GadgetHunter: Region-Based Neuro-symbolic Detection of Java Deserialization Vulnerabilities

KAIXUAN LI, AUMOVIO-NTU Corporate Lab, Nanyang Technological University, Singapore
JIAN ZHANG*, Beihang University, China
CHONG WANG, Nanyang Technological University, Singapore
SEN CHEN, Nankai University, China
ZONG CAO, Imperial Global Singapore, Nanyang Technological University, Singapore
MIN ZHANG, East China Normal University, China
YANG LIU, Nanyang Technological University, Singapore

Java deserialization vulnerabilities (JDVs) enable attackers to execute arbitrary code by crafting malicious serialized objects that trigger sequences of method calls (*gadget chains*) leading to dangerous operations. Existing detection approaches face a fundamental trade-off: static analysis achieves scalability but suffers from high false positives due to infeasible paths and imprecision with dynamic features like reflection; dynamic validation reduces false positives but incurs prohibitive costs and fails to explore deep exploitation chains.

We present GadgetHunter, a neuro-symbolic JDV detector that combines scalable static analysis with targeted LLM reasoning and JDV exploitation-oriented constraint solving. Our approach partitions gadget chains into *regions* based on analyzability: statically resolvable segments are processed via interprocedural taint analysis, while dynamic boundaries are delegated to LLMs for semantic validation. We then extract critical constraints from each gadget and compose them into SMT formulas to determine chain feasibility through satisfiability solving. Evaluation on the `ysoserial` benchmark demonstrates that GadgetHunter reduces false negatives by up to 32% and false positives by 12-85% compared to state-of-the-art tools, while discovering 197 previously unknown gadget chains and rediscovering 4 recent CVEs. Our results show that combining symbolic reasoning with semantic understanding achieves both precision and practical impact in vulnerability detection.

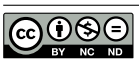
CCS Concepts: • **Security and privacy → Web application security**; **Software security engineering**.

Additional Key Words and Phrases: Neuro-Symbolic AI, SAST, LLMs, Java Deserialization Vulnerabilities

ACM Reference Format:
Kaixuan Li, Jian Zhang, Chong Wang, Sen Chen, Zong Cao, Min Zhang, and Yang Liu. 2026. GadgetHunter: Region-Based Neuro-symbolic Detection of Java Deserialization Vulnerabilities. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE003 (July 2026), 22 pages. https://doi.org/10.1145/3797065

*Corresponding author.

Authors' Contact Information: Kaixuan Li, AUMOVIO-NTU Corporate Lab, Nanyang Technological University, Singapore, Singapore, kaixuan.li@ntu.edu.sg; Jian Zhang, Beihang University, Beijing, China, zhangj3353@buaa.edu.cn; Chong Wang, Nanyang Technological University, Singapore, Singapore, chong.wang@ntu.edu.sg; Sen Chen, Nankai University, Tianjin, China, senchen@nankai.edu.cn; Zong Cao, Imperial Global Singapore, Nanyang Technological University, Singapore, Singapore, z.cao@imperial.ac.uk; Min Zhang, East China Normal University, Shanghai, China, mzhang@sei.ecnu.edu.cn; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg.



ACM 2994-970X/2026/7-ARTFSE003
https://doi.org/10.1145/3797065

1 Introduction

Java deserialization vulnerabilities (JDVs) [20] have repeatedly caused high-impact incidents across widely used middleware and frameworks such as Apache Dubbo, Spring, MyFaces, and WebLogic. By injecting crafted serialized objects, attackers can exploit sequences of method invocations to reach security-sensitive operations (e.g., remote code execution, arbitrary file upload/deletion). Numerous CVEs reported over the past decade highlight both the prevalence and the severity of these vulnerabilities [9, 11]. Consequently, proactive detection is essential to mitigate potential risks before exploitation.

In past years, significant efforts have been devoted to detecting JDVs, leading to two predominant methodological strands: static program analysis only and hybrid analysis that combines static gadget chain mining with fuzzing for validation [6, 7, 9, 10, 20, 31, 36, 38, 47]. Previous static-only approaches [10, 20, 31, 38, 47] typically operate by constructing call graphs via either lightweight Class Hierarchy Analysis (CHA) or, more precisely, whole-program Pointer Analysis (PA) and then identifying source-to-sink data flows within them. However, these methods are fundamentally limited in handling the dynamic semantics inherent to deserialization, such as reflection, dynamic proxies, and runtime polymorphism: CHA-based approaches, while scalable, introduce numerous unreachable (phantom) edges into the call graph, leading to high false positives and hindering efficient exploration [22, 26]. Conversely, PA-based techniques, though more precise, struggle to soundly model the flow of both newly instantiated and attacker-controlled deserialized objects, a critical requirement for this domain [29]. This often results in missing edges and false negatives. Recent works pair static discovery of potential gadget chains with directed fuzzing for dynamic verification [6, 7, 9, 36]. While this represents a pragmatic advance, their approaches remain limited by static imprecision and path explosion, and inadequate dynamic coverage with high overhead. Furthermore, the statically reported gadget chains are dominated by dynamic features (85.3%, see Section 5.3), which existing approaches fail to analyze precisely. Even hybrid techniques provide only partial coverage, since fuzzing struggles to explore these dynamic calls systematically, leading to missed exploitable chains.

Key idea. To address this gap, our key idea is to let static analysis provide the scalable backbone, while delegating unresolved or dynamic boundaries to LLMs for reasoning. Static analysis offers high-recall interprocedural tracking but becomes brittle in the presence of reflection, proxies, or dynamic dispatch. We therefore partition candidate chains into *regions*: segments that can be resolved statically are grouped, while boundaries involving dynamic ambiguity (e.g., reflective calls or polymorphic dispatch) are delegated to the LLM. At these boundaries, the LLM acts as an oracle to conduct semantic reachability analysis. Unlike prior hybrid methods that rely on fuzzing to blindly explore unresolved paths, our delegation is targeted and semantics-aware, applying LLM reasoning only where static analysis fundamentally fails. Finally, reachable chains are subjected to SMT-based path feasibility checking, enabling precise end-to-end exploitability validation.

Our approach. Based on this principle, we propose GadgetHunter for JDV detection. We retain the scalability of static analysis while compensating for its blind spots with LLM-based semantic reasoning. Our approach operates in three stages: ❶ comprehensive taint analysis to discover candidate chains, ❷ region-based semantic reachability analysis using LLMs to validate inter-region transitions, and ❸ JDV exploitation-oriented path feasibility checking using SMT solvers to ensure exploitability. The evaluation on the `ysoserial` and `Gleipner` benchmarks, GadgetHunter reduced false positives and false negatives effectively compared with six SOTA approaches, and further uncovered 197 previously unknown gadget chains, as well as four recently disclosed CVEs, within practical analysis time and financial cost.

Contributions. This paper makes the following three contributions:

- **Formulation and framework.** We introduce the first region-based neuro-symbolic framework for JDV detection. By formulating JDV detection as a feasibility-aware reachability problem, our tool GadgetHunter integrates interprocedural static taint analysis with solver-backed feasibility, while selectively delegating only inter-region semantic uncertainty to LLM reasoning.
- **Evaluation.** We conduct an extensive evaluation on the `ysoserial` and `Gleipner` benchmarks, and real-world Java projects. GadgetHunter consistently outperforms SOTA baselines in recall and precision, reducing FPR by 12–85% and FNR by up to 32%, discovering 197 new gadget chains, with modest computational and financial cost (23.5s and 0.04$/per chain).
- **Implementation and open science.** We implement a prototype of GadgetHunter and release it as open source at https://github.com/MarkLee131/GadgetHunter, facilitating future research on JDV detection.

2 Background

2.1 JDV Detection and Its Challenges

A *gadget chain* is a sequence of method invocations that connects a deserialization source (e.g., `ObjectInputStream.readObject`) to a security-sensitive sink (e.g., `Runtime.exec`). Each *gadget* in the chain is a method whose invocation can be influenced by deserialization and performs state changes useful for exploitation [9]. Unlike conventional vulnerabilities, gadget chains exploit the composition of benign library functionality to achieve critical impact through carefully crafted object graphs. JDV detection faces a fundamental *precision-recall dilemma* that existing approaches cannot resolve effectively. Not only do JDVs usually involve dynamic features, but **on average, 85.3% of gadgets in each statically detected chain involve dynamic features** that defeat conventional static analysis techniques [26, 27].

The fundamental challenge is that gadget chains contain both *statically analyzable segments* and *semantic boundaries* requiring dynamic understanding. Existing approaches either treat all boundaries uniformly (missing the opportunity for targeted reasoning) or rely on imprecise static approximations (suffering from the precision-recall trade-off). What is needed is a principled approach that can distinguish between different types of dynamic boundaries and apply appropriate reasoning techniques to each.

2.2 Related Work

2.2.1 Java Deserialization Vulnerability Detection. Prior JDV detection research mainly falls into two directions: ❶ Static analysis-based approaches [4, 7, 10, 20, 31, 38, 47]. This line of work aims to improve gadget-chain discovery through advanced program analysis. Representative tools include GadgetInspector [20] and Serianalyzer [4], which apply taint analysis on bytecode to trace untrusted data to sensitive sinks. Tabby [10] leverages code property graphs to support customizable and efficient gadget queries. Seneca [38] extends WALA with deserialization-oriented call-graph construction to reduce false negatives. Flash [47] further improves precision and scalability by recovering reflection-related edges involving controllable variables and pruning irrelevant paths within the Tai-e [43] framework. ❷ Hybrid analysis via dynamic validation [6, 9, 36, 42]. To address the inherent limitations of static analysis, these works combine it with dynamic techniques. Crystallizer [42] performs dynamic sink identification and builds gadget graphs for detection. SerHybrid [36] uses directed fuzzing to validate gadget chains extracted from heap access paths. ODDFuzz [6] introduces structure-aware and overriding-guided fuzzing to efficiently generate

proof-of-concept exploits. JDD [9] improves efficiency by reusing recurring gadget fragments across projects.

2.2.2 Studies on Java Deserialization Vulnerabilities. Several empirical studies specifically focus on JDVs [22, 24, 39]. Sayar et al. [39] conducted a large-scale experiment and found that patches are often incomplete or replaced with workaround fixes. Kreyssig et al. introduced Gleipner [22], a *synthetic* benchmark to evaluate JDV detection tools, and compared seven existing tools showing persistent difficulties in gadget-chain detection. More recently, Kreyssig et al. [24] studied the attack surface of gadget chains in Android by analyzing the Android SDK, official libraries, and popular third-party libraries. The results show the JDV's threat in Android is more nuanced. Kreyssig et al. [23] conducted an assessment, showing that small code changes in dependencies can activate or inject Java deserialization gadget chains, revealing dormant chains in 53 projects and underscoring JDVs as a serious supply chain risk.

2.2.3 LLM for Program Analysis. Recent work has explored integrating LLMs into program analysis. Iris [30] leverages LLMs to infer taint specifications and validate CodeQL's results on path traversal and injection vulnerabilities in Java, but does not address reachability or feasibility. LLMDFA [45] applies LLMs to data-flow analysis and bug detection for C programs. BugLens [25] presents a fully automated LLM-based workflow for Linux kernel bugs, using LLMs to reason about path reachability and the feasibility of static warnings. Their approach serves as an ablation baseline for GadgetHunter (denoted SA_LLMC in Section 5.3). In contrast, our work specifically addresses the unique characteristics of JDVs that prior approaches fail to handle effectively. By incorporating semantic reasoning at dynamic boundaries, supplementing static analysis, and validating chains via SMT-based feasibility checking, we achieve both scalability and precision. This design uniquely combines JDV domain knowledge with targeted reasoning techniques to achieve both scalability and precision.

2.3 Motivating Example

To illustrate the limitations of static analysis and the necessity of semantic understanding, we examine the classic Apache CommonsCollections CC1 gadget chain from the `ysoserial` benchmark [17]. As shown in Figure 1, this gadget chain exploits the `TransformedMap` and `AnnotationInvocationHandler` classes to achieve remote code execution through carefully constructed serialized payloads.

When analyzing this chain, static analysis struggles to distinguish true positives from false positives due to dynamic dispatch over-approximation. At `checkSetValue()` (line 15), CHA conservatively resolves the virtual call `valueTransformer.transform(value)` to *all* `Transformer` implementations, including benign ones such as `NOPTransformer`, `StringValueTransformer`, and dozens of other library classes. This directly produces false positives. For instance, consider a candidate chain: `readObject` → `setValue` → `checkSetValue` → `NOPTransformer.transform()`. This path is syntactically reachable under CHA, yet `NOPTransformer` simply returns its input unchanged and can never reach a dangerous sink. Static analysis reports it alongside the genuinely exploitable path through `ChainedTransformer` → `InvokerTransformer`, but cannot tell them apart.

Semantic reasoning is necessary to make this distinction. First, the attacker's serialized payload ensures that the `valueTransformer` field is initialized to a `ChainedTransformer` instance, a runtime constraint invisible to type-based static dispatch. Second, even within `ChainedTransformer`, the `iTransformers` array (line 19) must contain a precise sequence: a `ConstantTransformer` returning `Runtime.class`, followed by `InvokerTransformer` instances whose fields (`iMethodName`, `iParamTypes`, `iArgs`) satisfy strict type-compatibility constraints to ultimately reach the sink

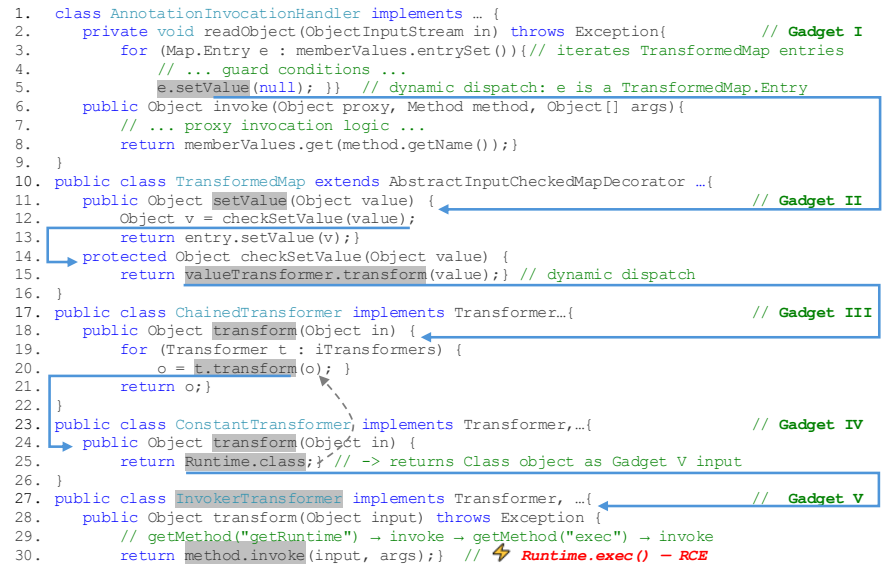


Fig. 1. A simplified motivating example from CommonsCollections (CC1) in the *ysoserial* benchmark.

`Runtime.exec()`. These constraints, including field initialization values, cross-method type propagation, and data-flow dependencies, are beyond the reach of syntactic reachability, causing static tools to report hundreds of candidate chains where most violate implicit semantic constraints (type incompatibility, unsatisfiable field bindings, or logical dependencies that cannot hold simultaneously).

3 Methodology

3.1 Problem Formalization

Given a Java program P with method set $\mathcal{M}$, class set C, and call graph edges $\mathcal{E}$, we define Targets(e) as the statically computed target methods for call edge e, and HasBody(m) as a predicate indicating whether method m has analyzable bytecode. The analysis requires two sets of security-sensitive methods: deserialization sources $\mathcal{S}_{src} \subseteq \mathcal{M}$ (e.g., `readObject`) and security-sensitive sinks $\mathcal{S}_{sink} \subseteq \mathcal{M}$ (e.g., `Runtime.exec`). The goal is to identify feasible gadget chains $\mathcal{G} = \{\pi_1, \pi_2, \ldots, \pi_k\}$, where each $\pi_i = \langle m_0, m_1, \ldots, m_n \rangle$ represents an exploitable call sequence from source $m_0 \in \mathcal{S}_{src}$ to sink $m_n \in \mathcal{S}_{sink}$ that is both **reachable** (admits a valid control-flow path) and **feasible** (satisfies all execution constraints without contradictions).

3.2 Overview

To address these challenges, GadgetHunter integrates static analysis with LLM-guided reasoning in a three-stage hybrid analysis pipeline (Figure 2). Each stage addresses specific limitations while building upon the strengths of the previous components:

Stage 1: Interprocedural Taint Analysis (Section 3.3). Comprehensive gadget chain detection requires systematic exploration of all potential source-to-sink data flows, but existing knowledge bases are incomplete and may miss novel attack vectors. To address this coverage problem, we design an enriched taint analysis approach that expands the search space while maintaining tractability. We first curate a comprehensive knowledge base of sources and sinks from historical

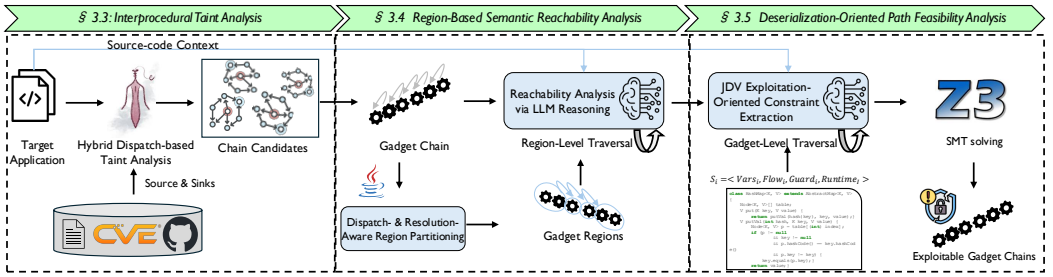


Fig. 2. Overview of GadgetHunter.

CVEs and prior works. Then, building upon prior work [47], we use the demand-driven pointer analysis [47], ensuring that potentially exploitable chains are missed as little as possible during the initial discovery phase while maintaining computational tractability. It yields an over-approximated candidate set $\widehat{\mathcal{G}}$ that captures potential gadget chains but also includes infeasible paths due to dynamic dispatch uncertainty.

Stage 2: Region-Based Semantic Reachability Analysis (Section 3.4). Static analysis suffers from phantom edges where syntactically valid method calls may be semantically unreachable due to type constraints, API contracts, or runtime conditions. To address this over-approximation problem, we design a region-based semantic filtering approach that systematically distinguishes between spurious and feasible call transitions. For each candidate chain $\pi \in \widehat{\mathcal{G}}$, we partition it into regions $\pi = \pi^1 \| \pi^2 \| \cdots \| \pi^k$ where static analysis remains precise, while employing LLM-based semantic reasoning to validate inter-region transitions where static analysis loses precision due to dynamic dispatch or unresolved targets. This produces a filtered set $\widehat{\mathcal{G}}_{\text{reach}} \subseteq \widehat{\mathcal{G}}$ of reachable chains.

Stage 3: Deserialization-Oriented Path Feasibility Analysis (Section 3.5). Even semantically reachable chains may be unexploitable if they violate execution constraints such as null checks, access control, or incompatible data flow requirements. To solve this path feasibility problem, we design a constraint-based validation approach that models execution prerequisites and verifies their satisfiability. For each chain $\pi \in \widehat{\mathcal{G}}_{\text{reach}}$, we extract gadget-level summaries in the form $\langle \text{Vars}, \text{Flow}, \text{Guard}, \text{Runtime} \rangle$ through LLM-guided constraint extraction. Vars captures required variable bindings and type constraints, Flow models data propagation across method calls, Guard represents blocking conditions (e.g., null checks, access control), and Runtime addresses JVM runtime limitations (e.g., reflection accessibility, module constraints). These summaries are composed into a global path formula $\Phi(\pi) = \bigwedge_{i=1}^{n} (\text{Vars}_i \wedge \text{Flow}_i \wedge \text{Guard}_i \wedge \text{Runtime}_i)$ and solved using SMT-based constraint satisfaction. For chains with $\text{SAT}(\Phi(\pi)) = \text{true}$, the SMT solver provides concrete variable assignments that can guide payload construction. Only these validated chains are retained as the final feasible set $\mathcal{G}$.

The three stages work together: Let $\widehat{\mathcal{G}}$ denote the candidate set from Stage 1. Each chain $\pi = \langle m_0, \ldots, m_n \rangle \in \widehat{\mathcal{G}}$ is partitioned into regions $\pi = \pi^1 \| \pi^2 \| \cdots \| \pi^k$ in Stage 2, and inter-region transitions are validated using LLM-based semantic reachability analysis, obtaining $\widehat{\mathcal{G}}_{\text{reach}}$. Finally, Stage 3 extracts path constraints $\Phi(\pi)$ from gadget summaries. The final output is: $\mathcal{G} = \{\pi \in \widehat{\mathcal{G}}_{\text{reach}} \mid \text{SAT}(\Phi(\pi))\}$.

3.3 Interprocedural Taint Analysis

3.3.1 Collecting and Classifying the Source–Sinks. Given a gadget chain $\pi = \langle m_0, m_1, \ldots, m_n \rangle$, the entry method $m_0 \in \mathcal{S}_{src}$ is a *source*, namely a deserialization entry point that ingests attacker-controlled input. The terminal method $m_n \in \mathcal{S}_{sink}$ is a *sink*, namely a security-sensitive API whose invocation may cause side effects such as code execution, file access, or class loading. For each adjacent pair (m_i, m_{i+1}), the execution of m_i must trigger the invocation of m_{i+1}, thereby ensuring control-flow or data-flow across the chain. We denote by $\mathcal{G} = \{\pi \mid m_0 \in \mathcal{S}_{src}, m_n \in \mathcal{S}_{sink}\}$ the set of all feasible gadget chains. To enable systematic mining of $\mathcal{G}$, we first construct a knowledge base of candidate sources $\mathcal{S}_{src}$ and sinks $\mathcal{S}_{sink}$, distilled from historical Java deserialization vulnerabilities.

Following prior work practices [7], we crawled the National Vulnerability Database (NVD) and GitHub Advisory Database to collect Java projects with security advisories. In total, we identified 5,911 Java open-source projects with reported CVEs. To focus on deserialization vulnerabilities, we filtered projects by CWE identifier (*CWE-502: Deserialization of Untrusted Data* [16]), resulting in 343 candidates. For reproducibility and to obtain reliable ground-truth exploitation semantics, we further excluded cases without publicly available patches, since patches are essential for understanding the exact gadget chains involved [28]. After this filtering, we obtained 165 confirmed Java deserialization vulnerabilities, each validated by corresponding patches referenced in the advisories, expanding beyond the 68 CVEs in [7].

We manually reviewed the corresponding patches and exploit descriptions to identify concrete deserialization entry points and dangerous API calls. CVEs without sufficient patch evidence or exploitable call sites were excluded. Two co-authors with expertise in Java deserialization further consolidated the results, cross-checking with prior work [6, 7, 9, 10]. Through this process, we distilled 20 unique sources and 40 unique sinks. Compared to prior works [6, 7, 9, 10], our knowledge base includes **9** additional sources and **15** new sinks, as shown in Table 1.

Table 1. Sources and sinks collected from historical Java deserialization CVEs. Gray items are newly incorporated.

Category	Type	Methods
Sources	JDK Serialization	`readObject`, `readExternal`, `readResolve`, `finalize`
		`readObjectNoData`, `writeReplace`, `validateObject`
	Third-party	`XMLDecoder.readObject`, `XStream.fromXML`
		`ObjectInputValidation.validateObject`
	Object Methods	`equals`, `hashCode`, `compare`, `compareTo`, `toString`
	Collection Methods	`get`, `put`, `call`, `doCall`
	Class Initialization	`<clinit>`
Sinks	RCE	`exec`, `start`, `exit`, `loadLibrary`
		`eval`, `GroovyShell.evaluate`, `clojure.core$eval`
		`Context.evaluateString`, `PyFunction.__call__`, `newTransformer`
	Reflection	`getDeclaredMethod`, `getConstructor`, `getMethod`, `invoke`
		`forName`, `newInstance`, `<init>`, `loadClass`
		`findClass`, `defineClass`, `invokeMethod`
		`invokeStaticMethod`, `invokeConstructor`
	Network Access	`getConnection`, `connect`, `openConnection`, `openStream`
		`InetAddress.getByName`
	JNDI Injection	`lookup`, `getObjectInstance`, `do_lookup`, `c_lookup`
	File Access	`newBufferedReader`, `newBufferedWriter`, `delete`
		`newInputStream`, `newOutputStream`
	Property Access	`setPropertyValue`, `getPropertyValue`, `ValueExpression.getValue`

Table 2. JVM call instructions and our region partitioning policy.

Call Type	JVM Instruction	Partitioned?	Example
STATIC	invokestatic	✗	`Math.max(x,y)`
SPECIAL	invokespecial	✗	`new A()`, `super.f()`
VIRTUAL	invokevirtual	✓	`obj.toString()`
INTERFACE	invokeinterface	✓	`list.add(e)`
DYNAMIC	invokedynamic	✓	`LambdaMetafactory`

3.3.2 Hybrid Dispatch-based Taint Analysis. Built on the collected source and sink methods, we aim to use an interprocedural taint analysis that prioritizes soundness over completeness. Inspired by Flash [47], we adopt their hybrid dispatch technique that strategically balances precision and recall based on variable controllability. Specifically, the hybrid dispatch resolves callees at call sites based on the controllability of their receiver variable: if controllable, a more comprehensive but potentially less precise technique (e.g., CHA, proxy dispatch) is used; otherwise, a more precise

technique (e.g., pointer analysis) is applied. This technique recovers missing call edges by handling reflection involving controllable variables [47], so it aligns with our design principle of prioritizing soundness over completeness in the deserialization vulnerability detection scenario.[1]

3.4 Region-Based Semantic Reachability Analysis

Given the over-approximated candidate set $\widehat{\mathcal{G}}$ from Stage 1's taint analysis, the central challenge is that many inter-region transitions represent *phantom edges*: call sites where static analysis conservatively includes all syntactically valid targets, but runtime semantics severely constrain actual reachability. The problem is that caller-side logic (parameter validation, type narrowing, guard conditions) often determines which callees can actually be invoked, but static analysis cannot capture these semantic dependencies at dynamic dispatch boundaries.

Our key insight is that deserialization gadget chains exhibit favorable properties for semantic validation: ❶ deserialization callbacks (`readObject`, `readResolve`, `equals`, `hashCode`) directly shape receiver and parameter types through object construction logic, ❷ inter-region transitions are typically "thin" semantic bridges (map lookups, comparator calls, simple delegation) rather than complex control flows, and ❸ caller-centric analysis can effectively validate transition feasibility without requiring whole-program reasoning. We therefore design a caller-centric semantic reachability analysis that systematically validates each inter-region transition by analyzing the caller's complete implementation context.

3.4.1 Dispatch- and Resolution-Aware Region Partitioning. Static taint analysis becomes imprecise when encountering dynamically dispatched calls (`invokevirtual`, `invokeinterface`), unresolved invocations (native methods, missing bytecode), or reflective constructs. In such cases, the static analyzer conservatively includes all potential targets according to CHA, yielding spurious paths that violate runtime type constraints, API contracts, or semantic dependencies. For instance, while CHA might connect a `List.get()` call to implementations across dozens of collection classes, the actual runtime type and calling context often restrict the feasible targets to a much smaller set.

Our region partitioning strategy identifies these precision boundaries by analyzing JVM dispatch semantics. Each region π^i contains a maximal sequence of statically resolvable call edges where traditional static analysis remains precise, while region boundaries demarcate transitions where static analysis loses precision due to dynamic dispatch or unresolved targets.

Dispatch-based boundaries. The JVM instruction set encodes invocation dispatch with distinct opcodes for statically bound (`invokestatic`, `invokespecial`) and dynamically bound calls (`invokevirtual`, `invokeinterface`, `invokedynamic`). This distinction directly informs analyzability in static analysis: the instruction's dispatch mechanism determines whether the callee can be precisely resolved at analysis time or must be treated as late bound. We therefore use the JVM call instruction as an operational heuristic for region partitioning, as summarized in Table 2. ❶ Calls via `invokestatic` and `invokespecial` are merged within a region, as their targets are fixed at compile time. ❷ Calls via `invokevirtual`, `invokeinterface`, and `invokedynamic` introduce region boundaries, because their targets depend on runtime type or linkage. This conservative policy prevents dynamic uncertainty from propagating inside a region and confines subsequent reasoning to inter-region links.

Resolution-based boundaries. Opcode alone, however, does not capture all sources of uncertainty. Even a statically-bound call (e.g., `invokestatic`) may be unresolvable or unanalyzable if the callee body is unavailable (library without bytecode), is native, is dynamically generated/loaded, or is

[1]We reuse Flash's core pointer analysis framework but remove its source-sink pair deduplication mechanism. Even when source and sink methods are identical, different attack paths between them can constitute distinct exploitable gadget chains with varying exploitability characteristics.

Algorithm 1: Dispatch- and Resolution-Aware Region Partitioning

Input: Candidate gadget chain $\pi = \langle e_1, \ldots, e_n \rangle \in \widehat{\mathcal{G}}$, where each $e_i = (c_i, m_i, k_i, \rho_i)$ with c_i: call site, m_i: callee method (signature), $k_i \in \{\texttt{STATIC}, \texttt{SPECIAL}, \texttt{VIRTUAL}, \texttt{INTERFACE}, \texttt{DYNAMIC}\}$: JVM call kind (opcode), ρ_i: receiver abstraction; and $\mathsf{UnresolvedInvoke}(e_i)$ indicates target resolvability as in Definition 1.

Output: Partitioned regions $\mathcal{R} = \langle R_1, R_2, \ldots, R_k \rangle$.

1 $\mathcal{R} \leftarrow \emptyset$ // Region sequence
2 $R \leftarrow \emptyset$ // Current region buffer
3 **for** $i \leftarrow 1$ **to** n **do**
4 $e_{\text{curr}} \leftarrow e_i$ // Current call edge
5 **if** $i = 1$ **then**
6 $R \leftarrow \langle e_{\text{curr}} \rangle$ // Start with the first edge
7 **continue**
8 $e_{\text{prev}} \leftarrow e_{i-1}$ // Previous call edge
9 **if** $k_i \in \{\texttt{STATIC}, \texttt{SPECIAL}\}$ ***and*** $\mathsf{UnresolvedInvoke}(e_{\text{curr}})$ **then**
10 **if** $R \neq \emptyset$ **then**
11 $\mathcal{R} \leftarrow \mathcal{R} \circ R$
12 $\mathcal{R} \leftarrow \mathcal{R} \circ \langle e_{\text{curr}} \rangle$ // Treat as singleton
13 $R \leftarrow \emptyset$ // Reset buffer
14 **else if** $k_i \in \{\texttt{VIRTUAL}, \texttt{INTERFACE}, \texttt{DYNAMIC}\}$ ***and*** $(m_i \neq m_{i-1}$ ***or*** $k_i \neq k_{i-1}$ ***or*** $\rho_i \neq \rho_{i-1})$ **then**
15 **if** $R \neq \emptyset$ **then**
16 $\mathcal{R} \leftarrow \mathcal{R} \circ R$ // Commit current region
17 $R \leftarrow \langle e_{\text{curr}} \rangle$ // Start new region
18 **else**
19 $R \leftarrow R \circ e_{\text{curr}}$ // Append to current region
20 **if** $R \neq \emptyset$ **then**
21 $\mathcal{R} \leftarrow \mathcal{R} \circ R$ // Commit final region
22 **return** $\mathcal{R}$

introduced by framework glue that eludes standard call-graph builders. Such invocations should also demarcate region boundaries, regardless of their dispatch type. We therefore complement the opcode-based rule with an *unresolved-invocation* predicate (Definition 1). Formally:

Definition 1 (Unresolved Invocation). *Using our established notation, we define*

$$\mathsf{UnresolvedInvoke}(e) \triangleq \big(\mathsf{Targets}(e) = \emptyset\big) \vee \big(\exists m \in \mathsf{Targets}(e).\ \neg\mathsf{HasBody}(m)\big).$$

An invocation is unresolved if no callee can be determined statically, or if some potential callees lack analyzable code (e.g., native methods, dynamically generated classes, or library stubs).

In practice, unresolved edges surface through framework-specific indicators: Soot may report missing source-position metadata (e.g., `LineNumberTag = -1`), WALA may insert synthetic call edges, and Doop may omit facts in its pointer-analysis database. While our formal rule relies solely on target resolvability, such indicators serve as practical heuristics to detect unresolved edges during implementation and to enforce precise region boundaries.

Algorithm 1 sequentially processes gadget chains to partition them based on dispatch semantics and target resolvability, ensuring that each region contains only statically analyzable transitions while explicitly marking boundaries that require LLM-based reasoning. For statically dispatched calls (`STATIC` or `SPECIAL`), the algorithm evaluates $\mathsf{UnresolvedInvoke}(e)$ (line 9): unresolvable calls terminate the current region and form singleton regions, while resolvable calls merge into the current region. For dynamically dispatched calls (`VIRTUAL`, `INTERFACE`, or `DYNAMIC`), the algorithm

Prompt for Region-Based Semantic Reachability Analysis

You are a Java deserialization vulnerability specialist. Decide if the call from the source method to the target method can execute during normal control-flow.

<TASK>
*Think it step by step and determine if the call *may* execute for **any** feasible program state. Treat variables as unconstrained unless a guard is provably impossible.*
Return:
*REACHABLE or UNREACHABLE(<reason>) if the call is impossible under *all* states.*
<INPUT>
<CODE>{CALL_SITE_SOURCE_CODE}</CODE>
<CALL_EDGE>
{CALLER_SIG, TAINT_LABEL,CALLEE_SIG} , *taint propagation labels: -3 means the return value is tainted, -1 means the `this` is tainted, 0/1/... means the n-th parameter is tainted.*
Note: Constraints come from static analysis and may be incomplete.
</CALL_EDGE>
</INPUT>
<OUTPUT_FORMAT>
REACHABLE (NO reason given)| UNREACHABLE (GIVE a short reason)
</OUTPUT_FORMAT>
<EXAMPLE>{EXAMPLE}</EXAMPLE>
</TASK>

Fig. 3. Prompt template for reachability analysis.

Prompt for Gadget Summary and Constraint Extraction

You are a Java deserialization vulnerability specialist. Analyze the vulnerability-oriented semantics and runtime feasibility analysis.

<TASK>
Think it step by step and extract deserialization-relevant constraints from the given method and call edge.
<INPUT>:
<GADGET_SIG>{FULL_METHOD_SIGNATURE}</GADGET_SIG>
<CODE>{SOURCE_CODE}</CODE>
{CALLER_SIG,TAINT_LABEL,CALLEE_SIG}
</INPUT>
<OUTPUT_FORMAT>
You must output strict JSON with four keys:
- vars: list of { "name":str, "kind": this/argN/field.x/ret, "constraint": constant value or type }
- flow: list of strings "src -> dst", where src,dst ∈ {this,argN,field.x,ret}
- guard: list of blocking predicates, e.g.,
(nonnull(x) | type(x,Cls) | cmp(a=="CONST") | range(l<x<h) | always_throw | early_return)
- runtime: list of JVM-level feasibility constraints, e.g.,
(reflection_accessible(C.f) | serializable(Type)
| module_opens(module,pkg) | class_available(C)
| not_anonymous_class(obj) | cast_compatible(src,dst))
</OUTPUT_FORMAT>
<EXAMPLE>{EXAMPLE}</EXAMPLE>
</TASK>

Fig. 4. Prompt template for JDV constraints extraction.

conservatively inserts region boundaries, grouping consecutive dynamic edges only when both receiver abstraction and invocation type remain unchanged (line 14) to preserve local semantic coherence. Finally, any residual region is committed to $\mathcal{R}$ (line 21), yielding a decomposition where intra-region reasoning relies on precise static analysis while inter-region transitions are delegated to LLM-based semantic inference.

3.4.2 Handling Anonymous and Inner Classes in Gadget Chain Analysis. Anonymous classes are common in Java-based gadget chains, particularly in deserialization scenarios that rely on dynamic method invocation. These classes are compiler-generated and lack standalone source files, often encoded with identifiers such as `dgm$1054`. As a result, their semantics are typically opaque to static analysis, resulting in incomplete representations of the chain. This limitation is especially problematic when the subsequent analysis relies on semantic reasoning to confirm path feasibility and exploitability.

To address this, we reconstruct the surrounding context of such classes by first inferring their host class based on naming conventions and domain knowledge of common libraries. For example, the identifier `dgm$1054` can be associated with `DefaultGroovyMethods` within the Groovy runtime. We then identify the lexical scope in which the anonymous class is instantiated and extract the enclosing method body as a coherent semantic unit. This extracted context serves as an approximation of the anonymous class's behavior, enabling the model to reason about its role in the chain. For instance, when analyzing a chain containing `dgm$1054:invoke`, the recovered context clarifies its function in facilitating reflective invocation, a common mechanism for triggering method calls in deserialization exploits. This mitigates the limitations of static analysis in handling compiler-generated constructs, ensuring that downstream semantic analysis remains informed and effective.

3.4.3 Semantic Reachability Analysis via LLMs. Built on the region-based decomposition $\pi = \pi^1 \| \pi^2 \| \cdots \| \pi^k$, we further conduct semantic reachability analysis via LLM reasoning across inter-region boundaries. Formally, let $\text{Boundaries}(\pi) = \{(m^i_{\text{out}}, m^{i+1}_{\text{in}}) \mid 1 \le i < k\}$ denote all inter-region transitions in chain π, where m^i_{out} is the last method in region π^i and m^{i+1}_{in} is the first method in region π^{i+1}. We define the LLM-based semantic reachability oracle as:

$$\text{LLMReach}(m_{\text{out}}, m_{\text{in}}, \text{context}) : \mathcal{M} \times \mathcal{M} \times \text{Context} \to \{\text{true}, \text{false}\}$$

Our goal is to determine the set of semantically reachable chains:

$$\widehat{\mathcal{G}}_{\text{reach}} = \{\pi \in \widehat{\mathcal{G}} \mid \forall (m_{\text{out}}, m_{\text{in}}) \in \text{Boundaries}(\pi),\ \text{LLMReach}(m_{\text{out}}, m_{\text{in}}, \text{context}) = \text{true}\}$$

We process each candidate chain π in a top-down manner, starting from the deserialization entry point. For each inter-region transition $(m_{\text{out}}, m_{\text{in}}) \in \text{Boundaries}(\pi)$, we construct a semantic context $\text{context}(m_{\text{out}}, m_{\text{in}})$ and leverage LLM reasoning to determine transition feasibility. This approach is motivated by the structural characteristics of deserialization vulnerabilities: most exploitable chains involve transitions through well-defined callback methods (`readObject`, `readResolve`, `equals`, `hashCode`, `compareTo`) where calling context is locally observable, and many inter-region transitions represent lightweight bridging patterns (map lookups, collection access, simple delegation) rather than complex control flows.

Given an inter-region edge $(m_{\text{out}}, m_{\text{in}})$ with calling edge $e = \langle m_{\text{out}}, s, m_{\text{in}} \rangle$ at statement s, we extract semantic context $\text{context}(m_{\text{out}}, m_{\text{in}})$ comprising the caller's complete method body (AST, Javadoc, enclosing class environment), call site information including JVM invoke types (as shown in Table 2), and taint annotations for receiver and parameters. The LLM then analyzes $\text{context}(m_{\text{out}}, m_{\text{in}})$ to examine type narrowing at the call site, explicit guards and early-exit exceptions, aliasing relationships between parameters and fields, literal patterns constraining parameter values, and deserialization lifecycle positioning.

For cases such as transitions involving framework event buses, script engines, or configuration-driven dispatch, the local analysis [12] is insufficient; so we apply conservative estimation following a *worst-case adversary assumption* [14]. When $\text{LLMReach}(m_{\text{out}}, m_{\text{in}}, \text{context})$ is undecidable and the missing prerequisites are typically attacker-controlled in deserialization scenarios (e.g., object field values, map key shapes, comparator instances), we conservatively retain the chain and record these prerequisites as assumptions for the feasibility phase. The whole analysis set an *early-stop mechanism* [35], where it terminates early if any transition is determined to be unreachable; otherwise, it continues until all inter-region edges have been validated.

Prompt Design. As illustrated in Figure 3, our prompt template adopts chain-of-thought (CoT) [46] and in-context learning (ICL) [15] prompting strategies to encourage intermediate reasoning before final judgment, yielding more consistent and interpretable results. Additionally, we follow the best practices to employ a schema-constrained format [2], i.e., in a predefined XML format, making it easier for LLM to parse.

3.5 JDV Exploitation-Oriented Path Feasibility Analysis

While Stage 2 establishes semantic reachability and produces $\widehat{\mathcal{G}}_{\text{reach}} \subseteq \widehat{\mathcal{G}}$, it does not answer the fundamental question for deserialization exploit construction: *does there exist a serializable object that can successfully traverse the entire gadget chain?* Formally, given a semantically reachable chain $\pi = \langle m_0, m_1, \ldots, m_n \rangle \in \widehat{\mathcal{G}}_{\text{reach}}$ where $m_0 \in \mathcal{S}_{src}$ and $m_n \in \mathcal{S}_{sink}$, the question becomes: does there exist a concrete payload object obj such that when obj is deserialized and triggers m_0, the subsequent execution can successfully traverse all intermediate method calls $m_1, \ldots, m_{n-1}$ and reach the sink m_n without constraint violations?

We address this through *deserialization-oriented path feasibility analysis* that models the constraint satisfaction problem inherent in gadget chain exploitation. Given a semantically reachable chain $\pi = \langle m_1, \ldots, m_n \rangle \in \widehat{\mathcal{G}}_{\text{reach}}$, we extract and compose per-gadget constraint summaries using a semantic quadruple $S_i = \langle \text{Vars}_i, \text{Flow}_i, \text{Guard}_i, \text{Runtime}_i \rangle$, where Vars_i captures serializable object requirements for method m_i, Flow_i models value propagation across method boundaries, Guard_i represents runtime conditions that must hold for m_i to execute successfully, and Runtime_i addresses JVM runtime limitations. The chain-level feasibility condition is the conjunction $\Phi(\pi) = \bigwedge_{i=1}^{n} (\text{Vars}_i \wedge \text{Flow}_i \wedge \text{Guard}_i \wedge \text{Runtime}_i)$, and π is *exploitable* iff $\text{SAT}(\Phi(\pi))$ holds, meaning there exists a concrete payload object satisfying all constraints. The final set of exploitable gadget chains is: $\mathcal{G} = \{\pi \in \widehat{\mathcal{G}}_{\text{reach}} \mid \text{SAT}(\Phi(\pi))\}$.

3.5.1 JDV Exploitation-Oriented Constraint Extraction. To distinguish exploitable gadget chains from spurious call graph paths, we perform exploitation-oriented constraint extraction that captures the concrete requirements for successful payload construction. The fundamental challenge in JDV exploitation is that *static call graphs overestimate reachability* [37, 41]: not every path corresponds to a feasible execution with concrete payload objects. Drawing from constraint-based program analysis [1, 12] and symbolic execution principles [5, 21], we identify four orthogonal constraint categories that collectively determine gadget chain exploitability. We model each method m_i in chain π using constraint quadruple $S_i = \langle \text{Vars}_i, \text{Flow}_i, \text{Guard}_i, \text{Runtime}_i \rangle$, where each constraint type addresses a fundamental requirement for successful JDV exploitation:

Variable Constraints (Vars**):** Capture the *object state requirements* inherent to deserialization attacks. Since JDV payloads are serialized objects with specific field values, attackers must control object fields to trigger desired behaviors (e.g., `InvokerTransformer.iMethodName = "exec"` to invoke `Runtime.exec()`). This reflects the core principle that deserialization attacks manipulate object state to subvert intended program logic [18].

Data Flow Constraints (Flow**):** Model *value propagation requirements* across method boundaries, ensuring that data flows from sources (serialized fields) to sinks (dangerous operations) through the gadget chain. This is essential because gadget chains are fundamentally data flow paths where malicious values must propagate correctly [3, 32]. For instance, the `valueTransformer` field must flow to enable `transform()` calls in the CC1 chain.

Path Guard Constraints (Guard**):** Represent *control flow feasibility conditions* that determine whether execution can reach dangerous code paths. These constraints capture runtime conditions (null checks, loop bounds, conditional branches) that must be satisfied for successful exploitation [19, 40]. Guards are crucial because many gadget methods contain defensive checks that can prevent exploitation if not properly handled.

Runtime Feasibility Constraints (Runtime**):** Address *JVM deployment and reflection constraints* that affect exploitation in real environments. These include serialization compatibility, class availability, security manager restrictions, and reflection accessibility [33, 34]. Runtime constraints bridge the gap between theoretical exploitability and practical deployment feasibility.

We employ LLM-based semantic analysis to extract these constraints from method source code and call-site context. As illustrated in Figure 4, our prompt template follows an aforementioned design: (i) task specification (constraint identification), (ii) method source code with taint annotations, and (iii) schema-constrained output requiring constraint categorization with justification. We also adopt CoT and ICL prompting to encourage systematic constraint identification. The extracted constraints are aggregated into a global chain constraint $\Phi(\pi)$ finally.

CC1 chain analysis. Consider the CC1 chain from our motivating example (Figure 1). Table 3 shows the extracted constraints for the CC1 chain. For `AnnotationInvocationHandler.readObject()`, we extract Vars_1 requiring `memberValues` to be a `Map`, Flow_1 capturing the flow from `this.memberValues` to `entrySet()`, and Runtime_1 ensuring the class is serializable. Similarly, `InvokerTransformer.transform()` requires Vars_4 with `iMethodName = "exec"`, Flow_4 from `this.iArgs` to `Method.invoke()`, Guard_4 ensuring non-null input, and Runtime_4 verifying reflection accessibility. Given the global constraint formula $\Phi(\pi)$, we determine chain feasibility through satisfiability solving using SMT solvers. The chain π is *exploitable* iff $\text{SAT}(\Phi(\pi))$ holds, meaning there exists a concrete payload object satisfying all constraints. The final set of exploitable gadget chains is: $\mathcal{G} = \{\pi \in \widehat{\mathcal{G}}_{\text{reach}} \mid \text{SAT}(\Phi(\pi))\}$. When satisfiable, the SMT solver provides a concrete model that assigns values to constraint variables, which can guide payload construction.

Table 3. Constraint extraction for motivating example (Figure 1).

Gadget	Vars	Flow	Guard	Runtime
AnnotationInvocationHandler.readObject	memberValues is Map	this.memberValues → entrySet()	–	serializable(AnnotationInvocationHandler)
TransformedMap.checkSetValue	valueTransformer is Transformer	this.valueTransformer → transform()	–	–
ChainedTransformer.transform	iTransformers is Transformer[]	this.iTransformers[i] → transform()	i < iTransformers.length	–
InvokerTransformer.transform	iMethodName == "exec"	this.iArgs → Method.invoke()	nonnull(input)	reflectionAccessible(method)

3.5.2 Constraint Lowering to SMT & Solving. After extraction, we unify all gadget summaries into the global constraint $\Phi(\pi)$ by conjoining constraint atoms across all Vars_i, Flow_i, Guard_i, and Runtime_i components. We compile $\Phi(\pi)$ to SMT formulae using a lightweight intermediate representation where type constraints become predicates (e.g., $\texttt{instanceof}(x, C)$, $\texttt{accessible}(C)$), aliasing relationships translate to equality assertions, state transitions follow single-assignment semantics, and method resolution uses $\texttt{resolves}(C, m, \sigma)$ predicates.

Based on the obtained $\Phi(\pi)$, we perform a structured lowering to Z3 SMT-LIB format, covering Java type-system fragments (e.g., subtyping, nullability) via axioms. When the generated Z3 script fails to parse or produces errors, we employ an LLM to analyze error messages and suggest repairs. The repair process is bounded with a maximum of 3 iterations to prevent infinite loops. Upon successful constraint solving, if $\text{SAT}(\Phi(\pi))$ holds, Z3 provides a concrete model that assigns values to constraint variables. This model can guide payload construction by providing specific field assignments and object configurations. This separation keeps the extraction flexible while ensuring the final feasibility judgment is auditable and reproducible.

4 Implementation

We implemented GadgetHunter using Java and Python. The taint analysis module is customized from Flash [47], whose backend is Tai-e [43]. Tai-e supports flexible configuration of taint sources and sinks, which we extend to capture deserialization-specific semantics. We further modify Flash components to extract JVM-level information required by the LLM module. For gadget context retrieval, we implement it based on Tree-sitter [44]. For semantic reasoning, we adopt GPT-4.1 as the LLM backend, following its demonstrated effectiveness in recent program analysis tasks [25, 30, 45]. For the SMT solver, we use Z3 [13], and the maximum fixing iterations for Z3 are set to 3.

All experiments were conducted on a machine with an Intel® Xeon® Gold 6248 CPU running at 2.50 GHz, with 188 GB of RAM and Ubuntu 22.04. (64-bit) as the operating system. To reduce randomness, we conducted three repetitions of each experiment and reported the average statistical results. The LLM temperature is set to 0.2, and all experiments are conducted under Java 8.

5 Evaluation

5.1 Setup

5.1.1 Research questions. We evaluate the performance of GadgetHunter and compare it with state-of-the-art tools via addressing four main research questions below:

- **RQ1. Effectiveness.** How effective is GadgetHunter in detecting gadget chains when compared with SOTA?
- **RQ2. Ablation Study.** What impact does each component of GadgetHunter have on the overall performance?
- **RQ3. Efficiency.** How efficient is GadgetHunter in gadget chain detection in terms of time and financial cost?
- **RQ4: Practicality.** How does GadgetHunter perform in analyzing recently real-world Java applications in practice?

Table 4. Gadget chain detection comparison among six SOTA tools and GadgetHunter (our approach). The number in parentheses indicates the confirmed known chains by directed fuzzing.

Application	Known	GadgetInspector		Tabby		Crystallizer		ODDFuzz		JDD		Flash		GadgetHunter	
		All	TP	All	TP	All	TP	All	TP	All	TP	All	TP	All	TP
AspectJWeaver	1	6	0	1	1	85	3	9 (1)	0	16 (6)	6	1,829	5	28	17
BeanShell	1	2	0	3	1	4	1	8 (0)	0	0 (0)	0	632	13	5	5
C3P0	1	2	0	6	4	0	0	13 (1)	1	15 (0)	0	6	6	4	4
Click	1	4	0	1	1	0	0	8 (1)	1	34 (1)	1	355	1	0	0
Clojure	1	12	1	2	1	0	0	184 (1)	1	169 (3)	3	13	1	7	4
CommonsBeantuils	1	2	0	1	1	2	1	8 (1)	1	9 (1)	1	320	1	10	5
CommonsCollections	5	4	1	17	13	80	6	97 (5)	3	196 (53)	53	4,975	26	44	38
CommonsCollections4	2	4	0	18	13	4	4	112 (2)	2	209 (26)	26	1,829	17	51	34
FileUpload	1	3	0	2	2	0	0	8 (0)	0	1 (1)	1	2	2	2	2
Groovy	1	4	0	2	0	7	1	13 (0)	0	580 (7)	7	489	0	11	8
Hibernate	2	3	0	4	4	0	0	8 (2)	2	0 (0)	0	0	0	4	2
JavassistWeld	1	2	0	3	1	0	0	8 (0)	0	2 (1)	1	7	1	3	2
JbossInterceptors	1	2	0	3	1	0	0	8 (0)	0	2 (1)	1	7	1	6	4
JDK	4	5	0	0	0	0	0	9 (1)	1	16 (8)	8	0	0	0	0
JSON	1	2	0	0	0	0	0	9 (0)	0	0 (0)	0	91	0	2	0
Jython	1	42	1	2	0	0	0	32 (1)	0	0 (0)	0	56	0	34	19
MozillaRhino	2	3	0	1	1	6	0	7 (2)	2	10 (1)	1	679	0	11	10
Myfaces	2	2	0	1	1	0	0	7 (0)	0	283 (3)	3	131	8	69	27
ROME	1	2	0	2	2	8	2	5 (1)	1	853 (5)	5	330	10	36	13
Spring	2	2	0	2	0	0	0	10 (0)	0	188 (0)	0	633	2	0	0
Vaadin	1	5	0	1	1	25	2	13 (1)	1	753 (39)	39	182	5	13	10
Wicket	1	3	0	2	2	0	0	7 (0)	0	1 (1)	1	2	2	2	2
Total	34	116	3	74	50	221	20	583 (20)	16	3,337 (157)	157	12,568	101	**362**	**221**

5.1.2 Baselines. We define two criteria for baseline selection: ❶ Tools must represent state-of-the-art static or hybrid approaches for gadget-chain detection. ❷ Tools must be publicly available or have reproducible evaluation results.

Following these criteria, we include six representative tools in our evaluation: (i) Static approaches: GadgetInspector, Tabby [10], and Flash [47]; (ii) Hybrid approaches: Crystallizer [42], ODDFuzz [6], and JDD [9]. Note that ODDFuzz is not open-sourced, so we reuse the evaluation results reported in its paper; JDD does not release its fuzzing module, so we contacted and requested their raw evaluation results. Other tools are excluded for the following reasons. ❶ Serianalyzer [4] and SerHybrid [36] are omitted due to consistently inferior performance compared with Tabby, JDD, and Flash in prior evaluations. ❷ GCMiner [7] and SerdeSniffer [31] are excluded because their source code is unavailable and lacks reproducible results, making the reproducibility non-trivial.

5.1.3 Benchmarks. To comprehensively evaluate both real-world effectiveness and challenge-specific robustness, we adopt a combination of established and controlled benchmarks. (1) Following prior work [4, 6, 7, 9, 10, 47], we first evaluate on the widely used `ysoserial` benchmark [17], which comprises 34 established gadget chains extracted from real-world Java applications. (2) We additionally experiment on the `Gleipner` benchmark [22], a synthetic benchmark designed to isolate key challenges in JDV detection, including *Depth* (20 cases with increasing call-chain depth), *Polymorphism* (20 cases stressing dynamic dispatch), and *Multipath* (10 cases with branching control flow). While `ysoserial` reflects practical exploitability, `Gleipner` enables fine-grained analysis of specific technical limitations. Finally, for RQ4, we evaluate GadgetHunter on four recent CVEs to assess its end-to-end practicality.

5.2 RQ1: Effectiveness Analysis

5.2.1 Evaluation on the ysoserial benchmark. For tools with dynamic validation modules (Crystallizer, ODDFuzz, and JDD), Crystallizer's design differs: it uses static analysis to identify candidate chains and dynamic analysis to generate concrete payloads, but only outputs the gadget chains confirmed by its dynamic module. By contrast, ODDFuzz and JDD expose both static and dynamically validated results. Therefore, the TP values correspond to results verified by their dynamic phase,

with the numbers in parentheses of ODDFuzz and JDD indicating the confirmed chains validated through directed fuzzing.

Overall, the results reveal clear differences consistent with each tool's design. GadgetInspector produces limited coverage and few exploitable chains due to its reliance on classical taint-based static analysis. Tabby detects a large number of chains with its CPG-based queries but suffers from high false positives. Crystallizer achieves only modest coverage, reflecting its reliance on limited sink modeling and dynamic probing. ODDFuzz benefits from directed fuzzing to validate exploits, but its coverage is bounded by fuzzing budgets. JDD reuses historical gadget fragments, yielding significantly more detections, although its static analysis module leads to severe false-positives. Flash expands call-graph edges through deserialization-guided heuristics, covering many candidates, yet its pruning strategy is not fine-grained enough to filter infeasible paths.

In contrast, GadgetHunter consistently outperforms all baselines, detecting 21–32 chains across targets and confirming 8–29 as exploitable. Crucially, it reduces false positives by 12–85% compared with JDD while maintaining high coverage, highlighting the benefit of combining static scalability with LLM-guided reasoning and SMT-based feasibility validation.

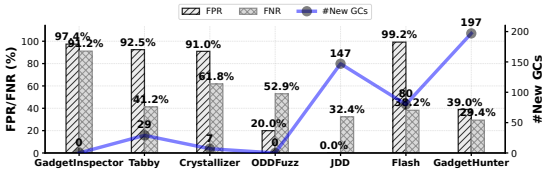


Fig. 5. False positive/negative analysis of GadgetHunter with baselines.

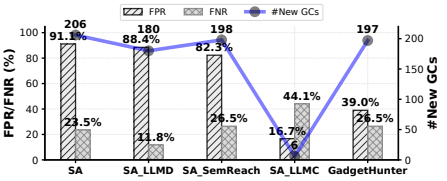


Fig. 6. Contribution of individual components within GadgetHunter.

False Positive and False Negative Analysis. To quantify precision and recall, we compute false positive rate (FPR) and false negative rate (FNR) following prior work [9, 47]: FPR = $\frac{\#\text{false positives}}{\#\text{all detected chains}}$, FNR = $\frac{\#\text{missed known chains}}{\#\text{total known chains}}$. Here, false positives are chains reported but not exploitable, while false negatives are known gadget chains in the benchmark missed by the tool. Figure 5 reports the FPRs and FNRs of each tool together with the number of "new" gadget chains beyond the benchmark. GadgetInspector, Tabby, and Flash exhibit extremely high FPR (above 90%), consistent with their over-inclusive expansions. Crystallizer suffers from high FNR (over 60%), failing to recover many known chains. ODDFuzz and JDD improve precision by validating chains through fuzzing, but remain limited by the inherent difficulty of exploring the constraints. ODDFuzz still reports 20% false positives, since its directed fuzzing only considers class hierarchies inferred from gadget chains and often generates incorrect object layouts. JDD eliminates false positives by encoding "if" conditions into fuzzing to guide object construction, but its reliance on fuzzing still leaves many feasible chains unexplored. This underscores the need for semantic constraints beyond fuzzing.

GadgetHunter achieves a balanced combined error rate: attains an FPR at 38.95%, FNR at 29.41%, and identifies 197 new chains (50 more than JDD), demonstrating improved precision as well as better generalization across diverse projects. The superior performance of GadgetHunter is attributed to its feasibility-aware design. Static-only approaches (e.g., GadgetInspector, Tabby, Flash) either *under-approximate* by missing reflection or dynamic-dispatch edges, leading to false negatives, or *over-approximate* by including all possible edges, resulting in excessive false positives. Hybrid approaches with dynamic validation (e.g., Crystallizer, ODDFuzz, JDD) provide concrete confirmation, but their effectiveness is constrained by fuzzing budgets and the difficulty of generating valid payloads, which limits coverage. In contrast, GadgetHunter partitions candidate gadget chains into regions: intra-region reasoning is handled by static taint analysis for scalability, while inter-region ambiguity

Table 5. Evaluation on the `Gleipner` synthetic benchmark.

Tool	Depth (20 TP)	Polymorphism (20 TP)	Multipath (10 TP)
GadgetInspector	20/20	20/20	3/10
Tabby	20/20	20/20	10/10
Crystallizer	3/20	0/20	10/10
ODDFuzz	–	–	–
Flash	5/20	7/20	1/10
JDD	5/20	0/20	10/10
GadgetHunter	14/20	13/20	1/10

is resolved by LLM-based semantic inference. Surviving chains are abstracted into JDV exploitation-oriented constraints and discharged as SMT constraints, which precisely eliminate infeasible paths. This design enables GadgetHunter to (i) retain high recall by preserving potential dynamic edges, (ii) reduce false positives through semantic pruning and SMT-backed feasibility checking, and (iii) generalize beyond the benchmark by identifying 197 new gadget chains.

5.2.2 Evaluation on the Gleipner benchmark. Table 5 summarizes the results. Early static analysis tools such as GadgetInspector and Tabby demonstrate strong effectiveness on the Depth and Polymorphism categories, achieving full coverage (20/20) on both, reflecting their effectiveness at handling deep call chains and polymorphic dispatch patterns. Tabby further achieves perfect Multipath coverage (10/10) due to its CPG-based representation that preserves fine-grained call-site information. Flash, which serves as the static analysis backbone of GadgetHunter, achieves 5/20 on Depth and 7/20 on Polymorphism under its default configuration, as its time-bounded path collection does not fully explore the deep and polymorphic chains in this benchmark. As shown below, GadgetHunter improves upon Flash's coverage (14/20 and 13/20, respectively) by extending the path collection budget and applying semantic reasoning at dynamic boundaries. GadgetHunter detects 14/20 Depth chains and 13/20 Polymorphism chains, but only 1/10 Multipath chains. We further analyze the root causes as follows.

Regarding the Depth and Polymorphism, the six missing Depth cases and seven missing Polymorphism cases are absent from GadgetHunter's call graph due to its time-bounded path collection and non-deterministic DFS traversal order inherited from Flash [47]. Experiments with extended collection time recovered additional chains (16/20 and 15/20, respectively), confirming that these gaps stem from configuration trade-offs rather than fundamental limitations of the approach.

Regarding the Multipath, the low score (1/10) reflects a structural limitation rooted in GadgetHunter's method-summary-based taint analysis. The Multipath test case in Gleipner uses a `switch`-based dispatch to invoke 10 distinct `LinkGadgetN.linkMethod()` intermediaries, each delegating to `SinkGadget.linkMethod()`. During interprocedural analysis, GadgetHunter computes a method summary for each `LinkGadgetN.linkMethod()` that collapses it to a direct transfer to `SinkGadget.linkMethod()`. Consequently, all 10 paths reduce to one identical chain in GadgetHunter's output. This reveals an inherent trade-off within GadgetHunter: method-summary-based analysis sacrifices path sensitivity for scalability since such structurally similar chains typically represent the same underlying vulnerability in practical security auditing.

Answer to RQ1: *Compared with the six baselines, GadgetHunter demonstrated superior effectiveness, achieving higher coverage of known gadget chains in* `ysoserial`*, substantially lower false positive and false negative rates (reducing FPR by 12–85% and FNR by up to 32%), and discovering 197 new chains beyond the benchmark. On the* `Gleipner` *benchmark, GadgetHunter covers 14/20 Depth and 13/20 Polymorphism chains; its Multipath gap stems from method-summary-based design trade-offs.*

5.3 RQ2: Ablation Analysis

To assess the contribution of each component in GadgetHunter, we construct ablation variants that progressively extend a static analysis baseline. Specifically, we consider four variants as follows:

- **SA** (Static Analysis Only): Performs only interprocedural taint propagation. This variant serves as the pure static baseline.
- **SA_SemReach** (Static Analysis + Semantic Reachability): removes the feasibility analysis.
- **SA_LLMD** (Static Analysis + LLM Direct): Builds on SA by letting an LLM directly validate candidate chains in a black-box manner.
- **SA_LLMC** (Static Analysis + LLM Constraint): Inspired by BugLens [25], replaces the feasibility analysis by using LLMs conduct end-to-end constraint extraction and analysis to decide feasibility.

5.3.1 Results. Figure 6 reports FPR, FNR, and the number of new gadget chains for each ablation variant. We discuss them in terms of component contribution. SA shows pure taint propagation yields the largest number of candidates but only 8.9% (237/2, 662) valid ones, resulting in FPR 91.13%. This confirms that static analysis ensures high coverage for exploitable chains, but suffers from massive path explosion. Compared to SA, SA_SemReach's performance reflects that our *Reachability Analysis* module cuts candidates almost by half, reducing FPR moderately (83.57%) while keeping FNR roughly stable. This shows that our *Reachability Analysis* module is effective in removing semantically unreachable chains.

SA_LLMD achieves the best FNR (11.76%) among all baselines by confirming more exact exploitable chains. However, it caused numerous false positives (FPR = 88.44%), as the LLM tends to over-accept semantically plausible yet infeasible chains. Additionally, new gadget-chain findings remain high at 180 ones, indicating SA_LLMD's generalization potential but at the cost of accuracy. The effectiveness of SA_LLMC suggests that when LLMs produce constraints and decide feasibility themselves, FPR drops sharply to 16.67%, but false negatives increase sharply (FNR 44.12%). This variant illustrates the risk of over-pruning: without solver-backed consistency checks, brittle constraints filter out not only false alarms but also many real chains. Compared with these variants, Figure 6 shows that GadgetHunter balances both sides: FPR is reduced to 38.95%, far lower than SA or SA_LLMD, with its FNR preserved at 26.47%, on par with SA_SemReach. Compared to SA_LLMD, the feasibility module prevents over-acceptance, thereby the FPR decreases from 88.44% to 38.95%. Compared to SA_LLMC, solver-backed reasoning restores recall, thereby the FNR decreases from 44.12%to 26.47%, with newly detected gadget chains increasing to 197.

5.3.2 Region partitioning analysis. We further examined the distribution of static and dynamic regions across projects (Figure 7). Across all 2,662 mined gadget chains on the `ysoserial` benchmark, we observed an average of 10.3 gadgets and 9.1 regions per chain, with only 14.7% of regions being fully resolved by static analysis. This distribution indicates that region partitioning is not an artificial decomposition: the number of regions strongly correlates with gadget counts, reflecting inherent program complexity. Overall, this validates the role of region partitioning in delegating hard cases to LLM reasoning while keeping intra-region reasoning purely static and efficient.

Answer to RQ2: *Each module in GadgetHunter plays a distinct role: static analysis ensures minimizing false negatives, the reachability analysis prunes trivially invalid chains, and path-feasibility analysis provides fine-grained feasibility checking. Together, these components achieve a principled trade-off between recall and precision, which neither pure static analysis nor direct LLM validation can achieve alone.*

Table 7. Efficiency and financial cost of GadgetHunter on the `ysoserial` benchmark.

Application	#Chains	Duration (s)	Avg. duration (s)	Cost ($)	Avg. cost ($)
AspectJweaver	162	4,047.86	24.99	5.65	0.03
BeanShell	185	2,895.66	15.65	6.61	0.04
C3P0	6	276.25	46.04	0.28	0.05
Click	9	34.31	3.81	0.07	0.01
Clojure	13	628.73	62.87	4.14	0.41
CommonsBeantuils	101	1,292.38	12.8	2.71	0.03
CommonsCollections	188	8,531.84	45.38	8.43	0.04
CommonsCollections4	276	6,678.67	24.2	11.96	0.04
FileUpload	2	79.05	39.52	0.11	0.05
Groovy	73	1,599.34	21.91	2.79	0.04
Hibernate	6	297.71	49.62	0.44	0.07
JavassistWeld	4	282.71	70.68	0.34	0.09
JbossInterceptors	8	725.59	48.37	0.73	0.05
JDK	19	725.96	38.21	1.12	0.06
JSON	294	2,813.19	9.57	6.07	0.02
Jython	51	7,751.75	152	4.26	0.08
MozillaRhino	471	7,883.89	16.74	13.16	0.03
MyFaces	124	5,623.88	45.35	6.76	0.05
Rome	398	5,945.58	14.94	12.34	0.03
Spring	11	85.00	7.73	0.17	0.02
Vaadin	259	4,317.78	16.67	6.95	0.03
Wicket	2	214.59	107.29	0.15	0.08
Total/Avg.	**2,666**	**62,731.72**	**23.53**	**95.24**	**0.04**

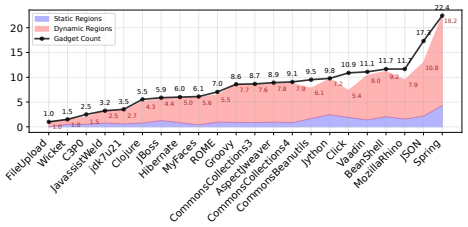


Fig. 7. The average number of regions (gadgets) per chain across applications in the `ysoserial` benchmark.

Table 6. Practicality evaluation on recent JDVs.

CVE ID	Application	Stars	#Known	Detected (#Chains)
CVE-2024-23636	Sofa-RPC	41k	1	✓(1)
CVE-2025-24813	Tomcat	7.9k	1	✓(13)
CVE-2022-1471	SnakeYaml	–	2	✓(1)
CVE-2023-23638	Dubbo	41.3k	4	✓(3)

5.4 RQ3: Efficiency and Cost Analysis

Table 7 reports the efficiency and financial cost of running GadgetHunter on the *ysoserial* benchmark. For each application, we record the total number of candidate chains processed (#Chains), total analysis time (Duration), average per-chain time, total cost, and average per-chain cost. Overall, GadgetHunter analyzes 2,662 chains in about 62.7k seconds, with an average of 23.53s per chain and an average cost of $0.04 per chain.

The results reveal substantial variation across projects. Smaller-scale projects such as *Click* and *Spring* complete within seconds per chain, while reflection-heavy frameworks like *Clojure* and *Jython* require significantly longer times (up to 152s per chain). This discrepancy aligns with the proportion of dynamic regions: projects dominated by reflection or dynamic dispatch involve more ambiguous edges, triggering more LLM queries and larger constraint systems for SMT solving. In contrast, projects with mostly static gadget flows (e.g., *FileUpload*, *Wicket*) remain lightweight.

Despite relying on GPT-4.1, the financial overhead is modest. The total expenditure across the entire benchmark is $95.24, averaging only $0.04 per chain. Two design choices drive this efficiency: (i) *region partitioning*, which restricts LLM calls to a small set of ambiguous edges rather than all possible paths, and (ii) *summary-based constraint extraction*, which reduces token usage by converting complex code into compact ⟨var, flow, guard⟩ formulas before SMT solving. As a result, the overall cost remains practical for large-scale evaluation. These findings show that GadgetHunter scales to thousands of gadget chains with manageable overhead. The efficiency

differences across projects further support the need for region partitioning: it aligns workload with inherent program complexity, preventing uncontrolled blowup in reflection-heavy code. Meanwhile, the per-chain cost remains sufficiently low to enable practical deployment in real-world audits.

Answer to RQ3: *GadgetHunter demonstrates practical efficiency with modest cost: processing 2,662 gadget chains required only $95.24 in total (about $0.04 per chain). Runtime variation across projects aligns with the proportion of dynamic regions, underscoring the role of region partitioning in limiting LLM invocations. By summarizing gadgets into compact constraints before SMT solving, GadgetHunter controls both execution time and API expenditure, supporting scalable use in real-world audits.*

5.5 RQ4: Practicality Analysis

To evaluate practicality, we applied GadgetHunter to four recently disclosed deserialization CVEs and three popular open-source applications without previously reported JDVs. As shown in Table 6, GadgetHunter successfully rediscovered the known gadget chains in all four CVEs and additionally surfaced several variants.

Across the four CVEs, GadgetHunter detected 18 exploitable chains overall. Counting against the ground truth, we matched 6 known chains and missed 2 (one in SnakeYaml and one in Dubbo), yielding an FNR of 25% (2/8). Especially, for *Tomcat* (CVE-2025-24813), although the advisory documents only a single chain, GadgetHunter uncovered 12 exploitable variants. Manual analysis confirms that this RCE leverages `CommonsCollections 3.2.1`, and GadgetHunter systematically enumerates feasible edge substitutions within the same chain schema, thereby producing multiple valid variants beyond the CVE's minimal record.

Answer to RQ4: *GadgetHunter successfully rediscovers 6 out of 8 known gadget chains across four recent CVEs (detecting 18 exploitable chains in total, with FNR = 25%). This confirms its practicality in real-world audits, while the Tomcat case illustrates its ability to generalize across semantically equivalent chain variants.*

6 Discussion

6.1 Limitation and Future Work

First, our current evaluation focuses on theoretical exploitability. The feasibility checks conducted by Z3 constraint solving ensure that the discovered chains are semantically valid, but we have not yet generated end-to-end proof-of-concept (PoC) exploits to confirm practical exploitability. A complete PoC generator, guided by the symbolic constraints obtained in GadgetHunter, would be an important extension to bridge the gap between theoretical feasibility and real-world exploitation.

Second, the integration between static analysis and LLMs in GadgetHunter is essentially performed as a post-processing step. While this design already improves precision by validating candidate chains, it leaves room for further improvement. In particular, embedding LLM reasoning directly into static analysis tasks such as pointer analysis or call-graph construction could significantly reduce false negatives. Such interleaving approaches [8] where LLM inference guides the resolution of hard-to-analyze language features within the analysis itself. Adopting this direction could enhance the soundness and strengthen the overall robustness of JDV detection.

6.2 Threats to Validity

6.2.1 External threats. ❶ **Generalizability.** The generalizability of GadgetHunter may be affected by dataset selection. To mitigate this, we adopted `ysoserial`, a widely used JDV benchmark, and compared GadgetHunter against six SOTA baselines that cover both static-only and hybrid static–dynamic approaches. Additionally, we extended the evaluation (Section 5.5: RQ4) to practical

settings by including four recently disclosed CVEs. GadgetHunter successfully rediscovered known gadget chains and revealed new exploitable variants. ❷ **Data leakage.** Another potential concern is that LLM training data may contain prior knowledge of the vulnerabilities, which could artificially inflate performance. We mitigate it in two ways. First, ablation studies showed that direct LLM reasoning (*SA_LLMD*) is substantially less effective than our integrated framework, indicating that the improvements cannot be attributed to memorization. Second, we evaluated two CVEs (i.e., CVE-2024-23636 and CVE-2025-24813) disclosed after the presumed cutoff date of LLM training data, which further reduces the likelihood that data leakage explains our results.

6.2.2 Internal threats. **Overhead.** The use of both LLMs and SMT solving introduces additional financial and computational costs compared to traditional methods. To mitigate this, we measured the average cost across all experiments. On the `ysoserial` benchmark, processing 2,662 gadget chains required only $95.24 in total, which corresponds to about $0.04 per chain and an average runtime of 23.53 seconds per chain. These results demonstrate that GadgetHunter achieves practical efficiency with modest cost. Given ongoing improvements in solver performance and model efficiency, we expect this overhead to decrease further over time.

7 Conclusion

This paper introduces GadgetHunter, a semantic-enhanced framework for detecting Java deserialization gadget chains. By integrating scalable static taint analysis with LLM-guided reachability reasoning and summary-based path feasibility checking, GadgetHunter addresses the imprecision of static analysis under dynamic features and the lack of semantic guidance in feasibility validation. Our evaluation demonstrates that GadgetHunter achieves notable improvements in accuracy while maintaining practical financial and time costs, highlighting the potential of combining static analysis with LLM semantic reasoning for advancing vulnerability detection in modern Java ecosystems.

Acknowledgments

This research/project is supported by A*STAR under the RIE2025 Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) Funding Initiative (Award: I2501E0045), as well as cash and in-kind contribution from the industry partner(s). This research is also part of the IN-CYPHER programme and supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] Alexander Aiken. 1999. *Introduction to set constraint-based program analysis.* Vol. 35. Elsevier. 79–111 pages.
[2] Anthropic. [n. d.]. Use XML Tags to Structure Your Prompts. https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/use-xml-tags [Online; accessed 2025-09-01].
[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.
[4] Moritz Bechler. [n. d.]. A static byte code analyzer for Java deserialization gadget research. https://github.com/mbechler/serianalyzer [Online; accessed 2025-09-09].
[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[6] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2726–2743.
[7] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 397–409.

[8] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Copenhagen, Denmark) *(SOAP 2024)*. Association for Computing Machinery, New York, NY, USA, 9–17.

[9] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. 2024. Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 3961–3978.

[10] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. 2023. Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 179–192.

[11] Common Weakness Enumeration. [n. d.]. 2024 CWE Top 10 KEV Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html [Online; accessed 2025-09-09].

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[14] Danny Dolev and Andrew Yao. 2003. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (2003), 198–208.

[15] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[16] Common Weakness Enumeration. [n. d.]. CWE - CWE-502: Deserialization of Untrusted Data (4.18). https://cwe.mitre.org/data/definitions/502.html [Online; accessed 2025-09-09].

[17] frohoff. 2025. *Ysoserial*. (Accessed on 31/01/2025).

[18] Chris Frohoff and Gabriel Lawrence. 2015. Marshalling pickles: how deserializing objects can ruin your day. In *AppSecCali 2015*. OWASP.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[20] Ian Haken. 2018. Automated discovery of deserialization gadget chains. *Proceedings of the Black Hat USA* 48 (2018).

[21] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[22] Bruno Kreyssig and Alexandre Bartel. 2025. Gleipner: A Benchmark for Gadget Chain Detection in Java Deserialization Vulnerabilities. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1–21.

[23] Bruno Kreyssig, Sabine Houy, Timothée Riom, and Alexandre Bartel. 2025. Sleeping Giants - Activating Dormant Java Deserialization Gadget Chains through Stealthy Code Changes. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 2668–2682.

[24] Bruno Kreyssig, Timothée Riom, Sabine Houy, Alexandre Bartel, and Patrick McDaniel. 2025. Deserialization Gadget Chains are not a Pathological Problem in Android: an In-Depth Study of Java Gadget Chains in AOSP. *arXiv preprint arXiv:2502.08447* (2025).

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

[26] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and evaluation on static application security testing (SAST) tools for java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 921–933.

[27] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1447–1470.

[28] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. 2024. PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Pages–590.

[29] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.

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

[31] Xinrong Liu, He Wang, Meng Xu, and Yuqing Zhang. 2024. SerdeSniffer: Enhancing Java Deserialization Vulnerability Detection with Function Summaries. In *European Symposium on Research in Computer Security*. Springer, 174–193.

[32] V Benjamin Livshits and Monica S Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. *USENIX Security Symposium* 14 (2005), 271–286.
[33] Oracle Corporation. 2021. Java Object Serialization Specification. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html [Online; accessed 2025-09-11].
[34] OWASP Foundation. 2017. Deserialization Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html [Online; accessed 2025-09-11].
[35] Lutz Prechelt. 2002. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.
[36] Shawn Rasheed and Jens Dietrich. 2020. A hybrid analysis to detect java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1209–1213.
[37] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. *ACM Transactions on Programming Languages and Systems* 17, 4 (1995), 526–566.
[38] Joanna C. S. Santos, Mehdi Mirakhorli, and Ali Shokri. 2024. Seneca: Taint-Based Call Graph Construction for Java Object Deserialization. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 1125–1153.
[39] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. 2023. An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–45.
[40] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 263–272.
[41] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69.
[42] Prashast Srivastava, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. 2023. Crystallizer: A hybrid path analysis framework to aid in uncovering deserialization vulnerabilities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1586–1597.
[43] Tian Tan and Yue Li. 2023. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1093–1105.
[44] Tree-sitter. 2025. Tree-sitter: A parser generator tool. https://tree-sitter.github.io/tree-sitter/. [Online; accessed 2025-09-11].
[45] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems* 37 (2024), 131545–131574.
[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[47] Yiheng Zhang, Ming Wen, Shunjie Liu, Dongjie He, and Hai. Jin. 2025. Precise and Effective Gadget Chain Mining through Deserialization Guided Call Graph Construction. In *Proceedings of the 34th Conference on USENIX Security Symposium*. USENIX Association, USA.

Received 2025-09-12; accepted 2025-12-22