INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

MACHINE LEARNING-GUIDED GREYBOX FUZZING FOR ADAPTIVE DISCOVERY OF DEEP-STATE SOFTWARE VULNERABILITIES

Talha Hussain[1], Muhammad Bilal[*2], Faheem Haroon[3]

[1] *Faculty of Computing, Riphah International University, Islamabad 46000, Pakistan*
[*2] *Department of Computer Science, University of Gujrat, Pakistan*
[3] *Bahria School of Engineering and Applied Sciences, Bahria University, Islamabad, Pakistan*

[1]thussain98ml@gmail.com; [*2]bilaloit786@gmail.com; [3]faheemharoon.buic@bahria.edu.pk

[2]ORCID: 0009-0005-2762-9188

DOI: https://doi.org/10.5281/zenodo.24080723

Keywords
Greybox fuzzing; machine learning; deep-state vulnerabilities; seed scheduling; neural program smoothing; reinforcement learning; multi-armed bandits; Large Language Models; vulnerability discovery; fuzzing benchmarks.

Article History
Received: *25 June 2026*
Accepted: *16 September 2026*
Published: *02 November 2026*

Copyright @Author
Corresponding Author: * Muhammad Bilal

Abstract

Coverage-guided greybox fuzzing (CGF) is the de-facto choice for automated discovery of memory safety errors and security vulnerabilities in production software. On the other hand, classical CGF makes use of pseudo-random mutation operators, has a hard time to break multi-byte magic values, cryptographic checksums or nested structures and state machines that restrict the input space. The chance of meeting a complicated 32-bit or 64-bit predicate randomly is so low that vanilla fuzzers are found wanting at easily reaching deep into programs. In this survey, we review ML-guided greybox fuzzing as an adaptive, data-driven optimization process that replaces blind heuristics with program surrogates learned from historical mutation data. We propose a novel five-stage architectural taxonomy which aligns machine learning paradigms such as deep neural networks, contextual multi-armed bandits, reinforcement learning and LLMs with the phases of the fuzzing lifecycle: seed generation, seed scheduling and energy allocation, mutation and operator selection, exploration of deep-state paths, and crash triage post-fuzzing. We study state-of-the-art solutions in depth, including the variant adversarial bandit formulation of EcoFuzz, the contextual LinUCB scheduler of LinFuzz, the neural program smoothing with gradient-guided mutation of NEUZZ and MTFuzz, taint-based solutions like Angora and GreyOne, as well as generative fuzzying driven by LLMs such as Fuzz4All, TitanFuzz, ChatAFL, and StateLens. In addition, we evaluate benchmarking approaches, comparing traditional synthetic suites like LAVA-M to contemporary ground- truth platforms, for example FuzzBench, Magma and FIRMRE- B... Finally, some promising research directions for addressing the open challenges of model obsolescence and retraining overhead, scalability in multithreaded and monolithic systems, and overfitting and hallucination in LLM-guided fuzzers are discussed to enable future work toward adaptive, generalizable deep-state vulnerability discovery.

1. INTRODUCTION

Coverage-guided greybox fuzzing or CGF has become the choice for automatically testing software and finding memory safety issues structural logic errors and security flaws in real-world systems (Ahmed et al. 2025). This method works by adding monitoring tools to the target binary to track which parts of the code are being run especially edges or basic blocks. Based on this feedback the fuzzer keeps changing the input data in ways to reach new parts of the program. Even though CGF has found thousands of bugs in practice it struggles when it comes to reaching parts of a program (Rawat et al., 2017).

The real problem lies in deep-state vulnerabilities. These bugs are hidden behind barriers like multi-byte checks, cryptographic checks, nested conditions and tightly controlled state changes (Güler, 2025). Traditional greybox fuzzers use random changes like flipping bits, changing numbers or swapping bytes.. The chance of randomly creating the exact sequence of bytes needed to pass a complex 32-bit or 64-bit check is extremely small like 2 to the power of negative 32 or negative 64. That means the fuzzer often gets stuck in parts of the code (Rieu, 2020).

To fix this researchers in software security have started using machine learning. The goal is to turn the mutation process into something smarter and adaptive. Fuzzing can be seen as a dimensional non-convex optimization problem (Xue et al. 2025). If we think of the input space, as I and the program we're testing as a state transition system S then the job of the fuzzer is to find a function f that maps inputs to states. The aim is to maximize the number of vulnerabilities found in a limited amount of time T (He et al., 2019). In a generalized greybox setting, any fuzzed input I_{fuzz} is generated from an original input I_{orig} via a mutation function parameterized by feedback R:

$$I_{\text{fuzz}} = f(I_{\text{orig}}, R)$$

where R encapsulates execution telemetry, structural data-flow relationships, or learned model gradients.

By replacing heuristic rules with machine learning paradigms such as deep neural networks (DNNs), contextual multi-armed bandits (MAB), dynamic reinforcement learning (RL), and Large Language Models (LLMs) ML-guided fuzzers construct explicit surrogates of program execution behavior (Sivamayil et al., 2023). These models dynamically predict high-reward mutation sites, balance the allocation of execution energy across candidate seeds, and generate syntactically valid inputs capable of traversing complex parsing logic (Chennakesavalu et al., 2026). This review provides a comprehensive analysis of machine learning-guided greybox fuzzing architectures, seed scheduling mechanisms, deep-state path navigation techniques, LLM integrations, and modern ground-truth benchmarking frameworks (Wang et al., 2024).

2. Taxonomy and Architectural Foundations of ML-Guided Fuzzing

When machine learning is added to greybox fuzzing, the step-by-step testing process changes (Narang et al. 2025). Classic greybox fuzzing runs in a loop that has four clear stages: first, creating and changing test cases; second running the target program; third, watching the program state with runtime tools; fourth sorting out crashes. Machine learning-guided greybox fuzzing improves this design by adding learning parts at key decision points inside the fuzzing engine (Li et al., 2018).

An ML-guided greybox fuzzer has five parts that reflect how the learning model works. In the part seed generation and corpus selection ML models create good valid seed inputs before fuzzing starts or they filter a big pool of initial seeds to cut down on repeated runs (Narang et al. 2025). In the part seed scheduling and energy allocation, reinforcement learning and multi-armed bandit techniques pick the best seeds, from the queue and give each seed the right amount of mutation effort using live program state data (Zhang & Cui 2025). In the part targeted input mutation and operator selection the models guess which byte places in a seed most affect branch decisions or change the chances of choosing each mutation operator on the fly (Zhu et al., 2023). In the part runtime state monitoring and neural program smoothing machine learning models learn smooth versions of the program's discrete paths turning sharp branch

conditions into smooth loss functions that help guide gradient-based input creation. The final part, post-fuzzing crash triage and exploitability analysis uses machine learning classifiers to find the root cause remove duplicate crashing inputs and judge how exploitable a vulnerability is (Seljan, 2025).

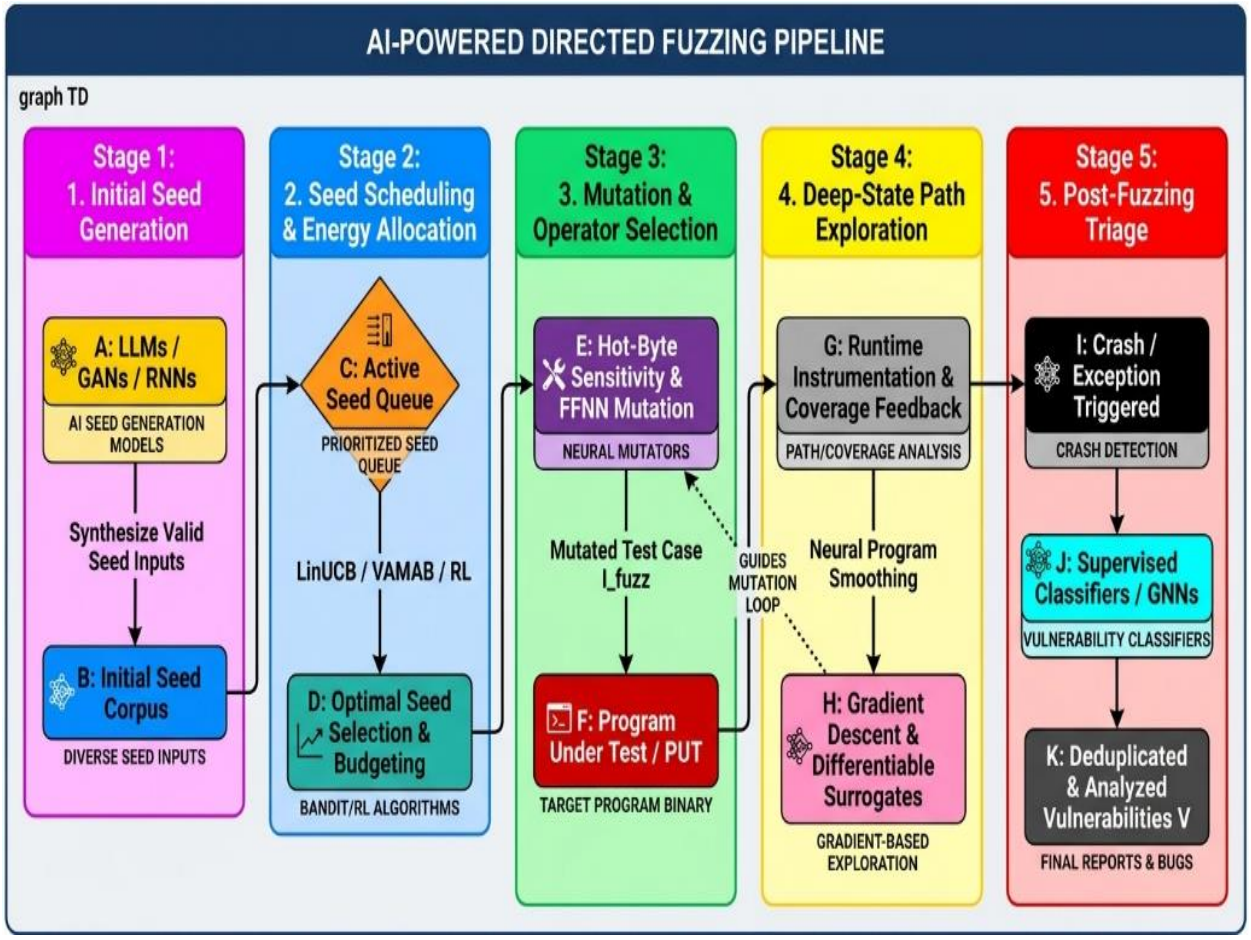


Figure 1: Comprehensive Five-Stage Architecture of the AI-Powered Directed Fuzzing Pipeline, Detailing Interoperable Machine Learning Components Across the Testing Lifecycle.

Table 1: Architectural Classification of Machine Learning Components across Fuzzing Lifecycle

Fuzzing Lifecycle Stage	Primary ML / Algorithmic Models	Operational Mechanism	Addressed Bottleneck
Initial Seed Generation	Large Language Models (LLMs), Generative Adversarial Networks (GANs), Recurrent Neural Networks (RNNs)	Grammatical inference, autoregressive token generation, masked input filling	Parsing stalls caused by strict input format specifications
Seed Scheduling & Energy Allocation	Contextual Multi-Armed Bandits (LinUCB), Adversarial MAB (VAMAB), Reinforcement Learning	Program-sensitive context vectors, reward probability attenuation estimation	Redundant execution of well-explored paths; suboptimal resource budgeting

Mutation & Operator Selection	Feed-Forward Neural Networks (FFNNs), Multi-Task Learning (MTL), Particle Swarm Optimization (PSO)	Hot-byte sensitivity mapping, dynamic operator weight recalibration	Blind random byte mutation leading to invalid payload generation
Deep-State Path Exploration	Gradient-Guided Smooth Neural Networks, Taint-Informed Gradient Descent	Differentiable program surrogates, loss function minimization ($\nabla f_x(I)$)	Hard conditional guards, complex arithmetic comparisons, magic values
Post-Fuzzing Triage	Supervised Classifiers (SVM, Random Forests), Graph Neural Networks (GNNs)	Crash stack trace clustering, control/data-flow vulnerability feature extraction	Labor-intensive manual crash triaging and false-positive filtering

The motivation to integrate machine learning in such an architecture stems from to the theoretical advantage of bridging the knowledge-gap that exists in greybox environments (Cianci et al., 2026). Blackbox fuzzing does not make use of any information about the internal state of the program being fuzzed, and whitebox fuzzing employs heavyweight symbolic execution (which suffers from very high computational cost and path explosion), ML-guided greybox fuzzing makes use of lightweight runtime telemetry to train dynamic surrogate models. This combination design enables the fuzzer to simulate whitebox style reasoning at nearly the speed of blackbox execution (Liang et al., 2018).

3. Seed Scheduling and Energy Allocation via Reinforcement Learning and Multi-Armed Bandits

In CFGF, the seed selection algorithm controls two core decisions: which seed to choose from the pool for mutation and how many mutated test cases vawe to produce based on this seed (Yue et al., 2020). Traditional fuzzers like American Fuzzy Lop (AFL) rely on deterministic heuristics or basic exponential decay functions relative to the size of the seed and the speed of execution. Yet these heuristics often devote too much computing time to the unpromising seeds, which explore dead-end code areas and hinder opening up deeper-state software faults (Zhao et al., 2024).

More recent ML-guided fuzzers formulate seed scheduling as a Multi-Armed Bandit (MAB) or Reinforcement Learning (RL) problem. In this setting, each seed in the active queue is conceptualized as an arm, and performing so that we generate a testcase that finds new paths or crashes produces a reward (Rahman et al., 2026). The key challenge lies in how to trade off exploration (choosing less-fuzzed seeds to find new code regions) and exploitation (distributing more energy to the seeds that previously discovered more paths), in a non-stationary, adversarial environment (Lavie et al., 2011).

3.1 EcoFuzz and the Variant Adversarial Multi-Armed Bandit (VAMAB)

Traditional multi-armed bandit (MAB) algorithms treat reward probabilities as constant. Fuzzing is different: as a seed explores more of its local code paths, it becomes less likely to uncover a new one. To account for this decline, EcoFuzz frames coverage-guided fuzzing (CGF) as a Variant of the Adversarial Multi-Armed Bandit (VAMAB) problem (Yue et al., 2020).

EcoFuzz tracks each seed through three stages. A newly discovered seed with no execution history starts in the Initial State (Hu, 2024). It then moves to the Exploration State, where it is fuzzed to gather initial reward data. After enough baseline data has been collected, the seed enters the Exploitation State. At this stage, EcoFuzz ranks seeds by their estimated reward probability, calculated using the State Probability Estimation Model (SPEM) (Khan et al., 2023).

Rather than using the fixed energy-doubling rule common in standard AFL variants, EcoFuzz adjusts each seed's reward probability over time.

For seed s at time step t, this changing probability is denoted by P_s(t). If a seed cannot find new paths within a calculated energy budget, its reward probability decreases and the fuzzer moves the seed out of the exploitation state (He et al., 2024). EcoFuzz saves energy on low-reward arms and, as a result, eliminates many redundant testcase executions while speeding up path discovery in complex software targets (Lin et al., 2026).

3.2 Contextual Bandits and LinFuzz

Unlike standard MAB, which evaluates each seed independently based on its historic success at execution, Contextual Multi-Armed Bandits take into account the environmental context at runtime when making a decision. LinFuzz uses the LinUCB (Linear Upper Confidence Bound) algorithm for program-sensitive seed scheduling (Su et al., 2026).

LinFuzz represents each fuzzing iteration t as a dynamic, d-dimensional feature context vector x_t∈R^d (Li & Fang, 2026). It is a multilevel execution feedback, such as how often the specific paths taken by the seed has been executed, the risk profile of basic blocks accessed (measured by the existence and concentration of high-risk operational primitives like memory allocations or pointer dereferences), and global coverage growth trends (Öğdü et al., 2025).

LinFuzz assumes that the expected reward $r_{t,a}$ for selecting seed a is a linear function of its context vector $x_{t,a}$ with an unknown parameter vector θ_a^*:

$$\mathbb{E}[r_{t,a} \mid x_{t,a}] = x_{t,a}^{\top}\theta_a^*$$

To estimate θ_a^*, LinFuzz applies ridge regression to historical execution matrices D_a and reward vectors c_a. At each iteration, the fuzzer selects the seed a_t that maximizes the upper confidence bound:

$$a_t = \arg\max_{a \in A_t}\left(x_{t,a}^{\top}\hat{\theta}_a + \alpha\sqrt{x_{t,a}^{\top}A_a^{-1}x_{t,a}}\right)$$

where $A_a = D_a^{\top}D_a + I_d$, $\hat{\theta}_a = A_a^{-1}b_a$, and α is a hyperparameter governing exploration variance.

By continuously updating the context matrix A_a and covariance vector b_a, LinFuzz dynamically steers computational energy toward seeds exercising high-risk or low-frequency basic blocks (Su et al., 2025). This program-sensitive contextual model allows LinFuzz to outperform non-contextual schedulers in discovering deeply nested vulnerabilities while maintaining lower time complexity than complex symbolic execution engines (Li & Fang, 2026).

4. Deep-State Path Navigation: Neural Program Smoothing and Gradient-Guided Fuzzing

Traversing exotic branch conditions, including multi-byte equality checks (if (x == 0xDEADBEEF)), hash checks, and the like, as well as structural parsing delimiters is however among the most stubborn bottlenecks in greybox fuzzing (Chen et al., 2021). Conventional CGF is based on blind, random mutations, which are unlikely to satisfy strict control-flow requirements. In contrast, dynamic symbolic execution (known as concolic fuzzing) relies on intensive program instrumentation and SMT solvers to solve constraints but is limited by path explosion, memory overheads, and significant solver latency (Luo et al., 2019).

Machine learning fills this gap Neural program smoothing and Gradient-guided fuzzing. By representing discrete program execution paths as continuous, differentiable neural network functions, these methods allow gradient descent algorithms to evaluate complex branch conditions in a matter of seconds (She et al., 2019).

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

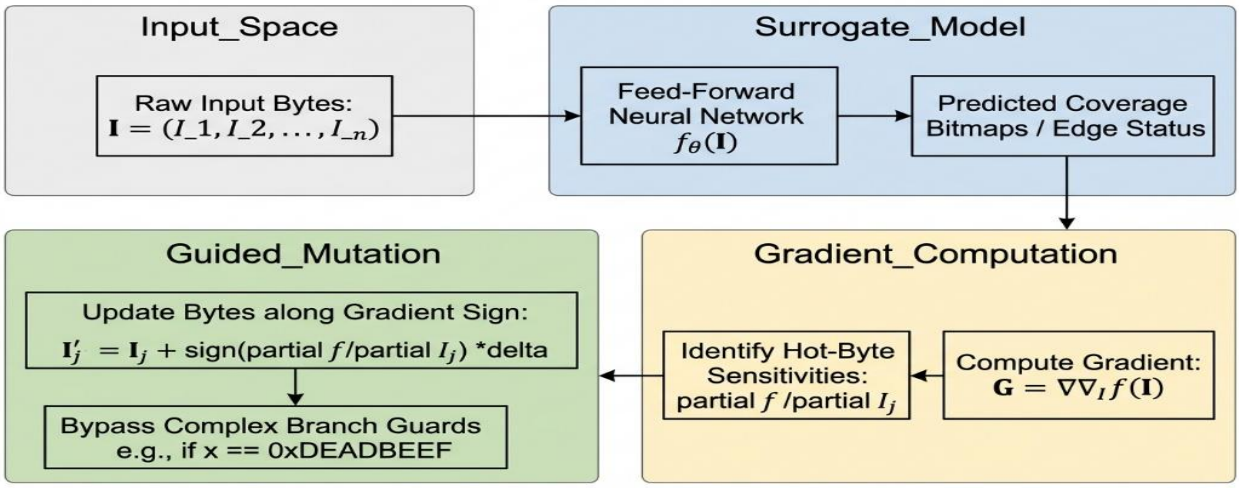


Figure 2: Workflow of Neural Program Smoothing, Gradient Computation, and Guided Mutation for Bypassing Complex Branch Guards during Deep-State Path Exploration.

4.1 Neural Program Learning and Gradient Descent: NEUZZ and MTFuzz

NEUZZ pioneered the application of neural program smoothing for deep-state vulnerability discovery. NEUZZ models the control-flow graph of a Program Under Test (PUT) as a continuous surrogate function (Wu et al., 2022). A Feed-Forward Neural Network (FFNN) is trained on historical input-coverage pairs, where the input represents raw byte arrays and the output vector represents edge coverage bitmaps (Laudani et al., 2015).

Because the neural network provides a smooth, differentiable approximation of the underlying program's discrete execution behavior, NEUZZ can compute the gradient of the objective function with respect to the input bytes (Macfarlane et al., 2026). The forward execution model $f_\theta(I)$ outputs target edge status, and the gradient is evaluated via backpropagation:

$$G = \nabla_I f_\theta(I) = \left(\frac{\partial f}{\partial I_1}, \frac{\partial f}{\partial I_2}, \ldots, \frac{\partial f}{\partial I_n}\right)$$

The magnitude of the directional derivative $\left|\frac{\partial f}{\partial I_j}\right|$ identifies hot bytes input byte positions that exert the highest statistical control over target branch outcomes. NEUZZ then performs gradient-guided byte mutations by updating input byte I_j along the sign of the computed gradient:

$$I_j' = I_j + \text{sign}\left(\frac{\partial f}{\partial I_j}\right) \cdot \delta$$

MTFuzz expands upon NEUZZ by replacing single-task neural architectures with a Multi-Task Deep Neural Network. MTFuzz simultaneously models multiple control-flow targets and branch dependencies as distinct tasks sharing underlying hidden representations (She et al., 2020). This multi-task learning formulation prevents the network from overfitting to specific individual paths, improving model generalization across unexplored code regions (Moon & Kim, 2022).

4.2 Taint-Informed, Data-Flow-Sensitive, and Dictionary Approaches

Purely neural program smoothing learns input-output mapping through observation, yet neural execution of program with dynamic branch predictions in tandem with its static counterpart maximizes branch solving throughput (Hu et al., 2024). Angora employs lightweight dynamic taint analysis to know the input bytes that influence the branch conditions. Rather than training complicated neural networks, Angora takes numerical gradient descent over the branch distance expressions, and solves byte constraints one at a time instead of using SMT solvers (Chen & Chen, 2018). In contrast, GreyOne adopts data-flow sensitive fuzzing by running lightweight taint inference. It evaluates byte control weights by

monitoring variable change of the run-time during byte mutations, and then high control bytes are dynamically prioritized for focused path exploration (Gan et al., 2020).

Along these lines, CDFUZZ (Customized Dictionary Fuzzing) shows that a judicious selection of tokens and dictionaries for certain seeds can come close to the performance of complex constraint-solving engines (Wu et al., 2024). Exploiting seed-specific dictionaries derived from program constants, CDFUZZ yields a drastic coverage improvement without incurring the computational cost associated with training the gradient model (Ota et al., 2026).

Table 2: Comparative Analysis of State-of-the-Art Deep-State Path Exploration Techniques

Fuzzer / Technique	State Exploration Mechanism	Constraint-Solving Paradigm	Model Retraining / Execution Overhead	Primary Limitation
NEUZZ	Neural Program Smoothing	Gradient Descent on Forward Neural Network ($\nabla_I f(I)$)	High (requires periodic retraining on new coverage data)	Gradients become stale as execution reaches new code branches
MTFuzz	Multi-Task Neural Learning	Multi-objective backpropagation across shared neural layers	Moderate to High (optimized multi-task forward passes)	High memory usage for large targets; susceptible to local minima
Angora	Dynamic Taint Analysis	Numerical Gradient Descent on Branch Distance Functions	Low (lightweight runtime taint tracking)	Scalability degradation on complex pointer aliasing and multi-dependent paths
GreyOne	Data-Flow-Sensitive Taint Inference	Variable dependency evaluation via direct mutation monitoring	Low to Moderate (no binary instrumentation for taint tracking)	Indirect branches and implicit data-flows reduce control sensitivity accuracy
CDFUZZ	Custom Seed-Specific Dictionaries	Dynamic string/constant insertion matched to local context	Very Low (dictionary extraction and pattern matching)	Cannot solve non-literal arithmetic constraints or complex transformations

5. Large Language Model (LLM) Integration in Generative and Adaptive Fuzzing

The transition from shallow statistical models to generative Large Language Models (LLMs) represents a structural evolution in automated vulnerability discovery. LLMs pre-trained on massive code repositories possess deep structural and semantic representations of programming languages, file formats, and communication protocols (Xia et al., 2024). Incorporating LLMs into greybox fuzzing addresses the input grammar generation bottleneck, enabling the automated creation of complex test cases that bypass initial target parsing logic (Chen et al., 2025).

5.1 Universal and Library Fuzzing: Fuzz4All, TitanFuzz, and FuzzGPT

Classic generation-based fuzzers rely on manually defined input grammars to generate syntactically valid inputs, a burdensome and fragile procedure. Contemporary LLM-guided generative fuzzers substitute static grammars with dynamic, prompt-based generation models (Li et al., 2025).

Fuzz4All builds a general-purpose fuzzing framework based on auto-prompting and seed mutation engines driven by LLMs including GPT-4 and StarCoder. Fuzz4All transforms system documentation, formal language definitions, or any user input into structured prompts (Xia et al., 2024). In every fuzzing round, Fuzz4All adjust its

prompt context by the coverage feedback in a dynamic way, semantically valid test case is generated for each new round of fuzzing in a variety of target languages (C, C++, Java, Python) directly without any identification to them (Sun et al., 2026).

TitanFuzz and FuzzGPT are designed for complex software libraries, such as deep learning frameworks. TitanFuzz utilizes generative and fill in the middle LLMs to generate valid sequences of APIs, therefore offering the maximum coverage of APIs edges. FuzzGPT improves on this methodology by concentrating on rare edge cases and previously bug-triggering events through few-shot learning methodologies (Deng et al., 2023).

5.2 Protocol and State-Machine Fuzzing: ChatAFL and Embedded Targets

Stateful network protocols and embedded systems are particularly challenging to fuzz as test inputs need to be syntactically correct and comply with the rigid state-machine manners (Meng et al., 2024). ChatAFL directly incorporates LLMs in the protocol fuzzing loops. By posing questions to LLMs with protocol rfc specifications and interaction histories, ChatAFL extracts protocol state machines, message types, and even field expectations automatically from the protocol rfc specifications (Xia et al., 2026). The LLM serves as a smart helper, producing sequences of domain-specific messages that push target network services into more profound protocol states that traditional greybox mutators are ineffective to attaining (Provvedi et al., 2026). Likewise, adapting LLMs for resource-limited CLI utilities and embedded software such as BusyBox leads to better edge coverage and more unique crashes than using a typical random seed corpus (Xu et al., 2026).

5.3 Neuro-Symbolic and Feedback-Guided Integration

Modern advancements combine LLM generative capabilities with low-level runtime execution feedback to form neuro-symbolic fuzzing loops (Zhang et al., 2025). Frameworks such as StateLens leverage execution-grounded instrumentation to monitor deep JavaScript engine semantics, feeding execution trace anomalies back into the LLM prompt context to iteratively mutate candidate payloads (Wong et al., 2026). Similarly, tools like SyzHarness analyze bug fix commits via LLMs to synthesize targeted execution harnesses, increasing bug reproduction and path exploration rates in kernel-level components (Li et al., 2026)

6. Benchmarking Methodologies and Ground-Truth Evaluation

Evaluating machine learning-guided greybox fuzzers requires rigorous, repeatable, and non-biased benchmarking frameworks. Historically, fuzzing research relied heavily on inconsistent metrics, such as raw line coverage or deduplicated crash counts (Lampert et al., 2016). These legacy metrics often produce misleading conclusions due to crash count inflation (where a single underlying bug triggers thousands of unique stack traces) and coverage inflation (where a fuzzer exercises non-security-critical code paths) (Scanlon et al., 2024).

6.1 Legacy Benchmarks: LAVA-M and Synthetic Bug Injection

Early fuzzing evaluations relied heavily on synthetic benchmark suites, most notably LAVA-M. LAVA-M injected thousands of synthetic bugs hardcoded 32-bit integer comparisons acting as canaries into four coreutils command-line targets (Bundt et al., 2021).

Despite its early adoption, research revealed severe structural flaws in LAVA-M. Injected synthetic canaries do not reflect the spatial, temporal, or control-flow complexity of real-world software vulnerabilities (Bundt, 2023). Furthermore, synthetic bugs are often clustered in shallow input-parsing paths, favoring tools with specific magic-value matching capabilities while failing to evaluate a fuzzer's ability to navigate deep state transitions (Wu, 2025).

6.2 Modern Evaluation Platforms: FuzzBench and Magma

To the best of our knowledge, a ground-truth evaluation platform To alleviate the drawbacks of synthetic benchmarks, the security community has

proposed to build common ground-truth evaluation platforms (Wang et al., 2026) within the.

FuzzBench from Google is an open-source benchmarking service that runs multiple fuzzers against a variety of real-world target applications. FuzzBench makes the runs comparable (usually 24h of execution budget with 20+ statistical repetitions) and computes most of the performance metrics based on code coverage profiles (Metzman et al., 2021).

The Magma Benchmark Framework cross-examines the ground-truth bug crisis by ones hot-backing documented, real world security vulnerabilities against modern software library target versions. Magma supports a wide range of C/C++ targets, and for each injected vulnerability it instruments a lightweight execution oracle (Hazimeh et al.

Magma introduces three execution stages-based bug-centric performance metrics. When the execution path covers the basic block that includes the buggy code chunk,a bug becomes Reached , while the inputs supplied to this path do not fulfilled to fault condition (Cheng et al., 2022).

A Bug is Triggered once input fulfills the precise predicate that is necessary to trigger the Fault (e.g. buffer overflow or integer exception), causing the program to enter an unknown state. And umin is Detected if the triggered fault leads to a specific crash or memory violation detected by the fuzzer's runtime monitoring oracle (Le et al., 2022).

6.3 The Leaky Oracle Problem and Firmware Benchmarking

The Leaky Oracle Problem is a common bug in modern benchmark design. In naive ground-truth implementations, bug instrumentation oracles are placed directly inside the target binary and change execution state, branch conditions, or memory layouts (Schneider et al., 2026). As a result, fuzzers may inadvertently take advantage of oracle feedback to optimize path exploration, leading to the benchmark overemphasizing certain evaluation results (Sherman & Nagy, 2025).

Frameworks like FIRMREBUGGER address the leaky oracle problem by decoupling the fuzzer's greybox execution loop from the benchmarking triaging engine (Duong et al., 2026). FIRMREBUGGER employs emulator-level introspection state capture during post-execution seed replay to verify bug state transitions using external formal descriptions, without altering the target runtime context while fuzzing (Kim et al., 2026).

Table 3: Evaluation Frameworks and Benchmark Taxonomy in Fuzzing Research

Benchmark	Vulnerability Source	Target Domain	Evaluation Metrics	Isolation Mechanism / Leaky Oracle Defense
LAVA-M	Synthetic injected canaries	Command-line utilities (coreutils)	Synthetic crash count	None (hardcoded canaries pollute target control flow)
Google FuzzBench	Real-world open-source software	File parsers, cryptographic libraries	Line and branch coverage profiles over time	High (standardized containerized environments, coverage calculated offline)
Magma (v1.0 - v1.3)	Forward-ported real CVEs	High-impact C/C++ libraries	Ground-truth Reached, Triggered, Detected metrics	Moderate (in-binary canary checks; relies on fatal/non-fatal oracle modes)
FIRMREBUGGER	Real-world firmware CVEs	Monolithic embedded firmware	Ground-truth state-matching expressions	Complete isolation (out-of-band emulator state introspection during seed replay)

7. Open Challenges and Future Research Trajectories

Despite substantial advances in machine learning-guided greybox fuzzing, several fundamental open challenges continue to hinder adaptive vulnerability discovery in enterprise-scale software systems (Paasi et al., 2026).

7.1 Model Obsolescence and Retraining Overheads

Neural program smoothing models like NEUZZ and MTFuzz suffer from stale gradient challenges. As a fuzzer successfully solves branch conditions and explores new subtrees within a target program's control-flow graph, the continuous surrogate model trained on historical paths becomes obsolete (Deo at al., 2016). Retraining deep neural networks online during an active fuzzing campaign introduces immense computational overhead, often negating the runtime performance advantage of greybox execution. Developing lightweight, incrementally updated neural architectures that perform online zero-shot adaptation remains a critical research priority (Zhang et al., 2026).

7.2 Scalability in Multithreaded, Monolithic, and Heterogeneous Systems

Current ML-guided fuzzers perform exceptionally well on single-threaded, file-parsing targets. However, scaling learning-guided fuzzing to monolithic firmware binaries, distributed systems, and highly concurrent multithreaded applications introduces state-space explosion (Ausavarungnirun et al., 2012). In multithreaded contexts, path execution becomes non-deterministic due to thread scheduling, introducing noise into feature context vectors and invalidating gradient calculations. Future research must formalize context vector abstractions capable of modeling concurrent thread states and transient asynchronous delays (Zhao et al., 2026).

7.3 Overfitting and Hallucination in LLM-Guided Fuzzers

Although LLMs are good at producing initial syntactically valid seed files, they suffer from prompt hallucination, where they generate the same or semantically similar code snippet multiple times (Luu et al., 2026). In addition, generative LLM-based engines may also accidentally overfit to publicly available benchmark targets that appear in their pre-training dataset, leading to a false sense of better vulnerability discovery performance that does not generalize to unseen, private targets (Mclntosh et al., 2025). Developing strict prompt isolation methods and context-aware feedback mechanisms are necessary for true LLM-based generalization in LLM-guided security testing (Qian, 2025).

Conclusion

This review has looked at how machine learning's being used in coverage-guided greybox fuzzing as a proper way to deal with the basic problems that traditional methods have. Traditional coverage-guided greybox fuzzing, even though it has worked well in practice doesn't have the tools to get to deep parts of a program that are protected by complicated rules, checks and state machine limits, where random changes almost never work. By thinking of fuzzing as a dimensional and not simple math problem machine learning guided methods make models that represent how a program runs and add learning parts in important places inside the fuzzing tool. Our five-step classification shows that machine learning is used through the fuzzing process. Large Language Models and models that create content help with the problem of input grammar when creating initial test cases; models that make choices based on context and learning methods help decide which test cases to use and how much effort to put in when the rewards change; models that take information and make decisions find the important parts of the test cases and change how they are modified; methods that use gradients to smooth the way a program runs change the parts where the program makes decisions into math problems that can be solved more easily which helps the test cases reach deeper parts of the program; and models that classify and use information from networks help with checking crashes and seeing if they can be used. All these

methods make the test process almost as good as having knowledge of the program but without the slow speed of full knowledge, which helps close the gap that greybox methods have. The comparisons in this review show that there is a set of options rather than one best way. A version of EcoFuzz uses a way to handle choices that removes old methods of using more effort by using a better way to change rewards while LinFuzz adds a way to choose which test cases to use that takes into account the details of the program to focus effort on parts that are more risky and used less. When it comes to exploring parts of the program NEUZZ and MTFuzz show that using models to make the program run more smoothly is helpful but they have problems with old math and needing to retrain often; Angora and GreyOne are good at solving problems quickly using simple checks but can't handle complicated parts of the program; and CDFUZZ shows that making a list of important parts can be as good as solving problems with little cost but it can't handle math that isn't exact. Fuzzers that use language models like Fuzz4All, TitanFuzz, ChatAFL and StateLens can create test cases for different types of language and systems but they can still have problems with making up information and only working well on certain test cases. Our look at how these methodsre tested shows that testing properly is needed to say something is better. Old tests like LAVA-M add parts that don't match real problems and focus on simple parts of the program which makes the tests favor finding simple mistakes. Newer places like FuzzBench use tests in containers Magma uses real problems with certain measures and FIRMREBUGGER uses special tools to check what is happening which makes the tests more accurate. The problem of having a way to know if something is wrong is still a big issue because checking the program can change how it runs and let the test cases use this to get better results so keeping the test process separate from checking the results like in FIRMREBUGGER is the best way. Looking ahead three things need work. First models getting old: as the test cases solve parts of the program and explore parts the models made from old paths become out of date and updating them in real time adds cost that can cancel out the benefits of greybox testing; models that can change quickly and without needing to retrain a lot are needed. Second being able to handle more: machine learning guided fuzzers work well on simple tasks but have problems with big programs, systems that work together and programs with many parts running at once where the order of tasks can change the information and make the math wrong; finding a way to model all the parts and delays in a way that works is still not done. Third being able to use language models in fuzzing: the models can make up information repeat the same ideas and focus too much on what they have seen before which makes it look like they find more problems than they really do; making sure the tests don't use too much of what they already know and use good ways to check their results is important. Together using machine learning in greybox fuzzing is a big step forward in finding problems automatically changing random changes into a better way that uses feedback. The way the field is moving from choices and ways to make programs run more smoothly toward using large language models that mix different types of methods suggests that the best future tools will use different ways together: models to help find new paths checking and flow analysis to understand rules models that create new test cases and tests that are done properly to make sure the results are true. Making this idea happen will need to solve problems with how fast models can be updated, how to handle when many parts work at once and how to make sure they work well in situations, which will help find deep problems, in big software.

REFERENCE

Ahmed, R. H., Zeeshan, M., Ali, U., Khan, M. S., Rasheed, M. D., & Hussain, M. (2025). AI-FUZZ: Reinforcement Learning–Guided Stateful Fuzzing for Secure Smart Contracts. INTERNATIONAL JOURNAL OF ADVANCES IN SIGNAL AND IMAGE SCIENCES, 1040-1055.

Ausavarungnirun, R., Chang, K. K. W., Subramanian, L., Loh, G. H., & Mutlu, O. (2012). Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. ACM SIGARCH Computer Architecture News, 40(3), 416-427.

Bundt, J. (2023). Towards Rigorous Evaluation of Binary Testing and Analysis (Doctoral dissertation, Northeastern University).

Bundt, J., Fasano, A., Dolan-Gavitt, B., Robertson, W., & Leek, T. (2021, May). Evaluating synthetic bugs. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (pp. 716-730).

Chen, C., Dolan-Gavitt, B., & Lin, Z. (2025). {ELFuzz}: Efficient Input Generation via {LLM-driven} Synthesis Over Fuzzer Space. In 34th USENIX Security Symposium (USENIX Security 25) (pp. 6279-6298).

Chen, C., Xu, H., & Cui, B. (2021). PSOFuzzer: a target-oriented software vulnerability detection technology based on particle swarm optimization. Applied Sciences, 11(3), 1095.

Chen, P., & Chen, H. (2018, May). Angora: Efficient fuzzing by principled search. In 2018 IEEE Symposium on Security and Privacy (SP) (pp. 711-725). IEEE.

Cheng, X., Nie, X., Li, N., Wang, H., Zheng, Z., & Sui, Y. (2022). How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors. IEEE Transactions on Dependable and Secure Computing, 21(2), 542-558.

Chennakesavalu, S., Hu, F., Ibarraran, S., & Rotskoff, G. (2026). Aligning Transformers with Continuous Feedback via Energy Rank Alignment. Advances in Neural Information Processing Systems, 38, 117475-117509.

Cianci, E., Civera, M., De Biagi, V., & Chiaia, B. (2026). Physics-informed machine learning for the structural health monitoring and early warning of a long highway viaduct with displacement transducers. Mechanical Systems and Signal Processing, 242, 113659.

Deng, Y., Xia, C. S., Yang, C., Zhang, S. D., Yang, S., & Zhang, L. (2023). Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. arXiv preprint arXiv:2304.02014.

Deo, A., Dash, S. K., Suarez-Tangil, G., Vovk, V., & Cavallaro, L. (2016, October). Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In Proceedings of the 2016 ACM workshop on artificial intelligence and security (pp. 71-82).

Duong, M., Chesser, M., Farrelly, G., Nepal, S., & Ranasinghe, D. C. (2026, January). FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers. In Procs. of the USENIX Security Symposium.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., & Chen, Z. (2020). {GREYONE}: Data flow sensitive fuzzing. In 29th USENIX security symposium (USENIX Security 20) (pp. 2577-2594).

Güler, E. (2025). Scaling and mitigating automated bug analysis: parallelization, domain expansion, and countermeasures (Doctoral dissertation, Dissertation, Bochum, Ruhr-Universität Bochum, 2024).

Hazimeh, A., Herrera, A., & Payer, M. (2020). Magma: A ground-truth fuzzing benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 4(3), 1-29.

He, J., Balunović, M., Ambroladze, N., Tsankov, P., & Vechev, M. (2019, November). Learning to fuzz from symbolic execution with application to smart contracts. In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security (pp. 531-548).

He, Q., Zhang, L., Fang, H., Wang, X., Ma, L., Yu, K., & Zhang, J. (2024). Multistage competitive opinion maximization with Q-learning-based method in social networks. IEEE transactions on neural networks and learning systems, 36(4), 7158-7168.

Hu, J. (2024). Enhancing Fuzzing Using Stochastic Modeling and Generative AI (Doctoral dissertation, University of California, Riverside).

Hu, J., Duan, Y., & Yin, H. (2024, February). Marco: A stochastic asynchronous concolic explorer. In Proceedings of the 46th IEEE/ACM International conference on software engineering (pp. 1-12).

Khan, R. I. S., Yazdani, A. H., Fu, Y., Paul, A. K., Ji, B., Jian, X., ... & Butt, A. R. (2023). {SHADE}: Enable fundamental cacheability for distributed deep learning training. In 21st USENIX Conference on File and Storage Technologies (FAST 23) (pp. 135-152).

Kim, J., Kim, D., Han, H., Kim, Y., Park, S., Ji, S., ... & Kim, T. (2026, October). CtxFuzz: Context-Aware Directed Fuzzing via Runtime Fuzzer-Agent Cooperation. In Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (pp. 437-448).

Lampert, T. A., Stumpf, A., & Gançarski, P. (2016). An empirical study into annotator agreement, ground truth estimation, and algorithm evaluation. IEEE Transactions on Image Processing, 25(6), 2557-2572.

Laudani, A., Lozito, G. M., Riganti Fulginei, F., & Salvini, A. (2015). On training efficiency and computational costs of a feed forward neural network: A review. Computational intelligence and neuroscience, 2015(1), 818243.

Lavie, D., Kang, J., & Rosenkopf, L. (2011). Balance within and across domains: The performance implications of exploration and exploitation in alliances. Organization science, 22(6), 1517-1538.

Le, Q. L., Raad, A., Villard, J., Berdine, J., Dreyer, D., & O'Hearn, P. W. (2022). Finding real bugs in big programs with incorrectness logic. Proceedings of the ACM on Programming Languages, 6(OOPSLA1), 1-27.

Li, J., & Fang, J. (2026). Online recommendation in non-stationary environments based on knowledge graph enhancement and time-varying reward mechanism. Applied Intelligence, 56(3), 73.

Li, J., & Fang, J. (2026). Online recommendation in non-stationary environments based on knowledge graph enhancement and time-varying reward mechanism. Applied Intelligence, 56(3), 73.

Li, J., Zhao, B., & Zhang, C. (2018). Fuzzing: a survey. Cybersecurity, 1(1), 6.

Li, X., Pu, J., Li, H., Srivastav, A., Shehada, K., Krishnamurthy, S. V., & Qian, Z. (2026). SyzHarness: Patch-Based Kernel Bug Reproduction with LLM-Synthesized Fuzzing Harnesses. arXiv preprint arXiv:2609.23889.

Li, Z., Wu, J., Ling, X., Luo, T., Rui, Z., & Wu, Y. (2025, April). The seeds of the future sprout from history: Fuzzing for unveiling vulnerabilities in prospective deep-learning libraries. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1616-1627). IEEE.

Liang, H., Pei, X., Jia, X., Shen, W., & Zhang, J. (2018). Fuzzing: State of the art. IEEE Transactions on Reliability, 67(3), 1199-1218.

Lin, P., Wang, P., Zhou, X., Xie, W., Zhang, G., & Lu, K. (2026). Not All Paths Are Equal: Multi-path Optimization for Directed Hybrid Fuzzing. ACM Transactions on Software Engineering and Methodology, 35(3), 1-29.

Luo, Z., Zuo, F., Jiang, Y., Gao, J., Jiao, X., & Sun, J. (2019). Polar: Function code aware fuzz testing of ics protocol. ACM Transactions on Embedded Computing Systems (TECS), 18(5s), 1-22.

Luu, V. H., Pasdar, A., Charoenwet, W., Murray, T., Cohney, S., & Pham, V. T. (2026). Following Dragons: Code Review-Guided Fuzzing. arXiv preprint arXiv:2602.10487.

Macfarlane, M., Bonnet, C., van Hoof, H., & Lelis, L. (2026, April). Gradient-based program synthesis with neurally interpreted languages. In International Conference on Learning Representations (Vol. 2026, pp. 124292-124317).

McIntosh, T. R., Susnjak, T., Arachchilage, N., Liu, T., Xu, D., Watters, P., & Halgamuge, M. N. (2025). Inadequacies of large language model benchmarks in the era of generative artificial intelligence. IEEE Transactions on Artificial Intelligence, 7(1), 22-39.

Meng, R., Mirchev, M., Böhme, M., & Roychoudhury, A. (2024, February). Large Language Model guided Protocol Fuzzing. In NDSS.

Metzman, J., Szekeres, L., Simon, L., Sprabery, R., & Arya, A. (2021, August). Fuzzbench: an open fuzzer benchmarking platform and service. In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 1393-1403).

Moon, C., & Kim, D. (2022). Prediction of drug–target interactions through multi-task learning. Scientific Reports, 12(1), 18323.

Narang, K., Jadhav, A. N., Dhakane, V. N., Tripathi, M. K., Momin, S. M., & Thorat, N. A. (2025, November). ML-based Fuzzing for Vulnerability Detection: Methods, Benchmarks, and Open Challenges-A Review. In 2025 9th International Conference on Electronics, Communication and Aerospace Technology (ICECA) (pp. 1239-1244). IEEE.

Narang, K., Jadhav, A. N., Dhakane, V. N., Tripathi, M. K., Momin, S. M., & Thorat, N. A. (2025, November). ML-based Fuzzing for Vulnerability Detection: Methods, Benchmarks, and Open Challenges-A Review. In 2025 9th International Conference on Electronics, Communication and Aerospace Technology (ICECA) (pp. 1239-1244). IEEE.

Öğdü, Ç. U., Arslanoğlu, K., & Karaköse, M. (2025). An adaptive multi-agent LLM-based clinical decision support system integrating biomedical RAG and web intelligence. IEEE Access.

Ota, S., Yu, Q., Fujiwara, K., Ikehata, S., & Sato, I. (2026, September). Retrieving and refining winning noise tickets for diffusion-based motion generation. In European Conference on Computer Vision (pp. 130-147). Cham: Springer Nature Switzerland.

Paasi, J., Valkokari, K., & Suominen, A. (2026). Collaborative foresight: major research trajectories and future research agenda. Futures, 103784.

Provvedi, C., Seidenari, L., Picano, B., & Fantacci, R. (2026). Intent-LLM: A Framework for Automated Network Configuration Through Code Generation. IEEE Transactions on Cognitive Communications and Networking.

Qian, Y. (2025). Prompt engineering in education: A systematic review of approaches and educational applications. Journal of Educational Computing Research, 63(7-8), 1782-1818.

Rahman, T., Saha, S., Alhurubi, A. Y., Saha, S. K., Farahmandi, F., & Tehranipoor, M. (2026). Emulation-based System-on-Chip Security Verification: Challenges and Opportunities. arXiv preprint arXiv:2604.15073.

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). Vuzzer: Application-aware evolutionary fuzzing. In 2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings] (pp. 1-14). Internet Society.

Rieu, R. (2020). Development and verification of arbitrary-precision integer arithmetic libraries (Doctoral dissertation, Université Paris-Saclay).

Scanlon, J. M., Kusano, K. D., Fraade-Blanar, L. A., McMurry, T. L., Chen, Y. H., & Victor, T. (2024). Benchmarks for retrospective automated driving system crash rate analysis using police-reported crash data. Traffic Injury Prevention, 25(sup1), S51-S65.

Schneider, M. Y., Kholmanskikh, D. D., Romanov, K. Y., Perekina, E. A., Nikolenko, S. A., Lukin, R. Y., & Golov, I. V. (2026). De Novo structure prediction from tandem mass spectra: algorithms, benchmarks, and limitations. Molecules, 31(5), 769.

Seljan, G. (2025). Machine Learning in Security Vulnerability Research (Doctoral dissertation, Budapesti Corvinus Egyetem).

She, D., Krishna, R., Yan, L., Jana, S., & Ray, B. (2020, November). MTFuzz: fuzzing with a multi-task neural network. In Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 737-749).

She, D., Pei, K., Epstein, D., Yang, J., Ray, B., & Jana, S. (2019, May). Neuzz: Efficient fuzzing with neural program smoothing. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 803-817). IEEE.

Sherman, G., & Nagy, S. (2025, April). No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 165-177). IEEE.

Sivamayil, K., Rajasekar, E., Aljafari, B., Nikolovski, S., Vairavasundaram, S., & Vairavasundaram, I. (2023). A systematic study on reinforcement learning based applications. Energies, 16(3), 1512.

Su, Y., Xiong, D., Wan, Y., Shi, C., & Zeng, Q. (2024). LinFuzz: program-sensitive seed scheduling greybox fuzzing based on LinUCB algorithm. IEEE Access, 12, 74843-74860.

Su, Y., Xiong, D., Wan, Y., Shi, C., & Zeng, Q. (2024). LinFuzz: program-sensitive seed scheduling greybox fuzzing based on LinUCB algorithm. IEEE Access, 12, 74843-74860.

Sun, Y., Andersen, V. H., Choudhary, G., & Dragoni, N. (2026). On the Challenges and Opportunities of Fuzzing via Large Language Models: A Review. Applied Sciences (switzerland), 16(10), 5160.

Wang, P., Zhou, X., Yue, T., Lin, P., Liu, Y., & Lu, K. (2024). The progress, challenges, and perspectives of directed greybox fuzzing. Software Testing, Verification and Reliability, 34(2), e1869.

Wang, Y., Zhang, Z., Wang, C., Xu, X., Liu, M., Wang, Y., ... & Zheng, Z. (2026, July). RealSec-bench: A benchmark for evaluating secure code generation in real-world repositories. In Findings of the Association for Computational Linguistics: ACL 2026 (pp. 35866-35883).

Wong, W. K., Xiao, D., Lai, A. C. T., Ke, P. F., & Wang, S. (2026, September). State-Aware Fuzzing of JavaScript Engines with LLM-Guided Instrumentation. In Proceedings of the ACM SIGOPS 32nd Symposium on Operating Systems Principles (pp. 988-1003).

Wu, M., Jiang, L., Xiang, J., Zhang, Y., Yang, G., Ma, H., ... & Zhang, L. (2022, May). Evaluating and improving neural program-smoothing-based fuzzing. In Proceedings of the 44th International Conference on Software Engineering (pp. 847-858).

Wu, M., Xiang, J., Chen, K., Di, P., Tan, S. H., Cui, H., & Zhang, Y. (2024). Tumbling down the rabbit hole: How do assisting exploration strategies facilitate grey-box fuzzing?. arXiv preprint arXiv:2409.14541.

Wu, Y. (2025). Hardening the Fuzzing Ecosystem Through Automated Seed Generation, Report Deduplication, and Patch Correctness Validation (Doctoral dissertation, Northwestern University).

Xia, C. S., Paltenghi, M., Le Tian, J., Pradel, M., & Zhang, L. (2024, April). Fuzz4all: Universal fuzzing with large language models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

Xia, C. S., Paltenghi, M., Le Tian, J., Pradel, M., & Zhang, L. (2024, April). Fuzz4all: Universal fuzzing with large language models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

Xiao, Z., Qian, K., Xiong, D., Wang, Y., & Yan, H. (2026, June). DeepChatAFL: Research on Protocol Fuzz Testing Method Assisted by Large Models. In 2026 2nd International Conference on Communication, Remote Sensing and Information Technology (CRSIT) (pp. 83-90). IEEE.

Xu, K., Zhang, D., & Wang, X. (2026). LLM-Conductor: A Closed-Loop Resource-Adaptive Architecture for Secure LLM Deployment in Industrial Sensor Networks and IIoT Systems. Sensors, 26(9), 2733.

Xue, J., He, Y., Xiao, Y., Chen, Y., Wang, Z., & Lu, H. (2025, August). From Heuristics to Intelligence: Evolving Black-Box Scanners for Modern Web Applications. In 2025 IEEE 10th International Conference on Data Science in Cyberspace (DSC) (pp. 247-254). IEEE.

Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K., & Zhou, X. (2020). {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 2307-2324).

Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K., & Zhou, X. (2020). {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 2307-2324).

Zhang, D., Li, Z. Z., Zhang, M. L., Zhang, J., Liu, Z., Yao, Y., ... & Liu, C. L. (2025). From system 1 to system 2: A survey of reasoning large language models. IEEE Transactions on Pattern Analysis and Machine Intelligence.

Zhang, H., & Cui, B. (2025, November). Balancing Exploration and Exploitation: Fuzzing Based on Adaptive Scheduling. In 2025 7th International Conference on Electronic Engineering and Informatics (EEI) (pp. 481-485). IEEE.

Zhang, Y., Kim, Y., Choi, Y. G., Kim, H., Liu, H., & Hong, S. (2026). Backpropagation-free test-time adaptation via probabilistic gaussian alignment. Advances in Neural Information Processing Systems, 38, 76280-76310.

Zhao, H., Li, Z., Wu, Y., & Zou, D. (2026). Distributed Race Condition Detection in CPSS via Proactive DAG-Guided Hidden Markov Models. IEEE Transactions on Computational Social Systems.

Zhao, X., Qu, H., Xu, J., Li, X., Lv, W., & Wang, G. G. (2024). A systematic review of fuzzing. Soft Computing, 28(6), 5493-5522.

Zhu, S., Wang, J., Sun, J., Yang, J., Lin, X., Wang, T., ... & Cheng, P. (2023). Better pay attention whilst fuzzing. IEEE Transactions on Software Engineering, 50(2), 190-208.