Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations

Huaien Zhang
Southern University of Science and Technology, Shenzhen
The Hong Kong Polytechnic University, Hong Kong
China
cshezhang@comp.polyu.edu.hk

Yu Pei
The Hong Kong Polytechnic University
Hong Kong, China
csypei@comp.polyu.edu.hk

Junjie Chen
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Shin Hwei Tan*
Concordia University
Montreal, Canda
shinhwei.tan@concordia.ca

ABSTRACT

Static analyzers reason about the behaviors of programs without executing them and report issues when they violate pre-defined desirable properties. One of the key limitations of static analyzers is their tendency to produce inaccurate and incomplete analysis results, i.e., they often generate too many spurious warnings and miss important issues. To help enhance the reliability of a static analyzer, developers usually manually write tests involving input programs and the corresponding expected analysis results for the analyzers. Meanwhile, a static analyzer often includes example programs in its documentation to demonstrate the desirable properties and/or their violations. Our key insight is that we can reuse programs extracted either from the official test suite or documentation and apply semantic-preserving transformations to them to generate variants. We studied the quality of input programs from these two sources and found that most rules in static analyzers are covered by at least one input program, implying the potential of using these programs as the basis for test generation. We present Statfier, a heuristic-based automated testing approach for static analyzers that generates program variants via semantic-preserving transformations and detects inconsistencies between the original program and variants (indicate inaccurate analysis results in the static analyzer). To select variants that are more likely to reveal new bugs, Statfier uses two key heuristics: (1) *analysis report guided location selection* that uses program locations in the reports produced by static analyzers to perform transformations and (2) *structure diversity driven variant selection* that chooses variants with different program contexts and diverse types of transformations. Our experiments with five popular static analyzers show that Statfier can find 79 bugs in these analyzers, of which 46 have been confirmed.

*Corresponding author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616272

CCS CONCEPTS

• **Software and its engineering → Software reliability**.

KEYWORDS

software testing, rule-based static analysis, program transformation

ACM Reference Format:
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 '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616272

1 INTRODUCTION

Static analyzers check the behaviors of programs without executing them to learn about their properties. When a program's behavior violates a desirable property, a static analyzer produces a warning—or reports an issue—against the violation. Since they do not need to run the programs under analysis, static analyzers are highly applicable in various situations and have been widely used to ensure the standard compliance, performance, reliability, etc., of programs in various stages of their development processes.

As with other types of software systems, testing has been an indispensable way to help developers ensure that a static analyzer correctly implements its functionalities (i.e., performs its analyses as expected). For example, the developers of PMD, a popular static analyzer that detects performance and code style issues in programs, manually crafted 2,983 programs as the input to verify the tool's behavior via testing. As manually writing tests for static analyzers is tedious and expensive, various approaches have been developed in the past few years to automatically test static analyzers as black-box systems [19, 45, 67, 72]. For instance, prior work [72] relies on feeding the same manually written input program to a pair of static analyzers implementing similar rules to detect inconsistencies between them. Although existing approaches have helped developers detect various bugs in analyzers, the bug detection capacities of manually written or randomly generated input programs are insufficient because the number of viable ways to implement certain functionalities by combining different language constructs is typically huge, while the number of those combinations covered by the test programs of these approaches is often small.

We identify two challenges in the automated testing of static analyzers. The first challenge (**C1**) has to do with the *unavailability of automated test oracles* (i.e., given execution of a static analyzer, it is not always possible to *automatically* determine whether the produced reports are correct or not). The test oracle problem is universal to the automated testing of many different types of systems, and a popular technique to partially address the problem is differential testing, which essentially utilizes the behaviors of similar applications or different implementations of the same application as cross-references. Differential testing, however, is not always viable for testing static analyzers because static analyzers usually check different kinds of properties and correspondingly detect different types of issues [32, 72], making their analysis reports seldom directly comparable. Prior work manually devised tailored oracles for analyzers conducting specific analyses [19], but such an approach is hard to generalize or scale. The second challenge (**C2**) lies in the *automated preparation of high-quality input programs* for static analyzers. Given a property checked by a static analyzer, for the testing of the related analysis to be effective and efficient, the input programs should not only *contain the program elements essential for the property* so that the corresponding check will be activated but also be *minimal* so that the analysis reports are easy to comprehend. Such requirements make the preparation of high-quality test input programs for static analyzers even more demanding.

In this paper, we propose the Statfier technique that effectively detects bugs in a static analyzer by transforming existing test programs in a semantic-preserving way (i.e., the transformations should not change the behaviors of the programs), and comparing the reports produced by the analyzer on those programs. To address the first challenge, i.e., the test oracle problem, Statfier employs metamorphic testing. Metamorphic testing uses metamorphic relations, i.e., relationships between the expected outputs from multiple system executions, to determine whether the systems have executed correctly or not, and it has been successfully applied to verify compilers [27, 47, 68], machine learning systems [26, 74, 76], and interactive debuggers [69]. Specifically, given an initial input program for a static analyzer, Statfier iteratively applies semantic-preserving transformations to the program to generate a group of variant programs, runs the static analyzer on both the input program and the variant programs, and compares the produced analysis reports to determine whether the static analyzer contains any fault. The rationale here is that, since the transformations are semantic-preserving, input and variant programs should have same behaviors, and the analysis reports produced by the analyzer on the programs should also be equivalent. Therefore, if the analysis reports for a pair of input and variant programs differ, an issue has been found for the static analyzer. Furthermore, non-semantic preserving transformations will produce variant programs that are distinct from the input programs, almost inevitably violating the metamorphic relation on which Statfier is built. Based on this property, our transformations help ensure that the semantic errors in the original programs are preserved after the variant generation.

To address the second challenge, Statfier utilizes the programs within the existing test suites for the static analyzers or included in the tool documentation as the input programs. Developers of static analyzers often develop their own test suites, containing both test input programs and the expected analysis reports, for verification purposes, and they usually include example programs in tool documentation to help explain the performed analyses. Given that such programs are typically small, and they usually can activate the checking of most rules supported by the corresponding static analyzers, they make perfect input programs for Statfier.

Moreover, based on the observation that not all transformations applied at all possible locations stand the same chance of affecting the analysis reports, Statfier incorporates two heuristics to further improve its effectiveness in bug detection: When choosing locations from input programs to be transformed, the first heuristic, known as *analysis report guided location selection*, prioritizes the ones already included in analysis reports; When transforming variant programs, the second heuristic, known as *structure diversity driven variant selection*, prioritizes transformations that have not been applied yet.

We have implemented the Statfier technique into a tool with the same name. The tool supports 12 types of semantic-preserving program transformations gathered from existing literature [11, 23, 27, 49, 51, 65], and it reports a bug to the user if the results produced by a static analyzer on an input program and a corresponding variant program turn out to be different. To empirically evaluate the effectiveness of Statfier and its heuristics, we have applied the tool to detect bugs in five popular static analyzers, namely PMD, SpotBugs, Infer, SonarQube, and CheckStyle.

Statfier successfully detected 79 bugs, of which 46 have been confirmed by the developers of the corresponding static analyzers, and 26 have been fixed by the developers. Moreover, the variant programs generated by Statfier for detecting 26 bugs have been incorporated into the official test suites of the analyzers. The experimental results suggest both heuristics are essential for the Statfier's effectiveness.

Overall, this paper makes the following contributions:

- We propose the Statfier technique to enhance the bug detection capability of test suites in static analyzers by generating new input programs using semantic-preserving transformations and metamorphic testing. To improve the test generation, Statfier employs two novel heuristics, namely analysis report guided location selection and structure diversity driven variant selection;
- We empirically evaluated and confirmed the appropriateness of using existing input programs to drive automated testing of static analyzers with Statfier. Input programs from official test suites and examples in documentation of static analyzers can activate the checking of a high percentage of rules (e.g., up to 100% for PMD and CheckStyle) during testing, and they are small in size;
- We have implemented the Statfier technique into a tool with the same name and experimentally evaluated the tool by applying it to detect bugs in five popular static analyzers. The experimental results show that both Statfier and its heuristics are effective.

2 BACKGROUND: STATIC ANALYZERS

Static analyzers are software for automating program reasoning with a myriad of applications in verification, optimization, and bug finding. In our paper, we only consider static analyzers that can produce analysis reports. Namely, given an input program, a static analyzer usually adopts different program analysis techniques (e.g., data flow analysis and control flow analysis) to obtain the program abstraction without execution. After that, they may produce an

analysis report for users to identify potential issues when they violate pre-defined desirable properties in the input program.

Although Statfier may be theoretically applied to any static analyzers that can produce analysis reports, we focus on five analyzers in this paper, including PMD, SpotBugs, SonarQube, CheckStyle, and Infer. PMD [24] is a cross-language static analyzer that finds common programming issues (e.g., unused variables). Rules in PMD are written either in Java or XPath. SpotBugs [25] is a fork of FindBugs (which is now deprecated) that detects common issues in Java code via a set of issue patterns. SonarQube [64] is a platform developed by SonarSource for continuous code inspection. CheckStyle [39] is used for checking if Java code adheres to a coding standard. Infer [8] is a static analyzer designed by Meta that detects issues for Java, C, C++, and Objective-C programs. We select these tools because they (1) are widely used open-source tools for Java programs, (2) have been used in prior studies on static analyzers [3, 4, 18, 32, 42, 63], and (3) are representative of static analyzers adopted for different purposes (e.g., SonarQube supports automated code review, whereas CheckStyle checks coding style) and by different companies (e.g., SpotBugs is used in Google [3], whereas Infer is used in Meta [55]).

3 ILLUSTRATIVE EXAMPLE

Before presenting the heuristics used in Statfier, we first provide the following definitions:

Definition 3.1 (Structurally diverse). We construct a new test program P' by applying a sequence of transformations $T_1, ..., T_i$ to the input program with program contexts $C_1, ..., C_i$ (a program context C denotes the type and location information of program element to be transformed) where P' is represented by $\{C_1 : T_1, ..., C_i : T_i\}$. We consider P' to be a *structurally diverse* program if the sequence $\{C_1 : T_1, ..., C_i : T_i\}$ applied to the original program P to generate P' are distinct among all generated variants, namely for any pair $C_i : T_i$ and $C_j : T_j$ $(i \neq j)$ in the sequence are different.

Definition 3.2 (Differential analysis results). Given a program P and a transformed program P' (where P' is obtained via a semantic-preserving transformation to P), we consider a static analyzer S generates *differential analysis results* if o_p=invoke(S, P) and $o_{p'}$=invoke(S, P') where $|o_p| \neq |o_{p'}|$, $|o_p|$ represents the type and number of warnings from the report generated by running P on a static analyzer.

We illustrate the general workflow of Statfier using an example bug found by Statfier in PMD. Given a static analyzer under test S, Statfier first extracts input programs from either the test suite of S or the documentation of S. Listing 1 shows a program P extracted from the official test cases in PMD [57]. PMD developers designed the program P for testing the "HardCodedCryptoKey" rule (i.e., checks if hardcoded values are used for cryptographic key).

Prior Approaches. Existing approaches that test static analyzers [19, 67, 72] fail to find this bug because they rely on either static analysis rule pairs [72] or specialized oracles [19, 67]. Specifically, the "HardCodedCryptoKey" rule cannot be tested by a prior approach [72] that relies on differential testing of static analysis rule pairs (only PMD supports this rule, and SonarQube does not have the matching rule [73]). Meanwhile, an approach that randomly generates and selects variants [19] wastes time in evaluating variants that do not help discover new bugs in static analyzers.

To solve the limitations of prior approaches, we propose two key heuristics, including (1) analysis report guided location (AL) and structurally diverse variant selection (SS), which we will explain using the example in Listing 1.

```
1 import javax.crypto.spec.SecretKeySpec;
2 public class Foo {
3  void encrypt() {
4   SecretKeySpec keySpec = new SecretKeySpec(
5   "Hardcoded Crypto Key".getBytes(),"AES");//
        warning
6   }}
```

Listing 1: Original program P (input program) that detects `HardCodedCryptoKey` in PMD

```
1 import javax.crypto.spec.SecretKeySpec;
2 public class Foo {
3   void encrypt() {
4 +   String str = "Hardcoded Crypto Key";
5     SecretKeySpec keySpec =  new SecretKeySpec(str
          .getBytes(), "AES"); // warning
6   }}
```

Listing 2: Transformed program P' generated by Statfier

```
1 import javax.crypto.spec.SecretKeySpec;
2 public class Foo {
3   void encrypt() {
4 -   String str = "Hardcoded Crypto Key";
5 +   String str;
6 +   str = "Hardcoded Crypto Key";
7     SecretKeySpec secretKeySpec = new
          SecretKeySpec(str.getBytes(), "AES"); //
          False negative
8     }}
```

Listing 3: Generated program by transforming P' to P''

Analysis Report Guided Location (AL). Given an input program P for a given rule (e.g., "HardCodedCryptoKey" rule), Statfier invokes S to generate an analysis report for P. Each analysis report contains the line in which the rule violation occurs (e.g., line 4 in Listing 1). After obtaining the line L in which the violation occurs and all statements within the backward slice of L (no other statement is data/control dependent on L for this example), Statfier systematically explores all semantic-preserving program transformations that are valid (fulfills the prerequisite for a given transformation) at line L. Listing 2 shows one such valid transformation (program P') that modifies line 4 in Listing 1 by extracting a local variable `str`. For each transformed program that reports the same rule violation as the original program, we keep them in a queue for further modifications. In this example, program P' causes the same rule violation as the one reported in P, so Statfier further modifies it to P'' by moving the assignment at line 4 (Listing 3). As Statfier only introduces semantic-preserving program transformations, the static analyzer S should report the same rule violation for Listing 3. However, when Statfier runs PMD for P'', it gets differential analysis results, so we consider the missing rule violation as a false negative (FN). We have reported this bug to PMD developers, and it has been confirmed and fixed by developers. According to the

developer's feedback on the submitted issue, the problem occurs because "The rule currently only checks the initializer of the variable. In your code sample, the variable `str` is not initialized at all, so the rule does not see a problem" [58].

Structurally Diverse Variant Selection (SS). As stated in Definition 3.1, this approach represents each variant as $P' = \{C_1 : T_1, ..., C_i : T_i\}$ where $C_1, ..., C_i$ refers to the program contexts, and $T_1, ..., T_i$ denotes the sequence of transformation types applied to the original input program P. Specifically, we represent the variant in Listing 2 by $\{C_1 : T_1\}$ = {StringLiteral, Line=4, Column=[52,72], "Extract local variable"}. To guide the search toward structurally diverse variants, this approach avoids selecting a new variant k where the context and the selected type of semantic-preserving transformation is the same as in Listing 2 (e.g., C_k=C_1 and T_k=T_1). Compared to a random approach that searches through 112 variants, the SS approach is more efficient as it can find the same bug in Listing 3 after iterating through only 51 variants.

4 METHODOLOGY

Algorithm 1: Algorithm for Heuristic-based Automated Testing of Static Analyzers

Input: Input programs *Progs* for a rule R in a static analyzer S, a set of transformations *Trans*, timeout $timeL$
Output: A set *errP* with test programs that give differential analysis results

```
1  errP ← ∅
2  normP ← ∅
3  disSeq ← ∅
4  initializeTimer(execTime)
5  for P ∈ Progs do
6      Q ← INITQUEUE(P)
7      while execTime ≤ timeL do
8          currP ← DEQUEUE(Q)
9          (numW_B, typeW_B, loc_B) ← run(currP)
10         locs ← GETBACKWARDSLICE(currP, S, loc_b)
11         for loc ∈ locs do
12             nodeType ← GETASTNODE(currP, loc)
13             for t ∈ Trans do
                   /* select structurally diverse variant */
14                 if (nodeType, t) ∉ disSeq then
15                     disSeq ← disSeq ∪ (nodeType, t)
16                     newP ← TRANSFORM(currP, t, loc)
17                     if ISDIFFERENTIAL(newP, numW_B, typeW_B) then
18                         errP ← errP ∪ {newP}
19                     else
                           /* store new variants */
20                         normP ← normP ∪ {newP}
21             Q ← Q.ENQUEUE(normP)
22 return errP
23 Func ISDIFFERENTIAL (newP, numWarn_B, typeWarn_B):
       /* run new program on static analyzer */
24     (numWarn_A, typeWarn_A, loc_A) ← run(newP)
25     return numWarn_B ≠ numWarn_A or typeWarn_B ≠ typeWarn_A
```

Figure 1 gives an overview of StatFier. Given an input program, StatFier applies semantic-preserving transformations and uses variant selection to get a reduced set of variants, which are run against the static analyzers to check for differential analysis results. After obtaining variants that produce differential analysis results, StatFier will use them as input programs to trigger bugs in static analyzers under test. Algorithm 1 shows our test generation algorithm. Our algorithm consists of four components: (1) variant generation, (2) selection of program location for transformation, (3) variant selection, and (4) feedback-driven exploration. Given a set of input programs *Progs* for a rule R in a static analyzer S, a set of semantic-preserving transformations *Trans*, and a timeout $timeL$, our test generation algorithm produces a set of programs that indicate bugs for the rule R in the static analyzer S. Specifically, StatFier initializes the algorithm parameters at lines 1–4 of the algorithm, whereas lines 5–22 are the core logic of the algorithm. After initializing the output set *errP* (line 1), the normal variant set *normP* (line 2), and a set *disSeq* for variant selection (line 3), StatFier retrieves the first element of Q as the program *currP* to be transformed (line 8), and runs a static analyzer on *currP* to obtain the number, type, and location of warnings (line 9). After that, it performs backward slicing to obtain all related locations *locs* and gets the corresponding node type *nodeType* at lines 10–12. Finally, it applies each transformation t in *Trans* to *loc* iteratively and performs structure diversity driven variant selection at lines 13–16. At lines 17–20, the isDifferential function compares the two analysis reports of *currP* and *newP* to detect potential bugs. If no bugs are found, *newP* will be added into Q (line 21). Note that instead of generating an extensive test suite for checking all rules in a static analyzer, our test generation algorithm produces test programs for each rule separately because each rule has its own set of input programs (mixing together input programs of different rules will make it difficult to isolate the bug).

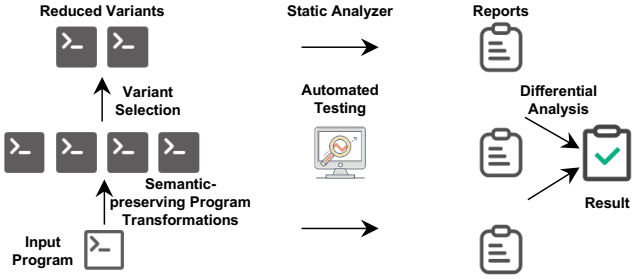

Figure 1: Overview of StatFier

4.1 Selection of Input Programs

As stated in **C2**, before automated testing static analyzers, we need to obtain input programs with high rule coverage as the initial set to be transformed. Static analyzers rely on rule checkers to detect problems under a given program. To help users understand the reported rule violations, static analyzers usually include code examples to explain the problematic code. Hence, our key insight to solve **C2** is *the input programs designed by the developers of static analyzers can address the challenge in constructing input programs which can trigger more rule violations*. Based on this insight, we extract input programs from two sources: (1) the official test suite and (2) code examples of each rule given in the documentation of the static analyzer. When extracting input programs from the provided test suite, we first obtain the folder storing the input programs of a

Table 1: Rule coverage of input programs for all evaluated static analyzers

Analyzer	Tests within the test suite		Documentation		Combined Rule Coverage (%)
	$\frac{\#of\ covered\ rules}{Total\ \#\ of\ rules}$	Rule Coverage (%)	$\frac{\#of\ covered\ rules}{Total\ \#\ of\ rules}$	Rule Coverage (%)	
PMD	$\frac{274}{274}$	100%	$\frac{273}{274}$	99.64%	100%
SpotBugs	$\frac{360}{458}$	78.60%	$\frac{52}{458}$	11.35%	78.60%
SonarQube	$\frac{581}{617}$	94.17%	$\frac{587}{617}$	95.14%	95.30%
CheckStyle	$\frac{183}{183}$	100%	$\frac{159}{183}$	86.89%	100%
Infer	$\frac{67}{92}$	72.83%	$\frac{30}{92}$	32.61%	77.18%
Overall	$\frac{1410}{1624}$	86.82%	$\frac{1101}{1624}$	67.8%	88.79%

static analyzer (e.g., src/test/resources/net/sourceforge/pmd/lang/-java/rules for PMD), and automatically crawl the folder to obtain the input programs for each rule. To reuse code examples in documentation, we manually create a complete input program from the code snippets in documentation by adding the missing (1) variable/method/class declarations and (2) import statements. Our goal is to obtain as many input programs as possible to maximize *rule coverage* (the percentage of rules checked by a static analyzer that contains at least one input program) and to test different behaviors of the covered rule checker. After obtaining the input programs, we run them against the static analyzer (in which the input programs originated from) to measure their quality. Furthermore, 86% of extracted input programs have more than one violation, 8% have one violation, and the rest 6% have no violations.

Since the input programs are from two sources (the official documentation and test suites), we study the rule coverage achieved by input programs from different sources, both separately and as a whole. Particularly, we manually analyzed the official documentation of each analyzer and counted the rules mentioned in the documentation as the total number of rules supported by each tool. Meanwhile, we ran each static analyzer on all input programs extracted for it and considered a rule to be activated, or covered, if and only if an issue was reported because a rule violation was detected in an input program. Table 1 shows the rule coverage information for all evaluated static analyzers. The second and the third columns denote the ratio of covered rules and the rule coverage given by the input programs within the test suite, whereas the fourth and fifth columns show similar measurements given by the documentation of each rule. Given the set T of the rule covered by the test suite and the set D of the rule covered by the documentation, the "Combined Rule Coverage (%)" presents the overall rule percentage of $T \cup D$.

Instead of obtaining input programs from a test suite, an alternative approach is to extract them from real-world projects. However, the prior evaluation [72] that uses input programs from 2,728 projects shows that this alternative approach can only achieve 74%–89% rule coverage for four static analyzers. Another alternative is to adopt Defects4J [43], which is a widely used dataset with defects for Java programs. However, prior studies [33] revealed that the rule coverage achieved by static analyzers in Defects4J is only around 2%–4%. Compared to these alternatives, our study shows that *reusing programs from a test suite of static analyzers can cover more rules and more analyzers.* As shown in the third column of Table 1, rule coverage achieved by the test suites of static analyzers ranges from 72.83%–100%, specifically all rules in PMD and CheckStyle can be covered. Our study of the test suite of analyzers also gives evidence for the hypothesis that most bugs have small counter-examples [41] because we observed that test input programs in analyzers are small, i.e., they have an average of 1.25 classes (max=30), 3.16 methods (max=106), 1.33 fields (max=1000), and 49.42 lines of code (max=65544).

Based on the combined rule coverage, we observe that the rule coverage of SpotBugs, CheckStyle, and SonarQube can be further improved if we add code examples from the documentation. Moreover, some analyzers are equipped with at least one input program for each implemented rule (e.g., CheckStyle have combined rule coverage of 100%). The high combined rule coverage achieved by these input programs confirms our intuition that *these input programs can be used as the initial set of input programs for testing analyzers.*

4.2 Variant Generation via Semantic-Preserving Program Transformations

Given an input program P, Statfier produces variants by applying a set of program transformations (line 16 in Algorithm 1), and these transformations should be semantic-preserving. To obtain a comprehensive set of transformations, we refer to existing related literature thoroughly [11, 23, 27, 49, 51, 65]. We use the following design principles when selecting transformations:

Multi-level Transformations: Based on the program elements to be transformed, we divide the space of transformations into five levels: variable, expression, statement, method, and class. We adopted the transformations from existing literature [11, 23, 27, 49, 51, 65] for the first four levels (i.e., variable, expression, statement, method). As there is no prior approach that uses class-level semantic-preserving transformations, we refer to GitHub issues of static analyzers for the class-level transformations [21, 22, 56].

Dead and Live Code Injections: We include both kinds of transformations which can inject dead code (e.g., "Unreachable code injection") [49, 65] and live code (e.g., "Statement wrapping") [27, 51] into the original program.

Incorporating Analysis Capability of Each Static Analyzer: Considering the trade-off between accuracy and efficiency, static analyzers may support different levels of analysis capabilities for producing more precise or faster analysis. For example, Infer and SonarQube support inter-procedural analysis, but PMD and SpotBugs only support intra-procedural analysis. For analyzers that

Table 2: Semantic-preserving program transformations supported in Statfier

Level	Transformation	Example	Source

Variable-level	Extract local variable	`- invoke_method("String Literal");`
		`+ String str = "String Literal";`
		`+ invoke_method(str);`
			[11, 51]

	Move assignment	`- int var = 10;`
		`+ int var;`
		`+ var = 10;`
			[11]

Expression-level	Equivalent boolean expression: Add $\|\|false$ or $\&\&true$ expression Swap symmetrical elements, e.g., $a == b \rightarrow b == a$	`- boolean tag = true;` `+ boolean tag = true || false;`	[27, 31]
	Equivalent arithmetic expression: Add +0, -0, or +1-1 expression	`- int value = 10;` `+ int value = 10 + 0;`	[2, 27]
	Add parenthesis to expression	`- int value = 10;` `+ int value = (10);`	[11, 23]

Statement-level	Equivalent statement conversion:
	Convert to equivalent for/while/do-while/lambda
		`- for(i = 0; i < 1; i++) {}`
		`+ i = 0;`
		`+ while(i++ < 1) {}`
			[11]

	Statement wrapping: Wrap statements with if/while/for/do-while	`- target_statement;` `+ if(true) { target_statement; }`	[27, 65]
	Dead code injection: Insert dead if/while/for statement	`target_statement;` `+ for(int i = 0; i < 0; ) { target_statement; }`	[49, 51]

Method-level	Encapsulate field	`- SecretKeySpec("Hardcode");`
		`+ String getHardcode() { return "Hardcode"; }`
		`+ SecretKeySpec(getHardcode());`
			[11, 23]

Class-level	Nested class wrapping	`- original_program;` `+ class NestClass { original_program; }`	[21]
	Anonymous class wrapping	`- original_program;` `+ Object c = new Object() { original_program; };`	[22]
	Enum wrapping	`- original_program;` `+ enum enumClass { original_program; }`	[56]

do not support inter-procedural analysis, variants generated via method-level program transformation (i.e., *Encapsulate field*) are not meaningful as detecting these variants is beyond its analysis capability. We integrate the analysis capability of each studied analyzer by excluding transformations that are beyond its capability.

No Style-Related Transformation: We exclude transformations that can change coding style (e.g., changing comments or identifier names) because (1) these transformations may trigger inaccurate differential analysis results in tools like CheckStyle that check coding standards, and (2) transformations such as renaming requires encoding Java naming conventions into the transformation rules, which is beyond the scope of this paper. Specifically, we exclude all transformations in "Level 1–Changes to Comments&Indentation" and "Level 2–Changes to Identifiers" from prior work [11].

Table 2 shows the 12 types of semantic-preserving program transformations supported in Statfier where we include an example for each transformation. The "Source" column indicates the relevant work in which we derived the corresponding transformation from.

4.3 Heuristic-Based Testing Process

After applying semantics-preserving transformations, Statfier first feeds all variants to static analyzers and obtains analysis reports, then performs differential analysis to determine if there is a bug. The core factors that affect the efficiency of testing are (1) identifying locations to be transformed and (2) removing redundant variants that are unlikely to trigger new bugs. Hence, we designed several heuristics below to accelerate the testing process:

Analysis Report Guided Location Selection The program locations in which we choose to apply the transformations will affect the effectiveness of variant generation and selection. Our main insight is that *we can select the program locations for transformations based on the locations that are listed in the analysis report generated by static analyzers*. Specifically, given an input program P, Statfier applies the static analyzer on P to obtain (1) the number of warnings reported by the static analyzer and (2) the program locations in which the analyzer reports violations (line 9 in Algorithm 1). Instead of relying solely on the reported program locations that may be incomplete, our goal is to obtain the set of locations that have either control or data dependency with respect to each program location stated in the analysis report. Hence, after executing the program against the static analyzer under test, Statfier obtains the backward slice of the program starting from each program location stated in the analysis report. Specifically, the getBackwardSlice($currP$, S, loc_b) function takes as input (1) the input program $currP$, (2) the static analyzer under test S, and (3) the program locations loc_b for the reported violations to produce a set of locations $locs$ that include all locations stated in the analysis report, together with their backward slices (line 10 in Algorithm 1). This step produces a set of program locations in which we will apply the set of semantic-preserving transformations. As we use an analysis report to guide the selection of program locations, we call this step *analysis report guided location selection*.

Structure Diversity Driven Variant Selection Given a set of variants, we propose structure diversity driven variant selection to obtain a subset of variants that are more likely to trigger distinct bugs in static analyzers. Specifically, given an original program P, we represent each of its variant $P' = \{C_1 : T_1, ..., C_i : T_i\}$ by checking (1) the program context $C_1, ..., C_i$ under which each transformation has been applied to, and (2) the transformation types $T_1, ..., T_i$ that have been applied starting from P. Given a transformed program location *loc*, we determine the program context by checking for the AST node type as follows: (1) if the AST node at *loc* is a leaf node, we use its AST node type (e.g., operand) as the program context, (2) if the AST node at *loc* is a non-leaf node, we use the AST node type of the root of the subtree (e.g., if-statement) to represent the program context. As shown at lines 11–16, Statfier checks if the current program context (represented by *nodeType*) and transformation type (represented by *t*) exist in the previously encountered sequence *disSeq*. This selection aims to eliminate variants with the same program context and use the same transformation type as previously chosen variants. Subsequently, all the selected variants are structurally diverse (refer to Definition 3.1).

Feedback-Driven Exploration The test generation algorithm of Randoop avoids extending illegal method sequences (e.g., those that lead to exceptional behavior) based on the feedback obtained from executing test inputs during test generation [53]. Inspired by Randoop's test generation algorithm, we use a feedback-driven approach for our test generation by avoiding further exploration of input programs that lead to differential analysis results (refer to Definition 3.2), essentially incorporating feedback obtained from running a static analyzer. At lines 17–21 in Algorithm 1, we store the newly transformed program *newP* into the *normP* set for further extension of *legal program sequences* (those that do not lead to differential analysis results) and return all programs within the *errP* set that contains programs that indicate bugs in a static analyzer.

Statfier inspects differential analysis results in the isDifferential function in Algorithm 1. The function (1) runs the transformed program on the given static analyzer (line 24), and (2) compares the number and type of warnings before and after applying the transformation and returns true if the number of warnings differs (line 25). In general, the differential analysis results represent two cases: (1) a false negative (FN) in S if the original program P produces a warning in S but the transformed program P' does not give a warning (a warning is missing), and (2) a false positive (FP) in S if the original program P does not produce a warning but the transformed program P' leads to a warning (indicates a false warning). Furthermore, we also add a filter that checks for the newly added types of reported warnings to remove false positives.

5 EVALUATION

We applied Statfier on five analyzers (PMD, SpotBugs, CheckStyle, SonarQube, and Infer) to address the research questions below:

RQ1: How many unique bugs can Statfier find? What are the characteristics of these bugs?
RQ2: Can the proposed heuristics in Statfier reduce the number of variants while preserving its bug finding capability?
RQ3: How many bugs can each transformation find?

5.1 Experimental Setup

We implemented semantic-preserving program transformations and backward slicing using the Eclipse JDT library with over 7000 lines of Java code. We did not use existing slicers because (1) using the same library for program transformations and slicing will make it easier to traverse and match the AST, and (2) Eclipse JDT library has been widely used for semantic-preserving transformations (e.g., it supports refactoring operations [28]). Specifically, Statfier constructed the program dependency graph based on Eclipse JDT and extracted the control and data dependency information. As most static analyzers cannot perform inter-procedural analysis well, we only construct the intra-procedural backward slicing without alias analysis. Based on our experiment [60] for selecting appropriate parameters, we set the time limit $timeL$ of each static analyzer to be six hours (which includes transforming and checking all input programs for an analyzer) for running Statfier in RQ1 and RQ2. For each rule in each analyzer, we reuse the configuration in the test case if available; otherwise, we use the default configuration recommended by the analyzer. For static analyzers that require compilation (e.g., SpotBugs), we compile each program using Oracle JDK 8 and 11. All experiments were conducted on a machine with Intel Xeon 6134 CPU and 192GB RAM.

5.2 RQ1: Assessing Effectiveness of Statfier

We use the number of discovered unique bugs to evaluate the effectiveness of Statfier. Notably, we consider a bug to be a ***unique bug*** if it is in (1) different rule checkers triggered by various transformations, (2) different faulty locations (determined by root cause diagnosis) in a static analyzer, and (3) not the 14 false positives discussed in Section 5.3. We use this definition of *unique bug* because static analyzer developers (1) adopt a similar definition when checking for duplicate bugs [20, 40], and (2) usually repair faults for different rules in different program locations for corresponding rule checkers. Specifically, we adopt the type of rule checker and applied transformation to automatically cluster variants which lead to differential analysis results. We consider variants in the same cluster as equivalent and randomly pick one representative variant from each cluster. After automated clustering differential analysis results based on our definition of a unique bug, Statfier can find 79 bugs that span across all of the evaluated static analyzers.

Status of Reported Bugs. We have reported the 79 unique bugs to developers of static analyzers. Table 3 shows the current status of our submitted issues. We classify these issues into four categories based on the responses that we have received from developers of static analyzers so far. The categories are listed below:

(1) Fixed: the issue was fixed by a merged pull request.
(2) Confirmed: the issue was confirmed by the developer but not fixed so far.
(3) Pending: we have not received a response from developers.
(4) Won't fix: the developer acknowledged that the bug is a limitation of the static analyzer but will not fix it.

Among all the evaluated static analyzers, we observe that Statfier finds the most significant number of bugs in PMD. This is because PMD has the highest rule coverage with the greatest number of input programs (as shown in Table 1). With more input programs, our approach can further transform these programs,

Table 3: The status of submitted GitHub issues

Issue status	PMD	SpotBugs	SonarQube	CheckStyle	Infer	Overall
Fixed	14	3	5	4	0	26
Confirmed	11	3	5	2	0	21
Pending	19	6	0	1	3	29
Won't fix	0	0	2	1	0	3
Total	44	12	12	8	3	79

Table 4: The number of detected bugs by different approaches

Approach	PMD	SpotBugs	SonarQube	CheckStyle	Infer	Overall
Statfier	44	12	12	8	3	79
[31]	1	0	0	0	0	1

making it more likely to discover new bugs in PMD. For each bug found by Statfier, we manually inspect it to filter FPs and report to developers only after checking for its validity. As illustrated in Table 3, except for "Won't Fix" where developers have decided not to fix the bugs, all our reports are perceived positively by developers (i.e., we did not have any rejected bug reports), and nearly half of them have been fixed or confirmed. It is worthwhile to note that Statfier *generated variants for 26 of the submitted bugs have been integrated into official test suites* by the developers of static analyzers, demonstrating the importance of input program generation for testing static analyzers. Moreover, as SonarQube is the only evaluated analyzer that uses Jira for issue tracking, we manually checked the assigned severity of the reported bugs. Among the 10 confirmed bugs in SonarQube, 7 are marked as major, 3 are minor.

Comparison with Mutation Testing Approach. We also evaluate Statfier against a mutation testing based approach [31] that compares the effectiveness of static analyzers. It considers a static analyzer kills a mutant when the number of warnings increases with mutation and adopts the Universal Mutator [30] for the mutation process. As the tool for prior approach [31] is not publicly available, we re-implement it by (1) using the open-source Universal Mutator [30], and (2) reproducing the oracle in their paper (i.e., the number of warnings or errors increases). To ensure a fair comparison, we measure its effectiveness in detecting bugs in static analyzers using the same timeout as Statfier. Table 4 demonstrates the comparison result. Among 472637 mutants generated for all static analyzers, it only finds one real bug in PMD and produces two false positives in SpotBugs. The result indicates the prior mutation testing approach is less effective than Statfier in finding bugs.

Limitations. We notice two limitations of our approach during the manual analysis of the bugs found. Firstly, as common in a testing tool, our approach may produce FPs (the discovered defects are not real defects). For example, when applying the "Statement wrapping" transformations to wrap code with if-statement, SpotBugs may report one more *DB_DUPLICATE_BRANCHES* warning, but this extra warning is triggered by our applied transformation (not a real bug) because this rule detects duplicate if-else branches. However, in our experiment, we only find 14 FPs, and they are excluded from Table 3 and Table 6. Our filter can remove these FPs automatically. The second limitation is that the tested static analyzers need to be able to generate an analysis report, and the report should include the warning types and locations. Currently, Statfier does not aim to test important static analyses such as call graph analysis and pointer analysis which may not produce reports that contain line information. In the future, it is worthwhile to investigate how to extend our technique to support other types of analysis (e.g., we can perform differential analysis on the reports of info-flow analysis between input program and its variants as their behaviors should be equivalent).

Table 5: The statistics for root causes of bugs in static analyzers

Root cause	PMD	SpotBugs	SonarQube	CheckStyle	Infer
Variable declaration	13	3	6	0	1
Complex class structure	12	4	0	3	1
Control flow structure	8	2	5	1	1
Compound expression	10	1	1	3	0
Java version and new features	1	2	0	1	0

Root Causes of Found Bugs. We manually analyze the root causes behind the found bugs and summarize them into six categories. Table 5 shows the numbers of each root cause for these bugs. This table is sorted in descending order by the number of discovered issues. We discuss representative examples of each category below:

5.2.1 Variable Declaration. When analyzing variables, rules in static analyzers either (1) only check for direct initialization or (2) fail to analyze declarations at the global level. For example, Listing 3 shows that PMD fails to detect the hardcoded key of the assignment statement at line 6 as it does not analyze the usage for the `str` variable. Statfier also found FNs due to incomplete analysis of global variables (fields). Figure 2 shows an example [6] where Infer fails to report the null pointer dereference for the field at line 6 but can detect the dereference if `color` is a local variable.

```
1  enum Color { BLACK, WHITE; }
2  public class SwitchCase {
3  + Color color = null;
4    public String switchOnNullIsBad() {
5  -   Color color = null;
6      switch(color) { ... // should report a warning
```

Figure 2: A null dereference bug in Infer

5.2.2 Complex Class Structure. Static analyzers need to retrieve methods or fields within classes, but the retrieval can be incomplete if the given program's class structure is complex, e.g., when there are nested classes. Figure 3 shows such an example in SpotBugs for the *MS_EXPOSE_REP* rule (this rule detects a security flaw when a public static method returns a reference to an array that is part of the static state of the class) [7]. Given the original input program, SpotBugs can report an issue as the method `faultMethod` leaks the private field `key`. However, after transforming the program via "Nested class wrapping", SpotBugs detector "FindReturnRef" for this rule cannot detect the reference in the nested class. The developer has prepared a fix for the bug soon after reporting.

5.2.3 Control Flow Structure. Our manual analysis shows that programs with complex control flow structures can lead to unexpected results in static analyzers. For example, the PMD rule *UseStringBufferForStringAppends* recognizes the use of the `+=` operator for appending strings and warns that the operator causes the JVM to use an internal `StringBuffer`, which is inefficient. Figure 4 shows an example [71] where PMD reports one warning for this rule at line 3 and two duplicated warnings (warnings that are exactly the

```
1  private static String[] key;
2  + static class nestedClass {
3      public static String[] faultMethod() {
4        return key; // should report a warning
5      }
6  + }
```

Figure 3: An FN for the rule *MS_EXPOSE_REP* in SpotBugs

same) for the same rule at line 5. Although line 3 and line 5 are equivalent, the duplicated warnings at line 5 show two problems in PMD: (1) an FP caused by the statements, (2) the failure to filter duplicated warnings. Moreover, our analysis also shows that some rule checkers in static analyzers may fail to support different control flow structures (e.g., SonarQube can detect infinite `for` and `while` loops but does not consider `do-while` loop [5], and the developer has fixed the bug upon receiving our bug report).

```
1  public void bar() {
2   String x = "foo";
3   x += "bar" + x; // report one warning
4  + if (false) {
5  +   x += "bar" + x; // report duplicate warnings
6  + }}
```

Figure 4: An FP for the rule *UseStringBufferForStringAppends*

5.2.4 Compound Expression. When analyzing compound expressions such as parenthesized expression and binary expression, static analyzers like PMD may fail to check the subexpression either due to (1) incomplete AST node representation (the Java AST library in PMD does not model `ParenthesizedExpression` as an AST node type, but its JavaScript AST library does not have this problem) or (2) fail to traverse the subexpression. Figure 5 shows a false positive example [79] for the PMD rule *RedundantFieldInitializer* that detects a redundant initialization (assigning a field to its default values). The FP occurs because PMD fails to check the subexpression by traversing only the first operand (value `0` is the default value for a `char`) in the binary expression and mistakenly reports the field `c` to be a redundant initialization.

5.2.5 Java Version and New Feature. With the release of new Java versions, new bugs may occur in static analyzers either due to (1) the differences in the generated Java bytecodes or (2) insufficient support of new language features (e.g., lambda expression). When communicating with developers of static analyzers, they noted that the bug failed to reproduce in different java versions, pinpointing the root causes to be the different Java versions used. For example, the *DMI_INVOKING_TOSTRING_ON_ARRAY* rule in SpotBugs can detect the issue shown in Figure 6 when we compile with Java 8 but SpotBugs fails to detect the issue when using newer Java versions (Java 11, 16, 17) [78]. In our experiment, we only tested input programs with Java 8 and 11. It is worthwhile to study a differential

```
1  - char c = 1;
2  + char c = 0 + 1; // should not report a warning
```

Figure 5: An FP for the rule *RedundantFieldInitializer*

Table 6: Number of bugs detected across five seeds

Analyzer	RL*RS [19]	AL*RS	RL*SS	AL*SS (Statfier)
PMD	(0, 0, 0)	(36, 40, 38)	(0, 0, 0)	44
SpotBugs	(0, 0, 0)	(11, 12, 12)	(0, 0, 0)	12
SonarQube	(0, 0, 0)	(11, 12, 12)	(0, 0 ,0)	12
CheckStyle	(0, 0, 0)	(7, 8, 7)	(0, 0, 0)	8
Infer	(0, 0, 0)	(3, 3, 3)	(0, 0 ,0)	3
Total/Avg	(0, 0, 0)	(68, 75, 72)	(0, 0, 0)	79

testing approach that checks the input programs against different Java versions in the future, especially for analyzers like SpotBugs that act on bytecodes. For the new language feature example, the CheckStyle rule *ParameterAssignment* that detects assignment to parameters fails to recognize parameters in the lambda expression `list.forEach((i)→{i*=10;});` [38]. We reported this bug and it has been fixed by the CheckStyle developer.

```
1  + final String[] gargs = new String[]{"1", "2"};
2  public void print() {
3  - final String[] gargs = new String[]{"1", "2"};
4    System.out.println(""+gargs);} //need a warning
```

Figure 6: An FN caused by Java version in SpotBugs

5.3 RQ2: Assessing Effectiveness of Heuristics

We construct several baselines below to measure the effectiveness of different heuristics in Statfier:

Random Location (RL): An approach that randomly selects program locations to transform.
Analysis Report Guided Location (AL): An approach that uses analysis reports for selecting program locations (see Section 4.3).
Random Variant Selection (RS): An approach that randomly selects variants generated via semantic-preserving transformations.
Structurally Diverse Variant Selection (SS): An approach that selects structurally diverse variants (see Section 4.3).

Although there are several prior approaches that test static analyzers [19, 45, 67, 72], most of them do not generate variants [67, 72] so we exclude them from comparison. To ensure a fair comparison with prior work that uses Csmith for generating variants for C programs via random mutations [19], we emulate prior work using the baseline **RL*RS** that reuses the same set of transformations and oracle (differential analysis results) but randomly selects variants and locations to transform. When generating new variants by **RL**, we remove duplicated variants and set the number of transformations equal to **AL** to ensure fairness. As all approaches (except for Statfier that uses AL*SS) rely on randomized algorithms that may produce different results across different runs, we re-run each randomized approach five times with different random seeds.

Table 6 shows the effectiveness of each evaluated approach where each cell represents the (minimum, maximum, median) number of detected bugs by the four approaches (note that Statfier that uses AL and SS is deterministic, so we do not need to rerun it five times). We observe that all approaches that use "Random location (RL)" fail to detect any bug in all evaluated static analyzers, including **RL*RS** that emulates prior work [19]. Indeed, as the RL approaches can skip the static analysis steps (i.e., analysis report

Table 7: Variant selection percentage for AL*SS versus AL*RS

Seed	PMD	SpotBugs	SonarQube	CheckStyle	Infer	Average
S1	329492(60.09%)	7444(56.56%)	58012(49.76%)	26301(15.23%)	28963(20.41%)	40.41%
S2	318988(62.06%)	7635(55.14%)	58749(48.83%)	25472(15.72%)	29367(20.13%)	40.38%
S3	339986(58.23%)	7298(57.69%)	54004(53.12%)	24795(16.15%)	27950(21.15%)	41.27%
S4	330262(59.94%)	7690(54.75%)	57277(50.40%)	25863(15.49%)	28375(20.83%)	40.28%
S5	328954(60.18%)	7594(55.44%)	56051(51.18%)	26371(15.19%)	29013(20.37%)	40.47%

generation and backward slicing), they perform transformations on many randomly selected program locations rather quickly, causing rapid growth of variants (leading to consuming too much time to analyze all variants). This indicates that *our proposed analysis report guided location selection heuristic plays an essential role in reducing the number of locations* selected for modifications, subsequently guiding the test generation towards producing more valuable variants. Compared to Statfier that discovers 79 bugs, the AL*RS approach that uses random variant selection only finds 68–75 bugs across the five runs. As we designed the structure diversity driven variant selection heuristic to avoid evaluating variants that trigger similar bugs, *the fact that* Statfier *outperforms the AL*RS approach shows that this heuristic guides* Statfier *in finding distinct bugs.* Besides, to assess the effectiveness of reducing redundant variants, we compare the structurally diverse variant selection heuristic with other baseline approaches. As shown in Table 6, RL*RS and RL*SS approaches fail to find any defects in static analyzers. Hence, we only compare the number of generated variants between AL*RS and AL*SS. We define *variant selection percentage* below:

$$Variant\ Selection\ Percentage = \frac{total\ \#\ of\ variants\ by\ AL{*}SS}{total\ \#\ of\ variants\ by\ AL{*}RS}\%$$

Based on the above equation, a *lower variant selection percentage denotes a better approach* (with greater selection power) that only needs to evaluate fewer variants (i.e., for all evaluated analyzers, Statfier can use fewer variants compared to AL*RS to find the same number of bugs). Table 7 depicts the variant selection percentage across the five seeds; each cell is of the form "number of generated variants(variant selection percentage)". On average, the AL*SS heuristic in Statfier selects 40.28%–41.27% variants compared to AL*RS. Based on Table 6 and Table 7, although AL*SS heuristic evaluates fewer variants, it still finds more unique bugs than AL*RS. Hence, we believe *AL*SS heuristic is effective in reducing the number of variants while preserving its bug finding capability.*

5.4 RQ3: Effectiveness of Transformations

We further analyze the effectiveness of each supported semantic-preserving program transformation. Figure 7 shows the number of bugs found using different types of transformations. Based on the bug distribution, each transformation can find at least one bug in the evaluated analyzers. This means *all implemented transformations are effective.* Moreover, we observe transformations that involve extracting a variable (e.g., "Extract local variable" and "Move assignment") are more likely to find bugs in the evaluated analyzers.

Comparison with Prior Mutation Testing Technique. Prior work [2] provides evidence on the correspondence between mutations and static warning types. However, it can not detect bugs in static analyzers. After comparing the set of transformations used in prior work, three transformations IOR (Overridden method rename), AOR (Arithmetic operator replacement), and AOI (Arithmetic operator insertion) are semantic-preserving and related to Statfier. As

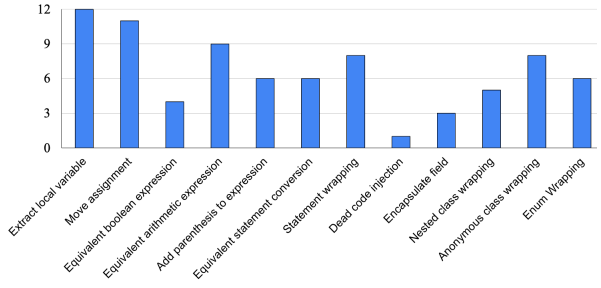

Figure 7: Number of transformation type

stated in section 4.2, we exclude style-related transformations like changing identifier names because of causing inaccurate differential analysis results. Hence, we do not consider the transformation IOR. The remaining AOI and AOR are similar to the transformation "Equivalent arithmetic expression". As shown in Figure 7, this transformation can find nine bugs, which is less than our findings.

Comparison with Prior Compiler Testing Technique. In a related approach for compiler testing [65], the Hermes tool synthesizes predicates representing known boolean values in control flow statements by executing input programs to obtain runtime information, whereas Statfier does not need to run input programs, which is also the feature of static analyzers. Besides, the Hermes is input sensitive as it relies on profiling analysis affected by the input, whereas our approach only requires static information and is more general. Additionally, we provide more transformations to make variants diverse. As stated in the design principle (Section 4.2), we incorporate the analysis capability of static analyzers by excluding transformations that are beyond the capability of existing analyzers. Hence, we adapted the prior compiler testing technique [65] for testing analyzers by representing it with the "statement wrapping" transformation, which uses literal boolean values as predicates. Figure 7 shows "statement wrapping" transformation can only detect eight bugs, which is less effective than Statfier.

6 THREATS TO VALIDITY

External. Our experiments may not generalize beyond the studied static analyzers and other programming languages beyond Java. While there are many other static analyzers available (e.g., Error Prone [61]), we only test five popular analyzers and construct only Java programs as inputs to these analyzers, although some of these analyzers can support multiple languages (e.g., PMD can analyze JavaScript). Moreover, our results may not generalize beyond the implemented transformations. To mitigate this threat, we include transformations that have shown promising results in prior work. Our experiments show that the implemented transformations are general enough to discover bugs in the evaluated static analyzers. During the root cause categorization, two authors of the paper independently analyze the bugs by inspecting analysis reports and logs of running each analyzer and discuss to resolve any disagreement. Moreover, we confirmed the validity of each identified bug with developers by filing a total of 79 bug reports.

Internal. Our code and automated scripts may have bugs that can affect the results. To mitigate this threat, we have made our tool and data publicly available.

7 RELATED WORK.

Program Transformations. Program transformations have been applied to enhance many software engineering jobs, including producing simulated source code plagiarism [11], improving testability (i.e., transform a program to make it easier for a given test generation method to generate test data) [34, 35], enhancing the generalizability of neural program models [59], and testing refactoring engines [23]. In the context of static analyzers, randomized program transformation has been used for testing static analyzers [19]. Meanwhile, a prior work formalizes the impact of program transformations on the results of static analysis by a mathematical framework [52]. Although we derive our set of semantic-preserving transformations from existing literature [11, 23, 51], our set of transformations is more comprehensive as it can cover program elements at different levels, and our set of transformations is used for testing static analyzers rather than for other applications. The most closely related to Statfier is the recent work that uses program transformations to resolve false positives of static analyzers [70]. Our approach differs from prior work in three aspects: (1) we generate input programs for testing static analyzers and file analysis reports to indicate issues in analyzers, whereas prior work improves static analyzers from the user perspective (no bug reports are filed as a result); (2) we use different transformations (general semantic-preserving transformations instead of rewriting tool-specific templates in prior work [70]); (3) Statfier finds issues in static analyzers, whereas prior work focuses on resolving FPs.

Static Analysis. Many prior studies focus on evaluating the effectiveness of static analysis [3, 4, 18, 32, 42]. Although our study in Section 4.1 also evaluates the effectiveness of rule checkers within static analyzers, none of these prior studies analyze the quality of existing input programs in the official documentation and test suites. Our study revealed the potential of using these input programs for generating variants that can be used for finding bugs in analyzers. To prioritize important warnings, prior techniques proposed improving the ranking of generated warnings [36, 44, 62]. Unlike these techniques that aim to enhance the results of static analyzers, our approach does not modify the ranking of generated warnings. We only report bugs in analyzers by generating test programs.

Several approaches have been proposed for testing static analyzers [19, 45, 67, 72]. These approaches have limited applicability because they either (1) rely on specialized oracles designed for a specific language or analyzer [1, 9, 19], (2) use a heavyweight an SMT solver for finding soundness and precision bugs [67], or (3) rely on programs from open-source projects instead of generating input programs [72]. Some differential testing approaches [45, 72] use several different implementations (model checkers [45], static analyzers [72]) as variants for testing. In contrast, we adopt metamorphic testing and use variants generated by transformations. Our experiments show our test generation approach is practical (compared to the approach that uses SMT solver) and can find bugs in many static analyzers. Bug injection approach is also adopted to test analyzers [10, 37, 50]. SolidiFI [29] evaluates the effectiveness of smart contract analyzers. It can generate mutants by injecting seven types of bugs. As injected bugs are well-known, SolidiFI can infer analysis results via inserted bug types. However, it cannot apply to general static analyzers because the issue types are determined by evaluated analyzers, and we cannot speculate whether evaluated tools can detect this issue without manual analysis. Besides, injected bugs in SolidiFI are specific to smart contracts. Parveen et al. [54] propose a mutation approach to evaluating taint analysis tools, but its mutators are designed for specific IoT applications, e.g., it mutates string literals in APIs like *sendPush*. Compared to previous approaches, our approach targets general-purpose analyzers.

Compiler Testing. Several techniques have been proposed for compiler testing [12–15, 65, 68, 75, 77], many of them rely on equivalent relation as the metamorphic relation to address lacking oracle problem [27, 47, 48, 51, 68]. Similarly, we also rely on equivalent relation (i.e., applying semantic-preserving program transformations to generate variants) to construct the oracle, but our transformation set is more diverse (e.g., the class-level transformations), which helps to test analyzers in different AST structures. Moreover, proposed heuristics can reduce locations to be transformed and select variants that are more likely to represent distinct bugs. Similar to existing techniques on JVM fuzzing [16, 17], our approach also generates Java programs. Different from these approaches that use various JVM implementations for differential testing, we perform metamorphic testing on static analyzers, and our heuristics find bugs in static analyzers instead of JVM implementations.

8 CONCLUSION AND FUTURE WORK

We present Statfier, a heuristic-based testing approach that automatically generates input programs via semantic-preserving program transformations for discovering bugs in static analyzers. Statfier relies on two key heuristics: analysis report guided location selection and structure diversity driven variant selection. Our experiments show that Statfier outperforms the evaluated baselines by finding more bugs yet iterating through fewer variants. Overall, Statfier has discovered 79 bugs, of which 46 have been confirmed. Our results suggest that developers of static analyzers can incorporate our approach into their test suites to further improve the bug detection capability. In fact, 26 of our generated input programs have been integrated into the official test suites of the evaluated static analyzers. While we focus on bugs that lead to differential analysis results in this paper, it would be worthwhile to study other bugs in static analyzers in the future (e.g., inconsistencies between documentation and the behavior of static analyzers [66]). Another direction is to improve the effectiveness of Statfier by tuning configurations of static analyzers as prior work shows that configurable software such as program verification tools can be tuned [46].

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their insightful comments. The work of Yu Pei was supported in part by Douyin Group (HK) Limited and in part by the CCF-Huawei Populus Grove Fund. This work was partially supported by the National Natural Science Foundation of China Grant Nos. 62002256, 62232001.

DATA AVAILABILITY

The experimental data and source code are available at: https://sa-research.github.io/.

REFERENCES

[1] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. 31–36.

[2] Cláudio A Araújo, Marcio E Delamaro, José C Maldonado, and Auri MR Vincenzi. 2016. Correlating automatic static analysis and mutation testing: towards incremental strategies. *Journal of Software Engineering Research and Development* 4, 1 (2016), 1–32.

[3] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. 241–252. https://doi.org/10.1145/1831708.1831738

[4] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and Yuqian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1–8.

[5] Belle. 2021. *[java] RSPEC-2189 should consider do-while loop.* Retrieved 2021-12-16 from https://community.sonarsource.com/t/java-rspec-2189-should-consider-do-while-loop/55080

[6] Belle-PL. 2022. *A false negative about NULL_DEREFERENCE.* Retrieved 2022-08-13 from https://github.com/facebook/infer/issues/1628

[7] Belle-PL. 2022. *MS_EXPOSE_REP cannot detect nested class.* Retrieved 2022-05-06 from https://github.com/spotbugs/spotbugs/issues/2042

[8] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.

[9] Ignacio Casso, José F Morales, Pedro López-García, and Manuel V Hermenegildo. 2020. Testing your (static analysis) truths. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 271–292.

[10] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 183–198.

[11] Hayden Cheers, Yuqing Lin, and Shamus P Smith. 2019. Spplagiarise: A tool for generating simulated semantics-preserving plagiarism of java source code. In *2019 IEEE 10th International conference on software engineering and service science (ICSESS)*. IEEE, 617–622.

[12] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 700–711.

[13] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[14] Junjie Chen and Chenyao Suo. 2022. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology* 31, 4 (2022), 1–33.

[15] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler Test-Program Generation via Memoized Configuration Search. In *45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2035–2047.

[16] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[17] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[18] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[19] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*. Springer, 120–125.

[20] Andreas Dangel. 2021. *How to contribute to PMD.* Retrieved 2021-09-06 from https://github.com/pmd/pmd/blob/49d35d0973d91dcd6526f67433ce701f2e291644/CONTRIBUTING.md

[21] Andreas Dangel. 2021. *[java] ArrayIsStoredDirectly doesn't consider nested classes.* Retrieved 2021-11-12 from https://github.com/pmd/pmd/issues/3613

[22] Andreas Dangel. 2021. *[java] UnusedFormalParameter doesn't consider anonymous classes.* Retrieved 2021-11-13 from https://github.com/pmd/pmd/issues/3618

[23] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 185–194.

[24] PMD Developers. 2023. *PMD Source Code Analyzer.* Retrieved 2023-05-30 from https://pmd.github.io/

[25] SpotBugs Developers. 2023. *SpotBugs.* Retrieved 2023-08-14 from https://spotbugs.github.io

[26] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 28–34.

[27] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (oct 2017), 29 pages. https://doi.org/10.1145/3133917

[28] Eclipse Foundation. 2023. *Eclipse Java development tools.* Retrieved 2023-06-16 from https://www.eclipse.org/jdt/

[29] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427.

[30] Alex Groce. 2023. *Regexp based tool for mutating generic source code across numerous languages.* Retrieved 2023-04-15 from https://github.com/agroce/universalmutator

[31] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qihong Chen. 2021. Evaluating and improving static analysis tools via differential mutation analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 207–218.

[32] A. Habib and M. Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 317–328. https://doi.org/10.1145/3238147.3238213

[33] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 317–328.

[34] Mark Harman. 2018. We need a testability transformation semantics. In *International Conference on Software Engineering and Formal Methods*. Springer, 3–17.

[35] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[36] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363–387.

[37] Yu Hu, Zekun Shen, and Brendan Dolan-Gavitt. 2022. Characterizing and Improving Bug-Finders with Synthetic Bugs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 971–982.

[38] Roman Ivanov. 2021. *ParameterAssignment does not detect problem for Lambda parameters.* Retrieved 2021-12-13 from https://github.com/checkstyle/checkstyle/issues/11038

[39] Roman Ivanov. 2023. *checkstyle.* Retrieved 2023-07-30 from http://checkstyle.sourceforge.net/

[40] Roman Ivanov. 2023. *How to report an issue?* Retrieved 2023-01-31 from https://checkstyle.sourceforge.io/report_issue.html

[41] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis.* MIT press.

[42] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[43] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[44] Sunghun Kim and Michael D Ernst. 2007. Which warnings should I fix first?. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 45–54.

[45] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 239–250.

[46] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S Foster, and Adam A Porter. 2021. SATune: a study-driven auto-tuning approach for configurable software verification tools. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 330–342.

[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[48] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[49] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.

[50] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 544–555.

[51] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE*

Asia Pacific Conference on Circuits and Systems (APCCAS). IEEE, 676–679.

[52] Kedar S Namjoshi and Zvonimir Pavlinovic. 2018. The impact of program transformations on static program analysis. In *International Static Analysis Symposium*. Springer, 306–325.

[53] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[54] Sajeda Parveen and Manar H Alalfi. 2020. A mutation framework for evaluating security analysis tools in IoT applications. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 587–591.

[55] Dino Distefano Peter O'Hearn and Cristiano Calcagno. 2015. *Open-sourcing Facebook Infer: Identify bugs before you ship*. Retrieved 2015-06-11 from https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/

[56] phoenix384. 2020. *[java] CompareObjectsWithEqualsRule: False positive with Enums*. Retrieved 2020-08-19 from https://github.com/pmd/pmd/issues/2716

[57] PMD. 2021. *HardcodedCryptoKey.xml*. https://github.com/pmd/pmd/blob/ac26d3dc6d7c121de72e6e7ddc1769caf8986e85/pmd-java/src/test/resources/net/sourceforge/pmd/lang/java/rule/security/xml/HardCodedCryptoKey.xml

[58] PMD. 2021. *[java] HardcodedCryptoKey false negative with variable assignments*. https://github.com/pmd/pmd/issues/3368#issuecomment-872794683

[59] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[60] sa-research. 2023. *Statfier Homepage*. Retrieved 2023-06-28 from https://sa-research.github.io/#timeout

[61] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[62] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. EFindBugs: Effective error ranking for findbugs. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 299–308.

[63] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 101–105.

[64] SonarSource. 2023. *Sonarqube code quality and code security*. Retrieved 2023-08-14 from https://www.sonarqube.org/

[65] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863.

[66] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.

[67] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 81–93.

[68] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*. IEEE, 270–279.

[69] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[70] Rijnard van Tonder and Claire Le Goues. 2020. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 824–834.

[71] vanguard 1024. 2022. *False positive about the rule UseStringBufferForStringAppends*. Retrieved 2022-08-13 from https://github.com/pmd/pmd/issues/4078

[72] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find Bugs in Static Bug Finders. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 516–527. https://doi.org/10.1145/3377811.3380380

[73] wuchiuwong. 2020. *Sonarqube and PMD*. Retrieved 2020-06-18 from https://github.com/wuchiuwong/Diff-Testing-01/blob/master/RulePair/Sonarqube%20and%20PMD.csv

[74] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. 2020. METTLE: A metamorphic testing approach to assessing and validating unsupervised machine learning systems. *IEEE Transactions on Reliability* 69, 4 (2020), 1293–1322.

[75] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. 2023. Silent Compiler Bug De-duplication via Three-Dimensional Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 677–689.

[76] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 132–142.

[77] Yingquan Zhao, Junjie Chen, Ruifeng Fu, Haojie Ye, and Zan Wang. 2023. Testing the Compiler for a New-Born Programming Language: An Industrial Case Study (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 551–563.

[78] Zustin. 2021. *New Java versions lead to FN about the rule DMI_INVOKING_TOSTRING_ON_ARRAY*. Retrieved 2021-12-14 from https://github.com/spotbugs/spotbugs/issues/1874

[79] Zustin. 2022. *A false positive about the rule RedundantFieldInitializer*. Retrieved 2022-07-28 from https://github.com/pmd/pmd/issues/4070

Received 2023-02-02; accepted 2023-07-27