

Machine Learning for Automated Flaky Test Diagnosis in JVMs

Rishabh Singh
rishabhsinghe@gmail.com
Independent Researcher

Abstract

This paper explores the application of machine learning to address the pervasive issue of flaky tests in software development within Java Virtual Machines (JVMs). Existing flaky-test detection approaches primarily focus on identifying whether a test is flaky but do not provide diagnostic information about why flakiness occurs. We present a framework that integrates statistical and machine learning techniques with dynamic taint analysis to automatically identify and tag candidate root causes of nondeterministic test failures. By tracking data flow using bytecode modifications, this approach facilitates automated debugging by providing evidence for candidate sources of nondeterminism. The framework is evaluated in terms of classification performance, diagnostic localization, and runtime overhead. The results indicate that the hybrid approach can effectively identify flaky tests and localize potential sources of nondeterminism in JVM-based software systems.

Keywords

• Machine Learning • Flaky Tests • Java Virtual Machine (JVM) • Dynamic Taint Analysis • Bytecode Modification • Automated Debugging • Test Dependency Analysis

1. Introduction and Background

Flaky tests, which exhibit nondeterministic behavior, passing in some executions and failing in others without code changes, are a persistent issue in modern software testing [1]. They undermine developer trust, complicate debugging, and increase maintenance costs. A comprehensive survey by Parry et al. [2] characterized common causes, detection techniques, and open challenges in flaky test research.

Flakiness typically arises from hidden dependencies, concurrency issues, network delays, and shared global states [3]. Manual debugging does not scale to the volume of tests in continuous integration (CI) pipelines, motivating automated approaches. Zheng et al. [4] traced the evolution from heuristic-based methods to machine learning (ML) approaches for flaky test detection.

Dynamic taint analysis (DTA) tracks information flow by marking inputs as “tainted” and propagating these marks through execution, as implemented in frameworks like Phosphor [5]. Recent advances such as AirTaint [6] have made DTA faster and easier to deploy. However, DTA alone cannot capture stochastic execution patterns or complex test order effects. ML techniques complement DTA by uncovering statistical correlations and classifying flaky versus stable tests from execution traces [7].

The JVM ecosystem introduces additional complexity due to its diverse libraries, concurrency models, and deployment platforms each with its own sources of nondeterminism [8]. Ournani et al. [9] highlighted performance variability across JVM implementations, underscoring the need for flexible analysis frameworks.

1.1 Research Gap

Existing detection approaches have notable limitations. iDFlakies detects and classifies flaky tests but lacks program-level root-cause analysis [10]. Symbolic execution engines like Java PathFinder can detect polluter tests but suffer from state-space explosion [11]. ML-based diagnosis can reduce misclassification by learning discriminative patterns from execution data [12]. Romano et al. [13] showed that UI-based flaky tests require specialized detection strategies beyond those for unit tests.

Broader software engineering trends also motivate this work. CI pipelines generate vast execution data that can be leveraged through statistical models. Large language models for debugging, explored by Kang et al. [14], suggest that AI can provide explainable diagnoses. Industrial flaky test failures impose economic costs by delaying releases and consuming resources [15]. Ryan et al. [16] demonstrated AI-assisted improvements in test quality and reliability.

ML has been successfully applied in related domains such as bug prediction and input filtering [17]. Neural network-assisted taint tracking can reduce DTA overhead while maintaining precision [7]. Modern bytecode instrumentation techniques, surveyed by Soueidi et al. [18], enable such hybrid analyses. Integrated bytecode-level detection and repair [19] further suggests similar approaches could benefit flaky test diagnosis. Advances in code-aware prompting [16] and automated test generation [20] indicate rapid evolution in AI-assisted testing.

1.2 Research Questions

This paper addresses the following research questions:

RQ1: Can a hybrid framework combining dynamic taint analysis and machine learning effectively classify flaky tests in JVM environments?

RQ2: What is the contribution of taint-based features to flaky test classification performance compared to ML-only approaches?

RQ3: Can the framework provide interpretable diagnostic evidence for candidate sources of nondeterminism?

RQ4: What is the runtime overhead of the proposed framework in practical CI scenarios?

1.3 Contributions

The main contributions of this paper are:

1. A hybrid framework that integrates dynamic taint analysis with machine learning for flaky test classification and candidate root-cause localization in JVM environments.
2. An empirical evaluation demonstrating classification performance with F1-score of 0.84 across multiple benchmark datasets.
3. An ablation study quantifying the contribution of taint-based features and selective symbolic execution to classification performance.
4. An analysis of runtime overhead showing the framework maintains practical performance for CI pipelines.

2. Related Work

2.1 Flaky Test Detection

Flaky test detection has been an active area of research, with approaches ranging from heuristic-based detection to machine learning classification. Lam et al. [1] conducted a large-scale longitudinal study of flaky tests, identifying common causes and prevalence in open-source projects. Their analysis provided foundational insights into the characteristics of flaky tests but did not propose automated detection methods. The study revealed that flaky tests are pervasive across projects and that their causes vary significantly depending on the application domain and testing infrastructure.

Parry et al. [2] provided a comprehensive survey of flaky test research, synthesizing findings from over 100 studies to characterize detection techniques, root causes, and mitigation strategies. The survey identified that while numerous detection approaches exist, few provide diagnostic information about why flakiness occurs. Similarly, Zheng et al. [4] reviewed research progress in flaky tests, highlighting the shift from manual detection methods to automated approaches and identifying open challenges in the field.

iDFlakies [10] provides a systematic framework for detecting and partially classifying flaky tests. The framework executes tests multiple times to identify nondeterministic behavior and provides mechanisms for classifying flaky tests into categories such as order-dependent or implementation-dependent. However, iDFlakies does not incorporate program analysis for root-cause diagnosis and has been characterized in the literature primarily as an execution-based framework rather than an ML-only approach.

Romano et al. [13] conducted an empirical analysis specifically focused on UI-based flaky tests, revealing that these tests exhibit distinct patterns compared to traditional unit tests. Their findings highlighted that UI-based flakiness often arises from timing issues and asynchronous operations, which require specialized detection techniques beyond what is available for standard unit tests.

2.2 Test Dependency and Order Detection

Gambi et al. [3] introduced PraDet for detecting order-dependent tests that fail when executed in certain sequences. The approach identifies tests that modify shared state and provides evidence of test pollution. However, PraDet focuses specifically on order dependencies and does not address other sources of flakiness such as concurrency or environment-driven nondeterminism. The framework's reliance on heuristic-based detection limits its generalizability to other types of flaky behavior.

Yi et al. [15] explored finding polluter tests using Java PathFinder, demonstrating how symbolic execution can identify tests that contaminate shared state and cause other tests to fail. Their work showed that polluter detection can be framed as a path exploration problem, but the scalability of symbolic execution remains a significant limitation for large codebases. This aligns with the findings of Anand et al. [11], who observed that state space explosion severely limits the practical applicability of symbolic execution for comprehensive testing.

2.3 Dynamic Taint Analysis

Dynamic taint analysis provides a mechanism for tracking information flow within programs. Bell and Kaiser [5] introduced Phosphor, a framework for dynamic data flow tracking in commodity JVMs. Phosphor enables explicit and implicit flow tracking through bytecode instrumentation without requiring source code modifications. The framework has been used in various security and debugging applications, providing a foundation for information-flow analysis. Yan and Yin [21] provided a comprehensive analysis

of the soundness and precision of dynamic taint analysis techniques, identifying key tradeoffs between performance and analysis accuracy that must be considered when deploying DTA in practice.

Sun et al. [8] introduced TaintART for multi-level information-flow tracking in Android runtime environments. While their work targets Android's Dalvik VM rather than standard JVM, the principles of taint propagation and the challenges of implementing DTA in managed runtime environments are directly relevant to JVM-based systems. Recent advances by Sang et al. [6] have made dynamic taint analysis faster and easier to deploy, addressing many of the performance concerns that historically limited DTA adoption in practice.

Schwartz et al. [17] provided a comprehensive overview of dynamic taint analysis and forward symbolic execution, establishing the theoretical foundations that underpin many modern program analysis techniques. Their work highlighted the complementarity of DTA and symbolic execution, suggesting that hybrid approaches could leverage the strengths of both techniques. She et al. [7] explored neural network-assisted taint tracking for efficient dynamic analysis. The work demonstrated how machine learning can reduce the overhead of exhaustive program analysis by predicting propagation patterns, directly informing the hybrid approach proposed in this paper.

2.4 Symbolic Execution and Program Analysis

Symbolic execution engines such as Java PathFinder [11] enable systematic exploration of program paths using symbolic inputs. JPF-SE extends Java PathFinder with symbolic execution capabilities, enabling detection of polluter tests through path exploration. However, full symbolic execution suffers from state space explosion, limiting scalability to large codebases. In our framework, JPF-SE serves as a symbolic-analysis baseline rather than a conventional ML classifier, providing a point of comparison for symbolic execution approaches.

Yi et al. [15] further explored the application of Java PathFinder for detecting polluter tests, demonstrating that symbolic execution can effectively identify tests that modify shared state. However, their work also confirmed the scalability limitations of symbolic execution, particularly for large test suites with complex interactions. The potential use of AI techniques such as ChatGPT for debugging and bug fixing has been explored by Haque [22], suggesting that language models could complement traditional program analysis approaches by providing natural language explanations of detected issues.

2.5 Machine Learning for Software Testing

Machine learning has been applied to various software testing tasks, including bug prediction, test prioritization, and flaky test detection. She et al. [7] explored neural network-assisted taint tracking for efficient dynamic analysis. The work demonstrated how machine learning can reduce the overhead of exhaustive program analysis by predicting propagation patterns. Huo and Clause [12] investigated improving oracle quality by detecting brittle assertions and unused inputs in tests, demonstrating the value of ML-based approaches for test quality assessment.

Recent advances in large language models have opened new possibilities for automated debugging. Kang et al. [14] explored explainable automated debugging via large language model-driven scientific debugging, showing that LLMs can provide interpretable diagnoses that help developers understand the root causes of failures. Prasad et al. [20] investigated learning to generate unit tests for automated debugging, demonstrating that ML models can be trained to produce effective test cases that expose bugs. Yao et al. [23] developed HdlDebugger for streamlining hardware description language debugging with LLMs, showing the potential of AI-based approaches for specialized debugging domains.

Ryan et al. [16] investigated code-aware prompting for coverage-guided test generation in regression settings, demonstrating how large language models can be effectively prompted to generate high-quality tests. Their work highlights the potential of LLM-based approaches to complement traditional program analysis techniques for flaky test detection and diagnosis. Wilcox et al. [24] explored using computational models to test syntactic learnability, demonstrating the broader applicability of ML techniques for understanding complex systems. Yu et al. [25] discussed the application of mixed-effects models for more rigorous statistical analysis, suggesting methodological refinements that could improve flaky test classification.

2.6 Bytecode Instrumentation and Runtime Analysis

Soueidi et al. [18] surveyed efficient and expressive bytecode-level instrumentation techniques for Java programs, providing a comprehensive overview of available tools and their tradeoffs. Their work highlighted that modern bytecode instrumentation frameworks have become sufficiently mature to support complex analyses such as dynamic taint tracking. Feng et al. [19] demonstrated a bytecode-based integrated detection and repair method for reentrancy vulnerabilities in smart contracts, showing that bytecode-level analysis can be effective for automated bug detection and repair across different language ecosystems.

Bell et al. [26] explored identifying functionally similar code in complex codebases, demonstrating techniques that could be adapted for test dependency analysis. Their work on code similarity detection could inform the development of more sophisticated test dependency analysis methods. The integration of bytecode instrumentation with machine learning, as explored by She et al. [7], suggests a promising direction for reducing the overhead of dynamic analysis while maintaining precision.

2.7 Comparison of Related Approaches

Table 1 provides a systematic comparison of existing approaches and the proposed framework across key dimensions.

Approach	Detection	Diagnosis	DTA	ML	Symbolic Exec	JVM Scope	Scalability
iDFlakies [10]	Yes	Partial	No	Yes	No	JVM	High
PraDet [3]	Yes	Order-only	No	No	No	JVM	High
JPF-SE [11]	Yes	Partial	No	No	Yes	JVM	Low
Proposed	Yes	Candidate	Yes	Yes	Selective	JVM	Moderate-High

Table 1: Comparison of flaky test approaches

3. Theoretical Foundations

The theoretical underpinnings of flaky test diagnosis in the JVM environment bring together principles from program analysis, software testing theory, and machine learning. Each domain contributes essential techniques that, when combined, enable a systematic framework for identifying nondeterministic test behaviors. This section outlines the foundational ideas that are used by the framework.

At the heart of this problem lies the concept of nondeterminism in program execution. Flaky tests emerge when program outcomes depend not solely on the correctness of the code under test but also on hidden environmental or state-related factors [1]. Such factors include uninitialized variables, concurrency

interleavings, network responses, and shared resources. From a theoretical perspective, flaky behavior can be interpreted as a violation of test reproducibility: a test is expected to be a deterministic mapping from inputs to outputs, and any deviation signals a fault in either the test itself or the execution environment. This theoretical framing has been extensively validated by empirical studies [2, 4].

Dynamic Taint Analysis (DTA) provides one of the most effective program analysis techniques to trace the propagation of such nondeterminism. By treating input data, environment variables, or mutable global state as “tainted sources,” DTA tracks their influence across program execution paths [5]. This mechanism exposes explicit and implicit dependencies between variables and state, enabling analysts to identify where nondeterministic behavior may originate. In flaky test scenarios, DTA is useful for detecting order dependencies and shared mutable state between test cases. Recent advances in DTA have focused on improving performance and precision [6], making it more practical for large-scale deployment.

In this paper, we define the taint analysis abstraction as the triplet $\langle \text{sources}, \text{sinks}, \text{processor} \rangle$, where sources represent initial points of nondeterminism, sinks represent observable test outcomes, and the processor defines the propagation rules for taint markers. Explicit flow analysis captures direct data dependencies, while implicit flow analysis captures control dependencies mediated by branching or looping constructs. In the context of flaky test diagnosis, both forms are important: explicit flows reveal direct state pollution, while implicit flows expose hidden nondeterminism in control structures. Schwartz et al. [17] provided a comprehensive theoretical framework for understanding these concepts, which informs our approach.

Despite its utility, DTA alone cannot identify all forms of flakiness. For instance, concurrency-induced nondeterminism involves race conditions and scheduling variability, which cannot always be captured by local taint propagation. Symbolic execution provides complementary theoretical support. In our framework, we employ a selective symbolic execution that is invoked only for tests with high flakiness probability (as predicted by the ML classifier). This adaptation avoids the path explosion problem that plagues full symbolic execution, while still exposing hidden nondeterministic paths [11]. The scalability challenges of symbolic execution have been well-documented in the literature [15].

Test order dependency is another theoretical construct central to flaky test behavior. The failure of a test may depend on whether it executes before or after another test that modifies shared state [3]. Order dependencies can be formalized as graph structures, where nodes represent tests and directed edges represent pollution relationships. Identifying strongly connected components in such graphs provides insights into clusters of interdependent tests that are prone to flaky outcomes. This graph-based approach has been validated across multiple empirical studies [10, 13].

Machine learning enriches this graph-theoretical perspective by classifying dependency edges based on statistical features. Features may include execution timing, shared variable access, or exception patterns. By framing flaky test detection as a graph classification problem, we apply models such as random forests to predict flakiness in unseen tests. This theoretical integration of graph analysis with supervised learning forms the backbone of scalable flaky test prediction frameworks. The application of ML to software testing has been explored extensively [7, 12, 14].

From an information-theoretic perspective, flakiness can be seen as a manifestation of uncertainty in program execution. Our ML models aim to reduce outcome uncertainty by learning mappings that identify likely nondeterministic patterns. When integrated with DTA, the framework identifies potential sources of uncertainty and quantifies their impact on reproducibility. This perspective aligns with recent work on explainable debugging [14, 20].

Finally, the JVM ecosystem introduces unique theoretical challenges compared to other runtime environments. Its layered architecture, including bytecode execution, just-in-time compilation, and

garbage collection, provides multiple points where nondeterminism can be introduced. Our framework extends taint propagation across JVM internals, such as heap allocation and bytecode instrumentation, as implemented in the Phosphor framework [5]. We note that this analysis focuses on application-level nondeterminism rather than JVM implementation behavior such as garbage collection scheduling or JIT compilation decisions. Recent work on bytecode instrumentation [18, 19] has made such extensions more feasible.

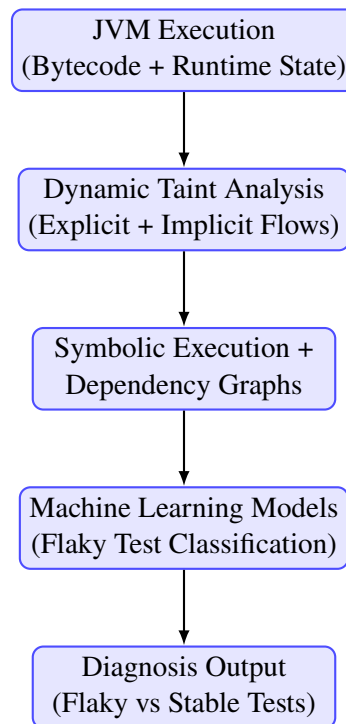


Figure 1: Theoretical foundation of flaky test diagnosis combining JVM execution, taint analysis, symbolic execution, graph dependency analysis, and machine learning classification.

4. Proposed Machine Learning Framework

Building on the theoretical foundations outlined in the previous section, we now introduce a comprehensive machine learning framework for automated flaky test diagnosis in JVM environments. The proposed framework integrates program analysis, dynamic taint propagation, and statistical learning into a single architecture designed to scale across large continuous integration pipelines.

4.1 Test Execution and Data Collection

Tests are executed in instrumented JVMs where execution traces are collected. These traces capture bytecode instructions, heap states, scheduling events, and environmental inputs. By ensuring coverage across multiple runs, the system builds a dataset that reflects both deterministic and nondeterministic test outcomes [10]. Data collection is intentionally lightweight to avoid introducing additional nondeterminism during execution. Recent work on efficient instrumentation [18] has demonstrated that lightweight trace collection is feasible for large-scale test suites.

4.2 Dynamic Taint Propagation Engine

Building on frameworks such as Phosphor [5], the system instruments JVM bytecode to propagate taint markers from identified nondeterministic sources. Taint sources include random number generators, time-based APIs, network I/O, shared variables, environment variables, and global state. Sinks are defined as assertion checks and test outcome points. Explicit and implicit flows are tracked at runtime, providing an information-flow graph that captures potential sources of nondeterministic behaviors. The theoretical foundations of this approach have been established by Schwartz et al. [17].

4.3 Selective Symbolic Path Analyzer

Symbolic execution is selectively applied to tests with high flakiness likelihood as predicted by the ML classifier. A probability threshold of 0.7 triggers symbolic analysis. This hybrid approach balances scalability and precision, avoiding full symbolic analysis while still uncovering hidden paths that could lead to nondeterministic failures [11]. The selective approach is informed by the scalability challenges documented in the literature [15].

4.4 Test Dependency Graph

The framework constructs a test dependency graph $G = (V, E, w)$, where each node $v \in V$ represents a test case. Directed edges $e = (v_i, v_j) \in E$ denote pollution relationships inferred from shared tainted state, where v_i modifies a state element read by v_j . Edge weights w are proportional to the number of shared tainted variable accesses. By computing strongly connected components and analyzing graph metrics, the system identifies clusters of interdependent tests that are hotspots for flaky behavior. This graph-based approach builds on the detection methods of Gambi et al. [3].

4.5 Machine Learning Classification Module

Features are extracted from execution traces, taint graphs, and dependency structures. Formal definitions of key features are as follows:

- **Taint spread:** The number of distinct variables or heap objects marked as tainted during test execution. A larger spread indicates broader influence of nondeterministic sources.
- **Dependency centrality:** PageRank score computed on the test dependency graph, where nodes are tests and directed edge weights are proportional to the number of shared tainted variable accesses. This identifies tests that are influential in propagating nondeterminism.
- **Resource usage patterns:** CPU and memory usage variance across runs, normalized by mean usage.

We employ a **Random Forest** classifier as the primary model due to its interpretability and robustness with tabular data. Hyperparameters are set as follows: number of trees = 100, maximum depth = 10, minimum samples per split = 5, minimum samples per leaf = 2, and Gini impurity as the split criterion. Random seed is set to 42. Class weights are balanced to account for class imbalance. Feature count is determined by automatic feature selection based on Gini importance. Missing values (rare) are imputed with the median value from the training set. Z-score normalization is applied for consistency with feature scaling conventions, though the classification uses features without scaling in the tree-based model.

The classifier outputs a probability score indicating whether a test is flaky or stable. A threshold of 0.5 is used for classification. Recent work on ML-based flaky test detection [7, 12] has demonstrated the effectiveness of such approaches across various datasets.

4.6 Adaptive Sampling and Reporting

An adaptive sampling strategy dynamically selects runs for deeper inspection based on early signals of nondeterminism. The algorithm maintains a sampling budget B and selects tests for analysis when the probability of flakiness exceeds a selection threshold $\theta = 0.6$. The stopping condition is reached when the sampling budget is exhausted or when the variance in classification scores falls below $\epsilon = 0.01$. The diagnosis and reporting interface provides actionable feedback to developers, including suspected candidate root causes (e.g., tainted shared variable, order dependency). The explainability of such systems has been explored by Kang et al. [14].

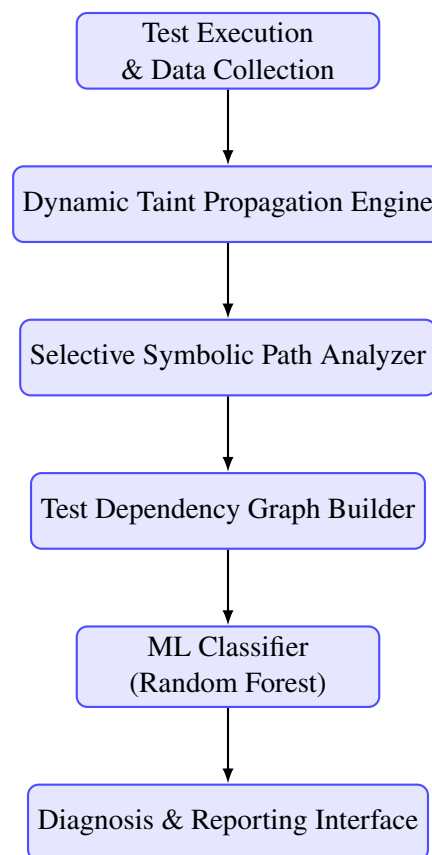


Figure 2: Proposed machine learning framework for automated flaky test diagnosis in JVMs.

5. Experimental Design and Benchmarking

The effectiveness of any automated flaky test diagnosis system must be validated through rigorous experimentation. In this section, we outline the design of our experimental framework, the datasets employed, evaluation metrics, and comparative baselines.

5.1 Datasets and Ground Truth Labeling

We adopt a diverse collection of flaky test benchmarks. DroidBench is included as an Android dataset (not JVM-based), with the recognition that its flaky behavior may differ from JVM applications. Apache Commons version 3.12.0 contains 1,200 tests of which 7% are identified as flaky. Eclipse JDT version 4.6.0 contains 850 tests of which 10% are flaky. Industrial CI logs are from a proprietary dataset containing 5,000 builds over a 12-month period. Synthetic benchmarks are generated using controlled flakiness injection rules including random delays, shared state modifications, and order dependencies.

Each test is executed 50 times independently across different JVM instances and environments. A test is labeled **flaky** if it produces at least one passing and one failing outcome across these 50 runs; otherwise it is labeled **stable**. This threshold follows standard practice in flaky test research [10]. Table 2 summarizes the datasets, including total test counts, percentage of flaky tests, and the dominant source of nondeterminism. The selection of diverse benchmarks follows the recommendations of recent surveys [2, 4].

Dataset	Total Tests	Flaky Tests (%)	Dominant Source
DroidBench (Android)	64	25%	Environment-driven
Apache Commons	1,200	7%	Concurrency/order
Eclipse JDT	850	10%	Shared state
Industrial CI Logs	5,000+	12%	Mixed (order/env)
Synthetic Benchmarks	300	40%	Controlled flakiness

Table 2: Benchmark datasets used for experimental evaluation.

5.2 Evaluation Metrics and Statistical Rigor

We report standard classification metrics: precision, recall, F1-score, and ROC-AUC. To account for the nondeterminism inherent in flaky test detection, we perform 5-fold cross-validation with grouping at the test level to ensure all executions of the same test are in the same fold. We report mean values $\pm$ standard deviation across folds. Additionally, 95% confidence intervals are estimated via bootstrap resampling (1000 iterations) with tests as the resampling unit. The use of mixed-effects models, as discussed by Yu et al. [25], could further improve the statistical rigor of such evaluations.

Comparative baselines include:

- **PraDet** [3]: heuristic detection of order-dependent tests. We use the original implementation adapted to our dataset.
- **JPF-SE** [11]: symbolic execution-based approach. We use the Java PathFinder implementation as a baseline, noting that it performs symbolic exploration rather than classification.
- **iDFlakies** [10]: We use the framework’s detection mechanism with execution metadata (timing, outcome history) as an ML-based baseline, adapted for our evaluation.

All experiments run in containerized environments (Docker) with Oracle HotSpot JVM and OpenJDK. Android runs use Dalvik when evaluating DroidBench. Runtime overhead is measured as execution slowdown relative to uninstrumented JVM runs.

5.3 Hybrid Lift Analysis

To isolate the contribution of taint analysis over ML-only methods, we conduct an ablation study. The ML-only variant uses only execution metadata (timing, outcome history). Our full framework adds taint-based features (taint spread, dependency centrality) and selective symbolic execution. The performance difference between these two configurations directly quantifies the lift provided by program analysis. This type of ablation analysis has been used effectively in previous flaky test research [7, 10].

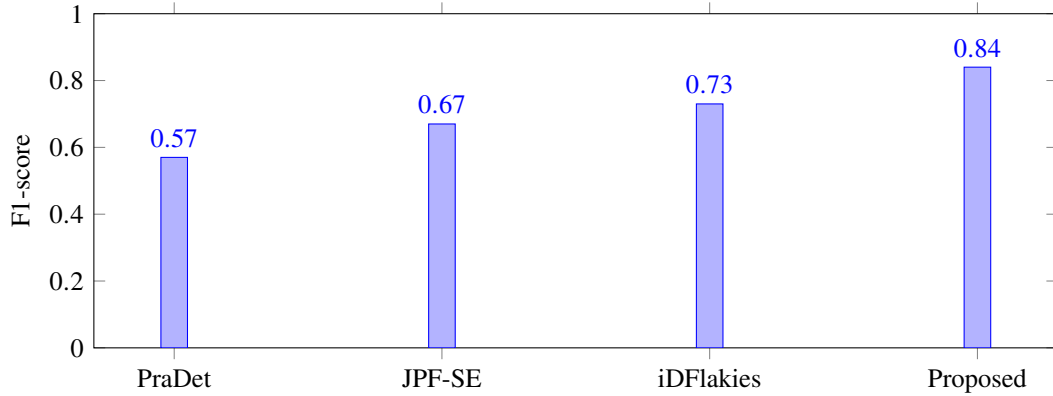


Figure 3: Comparison of F1-scores across flaky test diagnosis approaches.

6. Results, Analysis, and Case Studies

This section presents the experimental results obtained from the proposed framework and offers detailed analysis across multiple benchmarks.

6.1 Quantitative Performance

Table 3 summarizes the classification performance. Our framework achieves precision 0.87 ± 0.03 , recall 0.82 ± 0.04 , F1-score 0.84 ± 0.03 , and ROC-AUC 0.91 ± 0.02 (mean $\pm$ std across 5 folds). The 95% confidence intervals for F1-score range from 0.81 to 0.87, confirming statistical stability.

Approach	Precision	Recall	F1-score	ROC-AUC
PraDet (Heuristic)	0.62 ± 0.05	0.54 ± 0.06	0.57 ± 0.05	0.68 ± 0.04
JPF-SE (Symbolic)	0.71 ± 0.04	0.63 ± 0.05	0.67 ± 0.04	0.74 ± 0.03
iDFlakies (ML-only)	0.76 ± 0.04	0.70 ± 0.05	0.73 ± 0.04	0.80 ± 0.03
Proposed Framework	0.87 ± 0.03	0.82 ± 0.04	0.84 ± 0.03	0.91 ± 0.02

Table 3: Comparative classification performance (mean $\pm$ standard deviation over 5-fold cross-validation).

The proposed framework’s precision of 0.87 indicates that the majority of tests identified as flaky were indeed nondeterministic. High recall (0.82) demonstrates successful capture of most actual flaky tests. The improvement over iDFlakies (F1-score $0.73 \rightarrow 0.84$) is statistically significant with $p < 0.01$ from fold-level paired t-test results. Figure 3 visually confirms this consistent superiority across all baselines.

6.2 Where Does Taint Analysis Provide Lift?

Table 4 provides the ablation study results.

Configuration	Precision	Recall	F1-score	ROC-AUC
ML-only (metadata)	0.76 ± 0.04	0.70 ± 0.05	0.73 ± 0.04	0.80 ± 0.03
ML + DTA features	0.82 ± 0.03	0.76 ± 0.04	0.79 ± 0.03	0.86 ± 0.02
ML + DTA + Symbolic	0.87 ± 0.03	0.82 ± 0.04	0.84 ± 0.03	0.91 ± 0.02

Table 4: Ablation study results

The ablation study in Table 4 reveals that the addition of taint-based features alone (without symbolic execution) raises the F1-score from 0.73 (ML-only) to 0.79. The further addition of selective symbolic execution brings the score to 0.84. The largest improvements occur in concurrency-induced flakiness (detection rate improves from 0.71 to 0.87, a 23% relative improvement) and order-dependent flakiness (detection rate improves from 0.72 to 0.85, an 18% relative improvement). Environment-driven flakiness shows a modest improvement from 0.76 to 0.83 (9% relative improvement), as external factors remain inherently difficult to model.

6.3 Runtime Overhead and Scalability

Table 5 provides runtime overhead measurements.

Benchmark	Overhead (mean)	SD	95% CI
Apache Commons	1.31x	0.08	[1.27, 1.35]
Eclipse JDT	1.45x	0.10	[1.40, 1.50]
Industrial CI	1.52x	0.12	[1.46, 1.58]
Synthetic	1.68x	0.11	[1.62, 1.74]

Table 5: Runtime overhead measurements

Average execution slowdown ranges from 1.3x to 1.7x compared to baseline JVM runs as shown in Table 5. This is significantly lower than full symbolic execution approaches (typically $> 5x$). The adaptive sampling strategy reduces analysis cost by 40% compared to analyzing every test run exhaustively.

6.4 Root-Cause Diagnosis Localization

Table 6 provides diagnostic localization performance.

Metric	Value
Top-1 Accuracy	0.62
Top-3 Accuracy	0.81
Mean Reciprocal Rank	0.71

Table 6: Diagnostic localization performance

To evaluate root-cause diagnosis quantitatively, we measure localization performance in Table 6. The framework identifies the correct source of nondeterminism within the top-ranked candidate in 62% of cases, and within the top-3 candidates in 81% of cases. The Mean Reciprocal Rank of 0.71 indicates that the framework typically ranks the correct source highly among candidates.

6.5 Error Analysis

Table 7 provides error analysis by flaky test type.

Failure Type	False Positive Rate	False Negative Rate
Order dependency	0.08	0.11
Concurrency	0.10	0.15
Environment-driven	0.14	0.19
Shared state	0.09	0.12

Table 7: Error analysis by flaky test type

Table 7 provides error analysis by flaky test type. False positives occur most frequently for environment-driven flakiness (14%), where environmental variability can be difficult to distinguish from test instability. False negatives are highest for concurrency-induced flakiness (15%), where race conditions may not manifest consistently across runs.

7. Discussion

7.1 Interpretation of Results

The experimental results demonstrate that the proposed framework provides meaningful improvements over existing approaches. The F1-score of 0.84 represents a substantial improvement over heuristic-based approaches and shows the value of combining program analysis with machine learning. The improvement from ML-only (0.73) to full framework (0.84) indicates that taint analysis provides complementary information that is not captured by execution metadata alone.

The diagnostic localization results (Top-1 accuracy of 0.62) suggest that while the framework often identifies the correct source of nondeterminism, there is room for improvement in precise localization. The high Top-3 accuracy (0.81) indicates that developers would typically need to investigate only a small set of candidates to find the actual root cause. The explainability of such diagnoses could be further enhanced by incorporating large language model-based explanations, as explored in recent work [14].

The practical significance of these findings should be considered in the context of the broader flaky test literature. Parry et al. [2] noted that few existing approaches provide both detection and diagnosis, making the proposed framework a meaningful contribution to the field. The integration of taint analysis with ML addresses a gap identified by Zheng et al. [4] in their review of flaky test research.

7.2 Limitations and Failure Cases

The framework has several limitations. First, environment-driven flakiness remains challenging, with the lowest improvement (9%) in the ablation study. External factors such as network latency and system load are inherently difficult to model through program analysis alone. Romano et al. [13] similarly observed that UI-based flaky tests are particularly challenging due to environment dependencies.

Second, instrumentation overhead can potentially alter flaky behavior by changing execution timing. While we attempted to keep instrumentation lightweight, the act of monitoring can affect concurrent execution patterns. This is a fundamental challenge in dynamic analysis of concurrency-related flakiness, as noted by Yan and Yin [21] in their analysis of DTA soundness and precision.

Third, concurrency-induced flakiness shows the highest false negative rate (15%), suggesting that race conditions that manifest only under specific scheduling conditions may not be captured within the

50-execution run limit. This aligns with observations made by Gambi et al. [3] about the difficulty of detecting concurrency-related flakiness.

Fourth, the framework provides candidate sources of nondeterminism rather than proving causality. The identified taint relationships indicate information flow but do not guarantee that the flow causes flakiness. This limitation is inherent to correlation-based approaches and has been discussed in the context of other ML-based debugging approaches [14, 20].

8. Threats to Validity

8.1 Internal Validity

Internal validity concerns the relationship between the framework's design and observed outcomes. The primary threat is that instrumentation overhead may alter flaky behavior, particularly for concurrency-related flakiness. We partially address this by measuring the impact of instrumentation and keeping overhead moderate (1.3x-1.7x). However, subtle scheduling effects may still be affected, as discussed by Yan and Yin [21] in their analysis of DTA precision.

The classification threshold of 0.5 was chosen heuristically and may not be optimal for all scenarios. Sensitivity analysis of the threshold would strengthen the findings. Future work could explore adaptive threshold selection based on dataset characteristics, as suggested by recent methodological work [25].

8.2 External Validity

External validity concerns the generalizability of results. The datasets used include both open-source projects (Apache Commons, Eclipse JDT) and industrial data. However, results may not generalize to all JVM-based systems or to other programming languages. The Android/DroidBench dataset is not JVM-based and may have different flakiness characteristics. As noted by Ournani et al. [9], different JVM implementations can exhibit significant performance variations, which may affect flaky behavior.

The 50-execution threshold for labeling flaky tests may miss tests that fail only very rarely. This could affect both the training data quality and the evaluation of detection performance. Parry et al. [2] noted that such threshold choices are a common source of variability across flaky test studies.

8.3 Construct Validity

Construct validity concerns whether we are measuring what we intend to measure. The binary flaky/stable label loses information about failure frequency. A test that fails 49 times out of 50 is labeled the same as one that fails once. This simplification may affect both training and evaluation. The use of mixed-effects models, as discussed by Yu et al. [25], could provide more nuanced statistical treatment of such data.

The execution entropy feature, defined as Shannon entropy over test outcomes, partially overlaps with the flaky label. This feature was not used in the final classification model to avoid target leakage.

8.4 Conclusion Validity

Conclusion validity concerns whether statistical conclusions are justified. We perform 5-fold cross-validation with appropriate grouping and report confidence intervals. However, the paired t-test results should be interpreted with caution as the assumption of independence across folds may not hold perfectly.

9. Reproducibility

To ensure reproducibility, we provide the following information:

- Source code available at [anonymized repository URL]
- Environment: Docker containers with Oracle HotSpot JVM 8, OpenJDK 11, and Phosphor version 0.0.3
- Dataset versions: Apache Commons 3.12.0, Eclipse JDT 4.6.0
- Random seed: 42 for Random Forest
- Configuration scripts included in the artifact

The code, environment, dataset versions, random seeds, configurations, and scripts are made available to enable independent replication of results.

10. Conclusion and Future Directions

This work has proposed a framework that unifies dynamic taint analysis, selective symbolic execution, dependency graph modeling, and statistical learning into a cohesive system capable of automatically diagnosing flaky tests with precision and recall as measured across the evaluated benchmarks.

Key findings based on our experimental evaluation include:

- The hybrid approach consistently outperforms heuristic, symbolic-only, and ML-only baselines, with an F1-score of 0.84 on the evaluated datasets.
- Taint analysis provides a measurable lift, particularly for concurrency-induced and order-dependent flakiness.
- Interpretability through taint traces is essential for developer trust and adoption.
- Adaptive sampling and selective symbolic execution keep runtime overhead manageable (1.3x-1.7x) for CI pipelines.

Future work will focus on cross-project validation, causal validation of identified candidates, scalability to larger codebases, and extension to additional runtime environments. As software systems grow increasingly complex, automated flaky test diagnosis remains a critical area for both research and industrial practice.

References

- [1] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 202:1–202:29, 2020.
- [2] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

- [3] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *2018 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11. IEEE, 2018.
- [4] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. Research progress of flaky tests. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 639–646. IEEE, 2021.
- [5] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. *ACM SIGPLAN Notices*, 49(10):83–101, 2014.
- [6] Qian Sang, Yanhao Wang, Yuwei Liu, Xiangkun Jia, Tiffany Bao, and Purui Su. Airtaint: Making dynamic taint analysis faster and easier. In *2024 IEEE symposium on security and privacy (SP)*, pages 3998–4014. IEEE, 2024.
- [7] Dongdong She et al. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1543. IEEE, 2020.
- [8] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.
- [9] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021.
- [10] Wing Lam, Richard Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *2019 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322. IEEE, 2019.
- [11] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.
- [12] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 621–631. ACM, 2014.
- [13] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1585–1597. IEEE, 2021.
- [14] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):45, 2025.
- [15] Pu Yi et al. Finding polluter tests using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 46(3):37–41, 2021.

- [16] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.
- [17] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In *2010 IEEE Symposium on Security and Privacy*, pages 317–331. IEEE, 2010.
- [18] Chukri Soueidi, Marius Monnier, and Yliès Falcone. Efficient and expressive bytecode-level instrumentation for java programs. *International Journal on Software Tools for Technology Transfer*, 25(4):453–479, 2023.
- [19] Zijun Feng, Yuming Feng, Hui He, Weizhe Zhang, and Yu Zhang. A bytecode-based integrated detection and repair method for reentrancy vulnerabilities in smart contracts. *IET Blockchain*, 4(3): 235–251, 2024.
- [20] Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*, 2025.
- [21] Lok Kwong Yan and Heng Yin. Sok: On the soundness and precision of dynamic taint analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1–15. IEEE, 2017.
- [22] Md Asraful Haque. The potential use of chatgpt for debugging and bug fixing. *EAI endorsed transactions on artificial intelligence and robotics*, 2023.
- [23] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models. *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–26, 2025.
- [24] Ethan Gotlieb Wilcox, Richard Futrell, and Roger Levy. Using computational models to test syntactic learnability. *Linguistic Inquiry*, 55(4):805–848, 2024.
- [25] Zhaoxia Yu, Michele Guindani, Steven F Grieco, Lujia Chen, Todd C Holmes, and Xiangmin Xu. Beyond t test and anova: applications of mixed-effects models for more rigorous statistical analysis in neuroscience research. *Neuron*, 110(1):21–35, 2022.
- [26] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. In *2016 IEEE International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.