Testing Static Analyzers via Semantic-Preserving Mutators Learned from Real-World Refactoring Practice

Meilin Li*
Chongqing University
Chongqing, China
leemeii@stu.cqu.edu.cn

Kaixuan Li*
Nanyang Technological University
Chongqing, Singapore
kaixuan.li@ntu.edu.sg

Zifan Xie†
Chongqing University
Chongqing, China
xzff@cqu.edu.cn

Shiyu Qiu
Huazhong University of Science and Technology
Wuhan, China
qiusy@hust.edu.cn

Ming Wen
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Maolin Sun
Nanjing University
Nanjing, China
merlin@smail.nju.edu.cn

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Abstract

Static analyzers reason about program behavior without execution and report issues when code violates predefined rules. A key limitation is their production of inaccurate, incomplete results, generating spurious warnings and missing critical issues. Metamorphic testing is the most commonly used method to efficiently detect bugs in static analyzers. It requires designing a set of semantic-preserving mutators. When a mutator is applied to a seed program, any discrepancy in the analyzer's reports for the original and mutated versions indicates a potential bug. Thus, efficiently generating large numbers of semantic-preserving mutators is critical. We observe that during routine software maintenance, developers often restructure code to improve readability or optimize it without changing external program behavior. Our key insight is that mining and generalizing these patterns from real-world refactoring practice enables the automatic construction of numerous mutators that reflect the distribution of real-world code variants. We present SAFuzzer, a framework that mines such refactoring practice and distills these edits into semantic-preserving mutators. We ultimately generate 313 mutators and apply them to detect bugs in five widely used Java static analyzers. SAFuzzer detects 42 previously unknown bugs. Compared with five state-of-the-art baselines, SAFuzzer detects the most bugs, demonstrating the value of learning diverse mutators from real-world refactoring practice.

CCS Concepts

• **Software and its engineering → Maintaining software**.

*These authors contributed equally to this work
†Corresponding author

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837544

Keywords

Static Analyzer, Metamorphic Testing, Large Language Model

ACM Reference Format:
Meilin Li, Kaixuan Li, Zifan Xie, Shiyu Qiu, Ming Wen, Maolin Sun, and Hongyu Zhang. 2026. Testing Static Analyzers via Semantic-Preserving Mutators Learned from Real-World Refactoring Practice. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3837544

1 Introduction

Static analyzers [12, 13, 19, 23] play a vital role in modern software development. By reasoning about program behavior without execution, they detect bugs, enforce coding standards, and improve code quality at scale [11, 26, 32, 33]. Prominent tools such as SpotBugs [33], PMD [26], Infer [11], and SonarQube [32] have been widely adopted in both open-source projects and industrial practice [1, 28]. However, static analyzers are complex software systems and are therefore also susceptible to bugs. Under the documented semantics of a rule, a *false positive* [10] occurs when the analyzer reports a warning on compliant code. A *false negative* occurs when it misses a genuine violation. We treat such inaccuracies as bugs only when they violate the rule specification. Both types of inaccuracies erode developer trust and diminish the practical value of these tools [18, 37, 41]. Hence, systematically testing static analyzers to uncover such bugs is important and challenging [14].

To address this, prior work on testing static analyzers commonly instantiates metamorphic testing [5, 6, 30] for testing static analyzers. Metamorphic testing can in principle use diverse measurable metamorphic relations between inputs, with the corresponding relation checked in the outputs. In this work, we focus on the equivalence relation used by Statfier [41], where a *semantic-preserving transformation* is applied to a seed program to produce a behaviorally equivalent mutant. Following prior work [41], we use *semantic-preserving* to denote transformations that preserve

the observable behavior of the program, i.e., produce a behaviorally equivalent mutant. Since the transformation preserves program semantics, any discrepancy between the analyzer's reports on the seed and the mutant indicates a potential bug. The effectiveness of this strategy hinges on two key requirements: the *diversity* of the semantic-preserving transformations, which determines how many analyzer behaviors can be exercised, and their *correctness*, which ensures that any observed discrepancy is a genuine bug rather than a spurious artifact.

Existing approaches that broadly follow this metamorphic testing paradigm fall into two categories: *automated oracle construction* and *mutation-based fuzzing*, yet neither fully satisfy both requirements. For automated oracle construction, He et al. [15] design two categories of test oracles. The static oracle leverages seven types of equivalent boolean expressions. The dynamic oracle validates specific runtime-error properties. This design, however, is bound to specific runtime checkers such as null pointer dereference and out-of-bounds access. It also fails to explore the diverse syntactic forms of semantically equivalent program logic. For mutation-based fuzzing, Statfier [41] defines 12 manually curated semantic-preserving transformations gathered from existing literature. Although these transformations are carefully verified, their limited number restricts the range of syntactic patterns that can be explored, leaving many analyzer blind spots untested.

This raises a natural question: where can we find diverse semantic-preserving transformations beyond manually designed rules? We observe that real-world software development already provides such a source: *refactoring commits*. During routine maintenance, developers frequently refactor code to improve readability or optimize performance without altering external program behavior [21, 31]. These changes are recorded in version control systems and can be systematically identified from version histories at scale [9, 34, 35]. Because each refactoring commit restructures code while likely preserving its behavior, it inherently encodes a semantics-preserving transformation rooted in real-world programming practices. Our preliminary study (Section 2.1) supports this insight. Across 8,552 refactoring commits, such commits frequently change the rule sets covered by static analyzers. This confirms their potential as a rich source of transformation patterns for metamorphic testing.

However, turning refactoring commits into a practical testing framework presents a key challenge: not all refactoring commits truly preserve program semantics. Developers may bundle incidental behavioral changes alongside restructuring edits. A mutator derived from such a commit introduces subtle behavioral deviations, producing spurious report differences that compromise the validity of the metamorphic oracle.

In this paper, we propose SAFuzzer, a novel framework that mines real-world refactoring commits and distills them into validated semantic-preserving mutators. To harvest diverse transformation patterns, SAFuzzer mines refactoring commits and uses an LLM-based agent to extract the core semantic-preserving pattern from each refactoring. It then synthesizes the extracted pattern into an executable, Spoon-based [25] AST mutator. In this way, context-dependent refactoring edits are distilled into precise and reusable transformation specifications. To ensure semantic correctness, SAFuzzer adopts a *dynamic behavioral equivalence* strategy built on a staged validation pipeline. Each candidate mutator must satisfy a series of progressively stricter goals: compilation, termination, crash-freedom, mutant compilability, and behavioral equivalence. At the final stage, SAFuzzer compares the observable runtime behaviors of the original and mutated programs at two levels of granularity: coarse-grained observables (return values, exceptions, and standard output) and fine-grained variable traces that record every local variable's value at scope exit. When any goal is unmet, an LLM-based refinement loop automatically repairs the mutator for up to R iterations before discarding it. Only mutators that pass all validation goals are admitted into the final mutator pool.

Concretely, SAFuzzer operates in three phases. In the *Mutator Invention* phase, we collect 8,552 refactoring commits from 1,288 Java repositories on GitHub. We distill these commits into executable mutators through an LLM-based pipeline. This pipeline extracts transformation patterns and synthesizes Spoon-based AST transformations. In the *Mutator Refinement* phase, each candidate mutator undergoes the staged validation pipeline with LLM-driven refinement to fix or filter out those that fail to preserve semantics, yielding a validated mutator library of 313 mutators. In the *Testing Static Analyzer* phase, the validated mutators are applied at scale to seed programs, and the analysis reports of target static analyzers on each seed-mutant pair are compared to detect inconsistencies that signal potential bugs.

We evaluate SAFuzzer on five widely used Java static analyzers: SpotBugs, PMD, Infer, SonarQube, and Checkstyle. SAFuzzer detects 42 previously unknown bugs, of which 38 have been confirmed and 9 have been fixed by developers. Compared with five baselines, SAFuzzer detects the most bugs and achieves the highest rule coverage improvement, demonstrating the value of learning diverse mutators from real-world refactoring practice.

In summary, this paper makes the following contributions:

- **A refactoring-driven approach for generating semantic-preserving mutators.** We propose mining real-world refactoring commits to automatically construct diverse mutators, overcoming the limited diversity of manually curated mutators.
- **An automated framework with semantic validation.** We design and implement SAFuzzer, a three-phase framework that integrates LLM-based mutator synthesis with dynamic equivalence checking to ensure the correctness of generated mutators.
- **Extensive evaluation with real-world impact.** We evaluate SAFuzzer on five production static analyzers and detect 42 previously unknown bugs, of which 38 have been confirmed. We release our tool and dataset for replication at: https://doi.org/10.5281/zenodo.21722288.

2 Background and Motivation

2.1 Static Analyzers

Static analyzers [12, 13, 23] reason about program behavior without executing code. Given an input program, a static analyzer applies techniques such as data flow analysis and control flow analysis to construct an abstract representation of the program. It then checks this abstraction against a set of predefined rules and reports warnings when the program violates desirable properties. Because they do not require program execution, static analyzers are widely adopted in industrial software development for detecting bugs, enforcing coding standards, and improving code quality [32, 33].

For these analyzers, a core reporting unit is a *rule*. Each rule encodes a specific property that the analyzer checks. For example, SpotBugs defines the rule `NP_LOAD_OF_KNOWN_NULL_VALUE` to detect uses of values that are known to be null due to an earlier null check. When code violates a rule, the analyzer issues a warning. This warning specifies the error location and the relevant rule name. Table 1 shows the five widely used static analyzers used in our study, along with their versions, the number of rules each defines, and the type of analysis target they operate on.

Table 1: Static analyzers used in our study.

Static Analyzer	Version	#Rules	Analysis Target
Checkstyle [4]	13.3.0	73	Source code
PMD [26]	7.22.0	170	Source code
SpotBugs [33]	4.9.8	458	Bytecode
SonarQube [32]	8.0.1	590	Source code
Infer [11]	1.2.0	92	Compilation artifacts

2.2 Motivation and Insights

Bugs of Static Analyzers. Despite widespread adoption, static analyzers are prone to two types of inaccuracies. A *false positive* occurs when the analyzer reports a warning on code that does not actually violate the rule requirement. A *false negative* occurs when the analyzer fails to report a genuine violation. Both types of inaccuracies undermine developer trust and reduce the practical value of the tool [41]. Detecting and eliminating these inaccuracies is therefore essential for improving analyzer reliability.

```
1 class Foo {
2  public void foo() {
3   String s = "a";
4   if (s != null && s instanceof String) {
5    System.out.println(s); // Warning Reported
6   }}}
```

Listing 1: Seed program where PMD correctly reports a `SimplifyConditional` warning.

```
1 class Foo {
2  public void foo() {
3   String s = "a";
4   // Apply a semantic-preserving mutator
5   String s2 = "a";
6   if (s != null && s2 != null && s instanceof String){
7    System.out.println(s); // Warning not reported
8   }}}
```

Listing 2: Semantically equivalent program where PMD fails to report the warning (Issue-6513).

To further illustrate bugs in static analyzers, we present a concrete example of a false negative within PMD, a widely used static analyzer for Java. Listing 1 shows a Java method with a conditional statement. The statement checks if `s` is not null and if it is an instance of `String`. Since the `instanceof` operator inherently evaluates to false for null references, the explicit null check is logically redundant. PMD correctly identifies this redundancy and reports a `SimplifyConditional` warning. In Listing 2, a semantic-preserving code transformation is applied to the seed program. It declares a new non-null string `s2` and then inserts the condition `s2 != null` into the middle of the `AND` chain. Because `s2` is initialized to a valid string, this new condition always evaluates to true. The overall execution path remains exactly the same. Therefore, the null check for `s` remains completely redundant. However, PMD fails to report any warning on Listing 2. This omission constitutes a false negative.

This example reveals a systematic blind spot: the analyzer is sensitive to the syntactic form of a null check rather than its semantic meaning. To detect such bugs, metamorphic testing provides a general testing strategy. It defines measurable relations between inputs and checks whether the corresponding relations hold in the outputs. In this work, we instantiate this strategy with semantic equivalence. Specifically, we compare the reports of a single analyzer on a seed program and a behaviorally equivalent mutant produced by a semantic-preserving transformation. Any discrepancy in the reported warnings signals a potential analyzer bug. A key requirement of this strategy is to produce **semantic-preserving mutators** that can generate such equivalent program pairs.

Observation 1

Static analyzers are sensitive to syntactic variations of semantically equivalent code, and metamorphic testing with semantic-preserving mutators is essential to expose such blind spots.

Empirical Analysis of Real-World Refactoring Commits. The above study motivates the need for semantic-preserving mutators to effectively exercise diverse analyzer behaviors. To address this gap, we focus on real-world refactoring commits. During routine maintenance, developers frequently restructure code without altering its external behavior (e.g., annotated with `refactor` or `optimize`). We hypothesize that transformations derived from these commits elicit meaningful behavioral differences in static analyzers. By altering the structural patterns of the code, these changes effectively exercise different internal checking logic and analysis paths, making them a valuable mechanism for testing static analyzers.

To validate this hypothesis, we conduct a preliminary analysis. We extract 8,552 refactoring commits and their corresponding pre-/post- versions from well-known open-source Java repositories (see Section 3.1 for details). We then evaluate them against three popular source-level static analyzers: *SonarQube*, *PMD*, and *Checkstyle*. *Infer* and *SpotBugs* are excluded here, as compiling all repository versions is computationally expensive.

To characterize analyzer behavior, we focus on the concept of a rule being *covered*. A rule is applied only when the analyzed code satisfies specific structural conditions. Conversely, if the code lacks these structures, the analyzer simply skips the rule and it will never be covered. For example, the previously mentioned SpotBugs rule `NP_LOAD_OF_KNOWN_NULL_VALUE` requires a prior null check and a subsequent use of the variable. Without these structures, the

Table 2: Summary statistics of the rule coverage improvement (Δ) between the pre- and post- versions.

Analyzer	Q1	Median	Q3	Mean Δ
SonarQube	0.0	1.0	1.0	0.89
PMD	0.0	1.0	1.0	0.91
Checkstyle	1.0	2.0	2.0	1.49

rule remains uncovered. Therefore, a rule is **covered** when the code provides the required structures and the analyzer executes its internal checking logic. Since our goal is to evaluate whether transformations explore different analysis paths, we focus on changes in rule execution. Therefore, for each commit and selected analyzer, we calculate the symmetric difference in the set of **covered rules** between the before and after versions. We define this difference as the **rule coverage improvement** (Δ). For example, if the original code covers rules A and B, and the refactored code covers rules B and C, then Δ is 2 (rule A is no longer covered and rule C is newly covered).

Table 2 summarizes the distribution of Δ for each analyzer. For SonarQube, PMD, and Checkstyle, the median Δ ranges from 1.0 to 2.0. Across all three analyzers, the consistent trend is clear: a real-world refactoring commit effectively changes which rules are covered. This rule coverage improvement demonstrates that these refactoring-style modifications successfully steer the analyzers into different internal checking logic and new analysis paths. Since effective metamorphic testing relies on such structural variety to expose blind spots, transformations mined from refactoring commits serve as an ideal source of mutators. To ensure testing reliability, the semantic equivalence of derived mutators is empirically validated later during the Mutator Refinement phase (Section 3.2).

Observation 2

Real-world refactoring commits frequently alter the set of rules covered by static analyzers. This confirms that transformations mined from refactoring practice are a promising source for constructing semantic-preserving mutators capable of exposing analyzer bugs.

From Refactoring Commits to Mutators. The above findings show that refactoring commits offer diverse transformation patterns, but raw commit diffs are too noisy and context-dependent to serve directly as reusable mutators. Our key insight is to mine and generalize these patterns into executable transformation specifications. Figure 1 illustrates a refactoring commit from the Alibaba Druid repository, where a developer introduces a new variable `txInfo` and inserts a null check `txInfo != null` into an existing `AND` chain. We use an LLM to identify the core pattern: *insert '&& <non-null-var> != null' into an* `AND` *chain*, and distill it into an executable mutator. Applying this mutator to Listing 1 produces the mutant in Listing 2, which exposes the false negative in PMD. By mining such commits at scale, we construct a mutator set far more diverse than what manual design can achieve.

Ensuring Semantic Equivalence. However, not all refactoring commits truly preserve program semantics. Developers may bundle

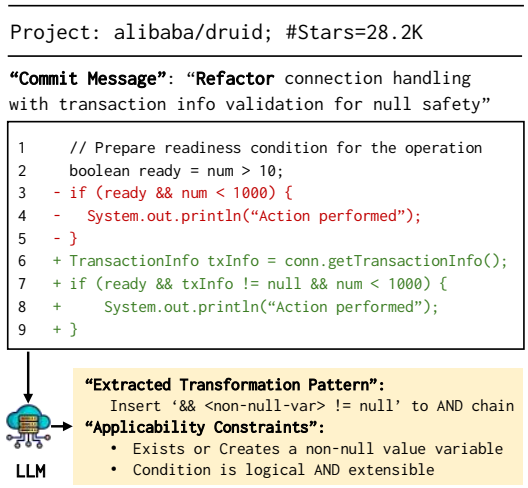

Figure 1: Illustration of a real-world refactoring commit and extracted transformation pattern.

incidental behavioral changes alongside restructuring edits. A mutator derived from such a commit introduces subtle behavioral deviations, producing spurious report differences that contaminate the test oracle. To address this, we design a dynamic behavioral equivalence strategy. As stated in prior work [41], semantic-preserving transformations must not alter the observable behavior of the program. Based on this principle, for each synthesized mutator, we apply it to a seed program and execute both the original and the mutated version under identical inputs. We then compare their observable runtime behaviors, including variable values, thrown exceptions, and standard output. Only mutators that exhibit full behavioral equivalence across all test executions are admitted into the final mutator set and used for large-scale metamorphic testing.

3 Approach

We describe the design of SAFuzzer. Figure 2 shows an overview of the three-phase pipeline: (1) *Mutator Invention*, which mines real-world refactoring commits and distills them into raw semantic-preserving mutators via an LLM; (2) *Mutator Refinement*, which validates each mutator's semantic-preservation through rigorous dynamic behavior validation; and (3) *Testing Static Analyzers*, which applies the validated mutators to test static analyzers via metamorphic testing. At a high level, SAFuzzer takes a large corpus of open-source Java repositories as input and produces a set of validated semantic-preserving mutators, which are then used to detect false negatives and false positives in production static analyzers.

3.1 Mutator Invention

Data Collection. The first step of SAFuzzer is to collect a large set of refactoring commits. Our current implementation targets the Java programming language. Specifically, we first gather 1,288 Java-primary repositories from GitHub, each with at least 100 stars and commit activity within the past five years. We then crawl the commit history of each repository's main branch. For each commit, we record the commit message, the full source code before and after

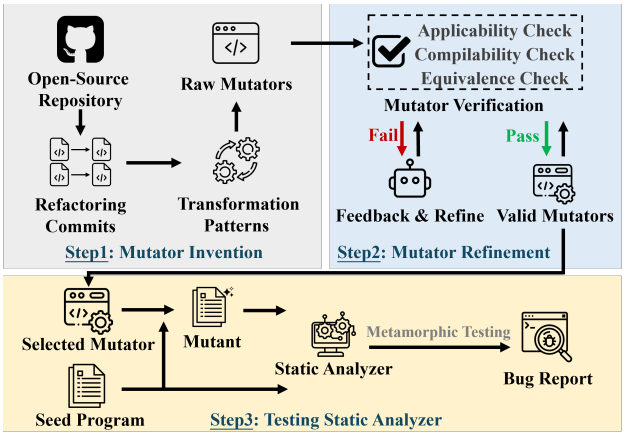

Figure 2: Overview of SAFuzzer.

```java
/* MutatorName.java */
public class [MutatorName] extends MutatorBase {
  @Override
  public String mutate() {
    ...
    // return the rewritten program content
    return getFullFileSource();
 }}

/* MutatorBase.java */
public abstract class MutatorBase {
 // return the updated file content
 abstract String mutate();
 // Several functions that perform AST analysis
 String getFullFileSource(){/* ... */}
 void sanitizeNewNodes(CtElement element){/* ... */}
 ...
}
```

Listing 3: Semantic-preserving mutator template.

the modification, and the corresponding diff file that captures the specific changes introduced. This process yields a total of 19,990 commits. Since our work focuses on refactoring at the function level, we first retain only those commits that modify a single Java method. We then apply a set of refactoring-related keywords (e.g., refactor, optimize) to filter commit messages.

To further verify the nature of each change, we employ a large language model to analyze the associated diff files. We exclude commits that clearly alter program semantics, such as those introducing new features or components. In addition, we perform deduplication by examining both commit messages and code modifications. Through this pipeline, we ultimately identify 8,552 commits that primarily involve code refactoring.

Note that we do not assume that all collected refactoring commits strictly preserve semantic equivalence. On the one hand, verifying semantic equivalence is challenging. This is because real-world code changes often rely on complex, project-specific execution environments that are difficult to reproduce and analyze automatically. On the other hand, our primary objective is to extract core transformation patterns to maximize the diversity of the mutators. If any mutators fail to maintain semantic equivalence, we try to fix or exclude them from the validated mutator set during the subsequent Mutator Refinement phase (see Section 3.2).

Transformation Pattern Extraction. Raw diffs are too noisy and context-dependent to serve directly as transformation patterns. A single commit diff often intermingles the core semantic-preserving pattern with unrelated edits such as whitespace adjustments, variable renames, and logging changes. Directly using such diffs as mutation templates would produce mutators that are either overly specific to one code snippet or contaminated by irrelevant modifications. Therefore, we prompt the LLM to isolate the essential transformation pattern from each commit.

Specifically, for each collected commit, we prompt the LLM with the refactoring commit and the source code before and after the modification. The prompt instructs the LLM to (1) identify the single core semantic-preserving pattern in the commit, ignoring unrelated changes; (2) generalize the pattern from concrete variable names and types to abstract placeholders; and (3) describe the applicability constraints under which the transformation is valid. The prompt template is shown in Figure 3a.

Raw Mutator Construction. The next step translates each natural-language transformation pattern into an executable, AST-level mutator. SAFuzzer prompts the LLM to generate a compilable Spoon-based mutator that traverses the AST, matches nodes satisfying the applicability constraints, and applies the transformation. The prompt requires the generated code to follow a predefined *mutator template* and to trigger the transformation only when the constraints are met. The prompt template is shown in Figure 3b. Listing 3 shows a simplified version of the mutator template. The template provides a base class `MutatorBase` that declares an abstract `mutate` method and a set of utility methods for common AST operations. Each concrete mutator extends this base class and overrides `mutate` to implement its specific transformation logic. This design ensures a uniform interface across all mutators, allowing the fuzzing loop to apply any mutator to any seed program through the same API.

3.2 Mutator Refinement

The raw mutators produced by the previous phase originate from real-world refactoring commits. However, their semantic-preserving properties have not been formally verified. The LLM may also introduce subtle errors during pattern generalization or code generation. These errors can silently break semantic equivalence. If a flawed mutator enters testing, each induced report difference becomes a false alarm. This makes the pipeline unreliable. Each raw mutator must undergo rigorous validation before it is admitted into the final mutator set.

However, verifying that a mutator preserves semantics across all possible inputs is undecidable in the general case. Semantic equivalence of two arbitrary programs is a well-known undecidable problem [29]. Rather than attempting a formal proof, we adopt a *dynamic behavioral equivalence* strategy: for each mutator, we apply it to a seed program and compare the observable runtime behaviors of the original and mutated versions. Only mutators that exhibit full behavioral equivalence are admitted into the validated mutator pool. This approach offers a high probability of semantic preservation.

Transformation Pattern Extraction Prompt

```
### You are a Java expert, given a refactoring
commit, your task is: (1) extract the core semantic-
preserving transformation pattern; (2) replace
concrete names or types with abstract placeholders;
(3) describe applicability constraints;
```

```
### Inputs
[Refactoring Commit], [Code Before], [Code After]
```

(a) Prompt template for transformation pattern extraction.

Raw Mutator Construction Prompt

```
### Please implement a mutator that performs
semantic-preserving code transformation using Spoon
framework, the mutator should: (1) be implemented
based on given mutator template; (2) follow the
given transformation pattern; (3) be applied only
when the applicability constraints are met.
```

```
### Inputs
[Transformation Pattern], [Mutator Template],
[Applicability Constraints]
```

(b) Prompt template for raw mutator construction.

Mutator Refinement Prompt

```
### The mutator failed validation at [Goal]. Your
task is to fix the mutator implementation so that
it: (1) correctly traverses and rewrites the Spoon
AST; (2) satisfies all applicability constraints;
(3) preserves the semantic behavior of the original
program.
```

```
### Inputs
[Current Implementation], [Feedback on Failure]
```

```
### accessible tools
[search_class] , [search_method], …
```

(c) Prompt template for mutator refinement.

Figure 3: Prompt templates used in mutator construction.

To systematically check for different categories of errors in the generated mutators, SAFuzzer decomposes validation into a sequence of *validation goals* with increasing stringency. Each goal targets a specific failure mode. SAFuzzer advances to the next goal only after the current one is satisfied. Table 3 summarizes the five validation goals along with the feedback provided upon failure.

Goals #1 – #3 require that the mutator itself compiles against the Spoon framework. When applied to seed programs, the mutator must terminate within the time budget and must not throw uncaught exceptions. **Goal #4** requires that the generated mutant P' compiles with the standard Java compiler. On failure, P, P', and the compilation error are jointly returned to the LLM for correction. **Goal #5** is the most challenging because it requires verifying that the mutant preserves the observable behavior of the original program. A mutator that passes the first four goals can still introduce subtle semantic deviations, such as altering a variable deep inside a loop or reordering side effects. Detecting such deviations demands fine-grained runtime comparison that goes beyond simple return-value checking.

Table 3: Staged validation goals for candidate mutator m. Goals are checked in order.

#	Validation Goal	Feedback on Failure
1	m compiles	<compilation error>
2	m terminates (no hang)	P, <timeout info>
3	m returns (no crash)	P, <stack trace>
4	m produces a compilable P'	P, P', <compilation error>
5	P' is behaviorally equivalent to P	P, P', <behavior diff>

To this end, SAFuzzer employs a two-level behavioral comparison protocol. For a candidate mutator m, we first select a seed program P that satisfies its applicability constraints and generate the mutant $P' = apply(m, P)$. SAFuzzer then generates N random inputs ($N = 20$ in our implementation) and executes both programs independently on each input. The mutator passes Goal #5 only if all N executions satisfy the following two levels of checks:

(1) **Coarse-grained observables**: The return values, thrown exceptions, and standard output/error streams must be identical between P and P'.
(2) **Fine-grained variable traces**: SAFuzzer instruments both P and P' to record the value of each local variable at the point where it leaves scope. For every variable shared by both the seed and the mutant, the recorded values must match. Variables introduced solely by the mutation (e.g., a newly declared auxiliary variable) are naturally excluded from comparison.

Refinement Loop. When any validation goal is unmet, SAFuzzer uses an LLM-based refinement agent to diagnose the failure and initiate a refinement iteration. The prompt template used for refinement is shown in Figure 3c. In each iteration, SAFuzzer supplies the refinement agent with three inputs: (1) the current mutator source code, (2) the index and description of the unmet goal, and (3) the corresponding diagnostic feedback specified in Table 3. The refinement agent leverages tools such as `searchClass` and `searchMethod` to query the Spoon API and refine the mutator accordingly. This loop continues until the mutator satisfies all five goals for a selected seed or the number of refinement iterations reaches R. We set $R = 10$ in our implementation. If the mutator still fails after R iterations, it is discarded.

Considerations on Semantic Equivalence. Despite our rigorous multi-stage validation process, we acknowledge that dynamic behavior verification cannot provide a theoretical guarantee of semantic equivalence. This limitation stems from the inherent incompleteness of the input space. For example, a mutator that inserts a *try-catch* block may appear semantically preserving for most executions. Yet its equivalence can fail on rare inputs if the statements inside the *try* block throw an exception that would otherwise propagate in the original program. For instance, an arithmetic expression may raise a division-by-zero exception only under a specially constructed input. If such inputs are absent from the N randomly generated test cases, the dynamic verification phase may incorrectly

accept the mutator as semantically equivalent. Hence, choosing a small N increases the chance of admitting non-equivalent mutators, while choosing a large N increases computational cost without substantially changing the outcomes. In our implementation, we select $N = 20$ based on an empirical convergence analysis. We vary N from 5 to 100 and measure how often the Goal #5 pass/fail verdict changes as N increases. Once N exceeds 20, the verdicts stabilize for more than 97% of mutators. Note that even a larger value of N does not fully guarantee that a mutator preserves semantics. Even if a non-equivalent mutator escapes the refinement phase, the resulting spurious inconsistency is discarded during the manual bug verification step before any report is submitted to developers.

3.3 Testing Static Analyzers

After collecting a set of validated semantic-preserving mutators, SAFuzzer enters its core fuzzing loop to systematically test the target static analyzers. Algorithm 1 details this process.

Given a corpus of seed programs $\mathcal{S}$ and a set of validated mutators $\mathcal{M}$, SAFuzzer iterates over each seed program $p \in \mathcal{S}$. For each seed, the framework performs up to $K = 50$ mutation attempts. In each attempt, a mutator m is randomly sampled from $\mathcal{M}$. If m is applicable to p (i.e., p contains at least one AST node satisfying m's preconditions), the mutator produces a mutant $p' = m(p)$. Both p and p' are then scanned by every target static analyzer A in the analyzer set $\mathcal{A}$ (e.g., *SpotBugs* and *Infer*), yielding analysis reports R_p and $R_{p'}$, respectively. The comparison of these reports serves as the metamorphic test oracle. Since p' is obtained from p via a semantic-preserving transformation, the two programs are expected to exhibit identical runtime behavior, and the analysis results should therefore remain consistent. Such an inconsistency indicates a potential false positive or false negative bug in the static analyzer. All detected inconsistencies are forwarded to human reviewers through `manual_check`. Only inconsistencies confirmed as real analyzer bugs are added to the report set $\mathcal{B}$.

Bug Verification and Reporting. To ensure the quality of bug reports, we conduct a rigorous manual verification process before submitting any bug report. As shown in Listing 2, the `SimplifyConditional` rule requires flagging all redundant `null` checks preceding an `instanceof` expression. If the static analyzer fails to issue a warning for this violation after the insertion of a semantically-preserving code snippet, we classify it as a confirmed false negative. In addition, seed programs undergo multiple rounds of mutation during the fuzzing loop, which may result in overly lengthy test cases that hinder developers from identifying the root cause of bugs.

To minimize developer effort, we manually reduce confirmed test cases before submitting the bug report. Specifically, we first identify the root cause that causes the inconsistency of analyzers and then remove irrelevant code. The final minimal test case meets a key requirement: removing any remaining statement will prevent the bug from being reproduced. This process generates concise, self-contained test cases that clearly demonstrate the root cause of bugs, enabling static analyzer developers to conduct efficient bug triage and resolution.

Algorithm 1: The fuzzing loop of SAFuzzer.

Input: A Corpus of Seed Programs: $\mathcal{S}$; Validated Mutators: $\mathcal{M}$; Static Analyzers: $\mathcal{A}$; Max Mutations Per Seed: K
Output: Bug Reports: $\mathcal{B}$

$\mathcal{B} \leftarrow \emptyset$;
foreach $p \in \mathcal{S}$ **do**
  for $i \leftarrow 1$ **to** K **do**
    $m \leftarrow$ random_select($\mathcal{M}$);
    if *is_applicable*(m, p) **then**
      $p' \leftarrow$ apply(m, p);
      foreach $A \in \mathcal{A}$ **do**
        $R_p \leftarrow$ analyze(A, p);
        $R_{p'} \leftarrow$ analyze(A, p');
        /* compare the differences between reports */
        if $R_p \neq R_{p'}$ **then**
          $b \leftarrow$ manual_check$(p, p', R_p, R_{p'})$;
          if $b \neq \perp$ **then**
            $\mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$;
    $p \leftarrow p'$
return $\mathcal{B}$;

3.4 Implementation

We implement SAFuzzer as an extensible testing framework for Java static analyzers, comprising approximately 26,364 lines of Python and Java code, including a core framework of 5,751 lines and 313 mutators totaling 20,613 lines.

For all LLM-assisted mutator invention and refinement phases (i.e., including transformation pattern extraction, raw mutator construction, and iterative refinement), we use Qwen3-Max [38] as the backbone model. We select Qwen3-Max for its strong code comprehension and generation capabilities, as well as its ability to follow structured prompts reliably [38]. All mutators are built on top of the Spoon framework, which provides fine-grained AST querying and rewriting support for Java programs.

4 Evaluation

The evaluation is guided by the following research questions:

- **RQ1 (Mutator Synthesis):** How many mutators can SAFuzzer synthesize, and what does the invention and validation process cost?
- **RQ2 (Bug Detection of SAFuzzer):** How effective is SAFuzzer in detecting real bugs in production static analyzers?
- **RQ3 (Comparison with Baselines):** How does SAFuzzer compare with existing baselines in terms of bug detection and the rule coverage of the analyzers?

4.1 Experiment Setup

Target Static Analyzers and Seed Programs. We evaluate SAFuzzer on five widely used Java static analyzers listed in Table 1: Checkstyle [4], PMD [26], SpotBugs [33], SonarQube [32], and Infer [11]. These analyzers collectively represent different analysis strategies and cover a wide range of rules as a comprehensive evaluation target. We use the versions listed in Table 1 for all analyzers. For the seed pool, we collect programs from the official regression test suites of static analyzers, following prior work [41].

Baselines. We compare SAFuzzer with the following baselines:

Table 4: Mutator synthesis pipeline of SAFuzzer. The 'Pass Rate' is calculated out of 8,552.

Pipeline Stage	Count	Pass Rate
Mined refactoring commits	8,552	-
Extracted Transformation Pattern	8,552	-
Generate raw mutators	8,552	-
Pass mutator itself compilation (Goal #1-#3)	4,789	56.0%
Pass mutant compilation (Goals #4)	2,162	25.3%
Pass semantic-equivalence (Goals #5)	313	3.7%

- **Statfier** [41]: a mutation-based fuzzer that applies 12 manually curated semantic-preserving transformations to test static analyzers via metamorphic testing.
- **He et al.** [15]: an automated oracle construction approach that uses dynamic execution to validate the presence of runtime errors. Since this approach originally targets C/C++, we re-implement it for Java by mapping its core oracle logic to equivalent Java runtime checkers (e.g., checking `NullPointerException` dynamically). To verify our migration, we evaluate it on 30 known Java analyzer bugs and confirm it reproduces the expected issues.
- **Fuzz4All** [36]: an LLM-based universal fuzzer that generates diverse test inputs for various software systems. To adapt it for metamorphic testing, we configure it to use only its *semantic-equiv* generation strategy.
- **WhiteFox** [39]: an LLM-based fuzzer that uses white-box information from compilers to guide test generation. To adapt it, we modify its system prompt to generate semantic-preserving mutants from seed programs.
- **LLM-Direct**: a direct LLM baseline using the same Qwen3-Max backbone. It asks the model to produce semantic-preserving patterns and mutators without refactoring commits. It then uses the same validation and refinement pipeline as SAFuzzer.

For a fair comparison, all tools use the same seed pool, target the same set of static analyzers, and share a 24-hour time budget.

4.2 RQ1: Mutator Synthesis

Overview of Mutators. Table 4 traces each candidate through the synthesis pipeline. Starting from 8,552 refactoring commits mined from open-source repositories (Section 3.1), SAFuzzer prompts the LLM to generate one candidate mutator per commit. Among the 8,552 initial candidates, 4,789 (56.0%) compile successfully against the Spoon framework and pass Goals #1–#3. From these compilable mutators, 2,162 (25.3%) successfully produce compilable mutants when applied to the seed programs (Goal #4). The remaining candidates undergo rigorous behavioral equivalence checking (Goal #5), resulting in **313 validated mutators** that pass all five staged validation goals, yielding an overall validation rate of 3.7%.

The refinement loop (Section 3.2) is critical for recovering mutators that initially fail validation. Among the validated mutators, 67.4% require at least one round of LLM-assisted refinement, averaging 2.71 iterations per repaired mutator. This suggests that the staged feedback mechanism is effective. Most fixable errors are resolved within a few iterations, as the LLM successfully uses structured diagnostics to produce targeted fixes.

Analysis of Invalidated Mutators. To understand why many candidates fail validation, we analyze their failure modes. Among the 3,763 (8,552-4,789) mutators that fail compilation, the most common issues are (1) incorrect usage of Spoon API methods (42.1%), (2) mismatched AST node type checks (31.5%), and (3) logical errors in applicability constraint checks (26.4%). For mutators that compile but fail behavioral equivalence, we observe two primary causes: (1) the transformation inadvertently alters control flow or side effects in edge cases not covered by the seed program (78.3%), and (2) the LLM over-generalizes the pattern, introducing changes that are not truly semantic-preserving (21.7%). We illustrate this with a concrete example. The following two boxes show a transformation pattern produced by the LLM that is clearly not semantic-preserving. Mutators derived from such patterns will be rejected during the behavioral equivalence checking stage (Goal #5).

StreamLogging: "*Add logging to a lambda expression within a stream operation to track the number of processed elements and filter results.*".

ExplicitNullInitializationRemoval: "*Remove explicit code that initializes a variable to null*".

Generation Costs. SAFuzzer is highly cost-effective. Processing 8,552 candidate mutators requires 31,823 LLM API calls. This consumes 105.6 million input and 42.6 million output tokens, averaging 3,318 input and 1,338 output tokens per call. Under Qwen3-Max [38] pricing, each validated mutator costs approximately $0.04. This estimate includes failed raw mutators, failed refinement attempts, and candidates discarded by the validation goals. Synthesizing the entire mutator set takes 38.2 wall-clock hours. LLM response wait times account for 81.2% of this duration. Compilation and dynamic testing occupy the remaining time.

Finding 1

SAFuzzer successfully synthesized 313 validated semantic-preserving mutators from 8,552 real-world refactoring commits, costing $0.04 and requiring 2.71 refinement iterations on average. This demonstrates the practicality of our LLM-assisted approach.

4.3 RQ2: Bug Detection of SAFuzzer

Detected Bugs. To evaluate bug-finding capability in the wild, we deployed SAFuzzer in an extended fuzzing campaign over a month. We mark an issue as *confirmed* only when developers acknowledge it as a genuine bug through comments, labels, or fixes. An unconfirmed issue is still pending developer review. Table 5 summarizes the bugs detected by SAFuzzer across the five target static analyzers during this period. In total, SAFuzzer reports 42 bugs, including 34 false negatives and 8 false positives, to the respective bug trackers. Developers confirm 38 of these reports, and 9 have already been fixed, showing that most reported issues are genuine and that several lead to fixes. SpotBugs receives the most reports (18), followed by PMD (10) and Infer (8). By identifying both false positives (spurious warnings) and false negatives (missed

Table 5: Status of bugs detected by SAFuzzer.

Status	SpotBugs	PMD	Infer	SonarQube	Checkstyle	Overall
Reported	18	10	8	4	2	**42**
Confirmed	17	8	8	3	2	**38**
Fixed	3	4	0	0	2	**9**
Won't Fix	1	2	0	1	0	**4**
FP	5	1	2	0	0	**8**
FN	13	9	6	4	2	**34**

violations), SAFuzzer demonstrates its capability to expose diverse inaccuracies in static analyzers.

Manual Validation Effort. To ensure the reliability of the bug reports, we manually inspect all inconsistencies to understand their root causes. In our experiments, SAFuzzer detects 45 inconsistencies, and the inspection takes about 5 hours. Of these 45 cases, only 3 (6.7%) are filtered out because the 3 corresponding mutators fail to preserve semantic equivalence under specific conditions. For example, one rejected case rewrites `Boolean.TRUE.equals(flag)` as `flag == true`. This rewrite is not equivalent when `flag` is null. The rewritten expression throws a `NullPointerException` due to unboxing, whereas the original expression safely returns false. The remaining 42 cases (93.3%) are reported to developers.

We further analyze the confirmed bugs. A recurring pattern is *syntactic sensitivity*: analyzers often fail to recognize semantically equivalent code. For instance, issue PMD-6513 (Listing 2) shows that inserting a redundant non-null conjunct into an `AND` chain prevents the `SimplifyConditional` rule from triggering, despite the unchanged logic. For another example, Issue SpotBugs#3886 demonstrates the following behavior. Swapping the two operands in a check for odd numbers using the modulus operator from "`i % 2 == 1`" to "`1 == i % 2`" leaves the program logic fully equivalent. However, this syntactic change prevents the `IM_BAD_CHECK_FOR_ODD` rule from being triggered. This confirms that static analyzers are brittle to syntactic variations, and our semantic-preserving mutators effectively expose such blind spots.

Effective Mutators. To identify which transformation patterns are most effective at uncovering analyzer bugs, we analyze the 42 reported bugs and trace each back to the mutator that exposed it. Note that a single bug may correspond to multiple mutators, as SAFuzzer applies mutators to seed programs iteratively.

Table 6 lists the top 10 mutators. The most effective ones are *EqualityCheckToInstanceof*, *ConditionalBlockInsertion*, and *ParenthesesAddition*, each of which triggers 4 bugs across multiple analyzers. Notably, these mutators target fundamentally distinct aspects of analyzer logic: the former rewrites type-checking idioms, while the latter inserts structurally redundant elements. Their high effectiveness indicates that analyzers are vulnerable to a broad range of syntactic changes rather than just a single type of rewrite.

Further investigation reveals two recurring themes among the top-ranked mutators: First, *condition-related* mutators dominate. Six of the top 10 mutators, such as *ConditionalBlockInsertion* and *IfConditionReordering*, manipulate conditional expressions or control flow guards. This reveals a systematic weakness: analyzers easily miss violations or introduce spurious warnings upon minor syntactic rearrangements of logically equivalent conditions. Second,

Table 6: The top 10 mutators involved in the 42 bug-triggering test cases.

Mutator Name	#	Triggered Bugs
`EqualityCheckToInstance.`	4	SB#3916, IF#2001, SQ#S2259, PMD#6513
`ConditionalBlockInsert.`	4	SB#3894, IF#2015, SB#3929, PMD#6518
`ParenthesesAddition`	4	SB#3904, IF#2015, PMD#6491, CS#19162
`IfConditionReordering`	3	SB#3886, SB#3920, SB#3963
`VariableAddition`	3	SB#3884, IF#2015, IF#1993
`NullCheckReordering`	3	SB#3920, SB#3886, SB#3916
`ConditionalLogicInsert.`	2	PMD#6518, SB#3978
`MethodChainCallSwap`	2	SB#3966, PMD#6494
`ConditionNegation`	2	PMD#6435, SB#3963
`SingleLineReturnToBlock`	2	PMD#6491, PMD#6519

Table 7: Sensitivity of SAFuzzer to different LLM backbones. Variants are evaluated on the same 200 refactoring commits.

Variant	#Mutators	Pass Rate	#Bugs	Avg. Cost
SAFuzzer$_{GPT}$	9	4.50%	2	$0.09
SAFuzzer$_{DS}$	6	3.00%	1	$0.05
SAFuzzer	11	5.50%	3	$0.04

structurally additive mutators are highly effective. For example, *ParenthesesAddition* reveals 4 bugs by adding redundant parentheses without altering data or control flow. This confirms that some rules rely excessively on surface-level syntactic structures over semantic properties.

Sensitivity to Backbone LLM. SAFuzzer uses Qwen3-Max (snapshot `qwen3-max-2025-09-23`) as the backbone LLM for pattern extraction, mutator construction, and refinement. To evaluate whether its effectiveness generalizes across LLM backbones, we construct two variants by replacing Qwen3-Max with alternative models. The first variant, SAFuzzer$_{GPT}$, uses GPT-4o [22] (snapshot `gpt-4o-2024-11-20`). The second variant, SAFuzzer$_{DS}$, uses DeepSeek-V3.2 [8] (snapshot `DeepSeek-V3.2-2025-12-01`). We evaluate all variants under the same setup. For each, we randomly sample 200 refactoring commits, synthesize mutators, deploy them in the fuzzing loop for 24 hours, and measure their effectiveness.

As Table 7 shows, SAFuzzer is not tightly coupled to a specific LLM. All three models successfully produce validated mutators that detect real bugs. Performance differences stem from code generation quality: Qwen3-Max and GPT-4o generate more compilable, semantically correct Spoon mutators, yielding higher validation pass rates. However, even the weakest model (DeepSeek-V3.2) detects new bugs affordably, meaning practitioners can swap LLMs based on budget or availability without sacrificing core capabilities.

Finding 2

SAFuzzer detected 42 bugs (34 false negatives and 8 false positives) across five production static analyzers, with 38 confirmed and 9 fixed. Moreover, SAFuzzer is not sensitive to the LLM backbone, as all three evaluated models successfully produce bug-detecting mutators.

4.4 RQ3: Comparison with Baselines

In this RQ, we compare SAFuzzer with the five baselines described in Section 4.1 in terms of bug detection capability and rule coverage improvement. We run each tool for the same 24-hour time budget using the same seed pool and target the same five static analyzers.

Bug Detection Capability. Figure 4 shows the bug detection results under the same 24-hour setting for all baselines. SAFuzzer detects 12 bugs, compared to Statfier (8 bugs), WhiteFox (7 bugs), Fuzz4All (5 bugs), LLM-Direct (4 bugs), and He et al. (3 bugs). This advantage stems from the diversity of SAFuzzer's mutators. Patterns mined from real-world refactoring commits exercise syntactic variations that fixed rule sets (Statfier), direct LLM-designed transformations (LLM-Direct), and general-purpose LLM generators (Fuzz4All, WhiteFox) cannot reproduce. For example, PMD#6513 (Listing 2) is revealed by inserting a redundant conjunct into an `AND` chain, a transformation directly learned from developer practice and absent from all baselines. Furthermore, while Fuzz4All and WhiteFox can produce diverse code, they lack semantic equivalence guarantees, which limits the reliability of their metamorphic oracles. LLM-Direct's lower bug count shows that refactoring commits provide useful transformation evidence beyond unguided LLM generation. The staged results further explain this gap. LLM-Direct generates 520 raw mutators as the starting pool. Among them, 296 (56.9%) pass mutator compilation and the structural checks in Goals #1 to #3. Only 140 (26.9%) produce compilable mutants in Goal #4, and 16 (3.1%) pass semantic equivalence validation in Goal #5. Thus, direct prompting yields many candidates, but most are filtered out before they become validated mutators. Overall, these results confirm that SAFuzzer achieves the strongest bug detection capability among all evaluated tools.

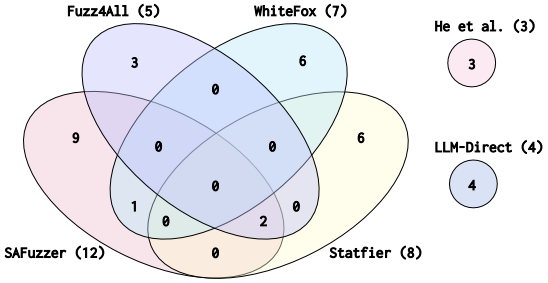

Figure 4: Comparison of bug detection capability.

Rule Coverage Improvement. Rule coverage measures the comprehensiveness of a testing approach by reflecting its ability to exercise diverse checking logic encoded in the analyzer's predefined rules. We report the distribution of the Rule Coverage Improvement metric Δ across all (seed, final mutant) pairs. Because the generated mechanisms differ across tools, we select the final mutant $m^\star$ for each tool as follows: $m^\star$ as the $50th$ mutant for SAFuzzer and LLM-Direct, the $30th$ mutant for Statfier, the $10th$ mutant for He et al., and the $100th$ mutant for Fuzz4All. For WhiteFox, $m^\star$ is defined as the $1st$ mutant generated by the LLM. Since each seed undergoes a sequence of mutations, the final mutant better reflects the cumulative complexity that each tool is designed to produce. For LLM-Direct, we use the same depth as SAFuzzer to isolate the effect of refactoring-derived transformations. To ensure a fair comparison, all tools operate under a uniform 24-hour time budget.

Table 8: Percentage of first-valid mutants that trigger at least one new analyzer rule ($\Delta > 0$).

Tool	Checkstyle	PMD	SpotBugs	SonarQube	Infer
SAFuzzer	46.8%	52.6%	48.3%	49.1%	22.8%
Statfier	19.6%	24.8%	15.9%	20.7%	10.6%
Fuzz4All	25.4%	30.7%	21.2%	28.4%	14.9%
WhiteFox	34.1%	36.9%	17.5%	31.6%	16.4%
He et al.	8.7%	7.4%	6.8%	6.5%	5.7%
LLM-Direct	22.3%	23.5%	13.6%	19.8%	9.8%

As shown in Figure 5, SAFuzzer achieves the highest median Δ across all five analyzers including the LLM-Direct comparison. For example, on PMD, SAFuzzer reaches a median of 48, outperforming WhiteFox (42), Fuzz4All (36), Statfier (33), LLM-Direct (39), and He et al. (7). This advantage remains consistent on the other analyzers. On SonarQube, SAFuzzer attains a median of 44, compared to 36 for the best baseline, WhiteFox. On Checkstyle, SpotBugs, and Infer, SAFuzzer achieves median Δ values of 33, 39, and 11, respectively, outperforming the best baselines in each case. Across Checkstyle, PMD, SpotBugs, SonarQube, and Infer, LLM-Direct obtains median Δ values of 31, 39, 29, 30, and 7, respectively. It remains below SAFuzzer on every analyzer. He et al. exhibit the lowest coverage overall, as their oracle construction approach targets narrow runtime properties rather than exercising diverse syntactic forms. These results confirm that the refactoring-derived mutators of SAFuzzer induce broader behavioral variation in static analyzers than both manually curated and LLM-generated transformations.

Normalized Depth-1 Transformation Quality. To quantify transformation quality at the same mutation depth, we also conduct a normalized depth-1 experiment. Specifically, we fix the mutation depth to one for SAFuzzer and all baselines. All tools use the same seed pool, target the same five analyzers, and run under the same 24-hour budget. For each seed, we keep only the first valid mutant generated by each tool and compare its analyzer-rule set with that of the original seed. If the mutant triggers at least one new rule ($\Delta > 0$), we count it as a successful first-valid mutant. We report the percentage of such mutants to measure how often a single semantic-preserving step creates analyzer-visible structural variation.

Table 8 shows that SAFuzzer achieves the highest new-rule coverage rate on all five analyzers. Its rate ranges from 22.8% on Infer to 52.6% on PMD. In contrast, the best baseline only reaches 34.1% on Checkstyle, 36.9% on PMD, 21.2% on SpotBugs, 31.6% on SonarQube, and 16.4% on Infer. Statfier is limited by a small set of manually designed transformations. Fuzz4All, WhiteFox, and LLM-Direct can generate diverse code, but their first valid mutants less consistently produce semantics-preserving structural changes around the seed. These results show that SAFuzzer's advantage comes from refactoring-derived mutators that expose new analyzer logic in a single semantic-preserving mutator.

Finding 3

SAFuzzer can detect the most bugs and achieves the highest rule coverage improvement compared to baselines.

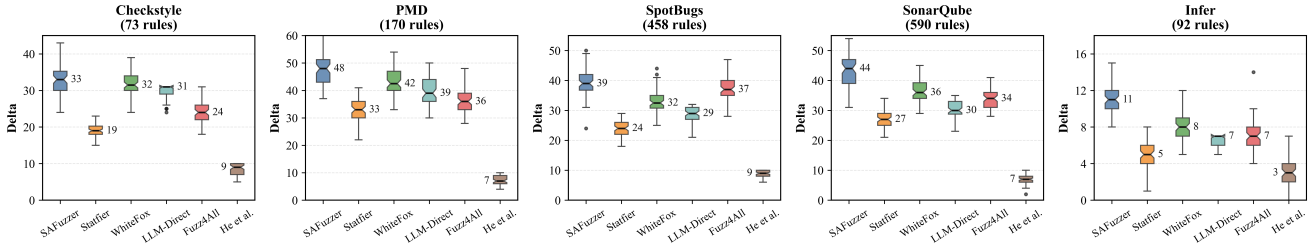

Figure 5: Comparison of rule coverage improvement (Δ).

5 Discussion

5.1 Generalizability of SAFuzzer

The core design of SAFuzzer is built on two language-dependent components: the AST manipulation layer (currently Spoon for Java) and the refactoring commit corpus. Higher-level components, including the LLM-based transformation pattern extraction, the staged validation pipeline, and the metamorphic testing loop, are entirely language-agnostic. When adapting to a new language, the process involves replacing the AST framework (e.g., Tree-sitter [2]), collecting refactoring commits from repositories written in the target language, and re-running the mutator invention and refinement phases. Because the LLM generalizes transformation patterns from natural-language descriptions rather than language-specific syntax, the extracted patterns remain largely transferable.

5.2 Threats to Validity

External Validity. The primary external threat concerns the representativeness of the refactoring commits used to construct our mutator set. We collect 8,552 refactoring commits from 1,288 open-source Java repositories on GitHub, filtered by star count and recent activity. Although this corpus is large and diverse, it may not fully represent refactoring practices in proprietary or domain-specific codebases. Moreover, as noted in Section 3.1, we do not assume that all collected commits strictly preserve semantic equivalence; some may bundle behavioral changes alongside structural edits. To mitigate this, our Mutator Refinement phase (Section 3.2) filters out any mutator that fails to preserve observable behavior, so impure commits do not propagate into the final mutator pool.

Internal Validity. The main internal threat lies in the soundness of our semantic equivalence verification. Because verifying semantics across all possible inputs is undecidable in the general case [29], dynamic validation may still incorrectly admit flawed mutators if the inputs fail to cover corner-case divergences. To mitigate this, as discussed in Section 3.2, we empirically tune the number of randomized inputs to effectively balance validation reliability with computational cost. Additionally, we manually inspect all candidate inconsistencies to confirm they arise from genuine analyzer bugs rather than from mutator-induced semantic changes before submitting reports to developers.

6 Related Work

Testing Static Analyzers. Testing static analyzers requires addressing the complex oracle problem and syntactic sensitivity [3, 20]. Statfier [41] detects analyzer inconsistencies via metamorphic testing by applying manually predefined semantic-preserving transformations to seed programs. Zhang et al. study annotation-induced faults in static analyzers [42] and characterize program representation faults of static analysis frameworks. Additionally, other approaches evaluate analyzer soundness and track warnings across versions by constructing differential benchmarks or applying dynamic filtering engines [16, 24]. While existing works rely on manually predefined rules, SAFuzzer mines real code evolution history to generate diverse semantic-preserving mutators.

LLM-based Generative Fuzzing. Recent advancements leverage large language models to automate and enhance software testing across various domains. Fuzz4All [36] leverages LLMs and dynamic autoprompting to generate highly diverse edge-case test inputs for various compilers across multiple languages. KNighter [40] transforms static analysis by using LLMs to synthesize new checkers for detecting kernel bugs, while Jiang et al. [17] enhance existing analyzer rules via fact-aligned, template-constrained LLM generation. Additional studies leverage neural networks for probabilistic structure generation or employ iterative feedback loops to resolve compilation and semantic bugs in model-generated code [7, 27]. By using LLMs solely to extract deterministic rules during mutator construction, SAFuzzer not only mitigates semantic hallucinations but also significantly reduces token consumption.

7 Conclusion

We are the first to systematically mine real-world refactoring commits to test static analyzers. We present SAFuzzer in this study, a novel metamorphic testing framework that distills these commits into diverse and verified mutators using large language models and dynamic validation. SAFuzzer identifies 42 new bugs in widely used Java static analyzers. Crucially, our approach successfully exposes structural blind spots that existing fuzzing baselines miss. We believe this work provides a practical foundation for building more reliable static analysis tools.

8 Data Availability Statement

All artifacts of this study have been made publicly available at: https://doi.org/10.5281/zenodo.21722288.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (No.62372193, No.U2436207, No.624B2067, No.62572081 and No.92582110), and the Major Technological Innovation Program of HUST (No. 2025ZDKJCX05).

References

[1] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER, Suita, Osaka, Japan, March 14-18, Volume 1*. IEEE Computer Society, 470–481. doi:10.1109/SANER.2016.105

[2] Max Brunsfeld et al. 2026. Tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io/tree-sitter/ Accessed: 2026-03.

[3] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An Empirical Study of Static Analysis Tools for Secure Code Review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Vienna, Austria, September 16-20, 2024*. ACM, 691–703. doi:10.1145/3650212.3680313

[4] CheckStyle. 2026. CheckStyle, a Java static analysis tool. https://checkstyle.org. Accessed: 2026-03.

[5] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[7] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Amsterdam, The Netherlands, July 16-21*. ACM, 95–105. https://doi.org/10.1145/3213846.3213848

[8] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, et al. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. arXiv:2512.02556 [cs.CL] https://arxiv.org/abs/2512.02556

[9] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of Object-Oriented Programming, 20th European Conference, Nantes, France (Lecture Notes in Computer Science, Vol. 4067)*. Springer, 404–428. doi:10.1007/11785477_24

[10] Carlo Dimastrogiovanni and Nuno Laranjeiro. 2016. Towards Understanding the Value of False Positives in Static Code Analysis. In *Proceedings of the Seventh Latin-American Symposium on Dependable Computing, LADC, Cali, Colombia, October 19-21*. IEEE Computer Society, 119–122. https://doi.org/10.1109/LADC.2016.25

[11] Facebook. 2026. Infer, A tool to detect bugs in Java and C/C++/Objective-C code before it ships. https://fbinfer.com/. Accessed: 2026-03.

[12] Iván García-Ferreira, Carlos Laorden, Igor Santos, and Pablo García Bringas. 2016. Static analysis: a brief survey. *Log. J. IGPL* 24, 6 (2016), 871–882. https://doi.org/10.1093/jigpal/jzw042

[13] Hristina Gulabovska and Zoltán Porkoláb. 2019. Survey on Static Analysis Tools of Python Programs. In *Proceedings of the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA, Ohrid, North Macedonia, September 22-25 (CEUR Workshop Proceedings)*. CEUR-WS.org. https://ceur-ws.org/Vol-2508/paper-gul.pdf

[14] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE, Montpellier, France*. ACM, 317–328. doi:10.1145/3238147.3238213

[15] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and understanding defects in static analyzers by constructing automated oracles. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1656–1678. doi:10.1145/3660781

[16] Gábor Horváth, Réka Nikolett Kovács, and Péter Szécsi. 2020. Report on the Differential Testing of Static Analyzers. *Acta Cybern.* 25, 4 (2020), 781–795. https://doi.org/10.14232/actacyb.282831

[17] Zongze Jiang, Ming Wen, Ge Wen, and Hai Jin. 2025. Fact-Aligned and Template-Constrained Static Analyzer Rule Enhancement with LLMs. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 1642–1654. doi:10.1109/ASE63991.2025.00138

[18] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the ICSE*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[19] 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, ESEC/FSE, San Francisco, CA, USA*. ACM, 921–933. doi:10.1145/3611643.3616262

[20] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. 2023. A Comprehensive Study on Quality Assurance Tools for Java. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 285–297. doi:10.1145/3597926.3598056

[21] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Trans. Software Eng.* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[22] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2026-07.

[23] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2022. A Survey of Parametric Static Analysis. *ACM Comput. Surv.* 54, 7 (2022), 149:1–149:37. https://doi.org/10.1145/3464457

[24] Ivan Pashchenko, Stanislav Dashevskyi, and Fabio Massacci. 2017. Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, Toronto, ON, Canada*. IEEE Computer Society, 163–168. https://doi.org/10.1109/ESEM.2017.24

[25] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Software—Practice & Experience* 46, 9 (2016), 1155–1179. https://doi.org/10.1002/spe.2346

[26] PMD. 2026. PMD, a Java static analysis tool. https://pmd.github.io. Accessed: 2026-03.

[27] Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. LLMLOOP: Improving LLM-Generated Code and Tests Through Automated Iterative Feedback Loops. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME, Auckland, New Zealand*. IEEE, 930–934. https://doi.org/10.1109/ICSME64153.2025.00109

[28] New Relic. 2024. 2024 State of the Java Ecosystem. https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem. Accessed: 2026-03.

[29] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366. doi:10.1090/S0002-9947-1953-0053041-6

[30] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[31] Danilo Silva, Nikolaos Tsantalis, and Marco Túlio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. *CoRR* abs/1607.02459 (2016). arXiv:1607.02459 https://arxiv.org/abs/1607.02459

[32] SonarSource. 2026. SonarQube, a Java static analysis tool. https://www.sonarqube.org. Accessed: 2026-03.

[33] SpotBugs. 2026. SpotBugs, a Java static analysis tool. https://spotbugs.github.io. Accessed: 2026-03.

[34] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[35] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE, Gothenburg, Sweden*. ACM, 483–494. doi:10.1145/3180155.3180206

[36] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 483–494. doi:10.1145/3180155.3180206

[37] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25, 2 (2020), 1419–1457. doi:10.1007/s10664-019-09750-5

[38] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[39] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735. doi:10.1145/3689736

[40] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*. ACM, 655–669. doi:10.1145/3731569.3764827

[41] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE, San Francisco, CA, USA, December 3-9, 2023*. ACM, 237–249. doi:10.1145/3611643.3616272

[42] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proc. ACM Softw. Eng.* 1, FSE (2024), 722–744. doi:10.1145/3643759