Target Domain	Evaluation Methodology/Metrics	Evaluated Performance and Key Findings	Limitation Compared with This Paper
Hardgrove and Xu [30]	Intelligent Fuzzing for Software Security	High-level review of using LLMs to automate fuzzing setup and input generation, with discussion of prompt engineering, multi-LLM integration, bug detection, and remediation opportunities.	Argues that LLMs can reduce manual overhead and support more adaptive fuzz testing, but the discussion is primarily opportunity-oriented rather than a systematic empirical evaluation with a unified taxonomy.	Broad and forward-looking; lacks stage-wise synthesis of concrete LLM-assisted fuzzing studies and does not cover pre-LLM deep learning-based fuzzing in depth.
This paper	General LLM-Assisted Fuzzing Across the Full Workflow	Systematic survey that first introduces fuzzing workflow and LLM-related background, then reviews pre-LLM deep learning-based fuzzing, analyzes how LLMs are integrated at different fuzzing stages, and finally synthesizes open challenges and future directions.	Provides a comprehensive, end-to-end view of the field, clarifies why LLMs are needed beyond earlier deep learning approaches, and organizes current work by concrete fuzzing stages rather than by isolated tasks or domains.	More comprehensive than prior work by jointly covering workflow background, pre-LLM baselines, stage-wise LLM integration, and future research directions across multiple domains instead of a single task, target, or application area.

References

1. Miller, B.P.; Fredriksen, L.; So, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* **1990**, *33*, 32–44. [\[CrossRef\]](#)
2. Xiao, X.; Li, S.; Xie, T.; Tillmann, N. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: Piscataway, NJ, USA, 2013; pp. 246–256. [\[CrossRef\]](#)
3. Pacheco, C.; Lahiri, S.K.; Ernst, M.D.; Ball, T. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*; IEEE: Piscataway, NJ, USA, 2007; pp. 75–84. [\[CrossRef\]](#)
4. Pan, M.; Huang, A.; Wang, G.; Zhang, T.; Li, X. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 153–164. [\[CrossRef\]](#)
5. Guo, D.; Yang, D.; Zhang, H.; Song, J.; Wang, P.; Zhu, Q.; Xu, R.; Zhang, R.; Ma, S.; Bi, X.; et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* **2025**, *645*, 633–638. [\[CrossRef\]](#)
6. OpenAI. Introducing GPT-5. 2025. Available online: <https://openai.com/index/introducing-gpt-5/> (accessed on 30 December 2025).
7. Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; Zhou, Y. Codegen2: Lessons for training llms on programming and natural languages. *arXiv* **2023**, arXiv:2305.02309. [\[CrossRef\]](#)
8. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv* **2024**, arXiv:2401.14196. [\[CrossRef\]](#)
9. OpenAI. Introducing Codex. 2025. Available online: <https://openai.com/index/introducing-codex/> (accessed on 30 December 2025).
10. Anthropic. Claude. 2026. Available online: <https://www.anthropic.com/claude> (accessed on 5 April 2026).
11. Wang, J.; Huang, Y.; Chen, C.; Liu, Z.; Wang, S.; Wang, Q. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936. [\[CrossRef\]](#)
12. OpenAI. ChatGPT: Optimizing Language Models for Dialogue. Version 3.5. 2023. Available online: <https://openai.com/blog/chatgpt> (accessed on 17 May 2026).
13. Petticrew, M.; Roberts, H. *Systematic Reviews in the Social Sciences: A Practical Guide*; Blackwell Publishing: Malden, MA, USA, 2006. [\[CrossRef\]](#)
14. Aldysty, A.R.; Moustafa, N.; Lakshika, E. A Holistic Review of Fuzzing for Vulnerability Assessment in Industrial Network Protocols. *IEEE Open J. Commun. Soc.* **2025**, *6*, 4437–4461. [\[CrossRef\]](#)
15. Asmita, A.; Tsang, R.; Ghimire, S.; Salehi, S.; Homayoun, H. Bare-Metal Firmware Fuzzing: A Survey of Techniques and Approaches. *IEEE Access* **2025**, *13*, 98253–98277. [\[CrossRef\]](#)
16. Black, G.; Vaidyan, V.M.; Comert, G. Evaluating Large Language Models for Enhanced Fuzzing: An Analysis Framework for LLM-Driven Seed Generation. *IEEE Access* **2024**, *12*, 156065–156081. [\[CrossRef\]](#)
17. Jiang, Z.; Wen, M.; Cao, J.; Shi, X.; Jin, H. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *ASE '24: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*; Association for Computing Machinery, Inc.: New York, NY, USA, 2024; pp. 1408–1420. [\[CrossRef\]](#)

18. Jiang, Y.; Liang, J.; Ma, F.; Chen, Y.; Zhou, C.; Shen, Y.; Wu, Z.; Fu, J.; Wang, M.; Li, S.; et al. When Fuzzing Meets LLMs: Challenges and Opportunities. In *FSE 2024: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*; ACM: New York, NY, USA, 2024; pp. 492–496. [\[CrossRef\]](#)
19. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 58. [\[CrossRef\]](#)
20. Miao, S.; Wang, J.; Zhang, C.; Lin, Z.; Gong, J.; Zhang, X.; Li, J. Deep learning in fuzzing: A literature survey. In *Proceedings of the 2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*; IEEE: Piscataway, NJ, USA, 2022; pp. 220–223. [\[CrossRef\]](#)
21. Huang, L.; Zhao, P.; Chen, H.; Ma, L. On the Challenges of Fuzzing Techniques via Large Language Models. In *2025 IEEE International Conference on Software Services Engineering, SSE*; Institute of Electrical and Electronics Engineers Inc. (IEEE): Piscataway, NJ, USA, 2025; pp. 162–171. [\[CrossRef\]](#)
22. Nakanishi, H.; Hasegawa, K.; Hidano, S.; Fukushima, K.; Hashimoto, K.; Togawa, N. Performance Comparison of the LLM Models on LLM-based Seed Generation Method for IoT Device Fuzzing. In *2025 IEEE International Conference on Consumer Electronics (ICCE)*; Institute of Electrical and Electronics Engineers Inc. (IEEE): Piscataway, NJ, USA, 2025; pp. 1–5. [\[CrossRef\]](#)
23. Panebianco, F.; Isgrò, A.; Longari, S.; Zanero, S.; Carminati, M. Guessing As A Service: Large Language Models Are Not Yet Ready For Vulnerability Detection. In *Proceedings of the Ceur Workshop Proceedings*; CEUR-WS: Aachen, Germany, 2025; Volume 3962.
24. Qiu, J.; Jiang, Y.; Miao, Y.; Luo, W.; Pan, L.; Zheng, X. A survey of coverage-guided greybox fuzzing with deep neural models. *Inf. Softw. Technol.* **2025**, *186*, 107797. [\[CrossRef\]](#)
25. Zhang, C.; Zheng, Y.; Bai, M.; Li, Y.; Ma, W.; Xie, X.; Li, Y.; Sun, L.; Liu, Y. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *ISSTA 2024: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery, Inc.: New York, NY, USA, 2024; pp. 1223–1235. [\[CrossRef\]](#)
26. Feng, X.; Lu, S.; Gu, T.; Nie, Y.; Yan, Q.; Yang, M.; Chen, J.; Kuang, X. Fuzz Driver Generation: A Survey and Outlook from the Perspective of Data Sources. *Big Data Cogn. Comput.* **2026**, *10*, 129. [\[CrossRef\]](#)
27. Zhu, X.; Zhou, W.; Han, Q.L.; Ma, W.; Wen, S.; Xiang, Y. When Software Security Meets Large Language Models: A Survey. *IEEE/CAA J. Autom. Sin.* **2025**, *12*, 317–334. [\[CrossRef\]](#)
28. Gubbi, K.I.; Tarighat, M.; Sudarshan, A.; Kaur, I.; Kota, P.D.; Sasan, A.; Homayoun, H. State of Hardware Fuzzing: Current Methods and the Potential of Machine Learning and Large Language Models. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*; IEEE Computer Society: Piscataway, NJ, USA, 2025; pp. 540–546. [\[CrossRef\]](#)
29. Cheng, Y.; Kang, H.J.; Shar, L.K.; Dong, C.; Shi, Z.; Lv, S.; Sun, L. Towards Reliable LLM-Driven Fuzz Testing: Vision and Road Ahead. *arXiv* **2025**, arXiv:2503.00795. [\[CrossRef\]](#)
30. Hardgrove, I.; Xu, S. Towards Intelligent Fuzzing: Leveraging Large Language Models for Improved Software Security. In *Proceedings of the Lecture Notes in Networks and Systems*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2026; Volume 1698, pp. 404–419. [\[CrossRef\]](#)
31. Manès, V.J.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331. [\[CrossRef\]](#)
32. Zalewski, M. American Fuzzy Lop (AFL)–Security-Oriented Fuzzer. 2026. Available online: <https://lcamtuf.coredump.cx/afl/> (accessed on 5 April 2026).
33. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, Online, 11 August 2020.
34. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 2329–2344. [\[CrossRef\]](#)
35. Wang, C.; Yang, Y.; Gao, C.; Peng, Y.; Zhang, H.; Lyu, M.R. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 382–394. [\[CrossRef\]](#)
36. Mo, S.; Cho, M.; Shin, J. Freeze the discriminator: A simple baseline for fine-tuning gans. *arXiv* **2020**, arXiv:2002.10964. [\[CrossRef\]](#)
37. Li, X.L.; Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021; pp. 4582–4597. [\[CrossRef\]](#)
38. Hu, E.J.; Shen, y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W. Lora: Low-rank adaptation of large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Virtual, 25–29 April 2022; Volume 1, p. 3.
39. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 10088–10115. [\[CrossRef\]](#)

40. Wang, W.; Zheng, V.W.; Yu, H.; Miao, C. A Survey of Zero-Shot Learning: Settings, Methods, and Applications. *ACM Trans. Intell. Syst. Technol.* **2019**, *10*, 13. [\[CrossRef\]](#)
41. Wang, Y.; Yao, Q.; Kwok, J.T.; Ni, L.M. Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Comput. Surv.* **2020**, *53*, 63. [\[CrossRef\]](#) [\[PubMed\]](#)
42. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of the Advances in Neural Information Processing Systems*; Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2022; Volume 35, pp. 24824–24837.
43. Yao, S.; Yu, D.; Zhao, J.; Shafraan, I.; Griffiths, T.; Cao, Y.; Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 11809–11822.
44. Zhou, Y.; Muresanu, A.I.; Han, Z.; Paster, K.; Pitis, S.; Chan, H.; Ba, J. Large Language Models are Human-Level Prompt Engineers. In *Proceedings of the Eleventh International Conference on Learning Representations*, Kigali, Rwanda, 1–5 May 2023.
45. Kang, S.; Yoon, J.; Yoo, S. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 2312–2323. [\[CrossRef\]](#)
46. Kojima, T.; Gu, S.S.; Reid, M.; Matsuo, Y.; Iwasawa, Y. Large Language Models are Zero-Shot Reasoners. In *Proceedings of the Advances in Neural Information Processing Systems*; Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2022; Volume 35, pp. 22199–22213.
47. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafraan, I.; Narasimhan, K.R.; Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the International Conference on Learning Representations*, Kigali, Rwanda, 1–5 May 2023.
48. Schick, T.; Dwivedi-Yu, J.; Dessi, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; Scialom, T. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Proceedings of the Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2023.
49. Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.R.; Yao, S. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Proceedings of the Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2023.
50. Liu, X.; Yu, H.; Zhang, H.; Xu, Y.; Lei, X.; Lai, H.; Gu, Y.; Ding, H.; Men, K.; Yang, K.; et al. AgentBench: Evaluating LLMs as Agents. In *Proceedings of the International Conference on Learning Representations*, Vienna, Austria, 7–11 May 2024. [\[CrossRef\]](#)
51. Yang, J.; Jimenez, C.E.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.R.; Press, O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Proceedings of the Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2024. [\[CrossRef\]](#)
52. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774. [\[CrossRef\]](#)
53. Cao, R.; Chen, M.; Chen, J.; Cui, Z.; Feng, Y.; Hui, B.; Jing, Y.; Li, K.; Li, M.; Lin, J.; et al. Qwen3-Coder-Next Technical Report. *arXiv* **2026**, arXiv:2603.00729. [\[CrossRef\]](#)
54. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **2012**, *25*. [\[CrossRef\]](#)
55. Graves, A. Generating sequences with recurrent neural networks. *arXiv* **2013**, arXiv:1308.0850. [\[CrossRef\]](#)
56. Yin, Z.; Shao, J.; Hussain, M.J.; Hao, Y.; Chen, Y.; Zhang, X.; Wang, L. Dpg-lstm: An enhanced lstm framework for sentiment analysis in social media text based on dependency parsing and gcn. *Appl. Sci.* **2022**, *13*, 354. [\[CrossRef\]](#)
57. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*. [\[CrossRef\]](#)
58. Hassan, H.; Aue, A.; Chen, C.; Chowdhary, V.; Clark, J.; Federmann, C.; Huang, X.; Junczys-Dowmunt, M.; Lewis, W.; Li, M.; et al. Achieving human parity on automatic chinese to english news translation. *arXiv* **2018**, arXiv:1803.05567. [\[CrossRef\]](#)
59. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **2002**, *86*, 2278–2324. [\[CrossRef\]](#)
60. Graves, A. Supervised sequence labelling. In *Supervised Sequence Labelling with Recurrent Neural Networks*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 5–13. [\[CrossRef\]](#)
61. Godefroid, P.; Peleg, H.; Singh, R. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: Piscataway, NJ, USA, 2017; pp. 50–59. [\[CrossRef\]](#)
62. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*; USENIX Association: Berkeley, CA, USA, 2020; pp. 2255–2269.
63. Monsefi, A.K.; Zakeri, B.; Samsam, S.; Khashehchi, M. Performing software test oracle based on deep neural network with fuzzy inference system. In *Proceedings of the International Congress on High-Performance Computing and Big Data Analysis*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 406–417. [\[CrossRef\]](#)

64. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software defect prediction via convolutional neural network. In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*; IEEE: Piscataway, NJ, USA, 2017; pp. 318–328. [\[CrossRef\]](#)
65. Dias, T.; Batista, A.; Maia, E.; Praça, I. Testlab: An intelligent automated software testing framework. In *Proceedings of the International Symposium on Distributed Computing and Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 355–364. [\[CrossRef\]](#)
66. Zhang, Z.; Klees, G.; Wang, E.; Hicks, M.; Wei, S. Fuzzing configurations of program options. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 53. [\[CrossRef\]](#)
67. Li, Y.; Wang, S.; Nguyen, T.N. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 511–523. [\[CrossRef\]](#)
68. Li, S.; Xie, X.; Lin, Y.; Li, Y.; Feng, R.; Li, X.; Ge, W.; Dong, J.S. Deep learning for coverage-guided fuzzing: How far are we? *IEEE Trans. Dependable Secur. Comput.* **2022**, early access. [\[CrossRef\]](#)
69. Li, Z.; Wu, J.; Ling, X.; Luo, T.; Rui, Z.; Wu, Y. The Seeds of the FUTURE Sprout from History: Fuzzing for Unveiling Vulnerabilities in Prospective Deep-Learning Libraries. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2025; pp. 1616–1627. [\[CrossRef\]](#)
70. Riccio, V.; Tonella, P. When and why test generators for deep learning produce invalid inputs: An empirical study. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1161–1173. [\[CrossRef\]](#)
71. The MITRE Corporation. Common Weakness Enumeration (CWE). 2026. Available online: <https://cwe.mitre.org/> (accessed on 5 April 2026).
72. Zhang, Q.; Fang, C.; Ma, Y.; Sun, W.; Chen, Z. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 55. [\[CrossRef\]](#)
73. Lutellier, T.; Pham, H.V.; Pang, L.; Li, Y.; Wei, M.; Tan, L. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 101–114. [\[CrossRef\]](#)
74. Alagarsamy, S.; Tantithamthavorn, C.; Aleti, A. A3test: Assertion-augmented automated test case generation. *Inf. Softw. Technol.* **2024**, *176*, 107565. [\[CrossRef\]](#)
75. Shin, J.; Hashtroudi, S.; Hemmati, H.; Wang, S. Domain Adaptation for Code Model-Based Unit Test Case Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*; ACM: New York, NY, USA, 2024; pp. 1211–1222. [\[CrossRef\]](#)
76. Sun, M.; Yang, Y.; Wang, Y.; Wen, M.; Jia, H.; Zhou, Y. SMT Solver Validation Empowered by Large Pre-Trained Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2023; pp. 1288–1300. [\[CrossRef\]](#)
77. Deng, Y.; Xia, C.S.; Yang, C.; Zhang, S.D.; Yang, S.; Zhang, L. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*; ACM: New York, NY, USA, 2024; pp. 1–13. [\[CrossRef\]](#)
78. Zhang, H.; Rong, Y.; He, Y.; Chen, H. LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing. In *Proceedings of the 2026 IEEE/ACM International Workshop on Automation of Software Test (AST)*, Rio de Janeiro, Brazil, 12–18 April 2026.
79. Yang, L.; Wei, C.; Yang, J.; Xia, W.; Yang, Y.; Luo, Y.; Niyato, D.; Sun, L.; Liu, Z. FuzzCoder: Code Large Language Model-Based Fuzz Testing for Industrial IoT Programs. *IEEE Internet Things J.* **2025**, *12*, 36842–36851. [\[CrossRef\]](#)
80. Eom, J.; Jeong, S.; Kwon, T. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*; ACM: New York, NY, USA, 2024; pp. 1656–1668. [\[CrossRef\]](#)
81. Xia, C.S.; Paltenghi, M.; Tian, J.L.; Pradel, M.; Zhang, L. Fuzz4ALL: Universal Fuzzing with Large Language Models. In *ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2024. [\[CrossRef\]](#)
82. Li, J.; Zhao, Y.; Li, Y.; Li, G.; Jin, Z. AceCoder: Utilizing Existing Code to Enhance Code Generation. *arXiv* **2023**, arXiv:2303.17780. [\[CrossRef\]](#)
83. Asmita; Oliinyk, Y.; Scott, M.; Tsang, R.; Fang, C.; Homayoun, H. Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, USA, 14–16 August 2024; pp. 883–900.
84. Ackerman, J.; Cybenko, G. Large Language Models for Fuzzing Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop, Fuzzing 2023*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 31–38. [\[CrossRef\]](#)

85. Chen, Y.; Hu, Z.; Zhi, C.; Han, J.; Deng, S.; Yin, J. Chatunitest: A framework for llm-based test generation. In *FSE 2024: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 572–576. [\[CrossRef\]](#)
86. Yuan, Z.; Liu, M.; Ding, S.; Wang, K.; Chen, Y.; Peng, X.; Lou, Y. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* **2024**, *1*, 1703–1726. [\[CrossRef\]](#)
87. Shou, C.; Liu, J.; Lu, D.; Sen, K. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. *arXiv* **2024**, arXiv:2401.11108. [\[CrossRef\]](#)
88. Hu, J.; Zhang, Q.; Yin, H. Augmenting Greybox Fuzzing with Generative AI. *arXiv* **2023**, arXiv:2306.06782. [\[CrossRef\]](#)
89. Li, Z.; Wang, C.; Liu, Z.; Wang, H.; Chen, D.; Wang, S.; Gao, C. CCTEST: Testing and Repairing Code Completion Systems. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1238–1250. [\[CrossRef\]](#)
90. Dakhama, A.; Even-Mendoza, K.; Langdon, W.B.; Menendez, H.; Petke, J. SearchGEM5: Towards Reliable Gem5 with Search Based Software Testing and Large Language Models. In *Proceedings of the Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2024; Volume 14415, pp. 160–166. [\[CrossRef\]](#)
91. Lemieux, C.; Inala, J.P.; Lahiri, S.K.; Sen, S. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 919–931.
92. Xia, C.S.; Wei, Y.; Zhang, L. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1482–1494.
93. Deng, Y.; Xia, C.S.; Peng, H.; Yang, C.; Zhang, L. 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*; ACM: New York, NY, USA, 2023; pp. 423–435. [\[CrossRef\]](#)
94. Peng, Y.; Wu, J.; Ling, X.; Li, Z.; Luo, T.; Wu, Y. OptionFuzz: Fuzzing SMT Solvers with Optimized Option Exploration via Large Language Models. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 260–270. [\[CrossRef\]](#)
95. Meng, R.; Mirchev, M.; Böhme, M.; Roychoudhury, A. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 2024 Network and Distributed System Security Symposium*; Internet Society: Reston, VA, USA, 2024. [\[CrossRef\]](#)
96. Yang, Z.; Peng, H.; Jiang, Y.; Li, X.; Du, H.; Wang, S.; Liu, J. ChatHTTPFuzz: Large language model-assisted IoT HTTP fuzzing. *Int. J. Mach. Learn. Cybern.* **2025**, *16*, 4577–4598. [\[CrossRef\]](#)
97. Cheng, M.; Zhu, K.; Chen, Y.; Yang, G.; Lu, Y.; Lu, C. MSFuzz: Augmenting Protocol Fuzzing with Message Syntax Comprehension via Large Language Models. *Electronics* **2024**, *13*, 2632. [\[CrossRef\]](#)
98. Shi, W.; Zhang, Y.; Xing, X.; Xu, J. Harnessing Large Language Models for Seed Generation in Greybox Fuzzing. *arXiv* **2024**, arXiv:2411.18143. [\[CrossRef\]](#)
99. Yang, C.; Deng, Y.; Lu, R.; Yao, J.; Liu, J.; Jabbarvand, R.; Zhang, L. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. Acm Program. Lang.* **2024**, *8*, 709–735. [\[CrossRef\]](#)
100. Cao, D.; Hong, Y.; Pan, Q.; Wu, J. Program Interoperable Large Language Model Software Testing Scheme: A Case Study on JavaScript Engine Fuzzing. *IEEE Trans. Dependable Secur. Comput.* **2025**, *22*, 6179–6195. [\[CrossRef\]](#)
101. Zhang, Y.; Zhu, K.; Peng, J.; Lu, Y.; Chen, Q.; Li, Z. StatePre: A Large Language Model-Based State-Handling Method for Network Protocol Fuzzing. *Electronics* **2025**, *14*, 1931. [\[CrossRef\]](#)
102. Hao, Y.; Zhang, H.; Li, G.; Du, X.; Qian, Z.; Sani, A.A. Demystifying the dependency challenge in kernel fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 659–671. [\[CrossRef\]](#)
103. Jeong, D.R.; Kim, K.; Shivakumar, B.; Lee, B.; Shin, I. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*; IEEE: Piscataway, NJ, USA, 2019; pp. 754–768. [\[CrossRef\]](#)
104. Teplyuk, P.; Yakunin, A.; Sharlaev, E. Study of security flaws in the Linux kernel by fuzzing. In *Proceedings of the 2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*; IEEE: Piscataway, NJ, USA, 2020; pp. 1–5. [\[CrossRef\]](#)
105. Yang, C.; Zhao, Z.; Zhang, L. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *ASPLOS '25: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2025; Volume 2, pp. 560–573. [\[CrossRef\]](#)
106. Kapusta, P.; Duch, P.; Majchrowicz, M.; Królik, A. System for Automatic Bug Detection in Code and Programs Using LLMs and AI Agents. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2026; Volume 15948, pp. 114–125. [\[CrossRef\]](#)

107. Wang, E.; Xie, W.; Li, S.; Liu, R.; Zhou, Y.; Wang, Z.; Ma, S.; Yang, W.; Wang, B. Large Language Model-Powered Protected Interface Evasion: Automated Discovery of Broken Access Control Vulnerabilities in Internet of Things Devices. *Sensors* **2025**, *25*, 2913. [\[CrossRef\]](#)
108. Ding, Y.; Zhao, X.; Wu, Z.; Zhu, Y.; Bai, L.; Han, H. ADFEmu: Enhancing Firmware Fuzzing with Direct Memory Access (DMA) Input Emulation Using Concolic Execution and Large Language Models (LLMs). *Comput. Mater. Contin.* **2025**, *84*, 5977–5993. [\[CrossRef\]](#)
109. Yang, W.; Wang, E.; Gui, Z.; Zhou, Y.; Wang, B.; Xie, W. An MLLM-Assisted Web Crawler Approach for Web Application Fuzzing. *Appl. Sci.* **2025**, *15*, 962. [\[CrossRef\]](#)
110. Lin, X.; Xie, Q.; Zhao, B.; Tian, Y.; Zonouz, S.; Ruan, N.; Li, J.; Beyah, R.; Ji, S. PromFuzz: Leveraging LLM-Driven and Bug-Oriented Composite Analysis for Detecting Functional Bugs in Smart Contracts. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: Piscataway, NJ, USA, 2025; pp. 1044–1056. [\[CrossRef\]](#)
111. Gu, F.; Guo, Q.; Lu, J.; Xie, Q.; Zhao, B.; Lu, K.; Li, H.; Gong, X. Sheep’s Clothing, Wolf’s Data: Detecting Server-Induced Client Vulnerabilities in Windows Remote IPC. In *Proceedings of the 32nd Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, 23–28 February 2025.
112. Liu, H.; Zheng, L.; Gan, S.; Zhang, C.; Gao, Z.; Zhang, H.; Zeng, Y.; Jiang, Z.; Yang, J. EAGLEYE: Exposing Hidden Web Interfaces in IoT Devices via Routing Analysis. In *Proceedings of the 32nd Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, 23–28 February 2025.
113. Liu, F.; Zhang, Y.; Luo, J.; Dai, J.; Chen, T.; Yuan, L.; Yu, Z.; Shi, Y.; Li, K.; Zhou, C.; et al. Make Agent Defeat Agent: Automatic Detection of Taint-Style Vulnerabilities in LLM-based Agents. In *Proceedings of the 34th Usenix Security Symposium*; USENIX Association: Berkeley, CA, USA, 2025; pp. 3767–3786.
114. Jiang, S.; Zhu, C.; Khurshid, S. OBsmith: LLM-Powered JavaScript Obfuscator Testing. *Proc. ACM Program. Lang.* **2026**, *10*, 114–142. [\[CrossRef\]](#)
115. Zhang, H.; Wang, D.; Liu, Y.; Chen, K.; Wang, J.; Ying, X.; Liu, L.; Wang, W. ORFuzz: Fuzzing the “Other Side” of LLM Safety-Testing Over-Refusal. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: Piscataway, NJ, USA, 2025. [\[CrossRef\]](#)
116. Zhao, Y.; Wu, M.; Hu, X.; Xia, X. HFuzzer: Testing Large Language Models for Package Hallucinations via Phrase-based Fuzzing. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: Piscataway, NJ, USA, 2025. [\[CrossRef\]](#)
117. Cohen, A. In-Browser LLM-Guided Fuzzing for Real-Time Prompt Injection Testing in Agentic AI Browsers. *arXiv* **2025**, arXiv:2510.13543. [\[CrossRef\]](#)
118. Jiang, S.; Zhu, C.; Khurshid, S. Generating executable oracles to check conformance of client code to requirements of JDK Javadocs using LLMs. *arXiv* **2024**, arXiv:2411.01789. [\[CrossRef\]](#)
119. Paria, S.; Dasgupta, A.; Bhunia, S. DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection. *arXiv* **2023**, arXiv:2308.06932. [\[CrossRef\]](#)
120. Tang, Z.; Lin, J.; Dong, W.; Ma, H.; Liu, T. SyzForge: An Automated System Call Specification Generation Process for Efficient Kernel Fuzzing. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15747, pp. 118–139. [\[CrossRef\]](#)
121. Yang, Y.; Yao, S.; Chen, J.; Lee, W. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *Proceedings of the 34th Usenix Security Symposium*; USENIX Association: Berkeley, CA, USA, 2025; pp. 6299–6318.
122. Chen, P.; Liu, X.; Wang, Y. Fine-Tune LLMs for PLC Code Security: An Information-Theoretic Analysis. *Mathematics* **2025**, *13*, 3211. [\[CrossRef\]](#)
123. Kang, S.; Chen, B.; Yoo, S.; Lou, J.G. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *Empir. Softw. Eng.* **2025**, *30*, 45. [\[CrossRef\]](#)
124. Liu, S.; Marefat, S.; Tsai, O.; Chen, Y.; Deng, Z.; Wang, J.; Tayebi, M.A. PrediQL: Automated Testing of GraphQL APIs with LLMs. *arXiv* **2025**, arXiv:2510.10407. [\[CrossRef\]](#)
125. Guan, H.; Bai, G.; Liu, Y. CrossProbe: LLM-Empowered Cross-Project Bug Detection for Deep Learning Frameworks. *Proc. Acm Softw. Eng.* **2025**, *2*, 2430–2452. [\[CrossRef\]](#)
126. Ngo, L.Q.; Nhat, T.T.; Quyen, N.H.; Pham, V.H.; Duy, P.T. UniFuzz: A Unified Fuzzing Framework for Smart Contract Vulnerability Detection with LLMs and Audit Reports. In *Proceedings of the 2025 2nd International Conference on Cryptography and Information Security (VCRIS)*; IEEE: Piscataway, NJ, USA, 2025; pp. 1–6. [\[CrossRef\]](#)
127. Ali, G.M.A.; Chen, H.; Wang, Z.; Du, C. CrossGuard: Runtime-Adaptive LLM Fuzzing for Cross-Contract Vulnerabilities Detection. *Concurr. Comput. Pract. Exp.* **2025**, *37*, e70421. [\[CrossRef\]](#)
128. Fu, Y.; German, R.; Sardy, L.A. HyFuzz: A Hybrid AI-Enhanced Vulnerability Detection Framework. In *2025 IEEE 9th Forum on Research and Technologies for Society and Industry (RTSI)*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 350–355. [\[CrossRef\]](#)

129. Abtahi, S.M.; Azim, A. Securing LLM-Generated Embedded Firmware through AI Agent-Driven Validation and Patching. In *2025 IEEE International Conference on Collaborative Advances in Software and Computing, CASCON 2025*; IEEE: Piscataway, NJ, USA, 2025; pp. 210–218. [\[CrossRef\]](#)
130. Wang, W.; Gao, W.; Wang, Z.; Cao, S.; Lin, X.; Gu, D. AePPollo: Automated Exploit Generation for Prototype Pollution Vulnerabilities in Node.js Application. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*; Knowledge Systems Institute Graduate School: Skokie, IL, USA, 2025; pp. 19–24. [\[CrossRef\]](#)
131. Qiu, F.; Ji, P.; Hua, B.; Wang, Y. CHEMFUZZ: Large Language Models-Assisted Fuzzing for Quantum Chemistry Software Bug Detection. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2023; pp. 103–112. [\[CrossRef\]](#)
132. Mashhadi, E.; Hemmati, H. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*; IEEE: Piscataway, NJ, USA, 2021; pp. 505–509. [\[CrossRef\]](#)
133. Zhang, Y.; Wang, J.; Berzin, D.; Mirchev, M.; Liu, D.; Arya, A.; Chang, O.; Roychoudhury, A. Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*; ACM: New York, NY, USA, 2026.
134. Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; Svyatkovskiy, A. InferFix: End-to-End Program Repair with LLMs. In *ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1646–1656. [\[CrossRef\]](#)
135. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*; IEEE: Piscataway, NJ, USA, 2023; pp. 2339–2356. [\[CrossRef\]](#)
136. Kim, Y.; Shin, S.; Kim, H.; Yoon, J. Logs In, Patches Out: Automated Vulnerability Repair via {Tree-of-Thought}{LLM} Analysis. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA, USA, 13–15 August 2025; pp. 4401–4419.
137. Herter, P.; Ahlrichs, V.; Açılan, R.; Horsch, J. GPTrace: Effective Crash Deduplication Using LLM Embeddings. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*, Rio de Janeiro, Brazil, 12–18 April 2026.
138. Chen, J.; Liu, J.; Wang, J.; Li, D. LLM-Based Method for Generating Vulnerable Code Equivalents. In *Proceedings of the SPIE—The International Society for Optical Engineering*; SPIE: Bellingham, WA, USA, 2025; Volume 13519. [\[CrossRef\]](#)
139. Sim, H.; Cho, H.; Go, Y.; Fu, Z.; Shokri, A.; Ravindran, B. Large Language Model-Powered Agent for C to Rust Code Translation. *arXiv* **2025**, arXiv:2505.15858. [\[CrossRef\]](#)
140. Bai, Y.; Palit, T. RustAssure: Differential Symbolic Testing for LLM-Transpiled C-to-Rust Code. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*; IEEE: Piscataway, NJ, USA, 2025; pp. 534–546. [\[CrossRef\]](#)
141. Xia, C.S.; Zhang, L. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*; ACM: New York, NY, USA, 2024; pp. 819–831. [\[CrossRef\]](#)
142. Sheng, Z.; Xu, Q.; Huang, J.; Woodcock, M.; Huang, H.; Donaldson, A.F.; Gu, G.; Huang, J. All You Need Is A Fuzzing Brain: An LLM-Powered System for Automated Vulnerability Detection and Patching. *arXiv* **2025**, arXiv:2509.07225. [\[CrossRef\]](#)
143. Huang, J. Demo: Dynamic Defense Deployment with LLM-assisted Client-side Fuzzing in Web Applications. In *Proceedings of the 2025 Silicon Valley Cybersecurity Conference*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 1–3. [\[CrossRef\]](#)
144. Mathews, A.; Rao, S.M.; Faizan, S.; Khatiwada, R.; Bhatta, Y. Comprehensive Web Fuzzer with AI Mitigation and Custom Payload. In *Proceedings of the 7th International Conference on Inventive Material Science and Applications*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 1463–1468. [\[CrossRef\]](#)
145. Yang, K.; Zhang, Y.; Li, Z.; Tao, G.; Xu, J.; Liao, X. HarnessAgent: Scaling Automatic Fuzzing Harness Construction with Tool-Augmented LLM Pipelines. *arXiv* **2025**, arXiv:2512.03420. [\[CrossRef\]](#)
146. Liu, Y.; Deng, J.; Jia, X.; Wang, Y.; Wang, M.; Huang, L.; Wei, T.; Su, P. PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models. In *Proceedings of the CCS 2025-Proceedings of the 2025 Acm Sigsac Conference on Computer and Communications Security*; Association for Computing Machinery, Inc.: New York, NY, USA, 2025; pp. 1559–1573. [\[CrossRef\]](#)
147. Zhang, Z.; Li, L.; Liang, R.; Chen, K. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG. *Proc. Acm Softw. Eng.* **2025**, *2*, 848–870. [\[CrossRef\]](#)
148. Li, S.; Xie, K.; Li, Y.; Li, H.; Ren, Y.; Sun, L.; Zhu, H. TransferFuzz-Pro: Large Language Model Driven Code Debugging Technology for Verifying Propagated Vulnerability. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2396–2411. [\[CrossRef\]](#)
149. Lu, M.; Ding, S.; Alaca, F.; Charland, P. Semantic-Aware Fuzzing: An Empirical Framework for LLM-Guided, Reasoning-Driven Input Mutation. *arXiv* **2025**, arXiv:2509.19533. [\[CrossRef\]](#)
150. OpenAI. Introducing GPT-5.5. 2026. Available online: <https://openai.com/index/introducing-gpt-5-5/> (accessed on 3 May 2026).

151. Anthropic. Introducing Claude Opus 4.7. 2026. Available online: <https://www.anthropic.com/news/claude-opus-4-7> (accessed on 3 May 2026).
152. Anthropic. Project Glasswing: Securing Critical Software for the AI Era. 2026. Available online: <https://www.anthropic.com/glasswing> (accessed on 3 May 2026).
153. Chen, C.; Dolan-Gavitt, B.; Lin, Z. ELFUZZ: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *Proceedings of the 34th Usenix Security Symposium*; USENIX Association: Berkeley, CA, USA, 2025; pp. 6279–6298.
154. Androutsopoulos, G.; Bianchi, A. deepSURF: Detecting Memory Safety Vulnerabilities in Rust Through Fuzzing LLM-Augmented Harnesses. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*; IEEE: Piscataway, NJ, USA, 2026.
155. Ma, W.; Cui, B. Fuzzing IoT Devices via Android App Interfaces with Large Language Model. In *Lecture Notes on Data Engineering and Communications Technologies*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2024; Volume 193, pp. 87–99. [\[CrossRef\]](#)
156. Mei, L.; Yao, J.; Ge, Y.; Wang, Y.; Bi, B.; Cai, Y.; Liu, J.; Li, M.; Li, Z.Z.; Zhang, D.; et al. A Survey of Context Engineering for Large Language Models. *arXiv* **2025**, arXiv:2507.13334. [\[CrossRef\]](#)
157. Mohsenimofidi, S.; Galster, M.; Treude, C.; Baltes, S. Context Engineering for AI Agents in Open-Source Software. In *Proceedings of the 23rd IEEE/ACM International Conference on Mining Software Repositories*; Association for Computing Machinery: New York, NY, USA, 2026. [\[CrossRef\]](#)
158. Lopopolo, R. Harness Engineering: Leveraging Codex in an Agent-First World. 2026. Available online: <https://openai.com/index/harness-engineering/> (accessed on 11 April 2026).
159. Rajasekaran, P. Harness Design for Long-Running Application Development. 2026. Available online: <https://www.anthropic.com/engineering/harness-design-long-running-apps> (accessed on 11 April 2026).
160. Zheng, T.; Shao, J.; Dai, J.; Jiang, S.; Chen, X.; Shen, C. RESTLess: Enhancing State-of-the-Art REST API Fuzzing with LLMs in Cloud Service Computing. *IEEE Trans. Serv. Comput.* **2024**, *17*, 4225–4238. [\[CrossRef\]](#)
161. Chen, J.; Chen, Y.; Pan, Z.; Chen, Y.; Li, Y.; Li, Y.; Zhang, M.; Shen, Y. DynER: Optimized Test Case Generation for Representational State Transfer (REST)ful Application Programming Interface (API) Fuzzers Guided by Dynamic Error Responses. *Electronics* **2024**, *13*, 3476. [\[CrossRef\]](#)
162. Chen, X.; Liu, J.; Zhang, Y.; Hu, Q.; Han, Y.; Zhang, R.; Ran, J.; Yan, L.; Huang, B.; Ma, S. TraceAwareness and dual-strategy fuzz testing: Enhancing path coverage and crash localization with stochastic science and large language models. *Comput. Electr. Eng.* **2025**, *123*, 110266. [\[CrossRef\]](#)
163. Gu, A.; Roziere, B.; Leather, H.J.; Solar-Lezama, A.; Synnaeve, G.; Wang, S. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *ICML'24: Proceedings of the 41st International Conference on Machine Learning*; JMLR: Norfolk, MA, USA, 2024; Volume 235, pp. 16568–16621.
164. Chen, N.; Li, Z.; Bao, K.; Lin, J.; Liu, D. Chain of Execution Supervision Promotes General Reasoning in Large Language Models. In *Proceedings of the Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2025.
165. Bazalii, M.; Fleischer, M. Orion: Fuzzing Workflow Automation. *arXiv* **2025**, arXiv:2509.15195. [\[CrossRef\]](#)
166. Ugare, S.; Chandra, S. Agentic Code Reasoning. *arXiv* **2026**, arXiv:2603.01896. [\[CrossRef\]](#)
167. Lin, Y.; Tang, S.; Lyu, B.; Wu, J.; Lin, H.; Yang, K.; Li, J.; Xia, M.; Chen, D.; Arora, S.; et al. Goedel-Prover: A Frontier Model for Open-Source Automated Theorem Proving. In *Proceedings of the Conference on Language Modeling*, Montreal, QC, Canada, 7–10 October 2025.
168. Luo, H.; Feng, H.; Sun, Q.; Xu, C.; Zheng, K.; Wang, Y.; Yang, T.; Hu, H.; Tang, Y. AgentMath: Empowering Mathematical Reasoning for Large Language Models via Tool-Augmented Agent. In *Proceedings of the International Conference on Learning Representations*, Rio de Janeiro, Brazil, 24–28 April 2026.
169. Chen, Y.; Chen, J.; Meng, R.; Yin, J.; Li, N.; Fan, C.; Wang, C.; Pfister, T.; Yoon, J. TUMIX: Multi-Agent Test-Time Scaling with Tool-Use Mixture. In *Proceedings of the International Conference on Learning Representations*, Rio de Janeiro, Brazil, 24–28 April 2026.
170. Li, X.; Yuan, Z.; Zhang, Z.; Sun, Y.; Zhang, L. Towards Large Language Model Guided Kernel Direct Fuzzing. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15693, pp. 33–42. [\[CrossRef\]](#)
171. Zhu, S.; Gan, S.; Wang, X.; Nan, Y. GLMFuzz: Vulnerability Knowledge Guided Prompting for Efficient Network Protocol Fuzzing. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*; Institute of Electrical and Electronics Engineers: Piscataway, NJ, USA, 2025; pp. 564–575. [\[CrossRef\]](#)
172. Ning, B.; Zong, X.; He, K. MALF: A Multi-Agent LLM Framework for Intelligent Fuzzing of Industrial Control Protocols. *arXiv* **2025**, arXiv:2510.02694. [\[CrossRef\]](#)
173. Sun, J.; Yin, Z.; Zhang, H.; Chen, X.; Zheng, W. Adversarial generation method for smart contract fuzz testing seeds guided by chain-based LLM. *Autom. Softw. Eng.* **2025**, *32*, 12. [\[CrossRef\]](#)

174. Nguyen, M.D.; Nguyen, H.N.; de Souza, L.M.; Karagiannis, S.; Cavalli, A.R.; de Oca, E.M. An Intelligent Network Fuzzer for Protocol Testing in Healthcare Systems. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15999, pp. 213–231. [\[CrossRef\]](#)
175. Tan, S.; Li, Z.; Liu, Z.; Patel, H.; Tan, Z. Automated Model-Based Fuzzing for 5G O-RAN. In *Proceedings of the 2025 the 31st Annual International Conference on Mobile Computing and Networking*; Association for Computing Machinery, Inc.: New York, NY, USA, 2025; pp. 201–215. [\[CrossRef\]](#)
176. Shi, C.; Yu, J.; Zhao, Z.; Chen, J.; Zhang, F. CGIFuzz: Enabling Gray-Box Fuzzing for Web CGI of IoT Devices. *IEEE Trans. Inf. Forensics Secur.* **2025**, *20*, 10921–10936. [\[CrossRef\]](#)
177. Chen, J.; Zhang, B.; Chen, J.; Li, R.; Tang, C. Chat4seed: Semantic-Awareness Highly Structured Seed Generation for Fuzzing. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*; IEEE: Piscataway, NJ, USA, 2025; pp. 552–563. [\[CrossRef\]](#)
178. Cheng, J.; Fang, D.; Gu, Z.; Lv, S.; Si, S.; Sun, L. Discovering PLC Web Application Vulnerabilities Impacting Physical Control Using LLM-Based Fuzzing. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15686, pp. 49–61. [\[CrossRef\]](#)
179. Luo, Z.; Du, Q.; Wang, Y.; Roychoudhury, A.; Jiang, Y. Enhancing Protocol Fuzzing via Diverse Seed Corpus Generation. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2693–2709. [\[CrossRef\]](#)
180. Qian, B.; Gu, Y.; Wu, C.; Mu, Y.; Sun, X. Fuzz Testing for Polymorphic Network Firmware: A Collaborative Optimization Framework of LLMs and Neural Networks. In *Proceedings of the Acm Conext-2025 Workshop on the Polymorphic Network*; ACM: New York, NY, USA, 2025; pp. 13–18. [\[CrossRef\]](#)
181. Dong, Y.; Meng, X.; Yu, N.; Li, Z.; Guo, S. Fuzz-testing meets llm-based agents: An automated and efficient framework for jailbreaking text-to-image generation models. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP)*; IEEE: Piscataway, NJ, USA, 2025; pp. 373–391.
182. Ni, Y.; Li, S. Interleaving Large Language Models for Compiler Testing. *Proc. ACM Program. Lang.* **2025**, *9*, 815–841. [\[CrossRef\]](#)
183. Zhang, K.; Li, Z.; Wu, D.; Wang, S.; Xia, X. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *Proceedings of the 34th Usenix Security Symposium*; USENIX Association: Berkeley, CA, USA, 2025; pp. 6999–7018.
184. Ou, S.; Li, Y.; Yu, L.; Wei, C.; Wen, T.; Chen, Q.; Chen, Y.; Tang, H.; Pan, Z. MirrorFuzz: Leveraging LLM and Shared Bugs for Deep Learning Framework APIs Fuzzing. *IEEE Trans. Softw. Eng.* **2025**, *52*, 360–375. [\[CrossRef\]](#)
185. Gong, H.; Yan, C.; Xue, Y.; Guo, Y. Network Protocol Security Evaluation via LLM-Enhanced Fuzzing in Extended Pro-FuzzBench. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15863, pp. 522–533. [\[CrossRef\]](#)
186. Quan, L.; Xie, X.; Guo, Q.; Jiang, L.; Chen, S.; Wang, J.; Li, X. TensorJSFuzz: Effective Testing of Web-Based Deep Learning Frameworks via Input-Constraint Extraction. In *Proceedings of the ACM Web Conference*; Association for Computing Machinery, Inc.: New York, NY, USA, 2025; pp. 3405–3414. [\[CrossRef\]](#)
187. Ke, Y.; Dong, H.; Chen, H.; Hou, J.; Wan, X.; Tian, X. LLM-RLFuzz: An Intelligent Fuzzing Framework for IoT Protocol. In *Proceedings of the Communications in Computer and Information Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2026; Volume 2539, pp. 177–188. [\[CrossRef\]](#)
188. Gao, G.; Gan, S.; Wang, X.; Zhu, S. LLMUZZ: LLM-based seed optimization for black-box device fuzzing. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Trustcom*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 1209–1216. [\[CrossRef\]](#)
189. Duan, B.; Mahmud, T.; Che, M.; Yan, Y.; Dong, N.; Kim, D.D.; Yang, G. Harnessing LLMs for Document-Guided Fuzzing of OpenCV Library. In *2025 IEEE International Conference on Software Maintenance and Evolution*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 73–84. [\[CrossRef\]](#)
190. Wang, J.; Yu, L.; Luo, X. Llmif: Augmented large language model for fuzzing iot devices. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP)*; IEEE: Piscataway, NJ, USA, 2024; pp. 881–896.
191. Tang, Z.; Lin, J.; Dong, W.; Ma, H. SyzMutateX: Optimizing Syzkaller Fuzzing with LLM-Driven Mutation and Adaptive Energy Allocation. In *Proceedings of the 2025 International Conference on Digital Management and Information Technology*; Association for Computing Machinery, Inc.: New York, NY, USA, 2025; pp. 327–333. [\[CrossRef\]](#)
192. Xu, Y.J.; Jia, H.R.; Chen, L.G.; Wang, X.; Zeng, Z.R.; Wang, Y.D.; Gao, Q.; Ye, W.; Zhang, S.K.; Wu, Z.H. ISC4DGF: Enhancing Directed Grey-Box Fuzzing with Initial Seed Corpus Generation Driven by Large Language Models. *J. Comput. Sci. Technol.* **2025**, *40*, 1662–1677. [\[CrossRef\]](#)
193. Zhang, K.; Wang, S.; Han, J.; Zhu, X.; Li, X.; Wang, S.; Wen, S. Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models. In *International Conference on Software Engineering*; IEEE Computer Society: Piscataway, NJ, USA, 2025; pp. 3110–3122. [\[CrossRef\]](#)

194. Wang, D.; Zhou, G.; Chen, L.; Li, D.; Miao, Y. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*; ACM: New York, NY, USA, 2024; pp. 735–749. [\[CrossRef\]](#)
195. Lin, R.; Wang, R.; Hu, G.; Xu, X. LMFuzz: Program repair fuzzing based on large language models. *Autom. Softw. Eng.* **2026**, *33*, 25. [\[CrossRef\]](#)
196. Lin, S.; Chen, F.; Xi, L.; Xie, K.; Zheng, Y.; Fei, H.; Sun, Y.; Zhu, H. ScenarioFuzz-LLM: Enhancing Diversity in Autonomous Driving Scenario Fuzzing with LLMs. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 1581–1586. [\[CrossRef\]](#)
197. Cao, B.; Tong, W.; Tang, Z.; Wang, Z.; Huang, H.; Yan, Y. Nüwa: Enhancing MLIR Fuzzing with LLM-Driven Generation and Adaptive Mutation. In *2025 IEEE International Conference on Software Maintenance and Evolution*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 541–552. [\[CrossRef\]](#)
198. Lohiya, D.; Golla, M.R.; Godbole, S.; Krishna, P.R. Poster: GptCombFuzz: Combinatorial Oriented LLM Seed Generation for effective Fuzzing. In *2024 IEEE Conference on Software Testing, Verification and Validation*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 438–441. [\[CrossRef\]](#)
199. Yu, B.; Song, Q.; Cai, C. Large Language Model guided State Selection Approach for Fuzzing Network Protocol. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 52–57. [\[CrossRef\]](#)
200. Nakanishi, H.; Hasegawa, K.; Hidano, S.; Fukushima, K.; Togawa, N. LLM-based Fuzzing Method Using UI-based Input Value Generation for IoT Devices. In *2025 IEEE 14th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*; IEEE: Piscataway, NJ, USA, 2025; pp. 16–20. [\[CrossRef\]](#)
201. Zhang, Q.; Shen, Y.; Liu, J.; Xu, Y.; Shi, H.; Jiang, Y.; Chang, W. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2024**, *43*, 4238–4249. [\[CrossRef\]](#)
202. Zhong, M.; Zeng, Z.; Guo, Y.; Zhao, D.; Zhang, B.; Li, S.; Peng, H.; Ding, Z. Intelligent test case generation method for fuzzing IoT protocols based on LLM. *Autom. Softw. Eng.* **2026**, *33*, 6. [\[CrossRef\]](#)
203. Mhiri, I.; Börsig, M.; Stark, A.; Baumgart, I. How to Train Your Llama—Efficient Grammar-Based Application Fuzzing Using Large Language Models. In *Proceedings of the Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15396, pp. 239–257. [\[CrossRef\]](#)
204. Zhu, T.; Skapars, A.; Mackenzie, F.; Kehoe, D.; Newton, W.; Embury, S.; Sun, Y. SaFliTe: Fuzzing Autonomous Systems via Large Language Models. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2026; Volume 16045, pp. 245–258. [\[CrossRef\]](#)
205. Chen, X.; Liu, J.; Zhang, Y.; Hu, Q.; Han, Y.; Zhang, R.; Ran, J.; Yan, L.; Huang, B.; Ma, S.; et al. WebFuzzAuto: An Automated Fuzz Testing Tool Integrating Reinforcement Learning and Large Language Models for Web Security. In *Proceedings of the 2024 12th International Conference on Information Systems and Computing Technology*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 1–10. [\[CrossRef\]](#)
206. Gu, Q. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the ESEC/FSE 2023-Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery, Inc.: New York, NY, USA, 2023; pp. 2201–2203. [\[CrossRef\]](#)
207. Nunez, A.; Islam, N.T.; Jha, S.K.; Najafirad, P. AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing. In *Proceedings of the NeurIPS 2024 Workshop on Safe and Trustworthy Agents*, Vancouver, BC, Canada, 15 December 2024.
208. Wang, T.; Wang, R.; Chen, Y.; Yu, L.; Pan, Z.; Zhang, M.; Ma, H.; Zheng, J. Enhancing Black-box Compiler Option Fuzzing with LLM through Command Feedback. In *International Symposium on Software Reliability Engineering*; IEEE Computer Society, Ed.; IEEE: Piscataway, NJ, USA, 2024; pp. 319–330. [\[CrossRef\]](#)
209. He, J.; Dai, H.; Sui, R.; Yuan, X.; Liu, D.; Feng, H.; Liu, X.; Yang, W.; Cui, B.; Li, K. EvilPromptFuzzer: Generating inappropriate content based on text-to-image models. *Cybersecurity* **2024**, *7*, 70. [\[CrossRef\]](#)
210. Ma, X.; Luo, L.; Zeng, Q. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *Proceedings of the 33rd Usenix Security Symposium*; USENIX Association: Berkeley, CA, USA, 2024; pp. 4783–4800.
211. Meng, R.; Duck, G.J.; Roychoudhury, A. Large Language Model assisted Hybrid Fuzzing. *arXiv* **2024**, arXiv:2412.15931. [\[CrossRef\]](#)
212. Zhou, Z.; Yang, Y.; Wu, S.; Huang, Y.; Chen, B.; Peng, X. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *2024 39th ACM/IEEE International Conference on Automated Software Engineering*; Association for Computing Machinery, Inc.: New York, NY, USA, 2024; pp. 1633–1644. [\[CrossRef\]](#)

213. Wang, Y.; Guan, Q.; Chen, J. Poster: Fuzzing for Command Injections in Medical Software with Large Language Models. In *2024 IEEE International Conference on Mobility, Operations, Services and Technologies*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 272–274. [\[CrossRef\]](#)
214. Ou, X.; Li, C.; Jiang, Y.; Xu, C. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2024; Volume 4, pp. 298–312. [\[CrossRef\]](#)
215. Hasegawa, K.; Nakanishi, H.; Hidano, S.; Fukushima, K.; Hashimoto, K.; Togawa, N. Automated Test Input Generation Based on Web User Interfaces via Large Language Models. In *Proceedings of the 10th International Conference on Internet of Things, Big Data and Security-IoTBDS*; SciTePress: Setubal, Portugal, 2025; pp. 297–304. [\[CrossRef\]](#)
216. Liao, R.; Yan, X.; Pang, Z.; Zhu, K. Balancing Validity and Vulnerability: Knowledge-Driven Seed Generation via LLMs for Deep Learning Library Fuzzing. *Appl. Sci.* **2025**, *15*, 10396. [\[CrossRef\]](#)
217. Zhang, Y.; Zhang, X. Beyond Imprecise Distance Metrics: Trace-Guided Directed Greybox Fuzzing via LLM-Predicted Call Stacks. *arXiv* **2026**, arXiv:2510.23101. [\[CrossRef\]](#)
218. Gao, H.; Yang, Y.; Sun, M.; Wu, J.; Zhou, Y.; Xu, B. Clozmaster: Fuzzing Rust Compiler by Harnessing LLMs for Infilling Masked Real Programs. In *International Conference on Software Engineering*; IEEE Computer Society: Piscataway, NJ, USA, 2025; pp. 1422–1435. [\[CrossRef\]](#)
219. Xu, H.; Zhao, Y.; Wang, H. Directed Greybox Fuzzing via Large Language Model. *arXiv* **2025**, arXiv:2505.03425. [\[CrossRef\]](#)
220. Shiraishi, M.; Cao, Y.; Shinagawa, T. Effective Command-line Interface Fuzzing with Path-Aware Large Language Model Orchestration. *arXiv* **2025**, arXiv:2511.20555. [\[CrossRef\]](#)
221. Dakhama, A.; Even-Mendoza, K.; Langdon, W.B.; Menéndez, H.D.; Petke, J. Enhancing search-based testing with LLMs for finding bugs in system simulators. *Autom. Softw. Eng.* **2025**, *32*, 63. [\[CrossRef\]](#)
222. Lin, S. Hybrid Fuzzing with LLM-Guided Input Mutation and Semantic Feedback. In *Proceedings of the 2025 Cyber Awareness and Research Symposium (CARS)*; IEEE: Piscataway, NJ, USA, 2025; pp. 1–9. [\[CrossRef\]](#)
223. Mcshane, J.; Celik, L.; Aideyan, I.; Brooks, R.; Pesé, M.D. LLM-Powered Fuzz Testing of Automotive Diagnostic Protocols. In *Proceedings of the Sae Technical Papers*; SAE International: Warrendale, PA, USA, 2025. [\[CrossRef\]](#)
224. Huang, L.; Zhao, P.; Chen, H. LLMs are All You Need? Improving Fuzz Testing for MOJO with Large Language Models. In *Proceedings of the 2025 32nd Asia-Pacific Software Engineering Conference (APSEC)*; IEEE: Piscataway, NJ, USA, 2025. [\[CrossRef\]](#)
225. Maklad, Y.; Wael, F.; Elersy, W.; Hamdi, A. Retrieval Augmented Generation Based LLM Evaluation for Protocol State Machine Inference with Chain-of-Thought Reasoning. In *Proceedings of the Lecture Notes in Networks and Systems*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 1416, pp. 313–323. [\[CrossRef\]](#)
226. Cheng, X.; Sang, F.; Zhai, Y.; Zhang, X.; Kim, T. Rug: Turbo Llm for Rust Unit Test Generation. In *International Conference on Software Engineering*; IEEE Computer Society: Piscataway, NJ, USA, 2025; pp. 2983–2995. [\[CrossRef\]](#)
227. Hou, L.; Liu, P.; Ding, J.; Sheng, J.; Le, H.; Chen, Y.; Wang, W. SSFuzz: State-Guided Fuzzing with Shared Feedback for Black-Box IoT Devices. *IEEE Internet Things J.* **2025**, *12*, 49296–49310. [\[CrossRef\]](#)
228. Even-Mendoza, K.; Menéndez, H.D.; Langdon, W.B.; Dakhama, A.; Petke, J.; Bruce, B.R. Search+LLM-based Testing for ARM Simulators. In *Proceedings of the IEEE/acm International Conference on Software Engineering-Software Engineering in Practice*; IEEE: Piscataway, NJ, USA, 2025; pp. 469–480. [\[CrossRef\]](#)
229. Park, S.; Lee, H.; Cha, S.K. Short Paper: Systematic Bug Reproduction with Large Language Model. In *Proceedings of the Lecture Notes in Computer Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 15264, pp. 39–48. [\[CrossRef\]](#)
230. Wu, L.; Rostami, M.; Li, H.; Rajendran, J.; Sadeghi, A.R. {GenHuzz}: An Efficient Generative Hardware Fuzzer. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA, USA, 13–15 August 2025; pp. 1787–1805.
231. Francisco Borja, G.D.R.; Lera, F.J.; Llamas, C.F.; Olivera, V.M. Fuzzing Robotic Software Using HPC and LLM. In *Proceedings of the Lecture Notes in Networks and Systems*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2024; Volume 957, pp. 66–76. [\[CrossRef\]](#)
232. Nakanishi, H.; Hisafuru, K.; Hasegawa, K.; Hidano, S.; Fukushima, K.; Hashimoto, K.; Togawa, N. Initial Seeds Generation Using LLM for IoT Device Fuzzing. In *Proceedings of the 2024 11th International Conference on Internet of Things: Systems, Management and Security*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2024; pp. 5–10. [\[CrossRef\]](#)
233. Sas, C.; Capiluppi, A. AutoFL: A Tool for Automatic Multi-granular Labelling of Software Repositories. *arXiv* **2025**, arXiv:2408.02557. [\[CrossRef\]](#)
234. Mohajer, M.M.; Aleithan, R.; Harzevili, N.S.; Wei, M.; Belle, A.B.; Pham, H.V.; Wang, S. SkipAnalyzer: An Embodied Agent for Code Analysis with Large Language Models. *arXiv* **2023**, arXiv:2310.18532. [\[CrossRef\]](#)
235. Masud, M.T.; Koroniotis, N.; Keshk, M.; Turnbull, B.; Kermanshahi, S.K.; Moustafa, N. Generative fuzzer-driven vulnerability detection in the Internet of Things networks. *Appl. Soft Comput.* **2025**, *174*, 112973. [\[CrossRef\]](#)

236. Li, H.; Li, X.; Nie, Y.; Du, S.; Wu, Q. DKFuzz: An Automated Approach for Testing the Underlying Kernels of Deep Learning Frameworks. In *Proceedings of the 2025 2nd International Conference on Algorithms, Software Engineering and Network Security*; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2025; pp. 163–175. [[CrossRef](#)]
237. Wang, Y.; Wang, W.; Li, D. Dynamic Feedback-Based Vulnerability Mining Method for Highly Closed Terminal Protocols. In *Proceedings of the SPIE-the International Society for Optical Engineering*; SPIE: Bellingham, WA, USA, 2025; Volume 13519. [[CrossRef](#)]
238. Lyu, C.; Wang, Y.; Jie, L.; Zhao, M. LLM-based Dynamic Differential Testing for Database Connectors with Reinforcement Learning-Guided Prompt Selection. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering: New Ideas and Emerging Results*; IEEE: Piscataway, NJ, USA, 2025. [[CrossRef](#)]
239. Kim, S.; Lee, S.Y. Performance Comparison of Prompt Engineering and Fine-Tuning Approaches for Fuzz Driver Generation Using Large Language Models. In *Lecture Notes on Data Engineering and Communications Technologies*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2025; Volume 259, pp. 111–120. [[CrossRef](#)]
240. Sun, Y. Automated Generation and Compilation of Fuzz Driver Based on Large Language Models. In *Proceedings of the ACM International Conference Proceeding Series*; Association for Computing Machinery, Inc.: New York, NY, USA, 2024; pp. 461–468. [[CrossRef](#)]
241. Lyu, Y.; Xie, Y.; Chen, P.; Chen, H. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*; ACM: New York, NY, USA, 2024; pp. 3793–3807. [[CrossRef](#)]
242. Xu, H.; Ma, W.; Zhou, T.; Zhao, Y.; Chen, K.; Hu, Q.; Liu, Y.; Wang, H. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *International Conference on Software Engineering*; IEEE Computer Society: Piscataway, NJ, USA, 2025; pp. 243–254. [[CrossRef](#)]
243. Tran, C.T.; Van Huy, P. A Study on the Application of Large Language Models for Library Security Testing Under Low-Resource Conditions. In *Proceedings of the Communications in Computer and Information Science*; Springer Science and Business Media Deutschland GmbH: Cham, Switzerland, 2026; Volume 2585, pp. 244–255. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.