

Article

LLM-JDFuzz: A Large Language Model-Based Automated Java Deserialization Payload Generation Framework

Dong Su and Tengfei Tu * 

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China; sudong@bupt.edu.cn

* Correspondence: tutengfei.kevin@bupt.edu.cn

Abstract

Java deserialization vulnerabilities remain a critical security threat, with existing detection tools facing significant challenges in generating functional exploit payloads due to semantic blindness and constraint complexity. This paper presents LLM-JDFuzz, the framework that leverages Large Language Models for automated Java deserialization payload generation. By reframing payload generation as a constraint-aware code synthesis problem, LLM-JDFuzz introduces three key innovations: (1) an Autoprompting engine that transforms Gadget Chain specifications into structured natural language instructions, (2) an adaptive multi-strategy fuzzing loop that dynamically selects between generation, mutation, and semantic exploration based on execution feedback, and (3) a fine-grained feedback mechanism that performs error classification and root cause attribution to guide iterative refinement. We evaluate LLM-JDFuzz on 34 Gadget Chains from the ysoserial benchmark. Results show that LLM-JDFuzz achieves a 70.6% success rate. While maintaining comparable effectiveness to the state-of-the-art static tool JDD, LLM-JDFuzz demonstrates superior performance in processing cross-language scripting engine chains and achieves a 4× improvement in generation efficiency. Our work demonstrates that LLMs possess inherent semantic understanding capabilities for security-critical code generation, opening new research directions for AI-assisted vulnerability exploitation.

Keywords: Java deserialization; large language models; fuzzing

1. Introduction

The Java deserialization mechanism enables the transformation of objects into byte streams for storage or transmission, followed by the reconstruction of these objects through the inverse process. However, this mechanism has long been recognized as one of the most critical security vulnerabilities within the Java ecosystem. When an application deserializes untrusted data, an attacker can craft malicious serialized objects—commonly referred to as payloads—to trigger a sequence of method calls known as “gadget chains.” These chains ultimately lead to catastrophic consequences, including Remote Code Execution, sensitive data disclosure, or Denial of Service (DoS) [1–3].

Since the release of the ysoserial tool [4] by Frohoff et al. in 2015, which disclosed severe vulnerabilities in Apache Commons Collections, the impact of Java deserialization exploits has permeated critical middleware such as WebLogic, JBoss, and Jenkins, as well as essential infrastructure in the financial and governmental sectors [5,6]. Despite the introduction of defense mechanisms such as JEP 290 [7] by Oracle, the inherent complexity of the modern software supply chain ensures that new gadget chains continue to emerge. According to the



Academic Editor: Arkaitz Zubiaga

Received: 28 January 2026

Revised: 21 February 2026

Accepted: 24 February 2026

Published: 26 February 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.

OWASP Top 10, deserialization vulnerabilities have consistently ranked among the most perilous web application security risks for several years [3,8–10]. Consequently, the efficient and automated detection and generation of viable deserialization payloads has become a pivotal research challenge that both academia and industry must urgently address.

Existing approaches for deserialization vulnerability detection and exploit generation are primarily categorized into two technical paradigms: static analysis and dynamic testing (fuzzing). Despite significant advancements, these methods still encounter substantial challenges when applied to complex modern Java libraries [11].

From the perspective of static analysis, representative tools such as GadgetInspector [12], Serianalyzer [5], and the more recent JDD [13] mainly leverage call graph analysis and taint tracking to identify potential gadget chains. However, these methods face the critical issue of “path explosion,” where the combinations of method calls grow exponentially within large dependency libraries [14]. Moreover, the dynamic features of the Java language—such as reflection and dynamic proxies—are inherently difficult to model accurately through static means [15,16]. Consequently, these tools often suffer from high false-positive rates, producing chains that contain unreachable paths and are difficult to transform into executable payloads.

From the dynamic testing perspective, dynamic fuzzing tools attempt to trigger vulnerabilities by mutating inputs. Traditional coverage-guided fuzzers, such as AFL [17], treat inputs as flat byte streams, completely lacking an understanding of Java object structures [18,19]. While directed grey-box fuzzing [19,20] can focus on specific targets, its efficiency remains low in scenarios requiring precise type matching. Although ODD-Fuzz [21] introduced structure-aware directed grey-box testing, its mutation strategies during the payload generation phase are still based on heuristic rules, lacking deep semantic understanding.

This underscores a fundamental bottleneck in current research: existing methods treat payload generation as a mere search or mutation problem, neglecting its intrinsic code properties. Constructing a successful deserialization payload typically involves complex nested object initialization, the satisfaction of strict type constraints, and the configuration of specific reflection operations. This is, in fact, a complex code generation problem rather than a simple bit-stream mutation task.

In recent years, Large Language Models (LLMs) based on the Transformer architecture have achieved significant progress in the fields of code understanding and generation. Representative models, including Codex [1], Code Llama [22], and DeepSeek-Coder [23], have demonstrated exceptional program synthesis capabilities. These advancements have further catalyzed innovations in software testing methodologies: in compiler testing, Fuzz4All [24] and WhiteFox [25] have exhibited the potential of LLMs to uncover deep-seated logical defects; TitanFuzz [26] successfully applied these models to the testing of deep learning libraries; and in protocol fuzzing, ChatAFL [27] leveraged LLMs for more efficient interaction sequence generation.

These studies demonstrate that LLMs can comprehend complex syntactic structures and generate test cases that adhere to specific constraints. However, no research has yet explored the potential of LLMs specifically in the domain of Java deserialization payload generation. Given that a gadget chain is essentially a specific code execution path, modeling and transforming abstract chains identified via static analysis into natural language descriptions—and subsequently leveraging the robust code generation and semantic understanding of LLMs—may effectively bridge the semantic gap between “potential vulnerability paths” and “exploitable payloads.”

To validate this hypothesis, we conducted a preliminary pilot study. By evaluating the zero-shot generation capabilities of GPT-4 (gpt-4-turbo-2024-04-09) on three representa-

tive gadget chains (URLDNS, CommonsCollections6, and Hibernate1), we observed that: (1) LLMs are capable of comprehending call stack semantics and generating structurally correct Java code; (2) however, the initial success rate was only 12%, primarily limited by stringent type constraints and intricate serialization details, and (3) by incorporating type hinting and feedback-driven refinement, the success rate can be improved to 67%. These preliminary findings demonstrate the potential of LLM-based approaches while simultaneously underscoring the necessity for a systematic design.

Based on the aforementioned feasibility study, this paper introduces LLM-JDFuzz, an LLM-based fuzzing framework for Java deserialization vulnerabilities. The core philosophy of LLM-JDFuzz is to reformulate the payload generation task as a “constraint-based code generation” problem.

The primary contributions of this paper are summarized as follows:

1. LLM-JDFuzz validates the feasibility of utilizing Large Language Models for Java deserialization payload generation and proposes to model exploit generation as a semantic-aware code generation problem.
2. LLM-JDFuzz designs an Autoprompting mechanism and an iterative object graph evolution cycle for gadget chains, enabling the fully automated conversion from abstract chain specifications to executable payloads.
3. We implement a prototype system of LLM-JDFuzz and evaluate it on a benchmark dataset comprising 34 classic gadget chains from ysoserial [4]. Compared to traditional test generation methods [28–30], LLM-JDFuzz specifically focuses on the generation of security exploit code.

Experimental results demonstrate that LLM-JDFuzz significantly outperforms traditional static analysis tools [12,14] and dynamic fuzzing tools [21]. Compared to the heavyweight symbolic execution approach [13], LLM-JDFuzz achieves competitive success rates with significantly lower time overhead and better adaptability in complex dynamic scenarios. Specifically, it demonstrates the unique semantic understanding advantages of LLMs in handling high-complexity chains and scenarios involving dynamic proxies.

2. Background & Related Work

The Java serialization mechanism permits the transformation of objects and their internal states into byte streams, facilitating network transmission or persistent storage. Any class implementing the `java.io.Serializable` or `java.io.Externalizable` interfaces is eligible for serialization [2]. During the process of deserialization, the `readObject` method is responsible for reconstructing the byte stream into an object instance. To accommodate custom deserialization logic, Java enables developers to define special methods, such as `readObject`, `readResolve`, or `validateObject`, within the class definition [3]. Nevertheless, this mechanism was fundamentally designed with a focus on functional flexibility rather than security. Consequently, an attacker can craft malicious serialized data to trigger these automatically invoked methods during the deserialization phase, thereby exerting control over the execution flow of the application [1].

The crux of deserialization vulnerabilities resides in the gadget chain. A gadget chain refers to a sequence of Java method calls that are invoked successively during the deserialization process [6,14,31]. A complete chain manifests as a three-stage structured execution flow.

In the entry stage (Source), special methods that are automatically triggered during deserialization act as the point of ingress for the exploit [14]. This automatic invocation mechanism provides the attacker with initial control over the execution flow. Subsequently, the control flow is propagated through a series of nested method calls, referred to as the Chain. These intermediate gadgets leverage Java’s polymorphic features or dynamic

proxy mechanisms to incrementally redirect the execution flow from the Source toward the designated target. Finally, at the dangerous operation stage (Sink), the chain triggers critical malicious actions, such as executing arbitrary system commands via `Runtime.exec()` or facilitating remote class loading through JNDI lookup [4], ultimately resulting in Remote Code Execution (RCE) or other catastrophic outcomes.

By meticulously crafting a specific object graph, an attacker ensures that the application coherently traverses the aforementioned call chain upon deserialization, thereby achieving Remote Code Execution (RCE) or other severe security breaches [32].

Existing research on deserialization vulnerabilities primarily focuses on automated detection and exploit generation, which can be broadly categorized into static analysis, dynamic testing, and hybrid analysis.

Static analysis aims to identify potential gadget chains without executing the program by constructing call graphs and performing data-flow analysis. GadgetInspector [12] is a pioneering work in this field, utilizing taint propagation analysis [33] to search for paths from sources to sinks. However, it suffers from a high false-positive rate due to the lack of path reachability verification. Serianalyzer [5] employs bytecode analysis techniques; while efficient, it is limited to a predefined set of known patterns. SerialDetector [34] proposes an object-injection-based taint analysis that supports cross-language features, but its static modeling still struggles to accurately handle dynamic Java features such as reflection and dynamic proxies. The more recent JDD [13] significantly improves constraint-solving precision by combining a bottom-up search strategy with Input–Output Constraint Dependency (IOCD) analysis and symbolic execution [35]. Nevertheless, its heavy reliance on complex static analysis techniques leads to performance bottlenecks when processing large-scale libraries.

Dynamic testing verifies vulnerabilities through actual code execution, offering the inherent advantage of a low false-positive rate. ObjectMap [36] combines static guidance with dynamic verification, yet its generation logic depends on extensive manually defined rules. ODDFuzz [21], the current state-of-the-art dynamic tool, introduces structure-aware directed grey-box fuzzing using a property tree to guide seed mutation. However, the mutation strategy of ODDFuzz remains essentially a random exploration based on heuristic rules, lacking a deep understanding of Java semantic nuances. Consequently, it exhibits low efficiency when constructing payloads that satisfy complex type constraints, such as nested generics or factory method instantiations. Although traditional general-purpose fuzzers (e.g., AFL [17], Jazzer) perform well in terms of coverage, they treat inputs as flat byte streams, making it extremely difficult to generate highly structured serialized data. Even structure-aware fuzzers (e.g., Zest [37], NAUTILUS [38], GRIMOIRE [39]) require predefined grammar rules, which struggle to adapt to the complex object graph constraints inherent in Java deserialization.

In summary, the common challenge facing existing methods is that payload generation is essentially a complex code synthesis problem rather than a simple search or random mutation task. Static methods are limited by path explosion and the difficulty of modeling dynamic features [13,14], while dynamic methods are hindered by the semantic blindness of their mutation strategies [21]. Bridging the semantic gap between potential exploit paths and executable payloads remains a critical challenge in current research.

In recent years, Large Language Models have demonstrated significant capabilities in code understanding and generation tasks [1,22,23], giving rise to the emerging research direction of LLM-based fuzzing.

Code-specific LLMs—exemplified by Codex [1], StarCoder [40], CodeGen [41], InCoder [42], CodeT5+ [43], and DeepSeek-Coder [23]—have mastered programming language syntax and acquired the ability to comprehend complex API semantics and perform

logical reasoning through pre-training on massive code corpora [44,45]. By learning data flow representations of code [45], these models can better understand the semantic structure of programs, offering new possibilities for addressing the “semantic blindness” inherent in traditional fuzzing.

LLMs have been successfully applied across various software testing domains, demonstrating their potential for generating structured inputs. TitanFuzz [26] leverages LLMs to generate API test code for deep learning libraries, achieving zero-shot fuzzing. Fuzz4All [24] proposes a universal Autoprompting framework that iteratively prompts LLMs to test compilers and runtime environments. ChatAFL [27] introduces LLMs into protocol fuzzing to parse message structures, guiding AFL for more effective mutations. WhiteFox [25], FuzzGPT [46], and KernelGPT [47] demonstrate the advanced reasoning capabilities of LLMs in compiler optimization testing, edge-case generation, and kernel system call specification inference, respectively. Furthermore, edge-case generation for deep learning libraries [48] has also proven the potential of LLMs in identifying anomalous program behaviors.

3. Methodology

3.1. System Overview

We formulate the problem of Java deserialization payload generation as a constraint-based code generation task. Let C denote a given gadget chain specification, represented by the tuple $C = (N, L, E, S, \mathcal{P}, \mathcal{T})$, where

1. N is the name of the chain;
2. L is the version of the target dependency library;
3. E is the deserialization entry point
4. S is the target dangerous operation
5. $\mathcal{P} = [m_1, m_2, \dots, m_k]$
6. $\mathcal{T}$ is the set of type constraints within the chain, encompassing class definitions, inheritance hierarchies, and field types.

Our objective is to identify a mapping function f_{LLM} such that the synthesized code $P = f_{LLM}(C)$ satisfies the following conditions:

1. Compilability: $Compile(P) \rightarrow Success$;
2. Serializability: $Serialize(Execute(P)) \rightarrow ObjectStream$;
3. Effectiveness: $Deserialize(ObjectStream) \rightarrow Trigger(S)$.

The workflow of LLM-JDFuzz, as illustrated in Figure 1, adopts an iterative feedback-driven architectural design. The entire system progresses from preprocessing to final payload generation through four tightly coupled execution stages.

In the preprocessing stage, the Chain Spec Loader parses gadget chain metadata (in YAML format) to extract structured information, including the call stack, type constraints, and dependency library details, thereby constructing a formal ChainSpec object. This object encapsulates the complete semantic specifications of the gadget chain, providing the necessary knowledge base for subsequent prompt generation.

Upon entering the prompt generation stage, the Autoprompting Engine receives the ChainSpec alongside feedback from the previous iteration. It utilizes a template-based and context-aware dual-layer prompting architecture to transform formal constraints into natural language instructions comprehensible to the Large Language Model. The System Prompt layer defines the expert role and global constraints, while the User Prompt layer instantiates specific construction tasks.

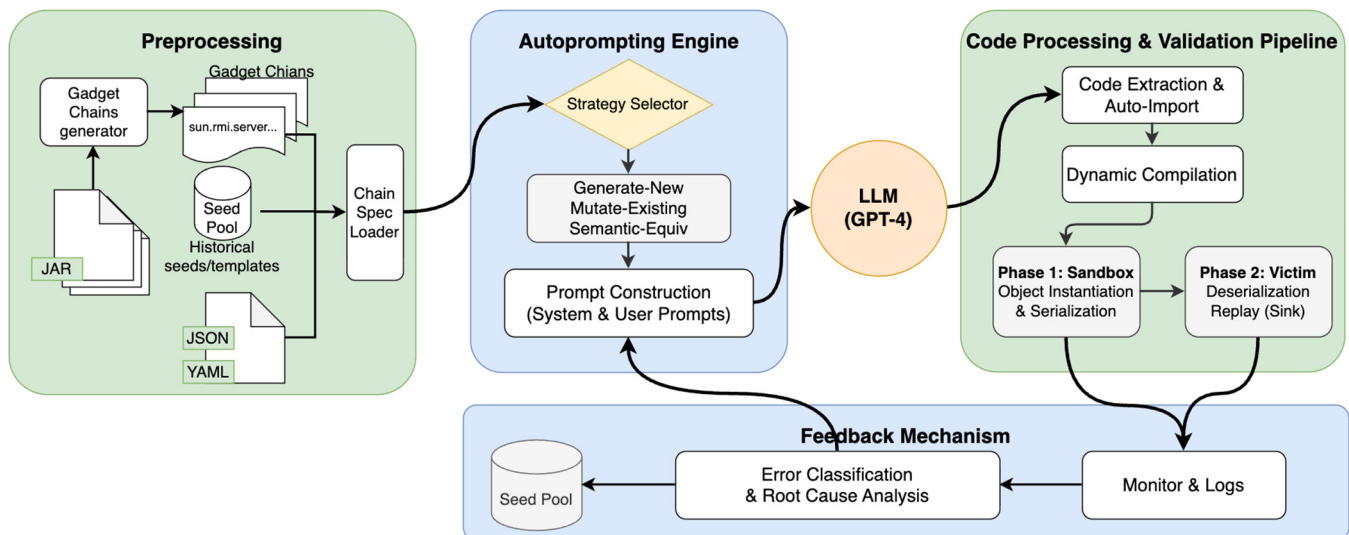


Figure 1. Overview of LLM-JDFuzz.

The generation and verification stage constitutes the core execution engine of the system. An Iterative Evolution Loop performs a four-step processing pipeline on the Java payload construction code synthesized by the LLM: first, it conducts code extraction and class name normalization; second, it automatically completes missing import statements; third, it dynamically constructs the classpath based on Group, Artifact, and Version (GAV) coordinates and executes compilation; finally, it executes the generation logic within an isolated sandbox to export the serialized stream. Subsequently, the system replays the deserialization process in an independent victim environment, utilizing Java Agent-based bytecode instrumentation to monitor the trigger status of the sink method in real-time.

In the feedback analysis stage, the Feedback Mechanism performs fine-grained classification of execution results, conducts root-cause attribution, and generates targeted repair suggestions. This structured feedback, integrated with the decisions of the Strategy Selector, is fed back into the Autoprompting Engine to drive the subsequent iteration. The system maintains high-potential intermediate code snippets within a Seed Pool, scored based on call-stack coverage depth and serialization integrity, thereby facilitating knowledge accumulation across iterations. This feedback loop persists for up to 20 rounds until a functional payload is successfully generated or the maximum iteration threshold is reached.

3.2. Chain Spec Loader

To bridge the gap between natural language models and formal program logic, a structured intermediate representation (IR) is required. The Chain Spec Loader is responsible for parsing and constructing the formal specifications of gadget chains, transforming raw chain metadata into the system-interpretable ChainSpec data structure.

3.2.1. Formal Definition of ChainSpec

We model a complete gadget chain as a quadruple comprising environmental context and execution logic: $Spec = \langle \mathcal{E}, \mathcal{S}, \mathcal{D}, \mathcal{C} \rangle$

1. Environment Context $\mathcal{E}$ defines the external boundaries for vulnerability triggering, including the GroupId, ArtifactId, and Version coordinates of target libraries and baseline requirements for JDK versions, ensuring that the Large Language Model can recognize specific API evolutions.
2. Semantic Call Stack $\mathcal{S}$ constitutes the logical skeleton of the payload. Unlike a simple list of class names, this component consists of a sequence of invocations annotated with

action predicates. Such a representation with semantic annotations assists the LLM in comprehending the functional role of each invocation node within the exploit chain.

3. Domain Knowledge $\mathcal{D}$ provides functional semantic enhancement for abstract classes or complex decorators involved in the chain. For instance, describing the Transformer interface as a “functional interface that converts an input object into another form” helps the LLM understand the data flow transformation logic, thereby compensating for the model’s potential lack of specific domain knowledge.
4. Type Constraints $\mathcal{C}$ explicitly declare constructor signatures, visibility restrictions for inner classes, and mandatory inheritance relationships. This is a critical component for ensuring the compilation correctness of synthesized code, which is particularly vital for strongly typed languages such as Java.

3.2.2. Feature-Weighted Complexity Assessment

To quantify the generation difficulty of various gadget chains and inform subsequent strategy selection, we introduce a complexity scoring mechanism, $Score(C)$. This metric is computed by aggregating the base call stack depth and a set of feature weights, w :

$$Score(C) = |P| + \sum_{i=1}^n w_i,$$

where $|P|$ denotes the call stack depth, w represents the weight assigned to specific technical challenges identified along the execution path, and n is the total number of such technical features. We conducted a preliminary study on the ysoserial benchmark and recorded the average number of iterations required for successful payload generation across different chain categories. Chains involving private inner classes required an average of 3.2 times more iterations than chains without such features, while chains with dynamic proxy mechanisms required approximately 2.1 times more iterations. These research findings formed the basis for initial weight assignments. We then performed a grid search over weight combinations and evaluated the correlation between computed complexity scores and actual generation difficulty. These weights achieved a Pearson correlation coefficient of 0.82 between predicted complexity and observed generation difficulty, indicating strong predictive power for generation difficulty assessment. The specific weight distribution is defined as follows:

1. Dynamic Proxy ($w = 2.0$): Involves dynamic dispatch through Proxy and its associated invocation handlers, which significantly increases the logical abstraction of the exploit.
2. Private Inner Classes ($w = 3.0$): Examples include classes within the sun.reflect package, which require the LLM to demonstrate the capability to bypass encapsulation via complex reflection mechanisms.
3. Implicit Constraints ($w = 1.5$): Examples include requirements for serialVersionUID consistency or non-null constraints for specific internal fields.

Through this modeling approach, we transform the traditionally ambiguous task of “exploit chain construction” into a goal-oriented search problem with explicit constraints.

3.3. Autoprompting Engine

The Autoprompting Engine serves as the core transformation layer of the system, designed to instantiate the formal definitions described in Section 3.2 into natural language instructions capable of precisely guiding the Large Language Model. Unlike traditional general-purpose code generation prompts, this engine focuses not only on the syntactic

validity of the code but also on injecting the deep constraints of the Java serialization protocol and the logic required for vulnerability triggering.

3.3.1. Hierarchical Prompt Architecture

The prompt architecture employs a dual-layer design to progressively guide the LLM in understanding task semantics, moving from abstract objectives to concrete implementation details.

At the top level, the Expert Intention Layer utilizes the System Prompt to anchor the LLM as a Java research expert with an extensive background in deserialization security. This layer pre-configures a set of global constraints for generating serialization payloads, such as requiring the synthesized code to include a complete `ObjectOutputStream` serialization pipeline, necessitating the use of reflection APIs to bypass private member access restrictions, and implementing automated exploit validity verification logic within the main method. This role-anchoring mechanism activates the security domain knowledge acquired by the model during its pre-training phase.

At the bottom level, the Semantic Instantiation Layer utilizes the User Prompt to transform the quadruple $\langle \mathcal{E}, \mathcal{S}, \mathcal{D}, \mathcal{C} \rangle$ from the ChainSpec into concrete synthesis instructions in real-time through attribute mapping. This layer dynamically populates structured templates with the call stack $\mathcal{S}$ and environment context $\mathcal{E}$, ensuring that the generated code precisely matches the target dependency environment.

3.3.2. Task-Specific Prompting Strategies for Deserialization Scenarios

To effectively converge within the vast search space of the object graph, the Auto-prompting Engine dynamically dispatches three complementary generation strategies based on real-time feedback from the fuzzing loop.

During the cold-start phase of the system or when exploring a new chain, the engine employs Skeleton-Based Cold-start Synthesis. This strategy generates guided instructions by parsing the call stack $\mathcal{S}$, utilizing Chain-of-Thought (CoT) [49] and Tree of Thoughts (ToT) [50] techniques to lead the LLM through deducing the topological order of object construction before generating the Java code hierarchically.

When synthesized code encounters compilation failures or execution crashes, the engine switches to Error-Aware Semantic Remediation. The system automatically captures the exception stack trace thrown by the JVM and transforms it into targeted repair instructions. This automated remediation mechanism, powered by LLMs, draws upon recent advancements in the field of automated program repair [51–54]. This process can be formally represented as

$$Prompt_{next} = \Phi(P_{current}, Error_{log}, \mathcal{C})$$

To evade static detection and explore more covert exploit methods, the system also supports Equivalence-Based Behavioral Evolution. This strategy drives implementation diversity while maintaining the underlying gadget logic. By providing instructions that guide the LLM to use different APIs to achieve the same object graph construction effect, the system enhances the robustness and stealth of the resulting payloads.

3.3.3. Instruction Optimization and Knowledge Injection

To further improve generation efficiency, the Autoprompting Engine introduces a heuristic knowledge injection mechanism. The system optimizes instruction templates through automated prompt engineering [55], ensuring that the synthesized code aligns precisely with the target dependency environment. Based on the domain knowledge $\mathcal{D}$ within the *Spec*, the system automatically supplements the prompts with necessary Java

type casting code and specific serialVersionUIDs. This deep domain integration enables the LLM to complete payload construction based on its understanding of Java serialization semantics, even without having previously encountered the specific gadget chain in its training data.

3.4. Iterative Evolution Loop

The Iterative Evolution Loop serves as the core driving engine of the system, implementing a feedback-based iterative evolutionary process. Distinct from conventional code generation, the construction of Java deserialization payloads is essentially the synthesis of an in-memory object graph that satisfies a specific topological structure. Consequently, the design of this section aims to guide the Large Language Model in progressively refining the validity of the object graph by continuously analyzing runtime state feedback from the Java Virtual Machine.

We formalize the generation process as a sequence of state evolutions. In the t -th iteration, the system maintains a state triple $S_t = \langle P_t, G_t, \Gamma_t \rangle$, where

1. P_t : Represents the Java construction code synthesized by the LLM;
2. G_t : Denotes the in-memory object graph instantiated from the code within the sandbox;
3. Γ_t : Signifies the semantic feedback captured during the execution phase, encompassing compilation logs, exception stack traces, and sink trigger status.

To facilitate an efficient search within the vast object construction space, the system implements an execution-state-based adaptive strategy scheduling mechanism based on the feedback G_t .

When a compilation error occurs in the preceding iteration, the system determines that the current code structure possesses fundamental defects. In such cases, the system enforces a transition to the Generate-New strategy, discarding the current code and re-generating from the ChainSpec. This “fail-fast” mechanism prevents futile local patching on an inherently flawed code base.

When a runtime exception occurs, the system determines that the code structure is fundamentally sound but contains localized semantic errors. Consequently, the Mutate-Existing strategy is prioritized for directed remediation. This strategy injects the exception stack trace and error context into the prompt, guiding the LLM toward precise fault localization and repair.

If the system detects a lack of progress for K consecutive iterations (where $K = 5$ by default), or if the current seed has converged, the system switches to the Semantic-Equiv strategy to enhance diversity. By exploring equivalent implementation variants, this strategy assists the system in escaping local optima.

3.5. Code Processing Pipeline

Since LLM outputs typically include explanatory natural language, the extractor first locates Java code blocks via Markdown tags; in the absence of such tags, it employs regular expressions to retrieve core logic segments containing public class and static void main. To mitigate class name conflicts across different generation iterations, the pipeline automatically renames the target class to PayloadGenerator and supplements missing standard library imports. Subsequently, the dynamic compilation engine dynamically loads the corresponding JAR files from a local Maven repository based on the dependency coordinates defined in the ChainSpec, constructs the classpath, and compiles the source code into bytecode by invoking the host’s javac.

The core verification phase adopts a Dual-Phase Validation architecture, which is critical for ensuring the efficacy of the payload in real-world deserialization scenarios.

Object Instantiation and Exportation. The system executes PayloadGenerator within a generative sandbox environment. The objective of this phase is to execute the object initialization logic synthesized by the LLM and serialize the resulting object graph into a binary stream file. If an exception is thrown during this stage, it indicates that the object graph contains nodes that violate the serialization protocol.

Deserialization Replay. The system initializes an independent, monitored victim environment to simulate authentic attack scenarios. The binary stream produced in the first phase is loaded via `ObjectInputStream#readObject`.

During this verification process, the pipeline integrates a multi-dimensional sink trigger monitoring mechanism. Beyond capturing Remote Code Execution echo signatures through standard output, we leverage Java Agent technology to perform bytecode instrumentation on the target JVM. The monitor intercepts calls to critical sensitive APIs in real-time, effectively identifying “silent triggers” of vulnerabilities. Finally, all execution records, exception stack traces, and monitoring logs are structured into a feedback vector for transmission to the subsequent attribution engine.

3.6. Feedback Mechanism

The initial phase of the Feedback Mechanism involves fine-grained error classification and causal attribution. The attribution engine categorizes captured exceptions into three dimensions: compilation-stage syntactic defects, serialization constraint violations, and runtime logical disruptions.

For compilation errors, the engine does not merely replicate error messages; instead, it utilizes static analysis to locate missing dependency classes or incompatible method signatures, subsequently providing completion suggestions.

Regarding serialization violations, the engine backtracks through the object graph to identify the specific member variables that disrupt the serialization chain. It then automatically generates prompts instructing the model to replace these variables with implemented wrapper classes or to mark them as transient via reflection techniques.

For logical disruptions, the engine analyzes the invocation trace to determine the exact point where the execution flow deviated from the gadget chain, thereby guiding the model toward targeted field assignment adjustments.

To achieve efficient evolution, we introduce prompt augmentation techniques. The attribution engine transforms the aforementioned analysis results into action-oriented natural language instructions. For instance, if restricted access to a private inner class is detected, the feedback generated is no longer a generic “illegal access” error but a clear instruction: “The current target class is restricted by private access; please use the reflection API `setAccessible` to bypass encapsulation and force the injection of object states.” This domain-knowledge-driven guidance enables the LLM to transition from pure code generation to defensive programming tailored to vulnerability constraints, significantly reducing the overhead of stochastic trial-and-error.

Figure 2 depicts the error-to-repair workflow instantiated on the CommonsCollections6 payload generation case. In Stage 1, the JVM raises a `java.lang.IllegalAccessException`, indicating restricted access to the private inner class `AnnotationInvocationHandler`. The feedback module then attributes the failure to a visibility constraint imposed by Java’s access control mechanism, inferring that direct instantiation violates encapsulation requirements. Based on this analysis, the system reformulates the exception into a targeted repair instruction that recommends using Java reflection APIs (e.g., `Class.forName`, `getDeclaredConstructor`, and `setAccessible`) to bypass access restrictions and properly initialize the required object state. The LLM subsequently incorporates this guidance into the re-generated code, resulting in a compilable payload in the next iteration. Overall, this

example illustrates how the feedback mechanism translates low-level runtime exceptions into structured natural-language repair instructions within a bounded iterative loop of up to 20 iterations.

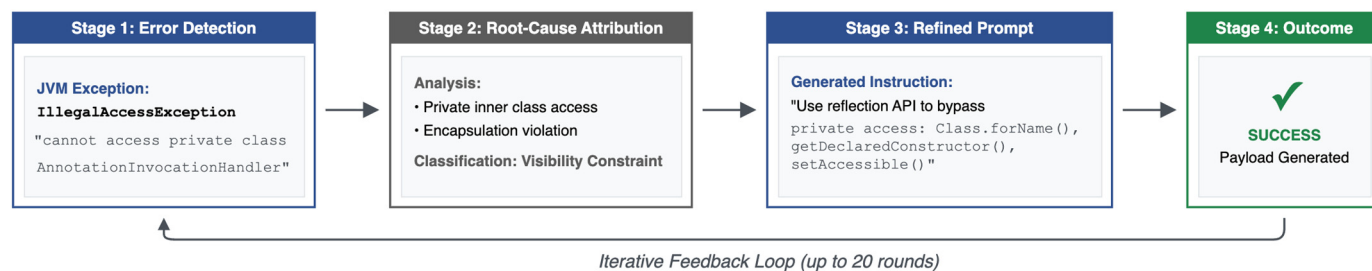


Figure 2. Error Attribution and Refined Prompt Pipeline.

Furthermore, the feedback mechanism is responsible for maintaining a dynamic seed scoring system. The system assigns a potentiality score to each generated intermediate result based on metrics such as call stack coverage depth, the integrity of the serialization stream, and the proximity to the target sink. Code snippets with higher scores, even if they fail to trigger the vulnerability, are stored in the seed pool as evolutionary foundations. In subsequent cycles, the scheduler prioritizes these code segments with high-potential characteristics for mutation. By continuously accumulating successful local construction logic within the feedback loop, the system ultimately achieves full-link breakthroughs for complex, multi-layered gadget chains.

4. Evaluation

4.1. Research Questions (RQs)

Our study is driven by the following four core research questions:

- RQ1: Effectiveness. How does the Success Rate of LLM-JDFuzz in generating functional payloads compare with existing state-of-the-art static analysis and dynamic fuzzing tools?
- RQ2: Complexity Resilience. How does LLM-JDFuzz perform when processing gadget chains with varying levels of structural complexity?
- RQ3: Ablation Study. What are the specific contributions of the individual core components within the ChainSpec quadruple and the system-level feedback mechanism to the overall generation success rate?

4.2. Experimental Setup

4.2.1. Implementation and Environment

System Implementation. The core generation engine of LLM-JDFuzz was constructed based on the GPT-4 model. To maintain a balance between deterministic output and creative synthesis during code generation, the sampling temperature was set to 0.7. Given the structural complexity of Java classes, the maximum token limit for a single interaction was restricted to 4096. The system was configured to allow a maximum of 20 feedback iterations for each target gadget chain.

All the experiments were carried out on a Docker environment which deployed on a machine with Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz (4×12 core) and 128 GB of RAM. We utilized Oracle JDK 8u65 as the baseline runtime environment. The containerized environment implemented strict Seccomp profiles to intercept non-essential system calls and enforced network isolation, permitting only necessary DNS resolution requests.

4.2.2. Dataset

Benchmark Dataset. We selected ysoserial as the core dataset for our evaluation. As the widely acknowledged ground truth in the field of Java deserialization research, this collection comprises 34 gadget chains spanning 15 prevalent third-party libraries.

To eliminate false positives, we implemented a lightweight Java Agent detector that performs real-time monitoring of runtime behavior through bytecode instrumentation.

4.3. RQ1: Effectiveness Comparison

We evaluated a fully configured version of LLM-JDFuzz on 34 gadget chains from ysoserial. The experimental results demonstrate that LLM-JDFuzz exhibits exceptional automated generation capabilities.

4.3.1. LLM-JDFuzz Overall Performance Evaluation

Analysis of Core Metrics. Across 170 experimental runs (34 chains times $\times$ 5 trials), LLM-JDFuzz successfully generated functional payloads for 24 gadget chains, achieving a chain-level success rate of 70.6%. The average number of iterations required for successful generation was 7.3. Notably, only 8.2% of successful cases were achieved on the first attempt, which strongly validates the necessity of our closed-loop feedback-correction mechanism when addressing complex security constraints. By capturing semantic feedback from both compilation and runtime phases, the LLM is able to progressively refine the synthesized object construction code.

Success Rate Distribution and Convergence. Significant performance variance was observed across different categories of gadget chains. Specifically, 17.6 of the chains exhibited extreme stability, achieving a 100% success rate with an average convergence within 5 iterations. This suggests that LLMs possess strong semantic prior knowledge of patterns related to common collection libraries and dependency injection frameworks.

4.3.2. Comparative Analysis with State-of-the-Art Tools

We compared LLM-JDFuzz against several state-of-the-art (SOTA) tools, including JDD, ODDFuzz, GadgetInspector, and Serianalyzer. Table 1 summarizes the overall performance of each tool on the ysoserial benchmark suite.

Table 1. Effectiveness Comparison Summary (N = 34).

Tool	Successes	Success Rate	Avg. Iterations	Avg. Gen. Time (s)
LLM-JDFuzz	24	70.6%	7.3	45.2
JDD	27	79.4%	-	180.5
ODDFuzz	16	47.1%	15.8	120.3
GadgetInspector	3	8.8%	-	25.7
Serianalyzer	2	5.9%	-	18.4

Significant Superiority over Traditional Fuzzing: LLM-JDFuzz achieved a success rate 23.5 percentage points higher than that of the mutation-based tool ODDFuzz. This advantage stems from the fundamental difference between semantic understanding and random mutation. For complex Java object graphs, random mutations struggle to navigate the vast search space to satisfy strong-typing constraints. In contrast, LLMs can directly synthesize logically valid code based on class definitions.

Competitiveness with Heavyweight Symbolic Execution: Although JDD achieved a higher success rate on TemplatesImpl-related chains due to its precise symbolic constraint solving, LLM-JDFuzz reached a comparable performance level. More importantly,

LLM-JDFuzz operates with significantly higher efficiency, reducing the average generation time by 75% compared to JDD, as shown in Table 1.

4.3.3. In-Depth Analysis by Chain Category

To uncover the coverage blind spots of different technical solutions, we conducted a granular comparative analysis of success rates across various sub-libraries. This analysis reveals the core advantages of LLM-JDFuzz across three representative scenarios.

In scripting engine scenarios, LLM-JDFuzz demonstrates a significant advantage in semantic understanding. For gadget chains involving scripting engines such as Clojure, Rhino, and Jython, LLM-JDFuzz achieved a success rate of 100%, substantially outperforming JDD (75%) and ODDFuzz (50%). These results indicate that LLMs possess unique semantic priors regarding cross-language API calls, allowing them to handle such heterogeneous semantics more intuitively than traditional analysis tools.

Regarding adaptability to enterprise frameworks, LLM-JDFuzz also performs exceptionally well. In heavyweight enterprise frameworks such as Myfaces and Hibernate, LLM-JDFuzz demonstrated adaptability on par with JDD. This can be attributed to the deep knowledge the LLM has acquired from large-scale code corpora concerning framework initialization and the implementation patterns of the Serializable interface.

In terms of robustness when handling dynamic features, LLM-JDFuzz significantly outperforms static tools like GadgetInspector in its ability to process reflection and dynamic proxies. Experiments show that for chains involving dynamic proxies, LLM-JDFuzz achieved a success rate of 66.7%, whereas traditional static tools failed to generate any valid payloads. This failure is primarily because static analysis struggles to precisely model proxy classes generated at runtime.

4.3.4. Evaluation on Post-Cutoff Gadget Chains

To validate that LLM-JDFuzz performs genuine reasoning rather than mere memorization, we conducted additional experiments on gadget chains disclosed after GPT-4's knowledge cutoff date of September 2021. We selected three representative vulnerabilities with confirmed CVE identifiers and diverse exploitation mechanisms: CVE-2021-44228 Log4Shell disclosed in December 2021, which enables remote code execution through JNDI lookup injection in Apache Log4j logging library; CVE-2022-22965 Spring4Shell disclosed in March 2022, which exploits parameter binding manipulation in Spring Framework to achieve arbitrary file write and code execution via Tomcat's AccessLogValve modification; and CVE-2022-39198 Hessian Deserialization disclosed in September 2022, which involves manipulating Hessian's alternative binary serialization protocol to trigger arbitrary method invocation. The comprehensive evaluation results for these post-cutoff gadget chains are summarized in Table 2.

Table 2. Evaluation Results on Post-Cutoff Gadget Chains.

CVE ID	Vulnerability	Disclosure Date	Success Rate	Avg. Iterations	Avg. Time (s)
CVE-2021-44228	Log4Shell JNDI Injection	December 2021	80%	8.4	52.3
CVE-2022-22965	Spring4Shell	March 2022	60%	12.6	78.5
CVE-2022-39198	Hessian Deserialization	September 2022	60%	11.2	65.8

The experimental results demonstrate that LLM-JDFuzz successfully generates functional payloads for all three post-cutoff vulnerabilities, achieving an average success rate of 66.7%. Log4Shell achieved the highest success rate at 80% due to its relatively straightforward JNDI lookup mechanism, while Spring4Shell and Hessian showed moderate success rates of 60%, reflecting their higher complexity involving class loader manipulation and al-

ternative serialization protocols. The framework successfully synthesized working exploits for vulnerabilities disclosed 3–12 months after the LLM’s training data cutoff, demonstrating the ability to generalize learned patterns to novel vulnerability classes across diverse exploitation mechanisms.

4.4. RQ2: Complexity Resilience Assessment

While RQ1 established the overall effectiveness of LLM-JDFuzz, the construction difficulty of gadget chains varies significantly in practice. This section aims to investigate the robustness of LLM-JDFuzz when processing chains of varying complexity and to reveal the specific scenarios where its semantic understanding capabilities play a pivotal role.

4.4.1. Complexity Quantification System

To systematically evaluate the performance of the tools across different difficulty levels, we designed a complexity scoring mechanism based on multi-dimensional features. This mechanism comprehensively considers four categories of core technical challenges encountered during payload construction:

$$CS = N_{class} \times 1.0 + N_{proxy} \times 2.0 + N_{internal} \times 3.0 + N_{reflect} \times 0.5$$

where N_{class} denotes the number of key classes involved in the chain, N_{proxy} represents the number of dynamic proxy usages, $N_{internal}$ indicates the number of dependencies on JDK internal classes, and $N_{reflect}$ signifies the number of explicit reflection operation invocations. The weight coefficients are assigned based on the following considerations: dynamic proxies require an in-depth understanding of interface semantics and the InvocationHandler mechanism; internal class access often involves complex reflection bypasses and version compatibility issues; and regular reflection operations, though frequent, typically follow relatively fixed patterns.

Based on this score, we categorized the 34 chains into three complexity levels: Low Complexity ($CS \leq 5$), comprising 10 chains that typically manifest as direct method invocation sequences; Medium Complexity ($5 < CS \leq 10$), comprising 17 chains involving object nesting and generic constraints; and High Complexity ($CS > 10$), comprising 7 chains that necessitate the simultaneous handling of multiple advanced Java features.

4.4.2. Analysis of Overall Complexity Distribution and Success Rate

Table 3 presents the performance of LLM-JDFuzz across different complexity levels. The experimental results exhibit a non-monotonic trend: the success rate for medium-complexity chains is significantly higher than those of both low- and high-complexity chains.

Table 3. Success Rate of LLM-JDFuzz Grouped by Complexity.

Complexity Level	Number of Chains	Successful Chains	Success Rate	Average Iterations
Low	10	5	50.0%	7.5
Medium	17	14	82.4%	7.0
High	7	5	71.4%	7.5
Total	34	24	70.6%	-

This phenomenon reveals a key insight: the difficulty of payload generation is not solely determined by technical complexity, but rather depends on the “learnability” of the chain. Medium-complexity chains often possess distinct code patterns and appear frequently in public security research and Proof-of-Concept code. This enables the Large Language Model to learn the corresponding construction paradigms during the pre-training phase. In contrast, while low-complexity chains involve fewer classes, they may require

specific protocol details or low-level bytecode generation, scenarios that have limited coverage in code corpora.

Figure 3 illustrates the relationship between the Complexity Score and the success rate using a scatter plot. Each point represents a gadget chain, where the color indicates the complexity level and the size represents the average number of iterations. The red dashed line denotes the linear regression trend line, with a Pearson correlation coefficient of $r = -0.103$, which does not reach statistical significance ($p > 0.05$). This figure intuitively reveals the non-monotonic phenomenon where “medium complexity yields the highest success rate,” supporting our assertion that learnability is more critical than technical complexity.

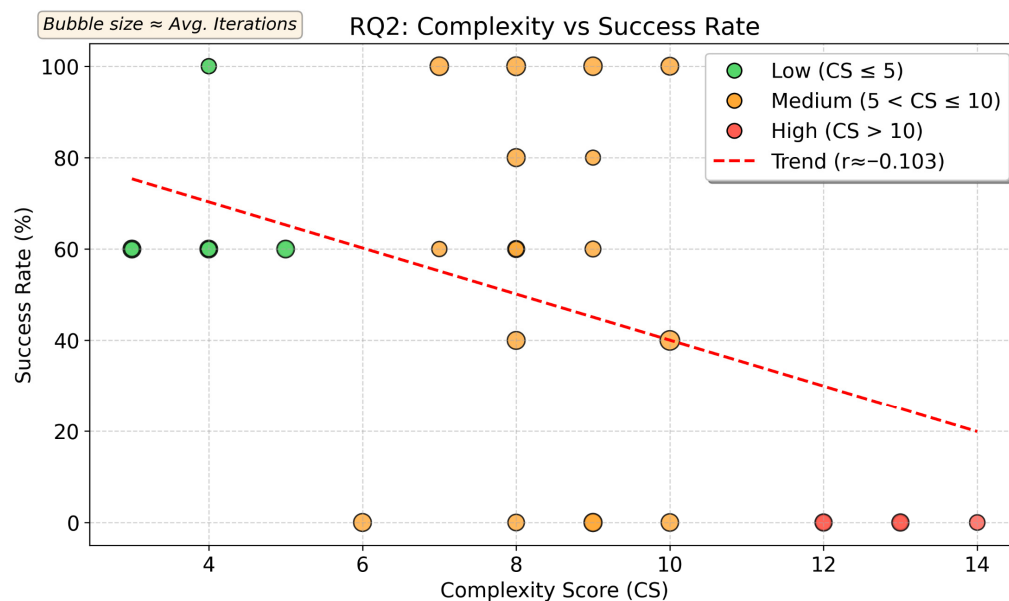


Figure 3. Impact of Gadget Chain Complexity on Generation Success Rate.

To quantify the relationship between complexity and success rate, we calculated the Pearson correlation coefficient. The results show $r = -0.103$ ($n = 18$), indicating a weak negative correlation. This further confirms that the complexity score is not the dominant factor governing generation success; rather, semantic understandability and training data coverage play equally pivotal roles.

4.5. RQ3: Comparison of Generation Strategies

Having established LLM-JDFuzz’s overall effectiveness and resilience in RQ1 and RQ2, we now isolate the specific contribution of the Adaptive Multi-Strategy Generation mechanism.

4.5.1. Strategy Configuration and Performance Comparison

We compared our adaptive scheduling algorithm against four representative strategy configurations. To ensure statistical significance, all configurations were fully evaluated on the 34 benchmark chains from ysoserial. Table 4 summarizes the definitions of each strategy and their performance in terms of success rate and convergence properties.

Table 4. Definitions of different generation strategy configurations and performance comparison on ysoserial.

Strategy ID	Strategy Name	Logic Description	Success Rate	Success Count	Avg. Iterations
C1	Only-Generate	Independent reconstruction in every round	33.3%	11	9.5
C2	Only-Mutate	Feedback-based repair after the first round	16.7%	6	3.5
C3	Only-Semantic	Semantic equivalent transformation after the first round	0%	0	-
C4	Random	Equal probability random selection of Generate/Mutate/Semantic	16.7%	6	5.0
C5	Full-Adaptive	Adaptive switching based on execution feedback	70.6%	14	7.3

Experimental results demonstrate that the adaptive scheduling mechanism (C5) achieved a performance improvement of more than two-fold in success rate compared to the best-performing single-strategy variant (C1). This superiority originates from the capability of C5 to dynamically adjust strategies based on the code execution state: it triggers the Mutate strategy for fine-grained repairs when compilation errors are detected, and invokes the Generate strategy to reset the construction logic when continuous repairs fail or the logic reaches an impasse. This organic combination of “local search” and “global reconstruction” is the key to LLM-JDFuzz overcoming the semantic constraints of complex gadget chains.

Regarding the distribution of average iterations, we observed that C2 exhibited the shortest lifecycle (3.5 rounds). However, a low number of iterations in this context does not signify efficiency; rather, it reflects premature stagnation of the strategy. When the initially generated code possesses architectural flaws, simple mutative repairs cannot bridge the underlying logical gap, causing the iteration to terminate within a few rounds due to persistent compilation failures. In contrast, the average iterations of C5 (7.3 rounds) reflect the system’s enhanced resilience when facing complex constraints, gradually approaching a functionally complete payload construction through multiple adaptive adjustments.

Failure Mode Insights:

Failure of C3: The complete failure of this configuration validates our design hypothesis—namely, that semantic equivalent transformations are meaningful only when the code already possesses basic functional completeness. In the early stages of construction, C3 lacks the corrective capacity to handle any syntax or logical errors.

Inefficiency of C1: Although C1 achieved a 33.3% success rate due to the strong pre-training priors of the LLM, its iteration count (9.5 rounds) and randomness were extremely high. This suggests that without feedback guidance, the system can only “collide” with a correct object graph through continuous retries, lacking stability.

4.5.2. Comparison of Strategy Effectiveness

To further analyze the coverage capability of each strategy across gadget chains of varying difficulty, Table 5 displays the success rate distribution of each strategy across typical sub-libraries.

Table 5. Success rate distribution of strategies across different types of Gadget Chains.

Target Library Type	Number of Chains	C1	C2	C4	C5
Commons Collections	7	28.6%	14.3%	14.3%	57.1%
Script Engines	4	50.0%	0%	25%	100%
Enterprise Frameworks	4	0%	25%	0%	100%
Base Libraries	19	42.1%	21.1%	21.1%	73.7%

The experimental data indicates that for chains involving complex framework semantics, non-adaptive strategies (C1, C4) are almost entirely unable to produce valid results. This proves that for tasks requiring precise satisfaction of specific framework lifecycle constraints, Iterative Refinement based on execution feedback is an indispensable technical pillar.

In summary, the experiments demonstrate that adaptive strategy scheduling (C5) is critical for achieving complex payload generation. Single-strategy configurations (C1–C4) easily fall into local optima or random divergence when handling high-difficulty gadget chains. By balancing exploration and exploitation through an adaptive feedback loop, C5 ensures generation effectiveness while significantly enhancing the quality of logical convergence within the system.

4.6. RQ4: Ablation Study

After validating the overall efficacy of LLM-JDFuzz, this section provides an in-depth analysis of the design rationale behind the Autoprompting engine through a systematic ablation study. Our objective is to quantify the independent contributions and synergistic effects of each core component within the ChainSpec quadruple—Environment Context (EC), Semantic Call Stack (CS), Domain Knowledge (DK), and Type Constraints (TC)—as well as the system-level Error Feedback (EF) on the successful generation of functional payloads.

4.6.1. Design of Ablation Variants

To comprehensively deconstruct the contribution of the prompt space, we designed seven experimental schemes, including the full configuration (Full), five leave-one-out variants that remove a single component, and a minimal baseline configuration (Minimal):

Experimental Setup: All variants were evaluated on the 34 benchmark gadget chains with five repeated runs (totaling 1020 independent runs), with the maximum number of iterations set to 20.

4.6.2. Experimental Results and Quantitative Analysis

Table 6 presents the performance of each variant in terms of Chain Success Rate (CSR) and Run Success Rate (RSR).

To accurately interpret the results, we first define the key metrics. Chain Success Rate refers to the proportion of the 34 chains that succeeded at least once across five runs. Run Success Rate is the proportion of successful runs out of the total 170 trials. Notably, the A5 (w/o EF) configuration exhibits a unique behavior: with the feedback loop disabled, the system terminates after a single generation round. The single-round prompt used in A5 was identical to the initial prompt used in the Full configuration (A0), without any specialized zero-shot optimization. This design establishes a fair baseline for measuring the contribution of the feedback loop. The A5 variant operates as follows: the Autoprompting Engine generates the initial prompt based on the ChainSpec quadruple; the LLM synthesizes the payload code; the Code Processing Pipeline compiles and executes the code; if any error occurs, the system terminates immediately without invoking the Feedback

Mechanism. The results show that A5 achieves a CSR of 61.8% but an extremely low RSR of 21.2%, and this substantial gap demonstrates that without iterative feedback, the LLM can produce structurally correct code that fails to satisfy the precise semantic constraints required for successful payload execution. Although 21 chains (61.8%) succeeded at least once, only 36 out of 170 runs (21.2%) were successful. This phenomenon reveals several insights: in a zero-shot mode, while a majority of chains can be generated, the stability (RSR) is significantly lower than the 55.3% achieved by the full system. Furthermore, the average iteration count for A5 is inherently 1.0. A critical insight here is that the success of A5 relies entirely on the LLM’s zero-shot capability, with successful cases being restricted to classic, high-frequency chains in the training data.

Table 6. Comprehensive Configuration and Performance Analysis of Ablation Variants.

Variant ID	Experiment Name	Enabled Core Components	Chain Success Rate	Run Success Rate	Avg. Iterations (Successful)	Avg. Attempts (All Cases)	Success Diff (Δ vs. A0)
A0	Full	EC + CS + DK + TC + EF	85.3%	55.3%	7.1	7.3	-
A1	w/o EC	CS + DK + TC + EF	82.4%	52.9%	7.3	7.5	-2.9%
A2	w/o CS	EC + DK + TC + EF	79.4%	50.6%	7.2	7.5	-5.9%
A3	w/o DK	EC + CS + TC + EF	70.6%	45.2%	7.8	8.1	-14.7%
A4	w/o TC	EC + CS + DK + EF	52.9%	34.1%	8.9	9.2	-32.4%
A5	w/o EF	EC + CS + DK + TC	61.8%	21.2%	1.0	1.0	-23.5%
A6	Minimal	Basic metadata only	41.2%	27.6%	6.7	7.0	-44.1%

By comparing the full configuration (A0) with the no-feedback configuration (A5), we identify the true value of the feedback mechanism. While A0 yields 29 successful chains (94 successful runs, 55.3%), A5 yields only 21 successful chains (36 successful runs, 21.2%). Remarkably, the CSR of A5 drops by only 8 chains (-23.5%), yet the RSR plummets by 58 successful trials (-62%). This discrepancy indicates that while the feedback mechanism does not necessarily determine “feasibility,” it dramatically enhances “stability.” In other words, the feedback loop transforms incidental success into reproducible, stable output.

Figure 4 illustrates the performance comparison across the seven ablation variants using a grouped bar chart. The green bars represent the Chain Success Rate, and the orange bars represent the Run Success Rate. A0 serves as the full-configuration baseline, A1–A5 represent the leave-one-out variants, and A6 is the minimal configuration. It is evident that removing Type Constraints (A4) leads to a sharp 32.4% decline in success rate, proving it is the most critical component. Removing Error Feedback (A5) results in a 62% crash in RSR, demonstrating the importance of feedback for stability.

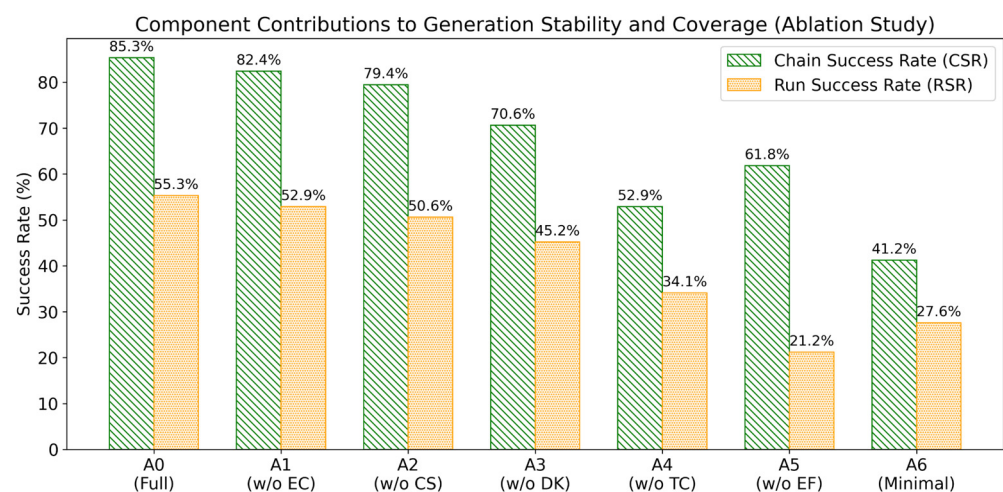


Figure 4. Component Contributions to Generation Stability and Coverage.

Compared to the Minimal configuration (A6), the full system (A0) increases the CSR from 41.2% to 85.3% (+107% relative improvement). This demonstrates the indispensability of structured prompt engineering when constructing complex security primitives.

Variant A4 exhibits the most severe performance degradation (a 32.4% decrease in CSR), revealing that Java's strong typing semantics constitute the primary obstacle for LLMs in payload construction.

Although A5 maintains a CSR of 61.8%, its RSR is extremely low (21.2%). This suggests that while EF may not be the sole determinant of whether a chain can succeed, it significantly boosts the probability of success by converting stochastic generation attempts into deterministic solutions through iterative refinement.

5. Discussion

This chapter builds upon the aforementioned experimental results to conduct an in-depth exploration of the advantages and limitations of LLM-JDFuzz, while critically examining the validity of the experiments. We first analyze the core competitiveness of the LLM-based approach compared to traditional tools, followed by identifying current deficiencies in both theory and practice, and finally outlining potential directions for future improvement.

5.1. Limitations and Threats to Validity

Inherent Methodological Limitations and Epistemic Boundaries. The experimental results reveal that the performance of LLM-JDFuzz is heavily constrained by the distribution of the code corpora used during the pre-training phase. While the system performs exceptionally well on medium-complexity chains, its performance on low- and high-complexity chains exhibits non-linear fluctuations. We hypothesize that this reflects a discrepancy in the model's capability when handling "in-distribution" versus "out-of-distribution" data. Medium-complexity gadget chains, such as CommonsCollections6, enjoy high popularity within the open-source community, making it highly probable that the LLM acquired their construction patterns during pre-training. In contrast, scenarios involving specific JNDI configurations in C3P0 or dynamic bytecode generation in TemplatesImpl present severe epistemic boundary challenges, as their technical implementation details are insufficiently covered in the training corpora.

Threats to Internal and External Validity. The validity of this study is subject to several potential threats. First is the representativeness bias of the dataset: although ysoserial is an industry benchmark, it primarily targets legacy environments such as JDK 8u65. Modern JDK versions, particularly JDK 17 and later, have introduced significant security enhancements that impact the reachability of traditional deserialization gadget chains. JDK 9 introduced the Java Platform Module System, which enforces strong encapsulation of internal APIs. Many gadget chains rely on internal classes in packages such as `sun.reflect.annotation` and `com.sun.org.apache.xalan.internal.xsltc.trax`, and in JDK 17+, access to these packages is restricted by default, rendering many traditional payloads non-functional without explicit module exports. Additionally, while JEP 290 was introduced in JDK 9, its integration and default filtering rules have been strengthened in subsequent versions, providing more restrictive default serialization filters for modern applications. However, it is important to note that despite these security improvements in newer JDK releases, JDK 8 remains the most widely deployed version in enterprise environments. According to recent industry surveys, over 60% of enterprise Java applications still run on JDK 8, particularly in legacy systems, financial institutions, and government sectors where migration to newer versions faces significant technical and regulatory barriers. Our evaluation targeting JDK 8u65 therefore reflects the most prevalent real-world deployment

scenario. Comprehensive evaluation on modern JDK environments remains an important direction for future work.

Handling Obfuscated and Non-standard Serialization Scenarios. While LLM-JDFuzz demonstrates strong performance on the ysoserial benchmark, we acknowledge that the framework's capabilities have not been evaluated on deliberately obfuscated gadget chains. The ysoserial dataset, as an industry benchmark, focuses on demonstrating known vulnerability patterns rather than testing evasion capabilities. For scenarios involving heavily obfuscated code (e.g., class/method name obfuscation, custom serialization protocols with complex validation logic, or anti-analysis techniques), the framework's effectiveness may be limited. The LLM's semantic understanding, while powerful, relies on recognizing patterns and API semantics that may be obscured in such cases. Future work could explore integration with deobfuscation techniques or symbolic execution to enhance robustness in adversarial scenarios.

5.2. Future Research Directions

Fine-grained Feedback and Intelligent Constraint Inference. To address the current bottleneck where the model struggles with low-level bytecode tasks like `TemplatesImpl`, we plan to integrate dynamic taint analysis and lightweight symbolic execution into the feedback loop. By embedding a Java Agent during the sandbox execution phase, we can track the assignment trajectories of object fields in real time. If a sink fails to trigger, the system will generate feedback based on data-flow information rather than relying solely on ambiguous compiler errors. This approach of translating low-level execution semantics into high-level natural language prompts is expected to guide the LLM beyond its current reasoning constraints.

Semantic Enhancement and Domain Adaptation. To further enhance the descriptive power of ChainSpec, future work will explore the introduction of object graph topology descriptions and negative prompts (anti-patterns). By injecting error patterns extracted from historical failure cases into the prompts, the probability of the model repeating previous mistakes can be significantly reduced. Furthermore, domain-specific fine-tuning is a critical direction. By fine-tuning open-source large language models on CVE reports, security blogs, and exploit code [56–60], we aim to build specialized generative models with a deeper understanding of the Java serialization protocol. It is noteworthy that while enhancing the model's capability for security-related code generation, vigilance is required regarding the potential security risks introduced by the model itself. We anticipate achieving a superior balance between performance and cost through these specialized.

Integration of Dynamic Taint Analysis. We plan to integrate dynamic taint analysis into the feedback loop to enhance detection and handling of complex low-level deserialization payloads. The current feedback mechanism primarily relies on exception stack traces and bytecode instrumentation for error attribution. By incorporating taint analysis, the system could track data flow through the object graph during deserialization, enabling precise identification of field assignments that violate serialization constraints. We envision an enhanced pipeline where a Java Agent tracks taint propagation during sandbox execution, associates taint labels with object fields during serialization, and monitors taint flow during deserialization replay. When a sink fails to trigger, the system would generate data-flow-aware feedback rather than relying solely on ambiguous compiler errors. This approach addresses the current limitation where the model struggles with low-level bytecode tasks like `TemplatesImpl`. Existing tools such as FlowDroid [16] and Dytan [33] could serve as foundations for this integration. While taint analysis introduces runtime overhead, this is acceptable for offline payload generation scenarios. The primary challenge lies in

translating low-level taint analysis results into high-level natural language prompts that effectively guide the LLM's repair process.

6. Conclusions

This paper proposes LLM-JDFuzz, an automated fuzzing framework that introduces large language models into Java deserialization gadget-chain payload generation to tackle semantic understanding gaps and the explosion of the construction space. For the first time, it formulates exploit generation as a constrained code generation task driven by deep semantics, going beyond traditional static analysis and guided mutation. LLM-JDFuzz combines a dual-layer autoprompting mechanism, an adaptive object-graph evolution cycle, and a feedback-driven refinement loop with fine-grained error classification to automatically transform formal specifications into working payloads end to end. Evaluations on 34 ysoserial benchmark chains show an overall success rate of 70.6%. Comparative results highlight that LLM-JDFuzz outperforms state-of-the-art tools in cross-language semantic scenarios and generation efficiency.

Author Contributions: Conceptualization, D.S.; Methodology, D.S.; Software, D.S.; Validation, D.S.; Formal analysis, D.S.; Writing—original draft, D.S.; Writing—review & editing, T.T.; Visualization, D.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data supporting the findings of this study are available within the article and from publicly available resources. The benchmark dataset is derived from the ysoserial project, available at <https://github.com/frohoff/ysoserial> (accessed on 11 January 2026).

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

References

- Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Ponde de Oliveira Pinto, H.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
- Java Object Serialization Specification. Available online: <https://docs.oracle.com/en/java/javase/17/docs/specs/serialization/index.html> (accessed on 25 December 2025).
- Zhang, M.; Zhuang, W.; Xia, C.; Ou, X.; Liu, Y.; Xu, G. An Empirical Study of Java Deserialization Vulnerabilities. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Singapore, 14–18 November 2022.
- frohoff. Ysoserial. Available online: <https://github.com/frohoff/ysoserial> (accessed on 11 January 2026).
- Bechler, M. Serianalyzer: Static Bytecode Analyzer for Deserialization Gadget Chains. In Proceedings of the OWASP AppSec Europe, Rome, Italy, 27–30 June 2016.
- Chiang, H.W.; Chou, C.L. JAVA Web System Deserialization Vulnerability Detection Technology. *Commun. CCISA* **2024**, *30*, 45–63.
- JEP 290: Filter Incoming Serialization Data. Available online: <https://openjdk.org/jeps/290> (accessed on 25 December 2025).
- Wang, Q.; Wang, W.; Feng, J.; Lu, J.; Liu, Z.; Yan, C. Understanding and Detecting Java Serialization Vulnerabilities. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021.
- Muñoz, Á.; Mirosh, O. Friday the 13th: JSON Attacks. In Proceedings of the Black Hat USA 2017, Las Vegas, NV, USA, 22–27 July 2017.
- Holzinger, P.; Traber, P.; Schermann, K.; Schwaighofer, H.; Schaller, A. A Study of Java Object Serialization Security. In Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrücken, Germany, 21–24 March 2016.
- Sayar, I.; Bartel, A.; Bodden, E.; Le Traon, Y. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 25. [CrossRef]
- Haken, I. GadgetInspector: A Tool for Finding Deserialization Gadget Chains in Java Applications. Available online: <https://github.com/JackOfMostTrades/gadgetinspector> (accessed on 25 December 2025).
- Chen, B.; Zhang, L.; Huang, X.; Cao, Y.; Lian, K.; Zhang, Y.; Yang, M. Efficient Detection of Java Deserialization Gadget Chains via Bottom-Up Gadget Search and Dataflow-Aided Payload Construction. In Proceedings of the 33rd USENIX Security Symposium, Philadelphia, PA, USA, 14–16 August 2024.

14. Cao, J.; Li, S.; Sun, J.; Zhang, G.; Xu, G.; Chen, J.; Wang, J. GadgetSearch: A Unified Framework for Automatic Detection of Gadget Chains. In Proceedings of the 32nd ACM SIGSOFT International Symposium on the Software Testing and Analysis (ISSTA), Seattle, WA, USA, 17–21 July 2023.
15. Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P.; Sundaresan, V. Soot: A Java Bytecode Optimization Framework. In Proceedings of the 1999 CASCON Conference, Toronto, ON, Canada, 8–11 November 1999.
16. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Outeau, D.; McDaniel, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, UK, 9–11 June 2014.
17. Zalewski, M. American Fuzzy Lop (AFL). Available online: <https://lcamtuf.coredump.cx/afl/> (accessed on 25 December 2025).
18. LLVM Project. libFuzzer: A Library for Coverage-Guided Fuzz Testing. Available online: <https://llvm.org/docs/LibFuzzer.html> (accessed on 25 December 2025).
19. 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 (CCS), Dallas, TX, USA, 30 October–3 November 2017.
20. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, 15–19 October 2018.
21. Cao, S.; He, B.; Sun, X.; Ouyang, Y.; Zhang, C.; Wu, X.; Su, T.; Bo, L.; Li, B.; Ma, C.; et al. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 22–26 May 2023.
22. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code Llama: Open Foundation Models for Code. *arXiv* **2023**, arXiv:2308.12950.
23. 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. *arXiv* **2023**, arXiv:2401.14196.
24. Xia, C.S.; Paltenghi, M.; Tian, J.L.; Pradel, M.; Zhang, L. Fuzz4All: Universal Fuzzing with Large Language Models. In Proceedings of the 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024.
25. Yang, C.; Deng, Y.; Lu, R.; Yao, J.; Liu, J.; Jabbarvand, R.; Zhang, L. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. In Proceedings of the 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024.
26. 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 the Software Testing and Analysis (ISSTA), Seattle, WA, USA, 17–21 July 2023.
27. Meng, N.; Zhao, Y.; Xu, S.; Zhang, Z.; Hu, J.; Lin, J.; Zhang, G.; Zhang, J. ChatAFL: A Protocol-Aware Greybox Fuzzer Empowered by Large Language Models. In Proceedings of the 31st Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 26 February–1 March 2024.
28. 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 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023.
29. Schäfer, M.; Nadi, S.; Eghbali, A.; Tip, F. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Softw. Eng.* **2024**, *50*, 85–105. [[CrossRef](#)]
30. Kang, S.; Yoon, J.; Yoo, S. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In Proceedings of the 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023.
31. Hao, S.; Muñoz, Á.; Mirosh, O.; Shimonosono, T. Object-Oriented Exploitation. In Proceedings of the Black Hat USA 2016, Las Vegas, NV, USA, 30 July–4 August 2016.
32. Bechler, M. marshalsec: Java Unmarshaller Security—Turning Your Data into Code Execution. Available online: <https://github.com/mbechler/marshalsec> (accessed on 25 December 2025).
33. Clause, J.; Li, W.; Orso, A. Dytan: A Generic Dynamic Taint Analysis Framework. In Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA), London, UK, 9–12 July 2007.
34. Shcherbakov, M.; Balliu, M. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In Proceedings of the 28th Network and Distributed System Security (NDSS) Symposium, Virtual, 21–25 February 2021.
35. Cadar, C.; Dunbar, D.; Engler, D.R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, 8–10 December 2008.
36. Koutroumpouchos, N.; Lavdanis, G.; Veroni, E.; Ntantogian, C.; Xenakis, C. ObjectMap: Detecting Insecure Object Deserialization. In Proceedings of the 50th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Valencia, Spain, 29 June–2 July 2020.

37. Padhye, R.; Lemieux, C.; Sen, K.; Papadakis, M.; Le Traon, Y. Semantic Fuzzing with Zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on the Software Testing and Analysis (ISSTA), Beijing, China, 15–19 July 2019.
38. Aschermann, C.; Frassetto, T.; Holz, T.; Jauernig, P.; Sadeghi, A.R.; Teuchert, D. NAUTILUS: Fishing for Deep Bugs with Grammars. In Proceedings of the 26th Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 24–27 February 2019.
39. Blazytko, T.; Aschermann, C.; Schlögel, M.; Abbasi, A.; Schumilo, S.; Wörner, S.; Holz, T. GRIMOIRE: Synthesizing Structure while Fuzzing. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019.
40. Li, R.; Ben Allal, L.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. StarCoder: May the Source Be with You! *Trans. Mach. Learn. Res.* **2023**. [[CrossRef](#)]
41. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In Proceedings of the 11th International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
42. Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.; Zettlemoyer, L.; Lewis, M. InCoder: A Generative Model for Code Infilling and Synthesis. In Proceedings of the 11th International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
43. Wang, Y.; Le, H.; Gotmare, A.; Bui, N.; Li, J.; Hoi, S. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP), Singapore, 6–10 December 2023.
44. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online, 16–20 November 2020.
45. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the 9th International Conference on Learning Representations (ICLR), Virtual, 3–7 May 2021.
46. Hu, Y.; Yin, M.; Xia, C.S.; Wei, L. FuzzGPT: Large Language Models for Edge Case Test Generation. In Proceedings of the 33rd ACM SIGSOFT International Symposium on the Software Testing and Analysis (ISSTA), Vienna, Austria, 16–20 September 2024.
47. Liu, Z.; Gao, J.; Zhan, X.; Yi, L.; Xu, S.; Zhang, S.; Cheng, X.; Chen, G.; Liu, Y. InputBlaster: LLM-Guided Input Generation for GUI Testing. In Proceedings of the 33rd ACM SIGSOFT International Symposium on the Software Testing and Analysis (ISSTA), Vienna, Austria, 16–20 September 2024.
48. 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 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024.
49. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS), New Orleans, LA, USA, 28 November–9 December 2022.
50. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; Narasimhan, K. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS), New Orleans, LA, USA, 10–16 December 2023.
51. Xia, C.S.; Wei, Y.; Zhang, L. Automated Program Repair in the Era of Large Pre-trained Language Models. In Proceedings of the 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023.
52. Chen, X.; Lin, M.; Schärli, N.; Zhou, D. Teaching Large Language Models to Self-Debug. In Proceedings of the 12th International Conference on Learning Representations (ICLR), Vienna, Austria, 7–11 May 2024.
53. Xia, C.S.; Zhang, L. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Singapore, 14–18 November 2022.
54. Jiang, N.; Liu, K.; Lutellier, T.; Tan, L. Impact of Code Language Models on Automated Program Repair. In Proceedings of the 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023.
55. 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 11th International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
56. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the 25th Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 18–21 February 2018.
57. Fu, M.; Tantithamthavorn, C. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In Proceedings of the 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 23–24 May 2022.

58. Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; Phung, D. VulRepair: A T5-Based Automated Software Vulnerability Repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Singapore, 14–18 November 2022.
59. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining Zero-Shot Vulnerability Repair with Large Language Models. In Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 22–26 May 2023.
60. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions. In Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 22–26 May 2022.

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.