An Empirical Study of False Negatives and Positives of Static Code Analyzers From the Perspective of Historical Issues

HAN CUI, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
JINGJING LIANG*, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
MENGLEI XIE, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
JIAHAO PENG, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
TING SU, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
CHENGYU ZHANG, Loughborough University, United Kingdom
SHIN HWEI TAN, Concordia University, Canada

Static code analyzers are widely used to help find program flaws. However, in practice the effectiveness and usability of such analyzers is affected by the problems of false negatives (FNs) and false positives (FPs). This paper aims to investigate the FNs and FPs of such analyzers from a *new* perspective, *i.e.*, examining the historical issues of FNs and FPs of these analyzers reported by their maintainers, users and researchers in their issue repositories — each of these issues manifested as a FN or FP of these analyzers in the history and has already been confirmed and fixed by the analyzers' developers. To this end, we conduct the *first* systematic study on a broad range of 1257 historical issues of FNs/FPs from four popular rule-based static code analyzers for Java (*i.e.*, PMD, SpotBugs, SonarQube, and ErrorProne). All these issues have been confirmed and fixed by the developers. We investigated these issues' root causes and the characteristics of the corresponding issue-triggering programs. It reveals several new interesting findings and implications on mitigating FNs and FPs. Furthermore, guided by some findings of our study, we designed a metamorphic testing strategy to find FNs and FPs. This strategy successfully found 15 new issues of FNs/FPs, 12 of which have been confirmed and 9 have already been fixed by the developers. Our further manual investigation of the studied analyzers revealed one rule specification issue and additional three FNs/FPs due to the weaknesses of the implemented static analysis. We have made all the artifacts (datasets and tools) publicly available at *https://zenodo.org/doi/10.5281/zenodo.11525129*.

CCS Concepts: • **Software and its engineering → Software reliability**.

Additional Key Words and Phrases: Static Code Analyzer, False Negative, False Positive

*Jingjing Liang is the corresponding author.

Authors' Contact Information: Han Cui, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, cuihan0928@163.com; Jingjing Liang, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, jjliang@sei.ecnu.edu.cn; Menglei Xie, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, ml_xie@hotmail.com; JiaHao Peng, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, 51285902022@stu.ecnu.edu.cn; Ting Su, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, tsu@sei.ecnu.edu.cn; Chengyu Zhang, Loughborough University, United Kingdom, c.zhang4@lboro.ac.uk; Shin Hwei Tan, Concordia University, Canada, shinhwei.tan@concordia.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/9-ART
https://doi.org/10.1145/3849701

Just Accepted

1 Introduction

Static code analyzers (*or* static checkers) [80] are commonly used to help find program flaws at the early stages of software development (after code compilation and before testing) [59, 83, 89, 95]. These analyzers usually target various types of program flaws including violations of best practice, code design issues, common programming mistakes and security vulnerabilities [65, 95, 104]. Otherwise, these flaws might be costly and difficult to find by manual code reviews or testing [57, 71, 111]. Typically, to find these flaws, these analyzers implement a number of *checking rules* (or *rules* for short), supported by some forms of static analysis with different complexities (*e.g.*, syntactic pattern matching, control-/data-flow analysis, symbolic execution).

In practice, the effectiveness and usability of analyzers could be affected by the problems of *false negatives* (*FN*s for short), *i.e.*, missing *true* program flaws [68, 99, 100], and *false positives* (*FP*s for short), *i.e.*, the reported warnings are *spurious* [71, 74, 105]. To investigate FNs and FPs, most of existing studies [68, 74, 75, 77, 78, 84, 99–101, 105] evaluate the fault detection abilities or the usability of these analyzers by running them on datasets with known faults. For example, Habib *et al.* [68] use the faults of Defects4J to investigate FNs of three static code analyzers, *i.e.*, ErrorProne [37], Infer [12], and SpotBugs [36]. The results show that these analyzers could *only* find 4.5% of the field defects. Wedyan *et al.* [105] use the known coding faults from some open-source projects to investigate FPs and find that 96% of the warnings reported by three analyzers (*i.e.*, IntelliJ IDEA [16], FindBugs [70], Jlint [4]) are spurious. The main goal of such studies is to assess the effectiveness and usability of the evaluated analyzers in terms of *general* FNs and FPs. For example, these studies find that most known defects are missed because these defects are not targeted by existing rules in the analyzers or these defects are domain-specific errors [68, 75, 78, 99, 100].

Different from prior studies that focus on evaluating the effectiveness of static analyzers on existing datasets (*i.e.*, measuring FN/FP), this paper aims to investigate the FNs and FPs from a new perspective, *i.e.*, examining the *historical*, *fixed* issues of FNs and FPs of these analyzers in their own issue repositories. Each issue represents a confirmed FN or FP that has been identified and resolved by the analyzer developers. Our study investigates two key aspects: (1) the *root causes* of why these issues arise, and (2) the *input characteristics* of programs that trigger them. Figure 1 shows an illustrative example: a false positive in ErrorProne that was reported and fixed. The input program declares and initializes an underscore (var '_') placeholder. ErrorProne flagged an UnusedVariable [40] warning on this variable, as the variable '_' was declared but never used in subsequent code. This is a false positive because Java 22's unnamed variable feature uses the underscore to explicitly indicate that the value is intentionally unused. The discussion reveals the root cause: ErrorProne had not been updated to support this new language feature. The triggering input characteristic is the declaration and initialization of underscore-based variables (var '_'). By studying such historical fixes, we identify that insufficient support for new language features is a cause of static analyzer false positives or false negatives, providing implications for analyzer developers. To our knowledge, no prior work systematically studies such historical fixed issues.

In this paper, we studied a broad range of 1257 historical issues (corresponding to 187 FNs and 1070 FPs), which were collected from the issue repositories of four popular, open-source, rule-based static analyzers for Java (*i.e.*, PMD [17], SpotBugs [36], SonarQube [60], and ErrorProne [37]). For each analyzer, we collected all historical issues from the first reported issue up to the start of our study in June 2025, covering a time span of eight to ten years. All these issues are valid and representative because they have been confirmed *and* already fixed by the developers. To our knowledge, this is the *first* systematic study to investigate the FNs and FPs of static code analyzers from the perspective of historical issues. We will discuss and compare with the relevant work [103, 109] in detail in Section 9.

Our study aims to answer the the following three research questions:

- **RQ1 (Root Causes)**: What are the common root causes of these historical issues of FNs and FPs affecting the static code analyzers? Root causes characterize the underlying reasons of FNs/FPs in the source code of

Just Accepted

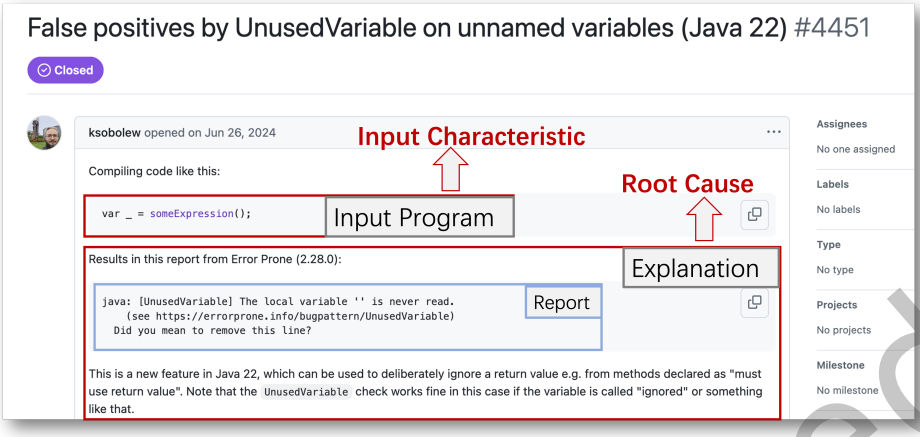


Fig. 1. A historical false positive issue fixed in ErrorProne (#4451).

static analyzers. We aim to investigate the root cause by examining the analyzers' documentations (*e.g.*, rule specifications), the issue reports, the issue-triggering programs and the fixing patches (detailed in Section 4).

- **RQ2 (Input Characteristics)**: What are the characteristics of input programs leading to these historical issues of FNs and FPs of the static code analyzers? Input characteristics capture the triggering conditions in input programs that are likely to expose FN/FP issues in static analyzers. We aim to investigate the input characteristics by examining the analyzers' documentations (*e.g.*, rule specifications), the issue reports and the issue-triggering programs and their representations (detailed in Section 5).

Answering these two questions is beneficial to both the analyzers' developers and the researchers in this field. The answers can provide new insights on how to improve these analyzers (*e.g.*, mitigating or unveiling FNs and FPs), thus complementing those general studies on only evaluating the effectiveness or the usability of these analyzers [68, 74, 75, 77, 78, 84, 99–101, 105]. Through answering **RQ1** and **RQ2**, we obtained several new interesting findings that shed light on mitigating FNs/FPs. Based on these findings, we further designed proof-of-concept demonstrations to improve the quality of static analyzers.

- **RQ3 (Usefulness of Our Findings)**: How can some of our study's findings from **RQ1** and **RQ2** help identify issues of FNs and FPs in these static code analyzers? We aim to show some proof-of-concept demonstrations (in Section 6) to demonstrate the usefulness of our study's findings.

Specifically, we designed a metamorphic testing strategy to automatically find FNs/FPs (Section 6.1). This strategy helped us find 15 new issues (13 FNs and 2 FPs) of the studied analyzers, 12 of which have been confirmed and 9 have been fixed. Additionally, we further manually examined the implementations of some rules of the studied analyzers (Section 6.2). We found one rule specification issue and additional three FNs/FPs due to the weaknesses of the implemented static analysis. We have reported these issues to the developers and all have been confirmed/fixed.

In summary, our work has made the following contributions:

- We conduct the *first* systematic study to investigate FNs and FPs of static code analyzers from the new perspective of the historical issues. We construct a dataset of 1257 historical issues of FNs and FPs from the studied analyzers to serve as the basis of our study and future research in this field.
- We study the 1257 historical issues of FNs and FPs to investigate their root causes and input characteristics and identify several new interesting findings.

Just Accepted

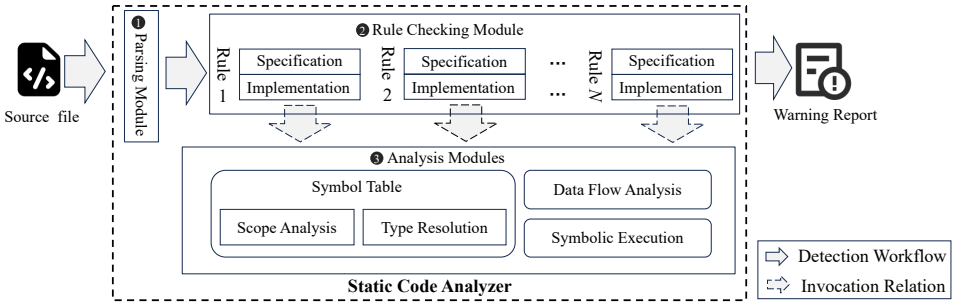


Fig. 2. Typical workflow and architecture of a static code analyzer.

- We design and implement two proof-of-concept demonstrations guided by our findings. These demonstrations successfully found 19 previously unknown bugs, demonstrating the usefulness of our findings in mitigating FNs and FPs.
- We discuss the implications of our study and demonstrations and provide four practical strategies for improving static code analyzer quality, along with one critical challenge for future research.
- We have made all the artifacts (datasets and tools) publicly available at *https://zenodo.org/doi/10.5281/zenodo.11525129*.

2 Background on Static Code Analyzers

Figure 2 shows the typical workflow and architecture of classic static code analyzers like PMD [17], SpotBugs [36], SonarQube [60], and ErrorProne [37]. Static code analyzers typically consist of three core components: the *Parsing Module*, the *Rule Checking Module*, and the *Analysis Modules* with various basic analysis techniques. Given a file under check, the parsing module transforms it into an intermediate representation (IR), *e.g.*, an AST or a CFG/DFG. The rule checking module then evaluates predefined rules over this IR, often invoking basic analysis techniques in the analysis modules (depending on the analyzer's design), such as syntactic pattern matching, data-flow analysis, or symbolic execution. Finally, detected violations are reported with diagnostic messages along with recommendations for potential fixes. While most analyzers generally involve these three components, their specific implementations may differ. We provide a detailed introduction to the four static analyzers (PMD, SpotBugs, SonarQube, and ErrorProne) examined in this study as follows:

The *parsing module* differs across analyzers. Although all of the four static analyzers operate on Java source code, they produce different forms of IRs to facilitate subsequent analysis. For example, PMD and ErrorProne parse source code into ASTs, while SpotBugs parses source code into CFGs. SonarQube parses the source code into both ASTs and CFGs.

In the *rule checking module*, an analyzer usually implements a number of rules, each of which has its own specification and implementation (*i.e.*, PMD's rules [13], SpotBugs's rules [5], SonarQube's rules [14], ErrorProne's rules [41]). Each rule is designed to detect a specific type of program flaw, such as *best practice violations*, *code design issues*, *common programming mistakes*, or *security vulnerabilities*.

The *analysis modules* implemented by different static analyzers vary. For example, all analyzers construct symbol tables in their analysis modules. Among them, only SonarQube combines symbolic execution with data-flow analysis. PMD, SpotBugs, and ErrorProne rely solely on data-flow analysis, though their capabilities differ. Specifically, PMD and ErrorProne provide only specific data-flow analysis,*i.e.*, PMD implements reaching-definition analysis, while ErrorProne offers the nullness related analysis [97]. In contrast, SpotBugs implements a more general data-flow analysis framework, supporting both forward and backward analyses.

Just Accepted

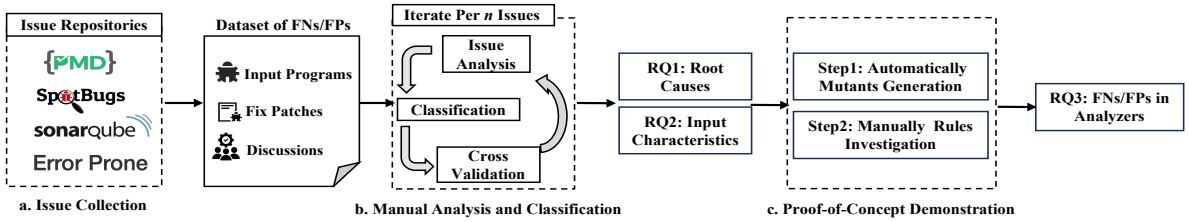


Fig. 3. Workflow of our study.

3 Study Methodology

This section details the methodology of our study. Specifically, Section 3.1 introduces the selected analyzers for our study, Section 3.2 presents how we collect the historical issues of FNs and FPs from the issue repositories of the studied analyzers, and Section 3.3 explains how we manually analyze these issues.

3.1 Static Code Analyzers Selected for Our Study

In this works, we focus on the static code analyzers for Java language because Java is one of the most popular programming language and targeted by (many) existing analyzers. Specifically, we selected the four representative rule-based static code analyzers for Java, *i.e.*, PMD [17], SpotBugs [36], SonarQube [60], and ErrorProne [37], as the subjects for our study based on the following reasons. First, these analyzers are popular. For example, PMD has been integrated into several industrial IDEs (*e.g.*, Eclipse, IntelliJ IDEA, Visual Studio Code); SpotBugs, the successor of FindBugs [70], and ErrorProne have been used by Google; SonarQube has been applied at the CI pipelines by many companies. Moreover, these analyzers have been widely studied by many prior work [62, 68, 74, 82, 90, 103, 105]. Second, these analyzers are open-sourced, facilitating issue collection and analysis. We can inspect the fixing patches and tool implementations to investigate FNs and FPs. Third, these analyzers adopt different analysis strategies, including syntactic pattern matching, data-flow analysis and symbolic execution. These characteristics can help us gain more overall contextual understanding on FNs and FPs. There are other static code analyzers for Java [88] like FindBugs [70], Infer [12] and CheckStyle [9], but we did not select these analyzers as the subjects. We did not select FindBugs because it is deprecated and has not been maintained for almost ten years (its latest version was released in 2015). FindBugs's development has moved to SpotBugs (which we studied in this work). Moreover, FindBugs do not maintain the issues of FNs and FPs (*e.g.*, not labeling the FNs/FPs or linking with the fixing patches), Infer has only a few checking rules (i.e., 25 rules) and thus few issues of FNs/FPs with respect to those rules, and CheckStyle focuses more on coding standards rather than program flaws.

Figure 3 illustrates the overall workflow of our study, which consists of three phases: (a) *Issue Collection*: we collect issues from the repositories of static analyzers and extract three types of artifacts for each issue: the input programs, the fix patches, and the developers' discussions and comments; (b) *Manual Analysis and Classification*: we apply open coding [96] to analyze each issue. This phase addresses RQ1 (root causes), RQ2 (input characteristics) and the findings derived from both RQs; (c) *Proof-of-Concept Demonstration*: we leverage the findings from RQ1 and RQ2 to identify new issues in static analyzers, investigating RQ3. We detail the collection process in Section 3.2, the analysis methodology in Section 3.3, and the proof-of-concept demonstration in Section 6.

3.2 Collecting Historical Issues of FNs and FPs

To obtain a broad range of issues, we collected *all* the reported FNs/FPs issues before the time of our study (which was started in Jun. 2025). We require that these FNs/FPs have been confirmed *and* fixed by the developers because

Just Accepted

Table 1. Statistics of the studied static code analyzers (K=1,000), and the dataset of valid FNs and FPs.

Analyzer	GitHub #Stars	#Total Java Rules	Date of first collected issue	Date of last collected issue	#Valid Issues (FP/FN)
PMD	4.3K	325	2017/2/27	2025/6/17	394(296/98)
SpotBugs	3K	468	2016/12/7	2025/5/23	98(79/19)
SonarQube	7.7K	618	2017/3/31	2025/5/16	706(642/64)
ErrorProne	7K	621	2014/10/31	2025/5/22	59(53/6)

the confirmed bug information, the committed patches, and the discussions between the developers provide us informative details to study the root causes and the input characteristics of these issues. Note that all fixed issues in our study represent implementation or rule bugs rather than specific design decisions. These decisions reflect intentional architectural choices and are not acknowledged or fixed by developers.

We detail the issue collection process below. We automatically collected all fixed issues based on their labels or keywords in the issue, such as "false positive", "false negative", "FP", "FN". We then manually filtered out non-FN/FP issues and duplicates. For PMD, an issue is considered as *confirmed* if PMD's developers explicitly labeled the issue with "a:false-positive" or "a:false-negative". For SpotBugs, SonarQube and ErrorProne, an issue is considered as *confirmed* if the developers explicitly confirmed the issue is a FN or a FP during discussions. An issue is considered as *fixed* if the issue report has been closed and associated with fixing commits.

- **PMD**. We collected all the *closed* issues with the labels *"a:false-positive"* and *"a:false-negative"* that are associated with fixing commits from PMD's issue repository on GitHub [42]. To collect issues that are related to Java programs, we filtered out the issues that do not contain the keyword *"java"* in the issue title or body. We initially obtained 168 FNs and 596 FPs.
- **SpotBugs & ErrorProne**. Both of the repositories of these two projects on GitHub do not provide explicit issue labels. Thus, we collected issues from their repositories (*i.e.*, SpotBugs [55] and ErrorProne [38]) and checked whether the issue title or body contains the keywords *"FP"*, *"FN"*, *"bug"*, *"error"*, *"false positive"*, *"false alarm"*, and *"false negative"* to filter issues. In total, we obtained 399 unclassified issues for SpotBugs and 504 for ErrorProne.
- **SonarQube**. We collected SonarQube issues from two sources: (1) Jira, the official issue tracker [54], and (2) the Sonar community [35],where issues are publicly discussed. *From Jira*, we collected issues in the SonarJava project labeled as *"False Positive"* or *"False Negative"* with status marked as *"finished"*, yielding 685 issues. We collected issues from the SonarJava project rather than from SonarQube itself, as our study focuses on Java-related FN/FP issues, which are reported and fixed only in SonarJava. SonarQube is a general framework that provides the core infrastructure, while the detection for individual programming languages is delegated to language-specific plugins (*e.g.*, SonarJava for Java, SonarPython for Python). Consequently, the issues reported in the SonarQube project are mostly framework-related and do not include FN/FP issues. *From the Sonar community*, we used the Sonar community APIs to collect issues labeled with *"Clean Code"*, *"Report False-positive / Report False-negative"*, *"java"*, *"answered"*, and *"closed"*, yielding 262 issues. After manually removing 97 duplicates across the two sources, we obtained a total of 850 unclassified issues.

We manually verified whether each collected issue is a valid FN or FP by inspecting the issue report and reproducing the issue if necessary. During this process, we excluded non-FN/FP or duplicated issues, won't-fix issues of deprecated rules, and the issues caused by user misconfigurations. We only retained those developer-confirmed *and* -fixed FNs/FPs. Finally, we obtained a dataset of 1257 FNs/FPs, including 98 FNs/296 FPs from PMD, 19 FNs/79 FPs from SpotBugs, 64 FNs/642 FPs from SonarQube, and 6 FNs/53 FPs from ErrorProne. Each issue is associated with at least one piece of issue-triggering program code from the issue report. Table 1 gives the detailed statistics of the collected issues. Note that many issues of Spotbugs were excluded because they were not confirmed by the developers, *e.g.*, the issue reporters misunderstood or misconfigured the rules,

Just Accepted

or the issue-triggering programs were deemed as unrealistic (not likely written by humans). Many issues of ErrorProne were excluded because they were crash bugs that are outside the scope of our study. The issues which were closed but not confirmed were also excluded from our study.

3.3 Analyzing Historical Issues of FNs and FPs

To answer RQ1 and RQ2, we investigated 1257 FNs/FPs in the dataset to understand the root causes and input characteristics, following the approach widely adopted in prior studies on software bug analysis [64, 76, 79, 93, 94, 107]. Specifically, for each issue, we assigned it to *exactly one category* for both root cause and input characteristics based on its primary evidence. *For identifying root causes*, we inspected the fixing patches, discussions, and the rule specifications. Issue labels (e.g., *dataflow analysis*) in the discussions served as supplementary indicators of the likely root cause. To confirm our understanding, we inspected the fixing patches (*i.e.*, fixing commits or pull requests) and reproduced the issue against the rule implementation if necessary. *For identifying input characteristics*, we inspected the issue-triggering program and the discussions in the issue reports. Keywords (e.g., *anonymous class*) in the issue titles served as additional indicators of the input characteristics leading to the issue. When the triggering input was unclear, we further performed targeted modifications to the program to identify the code constructs that were critical for triggering the FNs/FPs. Specifically, we manually removed or altered suspicious code constructs and reran the static analyzer. If the corresponding FN/FP disappeared after modifying a specific construct, we considered this construct to be pivotal in triggering the issue.

Specifically, to build taxonomies, we adopted the open card sorting approach [96] and conducted the preceding analysis in an iterative process. In each iteration, 30 issues in the dataset were randomly selected and two of the co-authors independently studied each of these issues. These two co-authors are familiar with Java, each having 5 years of Java programming experience, and the relevant static analysis techniques. According to their own understanding, they independently labeled each issue with the categories of root causes and input characteristics. Afterward, these two co-authors cross-validated and discussed the labels until they reached a consensus on the categorized results. When they could not reach a consensus, the other co-authors participated in discussions to help make the final decision. Such an iteration was repeated forty-two times until all 1257 issues were analyzed. We observed that this iterative process converged on the categories after the first 3 rounds.

Regarding annotation effort, the entire process took approximately ten person-months (~1,600 person-hours) in total. The two annotators each independently annotated all 1257 issues, spending approximately 700 hours each. The time required per issue varied depending on the sufficiency of the information provided (*e.g.*, whether the fix patch and discussions were detailed enough), averaging ~0.5 person-hours per issue. An additional ~200 person-hours were spent on resolving disagreements and reaching consensus through discussion among all authors.

During this manual analysis process, we computed the Cohen's Kappa coefficient [58] to evaluate the inter-rater agreement [56, 91] between the two co-authors for each of the dimensions analysed (root cause and input-program characteristic). Cohen's Kappa is a statistical measure used to quantify the level of agreement between two raters beyond what would be expected by chance. It is particularly useful when evaluating subjective judgments, where the goal is to determine how consistently two or more raters classify or label the same items. The high inter-rater agreement indicates that the raters are in close alignment in their judgments, while low agreement suggests differences in their evaluations. By applying Cohen's Kappa, we assessed whether the categories of root causes and input characteristics assigned by the co-authors were consistent and reliable, minimizing the impact of individual biases. In the first three rounds, the Cohen's Kappa coefficient was around 0.5, primarily because these two co-authors were still unfamiliar with the FNs and FPs of the static analyzers and hadn't yet reached a consensus. As their understanding deepened and they gained more experience, the coefficient increased to 0.9 in subsequent rounds. After each round, the co-authors discussed disagreements and refined the classification

Just Accepted

Table 2. The taxonomy of root causes of FNs and FPs from the four static code analyzers.

Category	Description	Example	#Issues	Ratio	Module
Rule Defects	Incorrect or incomplete rules.		74	5.9%	-
○○ Flawed Rule Specification	Some design flaws in rule specification.	Figure 4	42	3.3%	
◐○ Incomplete Rule Design	Missing relevant code patterns in rule specification.	Figure 5	32	2.5%	
Semantics Misunderstanding	Incorrect program semantics interpretation.		772	61.4%	**Rule Checking**
○● Unsupported Language New Features	Not adapted to language new features.	Figure 6	64	5.1%	
○○ Language Semantics Misunderstanding	Misunderstands established language semantics.	Figure 7	206	16.4%	
○○ Unsupported Frameworks and Libraries	Fails to handle third-party frameworks.	Figure 8	266	21.2%	
○○ API Semantics Misunderstanding	Misunderstands standard API behavior.	Figure 9	118	9.4%	
○○ Domain-Specific Semantics Misunderstanding	Lacks domain-specific knowledge.	-	118	9.4%	
○◐ **Mishandling Intermediate Representations**	Incorrect IR handling.	-	52	4.1%	**Parsing or Analysis**
○○ **Mishandling Configurations**	Incorrect configuration processing.	Figure 10	22	1.75%	**Rule Checking**
Analysis Module Error or Limitation	Errors or limitations in analysis modules.		337	26.8%	**Analysis**
○○ Scope Analysis Error or Limitation	Wrong resolving of the variable scopes.	Figure 11	65	5.2%	Scope
○○ Type Resolution Error or Limitation	Incorrect or limited type resolution.	Figure 12	138	11.0%	Type
○○ Dataflow Analysis Error or Limitation	Errors in the dataflow analysis or limited use of dataflow analysis.	Figure 13	114	9.1%	Dataflow
○○ Symbolic Execution Error or Limitation	Errors or limitations in the symbolic execution engine.	Figure 14	20	1.6%	Symbolic

The symbols "◐" and "○", "●", "◐", and "○" denote the differences between our study and the two most relevant work from Wang *et al.* [103] and Zhang *et al.* [109], respectively, in terms of the results of issue analysis. We discuss the differences in detail in Section 9.

criteria. Once the coding scheme stabilized, they revisited the annotations from earlier rounds and updated them when necessary according to the final consensus. Through this iterative refinement process, all annotations across all rounds eventually achieved perfect agreement with a Cohen's Kappa value of 1.0.

Our study primarily focuses on the root causes and input characteristics of FNs and FPs, and has not explored other possible dimensions (*e.g.*, severity, latency, and fixability) because these other dimensions are difficult to be analyzed or may bring additional threats to validity. For example, neither PMD nor SpotBugs maintains severity labels, and ErrorProne provides them for only 14% of issues. Since not all issues have severity labels, conducting a comprehensive and systematic evaluation of severity is difficult. The latency of fixing FNs/FPs may or may not indicate the difficulty of fixing FNs/FPs. The fixability is also difficult to measure in an objective manner because some issues were fixed by workarounds.

4 RQ1: Root Causes

In this section, we focus on the 1257 issues in the four analyzers to study the root causes. We identified the 5 major root causes and 13 subcategories from 1257 FNs/FPs. As shown in Table 2, we categorized the issues into disjoint groups of root causes based on the developers' discussion and the three core components in which the errors occur (see Figure 2). The column "Module" in Table 2 shows the three core components. Note that *Rule Defects* is independent of the implementation and is therefore not associated with any particular component. Root causes such as *Semantics Misunderstanding* and *Mishandling Configurations* are mainly due to errors in the *Rule Checking* module, while *Mishandling Intermediate Representations* is caused by errors in the *Parsing* or *Analysis* module. Other analysis-related limitations fall into the *Analysis* modules, which are further divided into

Just Accepted

submodules such as scope, type, dataflow, and symbolic execution modules. We explain and illustrate these root causes as follows.

4.1 Rule Defects

Rule defects refer to issues stemming from flaws in the design of detection rules, where the rule specification is logically incorrect or fails to comprehensively cover all intended detection scenarios, consequently leading to incorrect implementations and incorrect analysis results. This category accounts for 5.9% of all identified bugs.

4.1.1 Flawed Rule Specification (FRS). In this category, the rule specification itself (documented in natural language) is flawed, thus leading to FNs or FPs. As shown in Figure 4, PMD's rule *DoNotUseThreads* [11] intends to warn against the direct use of threads (*e.g.*, `Thread`, `ExecutorService`) in favor of J2EE's managed thread mechanism. However, this rule is incorrectly designed to warn against the use of `Runnable` on the assumption that developers misunderstand that `Runnable` is identical to `Thread`. In fact, `Runnable` is a class whose instances are intended to be executed by a thread. It is compliant with the managed thread environment in J2EE, and thus should not require a warning. Due to this flawed specification, this rule reports a FP when the input program uses `Runnable`. To resolve this issue, the developer fixed the rule specification and implementation[1].

```
1  @Override
2  public void eval(final ExecutionContext context) {
3      ...
4      // FP, but this is just a delayed computation, no threads involved...
5      final Runnable yieldEvaluation = ()-> ifPresent(yield, it ->it.evalInNewStackFrame(context, YIELD));
6      ...
7  }
```

Fig. 4. PMD#1627: Flawed Rule Specification.

4.1.2 Incomplete Rule Design (IRD). This category refers to bugs caused by incomplete design of detection rules. The rules themselves are conceptually sound, but they miss relevant code patterns, resulting in incomplete coverage. Although the rule's intent is correct, its specification fails to enumerate all scenarios that should be detected. For example, Figure 5 presents a false negative bug in PMD's *SimplifiedTernary* rule [49], which is designed to detect ternary expressions that can be simplified (*i.e.*, reduced to a more concise logical form), such as transforming ``` ``condition ? true : expr'' ``` into ``` ``condition || expr'' ```. However, the rule overlooks the pattern ``` ``condition ? true : false'' ```, which is can be simplified into ``` ``condition'' ```. This omission leads to false negatives because a simplifiable expression within the rule's intended scope is not reported. The issue was later fixed by updating both the rule specification and its implementation to incorporate this missing case[2].

```
1  public boolean test5() {
2      return condition ? true : false;   // FN, it can be as simple as return condition
3  }
```

Fig. 5. PMD#3603: Incomplete Rule Design.

[1]https://github.com/pmd/pmd/pull/2078/commits/5739041b164d0bb4cc94715aa3bed801c4565e02
[2]https://github.com/pmd/pmd/pull/3938

Just Accepted

Finding 1: *Rule Defects* account for 5.9% of all bugs, requiring updates to rule specifications beyond mere implementation fixes. This highlights specification quality as an important yet challenging aspect of analyzer development, as accurate and complete specifications are difficult to maintain due to evolving language features, libraries, and usage scenarios.

4.2 Semantics Misunderstanding

Semantic misunderstanding refers to cases where the analyzer fails to correctly interpret the intended semantics of the program, including runtime behavior, language feature, API contracts, framework conventions, or domain-specific abstractions. These issues stem from *knowledge gaps* in understanding what the code is intended to do according to language specifications, API documentation, framework semantics, or domain-specific knowledge. Bugs in this category account for 61.4% of all analyzed issues, making it one of the most significant root causes.

4.2.1 Unsupported Language New Features (ULF). This category captures bugs arising when static analyzers have not yet adapted to *newly introduced language features* from recent Java specifications. As the Java language evolves, new constructs such as `Record` classes (Java 14+), `Sealed` classes (Java 17+), pattern matching for `instanceof` (Java 16+), and enhanced `switch` expressions (Java 14+) are introduced. When analyzers fail to recognize or correctly handle these new features, they may produce false positives or negatives. Figure 6 illustrates one such case. Java 22 introduces ``` ``_'' ``` as a legal local variable name to deliberately ignore a method's return value, signaling to both the compiler and static code analyzers that the value is intentionally discarded. Prior to supporting this feature, ErrorProne's *UnusedVariable* rule [40] incorrectly reported a false positive. The developers fixed this issue by adding the variable name ``` ``_'' ``` to the list of variables exempted from the *UnusedVariable* warning[3].

```
1  public static void main(String[] args) {
2      var _ = new Object();    // FP, "_" is used to deliberately ignore a return value.
3  }
```

Fig. 6. ErrorProne#4451: Unsupported Language Features.

4.2.2 Language Semantics Misunderstanding (LSM). This category captures cases where the analyzer misunderstands the semantics of *long-established language features*. Unlike *Unsupported New Language Features*, these features have already been integrated into analyzers, but their inherent complexity causes incorrect semantic modeling, leading to false positives or negatives. We identified 42 concrete language features in this type, including *object-oriented principles* (*e.g.*, *modifiers, generics*), *exception handling*, *resource management*, *arrays accessing*, and *lambda expressions*. Table 3 presents the high-frequency features (appearing in more than 7 bugs). Notably, correctly modeling only these 8 features would prevent 49.0% of bugs in this category, corresponding to approximately 8.0% of all bugs in our dataset. Figure 7 illustrates a *type semantics* misunderstanding case, where the SonarQube rule *S3518* [33] reports a division-by-zero error on a floating-point expression as throwing an `ArithmeticException`. However, in Java, division by zero on `double` or `float` values yields `Infinity` or `NaN` rather than raising an exception. SonarQube incorrectly applies integer division semantics to floating-point operations, failing to distinguish between these different type behaviors, resulting in a false positive.

4.2.3 Unsupported Frameworks and Libraries. Bugs in this category occur when static analyzers fail to properly handle third-party frameworks and their annotations. The Java ecosystem includes numerous frameworks like Spring, Lombok, JUnit, and JPA that provide annotations or implicit behaviors essential for correct code analysis. When analyzers do not recognize these constructs, they generate false positives or negatives. We identified

[3]https://github.com/google/error-prone/commit/6bd568d418694043bfec7961dff0ba5559bb856a

Just Accepted

Table 3. Most Commonly Misunderstood Language Features (Count > 7)

	Lambda	Modifier	Exception	Resource	Type	Interface	ControlFlow	Array	Total
Count	28	15	15	11	8	8	8	8	101
Percent	13.6%	7.3%	7.3%	5.3%	3.9%	3.9%	3.9%	3.9%	49.0%

```
1  float d = 0;
2  float f = 1;
3  System.out.println(f / d);  // FP, No exception raised, expression evaluates to Infinity
```

Fig. 7. SonarQube#3026: Language Semantics Misunderstanding.

98 third-party frameworks that triggered such bugs. Table 4 shows the most frequently involved frameworks (appearing in more than 5 bugs). Supporting these 6 frameworks would prevent 52.6% of bugs within this category, which accounts for approximately 11.2% of all bugs in our dataset. Figure 8 illustrates an example involving `lombok.val`, which is semantically equivalent to a `final` local variable. PMD's *MethodArgumentCouldBeFinal* rule [48] did not account for this, resulting in a false positive. The developers fixed this issue by including `lombok.val` in the conditions for detecting `final` variables[4].

Table 4. Most Commonly Unsupported Frameworks (Count > 5)

	Lombok	Spring	JUnit5	AssertJ	Guava	JPA	Total
Count	54	32	23	13	12	6	140
Percent	20.3%	12.0%	8.7%	4.9%	4.5%	2.3%	52.6%

```
1  import lombok.val;
2  public class Test {
3      private StringBuilder createFileWatcher() {
4          val tost = "uaoleudaou";   // FP, lombok.val is equivalet to final class
5          return new StringBuilder(tost);
6  }
```

Fig. 8. PMD#5079: Unsupported Frameworks and Libraries.

4.2.4 API Semantics Misunderstanding. This category refers to bugs where the analyzer misunderstands the semantic behavior of *standard Java APIs* from packages such as `java.util.*`. The analyzer correctly understands the general rule but fails to recognize that certain API calls require exceptional handling in specific contexts. A common fix is to add the specific API to a whitelist or exception list. Figure 9 shows a false positive in SonarQube's rule *S2201* [24], which flags ignored return values of side-effect-free functions. However, SonarQube incorrectly treats `ConcurrentMap.putIfAbsent()` as side-effect-free, overlooking that it modifies the map by inserting a key-value pair when the key is absent. The developers fixed this by excluding `ConcurrentMap.putIfAbsent()` from the checked methods[5].

4.2.5 Domain-Specific Semantics Misunderstanding. This category captures bugs arising from the analyzer's lack of understanding of domain-specific idioms and semantics. Such specialized domain knowledge includes *logging frameworks, regular expressions, concurrency primitives, or date/time specifications.* SonarQube's Issue

[4]https://github.com/pmd/pmd/pull/5600
[5]https://github.com/SonarSource/sonar-java/pull/2461

Just Accepted

```
1  ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
2  map.putIfAbsent("key", "value");  // FP: putIfAbsent has side effects, ignoring the return value is
   acceptable
```

Fig. 9. SonarQube#3046: API Semantics Misunderstanding.

#3890 illustrates a false positive in SonarQube's rule *S5996* [34], which detects unreachable patterns in regular expressions. The rule flags the pattern `^foo$\r?\n?` as containing unreachable components, reasoning that since `$` denotes the end of a line, no characters should be allowed to follow it. However, `$` does not strictly prohibit subsequent characters. It matches the position before a line terminator and explicitly allows trailing newline characters (`\r` or `\n`) to follow. Since SonarQube lacks this domain-specific knowledge of regex anchor behavior, it incorrectly concludes that `\r?\n?` is unreachable, resulting in a false positive.

Finding 2: *Semantic Misunderstanding* affects 772 of the 1257 issues (61.4%), which is the most common root cause. Among its subcategories, *Unsupported Frameworks and Libraries* (266, 21.2%) and *Language Semantics Misunderstanding* (206, 16.4%) are the two dominant sources of bugs. Notably, a small subset of high-frequency patterns accounts for the majority of issues. Correctly modeling 8 language features could prevent 49.0% of *Language Semantics Misunderstanding* bugs, corresponding to 8.0% of all bugs. Similarly, supporting 6 frameworks could prevent 52.6% of *Unsupported Frameworks and Libraries* bugs, corresponding to 11.2% of all bugs. This concentration suggests that these high-frequency patterns can provide useful guidance for triaging FP/FN issues and prioritizing debugging and improvement efforts.

4.3 Mishandling Intermediate Representations

Static code analyzers usually convert input programs into some intermediate representation (IR), *e.g.*, abstract syntax trees or bytecode, for analysis. However, mishandling IRs could lead to FNs or FPs. For example, PMD's Issue #3949 reports a FN of rule *FinalFieldCouldBeStatic*. The rule warns if a `final` field is assigned by a compile-time constant but not modified by `static`. This rule correctly warns `public final int BAR = 42`, but fails to warn `public final int BAR = (42)`. Because the IR structures of these two cases are different at the level of abstract syntax tree due to the parenthesis.

4.4 Mishandling Configurations

Static code analyzers typically provide configuration files that allow users to customize the rule set, specifying which rules should be enabled or disabled. Several bugs occurred because static analyzers failed to process these configuration files correctly. For instance, as shown in Figure 10, PMD allows users to configure the *UnusedPrivateField* rule [52] to ignore classes annotated with specific annotations (`lombok.Getter` and `lombok.Data` in this example) through an XML configuration file. The purpose of this configuration was to override the default `ignoredAnnotations` list with only the specified annotations. However, PMD incorrectly processed this configuration as an extension to the default list rather than an override. This configuration mishandling led to the false negative reported in PMD's Issue #2876.

4.5 Analysis Module Error or Limitation

4.5.1 Scope Analysis Error or Limitation. Java adopts static scoping rules to determine the visibility of symbols. However, analyzers often exhibit imprecise scope analysis due to insufficient modeling of Java's scoping mechanisms, manifesting in four aspects: *(1) scope boundary misidentification*, where analyzers fail to correctly model scope boundaries introduced by complex control flow such as nested classes and anonymous classes; *(2) visibility rule misapplication*, where package-private accessibility and inner/outer class access privileges are

Just Accepted

```
<rule ref="category/java/bestpractices.xml/UnusedPrivateField">
      <properties>
            <property name="ignoredAnnotations" value="lombok.Getter|lombok.Data" />
      </properties>
</rule>
```

Fig. 10. PMD#2876: Mishandling Configurations.

inadequately handled; *(3) namespace conflict misresolution*, where incorrect import precedence handling (*e.g.*, explicit vs. wildcard imports) causes confusion between local variable names and package prefixes; and *(4) same-name symbol disambiguation failure*, where analyzers lack robust mechanisms to distinguish identically named symbols across different scopes.

Figure 11 showcases a FP of PMD's rule *InvalidLogMessageFormat* [47] due to *same-name symbol disambiguation failure*. The rule checks whether the numbers of arguments and placeholders ("`{}`") in SLF4J or LOG4J2 loggers are matched. In this case, the variable `logMessage` in Scope 1 (lines 2 to 4) is incorrectly mapped to the variable `logMessage` in Scope 2 (lines 5 to 9). As a result, the rule incorrectly warns that the argument `param` at line 7 is not matched with the three placeholders in the `logMessage` (in Scope 1).

```
1   public void foo() {
2       if (true) {         // Scope 1
3           final String logMessage = "Message with three para: {}, {}, {}";
4       }
5       if (true) {         // Scope 2
6           final String logMessage = "Message with one parameter: {}";
7           final Object param = null; // FP: missing arguments, expected 3 arguments but have 1
8           logger.trace(logMessage, param);
9       }
10  }
```

Fig. 11. PMD#3284: Scope Analysis Limitation.

4.5.2 Type Resolution Error or Limitation. Type resolution is a crucial ability of analyzers to determine the types of symbols. However, analyzers often exhibit significant limitations due to insufficient modeling of Java's type system: *(1) incomplete compilation context*, where analyzers lack complete dependency information to resolve external library types; *(2) complex type feature mishandling*, where analyzers struggle with nested generics (*e.g.*, `Future<String>`), and parameterized return types, often falling back to `Object`; *(3) type disambiguation failure*, where analyzers fail to distinguish identically named user-defined and JDK classes (*e.g.*, custom `Error` vs. `java.lang.Error`) and confuse primitive types with reference types (*e.g.*, `int` vs. `Integer`); and *(4) inadequate inheritance resolution*, where analyzers fail to correctly track type information across inheritance hierarchies. These deficiencies collectively lead to type resolution failures,

Figure 12 showcases a FP of PMD's rule *CompareObjectsWithEquals* [44] due to *type disambiguation failure*. It confuses primitive types with reference types. This rule requires using "`equals()`" instead of "==" to compare object references. However, at line 2, `a1.length` and `a2.length` are the type of `int` and it is fine to use "==". PMD's incorrect type resolution leads to a FP. We observe that these four analyzers do not utilize Java compilers for precise type resolution, but implement their own type resolutions.

4.5.3 Dataflow Analysis Error or Limitation. Dataflow analysis tracks the definition, use, and propagation of variable values throughout a program. However, analyzers often exhibit significant limitations due to insufficient

Just Accepted

```
1  public byte[] foo( byte[] a1, byte[] a2){
2      if (a1.length != a2.length) {          // FP: compare with equals()
3          ...
4      }
5  }
```

Fig. 12. PMD#2976: Type Resolution Limitation.

modeling of program behaviors: *(1) insufficient control-flow sensitivity*, where analyzers fail to correctly model exception paths, conditional branches, and loop structures; *(2) limited inter-procedural tracking*, where analyzers cannot trace values across method boundaries, lambda captures, and resource ownership transfers; *(3) insufficient expression evaluation*, where analyzers only partially evaluate string concatenations and compile-time constants; *(4) inadequate object state tracking*, where analyzers fail to track state evolution and field assignments in static initializers; and *(5) dataflow logic errors*, where analyzers incorrectly model def-use relationships and confuse array element operations with array reference operations. These deficiencies collectively lead to false positives or negatives.

For example, Figure 13 showcases a FP of PMD's rule *DataflowAnomalyAnalysis* [45] due to *dataflow logic errors*. This rule warns about data flow anomalies, *e.g.*, redefinition of a recently defined variable without prior usage. In this example, the variable a is defined at line 2, used at line 3, and then redefined at line 3. No data flow anomalies happen. However, the dataflow analysis module erroneously processes the variable definition and use from left to right at line 3 (assuming a is defined and then used), and flags an issue.

```
1  public void Test( ){
2      int a = 0;
3      a = a + 3;    // FP: DD-Anomaly
4  }
```

Fig. 13. PMD#1749: Dataflow Analysis Error.

4.5.4 Symbolic Execution Error or Limitation. SonarQube uses symbolic execution to implement some rules. Some FNs and FPs are caused by the errors in the symbolic execution engine, manifesting in four aspects: *(1) path explosion*, where the number of possible execution paths grows exponentially with branches, forcing analyzers to prune paths and potentially miss critical defects; *(2) constraint solving limitations*, where analyzers struggle with non-linear constraints, string operations, array/collection constraints, and floating-point arithmetic, leading to imprecise reachability analysis; *(3) inadequate state abstraction*, where analyzers fail to precisely model complex object states, container sizes, and symbolic value propagation across method boundaries; and *(4) loop and concurrency handling difficulties*, where analyzers cannot determine loop termination conditions, resulting in incomplete analysis. These deficiencies collectively lead to false positives or false negatives in symbolic execution-based rules.

For example, Figure 14 showcases a FP of SonarQube's rule *S2589* [28] due to *inadequate state abstraction*. This rule warns against the gratuitous boolean expressions that always evaluate to "true" or "false". In this example, the symbolic execution engine misinterprets that the boxed type `Boolean` has only two kinds of values `Boolean.TRUE` and `Boolean.FALSE` but missed the other value `null`. When the engine meets the condition `if(b == Boolean.TRUE)` at line 3, it tries to reach the `else if` branch with the "only" other value `Boolean.FALSE`, and thus incorrectly assumes that the boolean expression at line 4 always evaluates to "true".

Just Accepted

```
1  public void foo(Boolean b) {
2      if ( b == Boolean.TRUE ) {}
3      else if ( b == Boolean.FALSE ) {}    // FP: expression always evaluates to "true"
4  }
```

Fig. 14. SonarQube#3619: Symbolic Execution Error.

Table 5. The percentage of FNs/FPs due to different root causes in each analyzer.

Static Analyzers	RD		SM					MIR	MC	AME			
	FRS	IRD	ULF	LSM	UFL	ASM	DSM			SAE	TRE	DAE	SEE
PMD	4.1	2.3	3.8	**19.0**	15.0	6.6	6.9	7.9	2.8	9.4	13.2	9.1	0.0
SpotBugs	1.0	3.1	14.3	**19.4**	**19.4**	10.2	10.2	3.1	1.0	3.1	6.1	9.2	0.0
SonarQube	3.4	2.7	3.7	14.6	**24.6**	10.8	11.2	2.3	1.1	3.4	10.1	9.3	2.8
ErrorProne	1.7	1.7	15.3	15.3	**23.7**	10.2	3.4	3.4	3.4	1.7	15.3	5.1	0.0
Total	3.3	2.5	5.1	16.4	21.2	9.4	9.4	4.1	1.75	5.2	11.0	9.1	1.6

Finding 3: *Analysis Module Error or Limitation* affects 337 of the 1257 issues (26.8%), representing the second most common root cause category. These limitations stem from analyzers' incomplete modeling of Java's complex language semantics, including scope analysis, type resolution, dataflow analysis, and symbolic execution.

Table 5 presents the distribution of root causes across different static analyzers. The results of individual analyzers align closely with the overall results. *Semantics Misunderstanding* (SM) is the dominant root cause for all analyzers. Within SM, *Unsupported Frameworks and Libraries* (UFL) shows the highest percentages in SonarQube (24.6%), ErrorProne (23.7%), and SpotBugs (19.4%). *Language Semantics Misunderstanding* (LSM) also contributes significantly in PMD (19.0%) and SpotBugs (19.4%).

Finding 4: Root cause distributions are consistent across all four analyzers, with *Semantics Misunderstanding* dominating in each analyzer (ranging from 41.5% to 53.1%). Despite different implementation architectures, all analyzers struggle with the same challenges in handling language features and third-party libraries. This consistency suggests that (1) testing strategies can be generalized across analyzers, where inputs triggering bugs in one analyzer are likely effective for testing others, and (2) bug fixes are transferable, where solutions addressing specific issues (*e.g.*, improving lambda handling) in one analyzer can guide fixes in others.

5 RQ2: Input Characteristics

This section investigates the input characteristics leading to FNs and FPs of the studied analyzers. We further examine the correlation between input characteristics and root causes. From the 1257 issues, we excluded 124 issues unrelated to input programs. In total, we identified 66 distinct input characteristics across 1,133 analyzed issues. Figure 15 shows the top 10 most common characteristics. The remainder are grouped as "Others".

Library-related characteristics dominate. Among the analyzed issues, *Annotations* and *Java External Libraries* emerge as the top two input characteristics, responsible for 191 issues (16.86%) and 110 issues (9.71%), respectively. This highlights the challenges analyzers face in correctly interpreting third-party frameworks and libraries, likely due to incomplete modeling of framework-specific behaviors and annotation semantics. *Java Standard Libraries* contribute 103 issues (9.09%), indicating that even well-documented standard APIs pose modeling difficulties.

Just Accepted

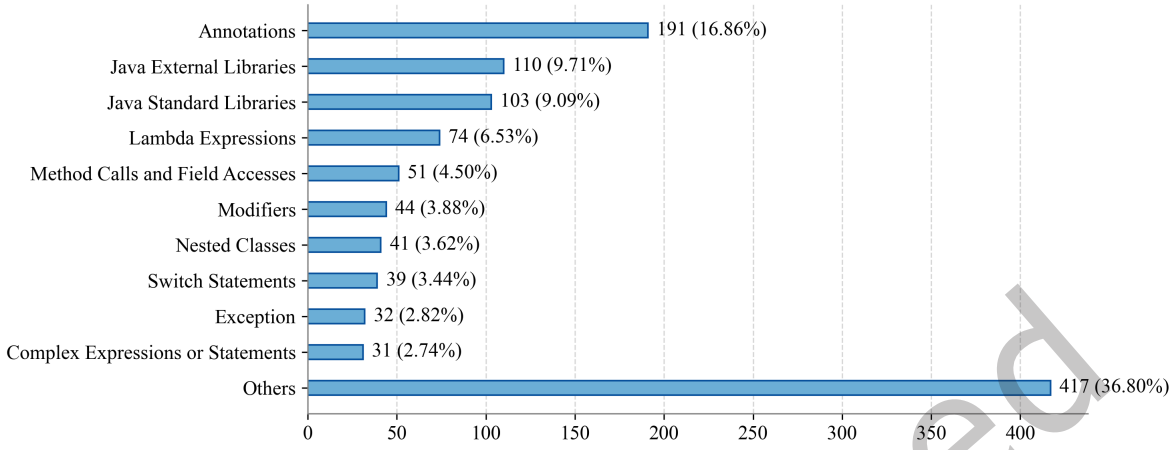


Fig. 15. Input characteristics leading to FNs/FPs.

Collectively, these three library-related characteristics account for 35.66% of all issues, revealing that library modeling remains a fundamental weakness in static analyzers.

Common Java language features are frequent bug triggers. Seven common Java language features collectively account for 27.53% of issues, making them the most impactful input characteristics after library-related ones, and highlighting their strong potential as targets for systematic testing. *Lambda Expressions* trigger 74 issues (6.53%), reflecting challenges with functional programming constructs, including closure semantics and type inference. *Method Calls and Field Accesses* (51 issues, 4.50%) expose difficulties in call graph construction and alias analysis. *Modifiers*, *Nested Classes*, *Switch Statements*, *Exception* and *Complex Expressions or Statements* collectively contribute 16.50% of issues. Importantly, beyond being bug-prone, these features are also straightforward to construct in test programs, as they require no third-party library dependencies or additional configuration. Static analyzer testing should therefore prioritize these features to maximize bug detection efficiency and ensure comprehensive quality assurance.

Others. The *Others* category (417 issues, 36.80%) encompasses less frequent issues such as inheritance (`extends`), enumerations (`enum`), and anonymous classes. Specific input characteristics for each issue can be found in our open-source artifacts. While individually less common, collectively they account for over one-third of all issues. This diversity indicates that ensuring the quality of analyzers is highly challenging, as analyzers must handle not only common patterns but also a wide range of corner cases across diverse Java features.

Finding 5: Library-related characteristics dominate analyzer bugs (35.66%), followed by common Java language features (27.53%). Seven Java language features frequently trigger bugs and are straightforward to construct without requiring third-party library dependencies, making them ideal targets for systematic testing to efficiently expose FNs and FPs in static analyzers.

To investigate the correlation between root causes and input characteristics, we analyzed the 716 cases associated with the top 10 input characteristics. Figure 16 presents a heat map where the X-axis represents 13 root cause subcategories (Table 2) and the Y-axis shows the top 10 input characteristics (Figure 15). Overall, the heat map shows a dense distribution with 68.5% of cells (89 out of 130) containing non-zero values, This indicates that a root cause can typically be triggered by different input characteristics, while an input characteristic can trigger different bugs with multiple root causes. Not all associations exhibit strong correlations.

Just Accepted

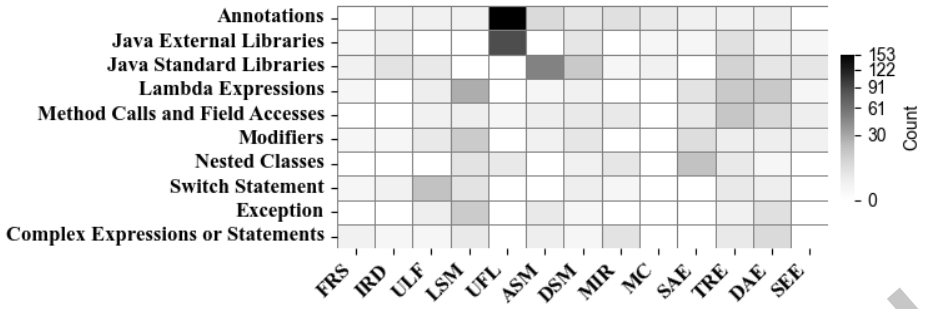


Fig. 16. Correlation between top-10 input characteristics and root causes.

Two root causes exhibit strong correlations on specific input characteristics. *Unsupported Frameworks and Libraries* (UFL) concentrates heavily on two input features: *Annotations* (153 cases, 57.5%) and *Java External Libraries* (89 cases, 33.5%), achieving 91.0% combined concentration. *API Semantics Misunderstanding* (ASM) shows similar result, with 42.4% of cases (50 out of 118) triggered by Java Standard Libraries. This high concentration suggests well-defined triggering conditions, making these root causes amenable to targeted mitigation strategies. In contrast, some root causes spread across many input characteristics, making them harder to address systematically. For example, *Scope Analysis Error or Limitation* (SAE) and *Unsupported Language New Features* (ULF) span 6-8 different input characteristics. However, despite this dispersion, a small number of characteristics still dominate. The top 3 features cover 50.8% of SAE cases (corresponding to 2.6% of all issues): Nested Classes (29.2%), Modifiers (12.3%), and Lambda Expressions (9.2%). Similarly, the top 3 features account for 42.2% of ULF cases (corresponding to 2.2% of all issues): Switch Statement (29.7%), Java Standard Libraries (6.2%), and Modifiers (6.2%). This dispersion demonstrates that rule-based static analyzers must handle diverse code characteristics, highlighting the fundamental challenge of ensuring analyzer quality. However, prioritizing support for the most common input characteristics can yield substantial quality improvements, enabling targeted progress even when comprehensive coverage of all characteristics remains challenging.

Finding 6: Concentrated root causes (UFL, ASM) with 42.4-91.0% of cases tied to 1-2 input characteristics enable targeted fixes, while dispersed root causes (*e.g.*, ULF and SAE) spreading across 6-8 characteristics with 42.2-50.8% top 3 concentration require systematic improvements. These findings provide useful guidance for triaging FP/FN issues and prioritizing further investigation and improvement efforts.

6 RQ3: Proof-of-Concept Demonstration of Our Findings

This section demonstrates how our findings can help identify issues in analyzers. Specifically, we automatically generate analytically equivalent program variants based on the most common input characteristics that trigger bugs to detect FNs and FPs (Section 6.1). We manually review the implementations and issue patches of historically error-prone rules, then construct targeted program variants to reveal weaknesses in static analysis modules (Section 6.2).

6.1 Finding the Analyzers' FNs/FPs by Automatically Generating Analytically Equivalent Programs

This section shows a proof-of-concept testing strategy to help find FNs or FPs of static code analyzers. The main idea is to automatically generate analytically equivalent input programs from existing ones to stress test the analyzers.

Just Accepted

Table 6. Analytically Equivalent Input Program Mutation Operators

Mutation Operators	Original Program/Mutated Program Examples

(a) Wrapping a method with a nested class.	`class foo{`
	`- method();`
	`+ class NestedClass{ method();}`
	`}`

(b) Converting an anonymous class to a lambda expression.	`- Runnable r = new Runnable(){`
	`- @Override`
	`- public void run() {...}`
	`- };`
	`+ Runnable r = () -> {...};`

(c) Renaming symbol names to create same identifiers.	`- int a;`
	`+ int b;`
	`public void b(){...}`

(d) Replacing with analytically equivalent expressions.	`- j++;`
	`+ ++j;`
	`---`
	`- if(false){...}`
	`+ if(false || false){...}`
	`---`
	`class foo{`
	`int a;`
	`void foo(){`
	`- bar(a);`
	`+ bar(this.a);`
	`}}`

Just Accepted

Analytically Equivalent Input Program Mutation. One insight is that we can perform *analytically equivalent input program mutations* to help find FNs or FPs. Specifically, given an original input program of an analyzer, we can generate some analytically equivalent program variants based on some program mutation operators. Here, analytically equivalent program variants mean that these variants are expected to have the same analysis results with the original program when checked by the analyzer [110], but also to have differing forms. If the analysis results (*e.g.*, the reported warnings) differ between the original program and its variants, some FNs or FPs are likely found.

Based on Figure 15 and Finding 5 of RQ2, we identify input characteristics that are more likely to trigger FNs/FPs in static analyzers to guide our mutation operator design. We implement a total of 6 mutation operators across 4 categories, targeting four frequent bug-triggering characteristics: (a) Nested Classes, (b) Lambda Expressions, (c) Same Identifier[6], and (d) Method Call and Field Access. Table 6 presents the corresponding mutation operators with a representative example for each category. Note that category (d) includes three analytically equivalent transformations, as illustrated in the table. The details are as follows.

(a) *Wrapping a method with a nested class*. Given a Java class, this mutation operator identifies all the methods in this class. For each (*static* or *non-static*) method, the operator creates a nested class to wrap this method and properly adjusts the original class fields or methods accesses to be compliant with the code changes.
(b) *Converting an anonymous class to a lambda expression*. In Java, an anonymous class can be equivalently represented by a lambda expression. Thus, given a Java class, this mutation operator identifies all the anonymous classes. For each anonymous class, it converts the class to a lambda expression.
(c) *Creating same identifiers*. Given a Java class, this mutation operator identifies all the methods and the fields in this class. For each field, it renames its symbol name to the name of one existing method's name.

[6] *Same Identifier* is the 12th most frequent input characteristic identified in our study.

(d) *Replacing with analytically equivalent statements or expressions*. We support three forms of analytically equivalent mutations: replacing (1) the statement `j++;` with `++j;` or vice versa, (2) `false` with `false || false`; and (3) the access of a non-static class field `a` with `this.a`.

Furthermore, Figure 16 and Finding 6 show that a single input characteristic can be associated with multiple root causes. Among these associations, some correspond to less frequent cases, while others represent prevalent cases. For example, *Nested Classes* are associated with seven distinct root causes. However, only *Scope Analysis Error or Limitation (SAE)* is prevalent, accounting for 19 issues in our dataset. Therefore, we focus on prevalent root causes (*i.e.*, those with more than 10 issues) when evaluating the coverage of our implemented input characteristics. Collectively, the four implemented input characteristics cover 30.8% of prevalent root-cause types (4/13). Beyond these four characteristics, the approach can be extended to other input characteristics. For instance, prior work [110] designs three metamorphic relations based on analysis equivalence to generate program variants annotated with specific *Annotation* for finding annotation-induced faults in static analyzers.

Implementation. We use JavaParser [15] to parse the input programs into abstract syntax trees (ASTs), manipulate the tree nodes according to the mutation operators and generate analytically equivalent program variants. The implementation consists of around 1000 lines of Java code.

Experimental Setup. We applied our testing strategy to test the latest versions of the four studied analyzers at the time of our study (PMD v7.0.0-rc3, Spotbugs v4.7.4, SonarQube v10.0.0.68432 and ErrorProne v2.41.0). We collected the regression test suites from these tools for the original input programs: 3,960 programs from PMD, 700 programs from SonarQube, 702 programs from Spotbugs, and 4,096 programs from ErrorProne. These test cases vary in size from 53 to 334 tokens, with an average of 85 tokens. These programs are directly derived from real-world bug reports and carefully constructed by analyzer developers, making them representative of practical usage. They have appropriate size for testing analyzers while covering complex analysis scenario such as data-flow analysis and call-graph construction. Moreover, as official regression test suites, they maintain broad coverage across supported rules.

We used JavaParser to parse, transform and create new test programs in Java 11. We used Python scripts to run the tests against the analyzers and analyze the outputs. We compared the analysis outputs by checking whether the numbers or the types of reported warnings are identical. If not, we likely find some FNs/FPs, and manually inspect each of them for confirmation. The testing process was conducted on a 64-bit Ubuntu 20.04 LTS machine with 16GB RAM. It took about 2 hours to run all the newly generated test programs for PMD, 1.5 hours for SonarQube, 1 hour for Spotbugs and 1.5 hours for ErrorProne.

Results and Analysis. Table 7 gives the issues of FNs and FPs (with Issue IDs from 1∼15) found by our testing strategy. We found 13 FNs and 2 FPs from the four static code analyzers. We reported all these issues to the developers. Up to now, 12 issues have been confirmed, 9 of which have been already fixed; and 3 issues are still waiting for feedback from the developers. These issues affected 13 different rules from PMD, SpotBugs and ErrorProne. We note that most issues were found by the mutation operator (b), which found 6 issues, and (d) (the three forms of analytically equivalent expression) found 1, 1 and 5 issues, respectively. The mutation (c) did not find new issues in the analyzers. We attribute this to the continuous improvement of these analyzers. Specifically, same identifier constructs are primarily associated with *Analysis Module Errors or Limitations*, particularly in type resolution and scope analysis. These issues appear to have been addressed in recent releases of the analyzers.

To our knowledge, Wang *et al.* [103] conducted the first work to find FNs or FPs of static code analyzers. They used a differential testing strategy, *i.e.*, comparing the outputs of similar rules between three different analyzers to find issues. Different from their work, our testing strategy is one form of metamorphic testing [61] which does not require a reference analyzer. Indeed, there were 11 issues found by us which cannot be found by the approach proposed by Wang *et al* [103]. Because the related 9 buggy rules in our experiment do not have similar rules in other analyzers. For example, for the rule *"UseIOStreamsWithApacheCommonsFileItem"* in PMD, no similar

Just Accepted

Table 7. Statistics of the 19 issues found by our study

ID	Tools	Mutation Operators	Issue Type	Affected Rules	Status
1	PMD	(b)	FP	AvoidAccessibilityAlteration	Fixed
2	PMD	(b)	FN	LawOfDemeter	Fixed
3	PMD	(d)-(2)	FN	WhileLoopWithLiteralBoolean	Fixed
4	PMD	(d)-(3)	FN	LawOfDemeter	Fixed
5	PMD	(d)-(3)	FN	ConsecutiveLiteralAppends	Fixed
6	PMD	(d)-(3)	FN	UseIOStreamsWithApacheCommonsFileItem	Fixed
7	PMD	(d)-(3)	FN	UnsynchronizedStaticFormatter	Fixed
8	PMD	(d)-(3)	FN	GuardLogStatement	Fixed
9	PMD	(d)-(1)	FN	UnusedPrivateField	Fixed
10	PMD	(b)	FP	DoNotTerminateVM	Confirmed
11	PMD	(b)	FN	LawOfDemeter	Confirmed
12	PMD	(b)	FN	AccessorClassGeneration	Pending
13	PMD	(a)	FN	UnusedAssignment	Pending
14	Spotbugs	(b)	FN	NP_ALWAYS_NULL	Pending
15	ErrorProne	(a)	FN	EqualsWrongThing	Confirmed
16	Sonarqube	—	Spec.	Java:S2384	Fixed
17	PMD	—	FP	CloseResource	Confirmed
18	Sonarqube	—	FP	Java:S2384	Confirmed
19	Sonarqube	—	FN	Java:S2259	Confirmed

rule in SonarQube exists. Thus, the issue (with ID: 6 in Table 7) could be missed by differential testing and our proof-of-concept testing strategy could complement the prior work. We believe that more mutation operators could be designed to help find more FNs or FPs of static code analyzers. We leave it as an interesting future work.

6.2 Investigating the Weaknesses of Static Analysis Modules

The static code analyzers use some forms of static analysis. Thus, weaknesses of the static analysis modules in the analyzers may lead to FNs/FPs (see **RQ1**'s Finding 3). To this end, we aim to manually investigate some typical rules of the studied analyzers to investigate their potential weaknesses based on our insights from **RQ1** and **RQ2**.

Investigation Method. To select typical rules for inspection, we ranked all the rules of each studied analyzer in terms of the number of historical fixed issues from the most to the least. To constraint our manual cost, we chose the top five percent of buggy rules from PMD, SpotBugs, SonarQube and ErrorProne for careful inspection. We suggest that investigating such "buggy" rules are more likely to reveal more systematic weaknesses of static analysis modules. Table 8 lists these selected rules from the four analyzers. The column "#Buggy Rules" denotes the total number of buggy rules identified in each analyzer, while "Top 5% Buggy Rules" lists the rule names. The column "#Fixed Issues" indicates the number of historical issues fixed for each rule.

During our investigation, for each rule in Table 8, we (1) reviewed its specification and implementation (understanding which kind of program flaws the rule checks and which form of static analysis the rule uses), (2) examined all its historical issues and the corresponding fixing patches (understanding which root causes and/or input characteristics trigger these issues, and whether the corresponding fixing patches are workarounds), and (3) applied some code mutations on the rule's test programs to manifest FNs/FPs. Here, the code mutations are applied based on the findings of **RQ1** and **RQ2**: the common input characteristics leading to FNs/FPs. From the top frequent input characteristics triggering FPs/FNs identified in Section 5 (also see Fig. 15), we selected these three input characteristics, *i.e.*, *method calls and field accesses*, *variable initialization and assignment* and *complex expressions and statements*, as the mutation strategies (operators) to manually investigate the weaknesses of the

Just Accepted

Table 8. Top 5% buggy rules of the studied analyzers in terms of the number of historical fixed issues.

Analyzer	#Buggy Rules	Top 5% Buggy Rules	#Fixed Issues
PMD	133	UnusedPrivateMethod [53]	25
		UnnecessaryImport [51]	19
		CloseResource [43]	13
		ImmutableField [46]	13
		UnnecessaryFullyQualifiedName [50]	13
		InvalidLogMessageFormat [47]	10
SpotBugs	69	NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE [7]	7
		NP_NONNULL_PARAM_VIOLATION [6]	4
		UPM_UNCALLED_PRIVATE_METHOD [8]	4
SonarQube	275	S1948: Fields in a "Serializable" class should either be transient or serializable [21]	22
		S2259: Null pointers should not be dereferenced [25]	16
		S2699: Tests should include assertions [30]	16
		S2187: TestCases should contain tests [23]	15
		S2095: Resources should be closed [22]	14
		S2637: "@NonNull" values should not be set to null [29]	14
		S1144: Unused "private" methods should be removed [18]	13
		S2755: XML parsers should not be vulnerable to XXE attacks [31]	12
		S2384: Mutable members should not be stored or returned directly [26]	11
		S1905: Redundant casts should not be used [20]	10
		S1612: Lambdas should be replaced with method references [19]	9
		S2386: Mutable fields should not be "public static" [27]	9
		S3457: Printf-style format strings should be used correctly [32]	9
ErrorProne	43	PreferJavaTimeOverload [39]	6
		UnusedVariable [40]	6

analyzers. We did not select the other input characteristics from the top ten because some input characteristics (*e.g.*, *nested classes*) are suitable for automatically generating analysis-equivalent programs (in Section 6.1) while some input characteristics (like *annotations*, *Java standard libraries* and *Java external libraries*) are difficult to apply for manual code mutations. In practice, we applied the input characteristic of *method calls and field accesses* to test those rules which may require dataflow information (*e.g.*, PMD's rule *CloseResource*). For example, we can encapsulate the sink (*e.g.*, closing some resource) into a new method, and call that new method at the original place. This mutation strategy can stress test the inter-procedure analysis ability of a rule. For another example, if the original test program involves some variable assignments, we can transform the assignments into different forms (*e.g.*, transforming the assignment of a local variable into that of a global variable or a class variable) by applying the input characteristic of *variable initialization and assignment* to stress test the rules. We can also apply the characteristic of *complex expressions and statements* to purposely transform the original expressions or statements into the complicated ones to stress test the rules. Note that these code mutations do not guarantee the mutated program is analysis-equivalent to the rule's original test program (like what we did in Section 6.1).

Table 7 shows the four issues (with Issue IDs from 16~19) found by us in these analyzers' latest versions, indicating the weaknesses of their static analysis modules (leading to FNs or FPs). All these issues were confirmed by the developers. Figure 17 shows the input programs that trigger the issues. Note that Figure 17b triggers two issues. We illustrate these issues below.

Just Accepted

```
1   import java.sql.Connection;
2   import net.sourceforge.pmd.....closeresource.Pool;
3
4   public class Foo {
5+      void closeConnection(Connection c) {
6+          c.close();
7+      }
8
9       void bar(Pool pool) {
10          Connection c = pool.getConnection();
11          try {
12              } catch (Exception e) {
13              } finally {
14+                 closeConnection(c);  // FP: c is not closed
15-                 c.close();
16              }
17      }
18  }
```

(a) PMD : FP of *CloseResource*

```
1   public class AISD {
2       private byte[] buf;
3       private AISD( final byte[] buf) {
4           this.buf = buf;    // FP
5       }
6       private void set(final byte[] buf) {
7           this.buf = buf;    // true positive
8       }
9       public AISD of( final byte[] buf) {
10          return new AISD(buf.clone( ));
11  }}
```

(b) SonarQube : FP of *S2384* (Mutable members stored directly in private constructors)

```
1   public class Foo {
2
3+      public void hasArguments(String name){
4-      public static void hasArguments(String name){
5           int length = name.length();
6           System.out.println("Name length: " + length);
7       }
8
9       public static void main(String[] args) {
10          String name = null;
11-         hasArguments(name);
12
13+         Foo foo = new Foo();
14+         foo.hasArguments(name);
15      }
16  }
```

(c) SonarQube : FN of *S2259* (Null pointers should not be dereferenced)

Fig. 17. Simplified code examples illustrating the weaknesses of static analysis modules in the studied analyzers.

Failure to handle basic method call flows (Issue 17 in Table 7). Figure 17a shows a FP of PMD's rule *CloseResource*. In this case, we hoisted the original statement of closing the connection `c` (line 15) into a method call `closeConnection` (line 14), and closed the connection `c` at line 6. However, PMD reports a FP at line 14, although `c` is correctly closed at line 6. The PMD developer confirmed that this issue is a valid FP and commented that "*since this is all within one class, it would be nice if PMD could detect this on its own (through some basic call flow)*". This case shows the weakness of data-flow analysis in PMD. In fact, we find that PMD's data-flow analysis is limited (i.e., it only supports reaching definition analysis procedure [10]).

Failure to choose the appropriate form of static analysis (Issue 16 and 18 in Table 7). Figure 17b shows a FP of SonarQube's rule *S2384*. This rule enforces that private mutable members in a class should not be directly stored or returned. In this example, this rule correctly warns line 7 because the private mutable member `buf` is directly stored. However, the rule reports a FP at line 4 which should not be reported because the private constructor `AISD` is only called by the public method `of` which safely clones the parameter `buf`. The SonarQube developer confirmed that this issue is a valid FP and commented that "*Unfortunately, this rule is (implemented as) AST-based, and it brings some limitations. So to eliminate these false positives the rule should rely on the data flow*". This case shows the weakness of SonarQube in choosing the inappropriate form of static analysis for some rules. We also found a flawed specification issue in this rule (Issue 16 in Table 7) which has been fixed [3].

Failure to track runtime types in symbolic execution (Issue 19 in Table 7). Figure 17c shows a FN of SonarQube's rule *S2259*. This rule warns against null pointer dereference. For the original input program, this rule can correctly warn against the dereferenced null pointer `name` (at line 5) because the string variable `name` is assigned as `null` (line 10). However, when we changed the static method `hasArguments` to an instance method, and changed the original static method call on `hasArguments` to the method call by the class instance `foo`, the rule has a FN at line 5. The SonarQube developer confirmed that this issue is a valid FN and commented that "*the FN is caused by a limitation of the symbolic execution engine*". The developer explained that, in the new code, the method `hasArguments` is an instance method and therefore requires a class instance to be called. The engine that runs this rule fails to track runtime types in the execution paths that it follows because it currently only supports tracking of final methods (*e.g.*, static methods). The developer said they have already started to enhance the engine. This case shows the weakness in resolving runtime types, affecting the precision of symbolic execution in SonarQube.

Just Accepted

7 Implications and Discussions

This section distills actionable implications and challenges from our findings in **RQ1-RQ3**. We identify four practical strategies that developers and researchers can adopt to reduce FNs and FPs, and discuss one fundamental challenge that requires long-term efforts.

Incrementally improving through high-frequency patterns. While ensuring the overall quality of static analyzers remains challenging, our findings suggest that incremental improvements targeting high-frequency patterns can yield practical benefits. Finding 2 demonstrates that correctly modeling 8 language features could prevent 49.0% of *Language Semantics Misunderstanding* bugs, corresponding to approximately 8.0% of all bugs, while supporting 6 frameworks could prevent 52.6% of *Unsupported Frameworks and Libraries* bugs, corresponding to approximately 11.2% of all bugs. This concentration suggests that analyzer developers can leverage the prevalence of different issue types to prioritize improvement efforts. While pursuing comprehensive coverage remains important, focusing on frequently occurring language features and popular frameworks can provide more efficient targets for debugging and quality improvement, especially when development resources are limited.

Mining analytically equivalent inputs from bug reports. User-submitted bug reports typically include both positive and negative examples, providing a valuable resource for mining analytically equivalent inputs. For instance, in PMD Issue #3148[7], the user reported that `Objects.nonNull(stmt)` triggers the CloseResource rule while `stmt != null` does not, despite their semantic equivalence. Based on our observations, these equivalent patterns include equivalent annotations (`@Nullable` vs `@CheckForNull`), equivalent APIs (`List.of()` vs `Arrays.asList()`), and equivalent expressions (lambda vs anonymous class), among others. Future work can extend our tool in Section 6.1 by automatically mining more equivalence relations from reference cases in bug reports and generating test cases, for example through LLMs, thereby achieving more comprehensive testing.

Cross-analyzer testing strategy through shared weaknesses. Different static analyzers exhibit highly consistent root cause distributions, indicating that these are common challenges in rule-based static analyzers rather than analyzer-specific issues. This is because such analyzers must continuously model evolving language features and library semantics while maintaining complex analysis logic, which makes accurate and complete rule implementation inherently challenging. Finding 4 reveals consistency in root cause distributions across four analyzers, with *Semantic Misunderstanding* dominating in all analyzers (41.5%-53.1%). This consistency enables practical cross-analyzer testing strategies. When a bug is discovered in one analyzer, the test case that triggers this bug can be applied to other analyzers. If multiple analyzers implement similar rules, a bug in one analyzer likely exists in others. This shared weakness also guides testing priorities by highlighting which root cause categories need more attention. This proposed cross-analyzer approach improves testing efficiency through one-time discovery and multi-analyzer validation, enables analyzers to learn from each other's fixes, and promotes community collaboration through shared issue pattern repositories.

Paying attention to historically error-prone rules. Rules that have historically been prone to errors continue to exhibit problems, especially when static analyzers are updated. In Section 6.2, we identify the top 5% buggy rules. Although these rules have been fixed multiple times, we still identified four bugs in them through simple manual investigation, suggesting that they require more attention. Analyzer maintainers should maintain a watchlist of these error-prone rules and proactively test them. These rules need more rigorous code review, enhanced test coverage, and stricter quality gates. This proactive strategy enables preventive maintenance and optimizes testing resources by focusing on high-risk areas.

Static analysis module improvements are critical but challenging. Improving underlying static analysis modules is crucial for mitigating FNs/FPs, as Finding 3 reveals that *Analysis Module Error or Limitation* affects 337 of the 1257 issues (26.8%). However, in practice, developers often do not immediately address these modules even

[7]https://github.com/pmd/pmd/issues/3148

Just Accepted

after confirming issues. This is because they need to trade off many factors like investment costs, development priorities, tool performance, and the balance between precision and soundness. For example, SonarQube Issue #90849 [2] is an FN caused by symbolic execution engine limitations. Developers confirmed the issue but chose not to fix it immediately due to the substantial investment required. They explained that the current engine is purposely designed to avoid FPs at the cost of inducing FNs, but noted that "*we are working on a new bug-detection engine, that should be, at some point, able to handle such cases and would allow us to replace this rule with a more performant version of it*". Similarly, PMD developers confirmed Issue #4127 [1], an FP caused by type resolution limitations, but it took over one year to improve the type resolution module and resolve these issues. These cases reveal that module-level improvements demand significant engineering effort and careful system-wide consideration, often requiring architectural evolution rather than quick patches.

Conceptual overlap between root causes and input characteristics. Although root causes and input characteristics capture different dimensions of static analyzer issues, some categories exhibit conceptual overlap between the two dimensions, as shown in Figure 16. For example, issues triggered by *Java External Libraries* are often attributed to the root cause of *Unsupported Frameworks and Libraries* (UFL). Similarly, issues involving *Java Standard Libraries* are frequently related to *API Semantics Misunderstanding* (ASM). Such correlations provide practical guidance for static analyzer developers. When an FP/FN issue is identified for a specific input characteristic, developers can leverage these associations to prioritize potential root causes, facilitating more efficient issue diagnosis.

8 THREATS TO VALIDITY

Internal Validity. We studied 1257 issues which include 1070 FPs and 187 FNs (the FPs are more than the FNs). This disparity between FNs and FPs may bring some threats in the computed percentages of the categories of root causes and input characteristics. But the readers should know that this arose naturally from the in-vivo datasets employed and so might be difficult to correct and it does not mean that FPs are more important than FNs. In practice, FNs can be more critical from practitioners' perspectives because missed defects may remain undetected and affect software reliability. The larger number of FPs in our dataset is mainly because (1) the users are more likely to report FPs than FNs, and (2) many FNs are not reported because no checking rules exist [68, 75, 78, 99] (such FNs may not be considered valid issues of the analyzers). Therefore, our taxonomy characterizes the distribution of known analyzer issues rather than the relative importance of FPs and FNs.

Our dataset is collected from historically reported and fixed issues, which may introduce selection bias. Specifically, issues that are easier to detect and reproduce are more likely to be reported, while subtle false negatives may remain underrepresented. Therefore, our taxonomy may not cover all possible failure modes of static analyzers. To mitigate this limitation, we collect issues from four widely used static analyzers to capture diverse failure modes, as an issue missed by one analyzer may be detected by another. We further design a proof-of-concept demonstration to uncover previously unknown issues, and will continue expanding our dataset in future work.

External Validity. We collected and investigated 1257 issues of FNs and FPs. The number of these issues might be not large enough and may bring some threats to the generability of our study's findings. But we believe that our study takes great effort to achieve ecological validity in the context of these four analysers: We have collected all the historical, fixed issues of FNs and FPs from the four representative PMD, SpotBugs, SonarQube and ErrorProne prior to the time of our study. These issues are diverse and were reported by the analyzers' own developers (who found FNs/FPs during development), real users (who found FNs/FPs by scanning real-world projects), and researchers (who found FNs/FPs by developing automated testing techniques). In addition, compared to the prior relevant work [68, 99, 103, 109], our study investigates the largest number of issues of FNs and FPs and their patches—more than ten times the size of the largest previous study. We have looked into the

Just Accepted

FNs/FPs which were analyzed by prior work. We find our study's findings have included all their analysis results (*i.e.*, root causes, input characteristics).

One possible method of further generalizing some of our findings might be analyzing open-source Java projects by using these analyzers, and determine to what extent the coding practices in these projects are likely to "trigger" any of the issues we found. However, this method may face some challenges in the further generalizability of our findings. First, it may be limited to investigating the issues of FPs if we do not have the ground-truth of program flaws in these projects. Second, if we do have the ground-truth, the missed flaws may not indicate the issues of FNs because the checking rules of these analyzers cannot cover all possible program flaws. But we can access the generalizability of our findings from the perspective of Wang *et al.*'s work [103]. They analyze 2,728 open-source Java projects by using these analyzers and find 46 issues of FNs/FPs based on differential testing. The root causes and input characteristics of these 46 issues are all included in our findings (which we will discuss in detail in Section 9). This suggests the generality of our findings.

Our study only considers four static code analyzers. So our conclusions may not generalize beyond these studied analyzers. However, these four analyzers are representative and widely used in practice and implement different forms of static analysis. In the future, to further mitigate the threats, we would expand our analysis to more static code analyzers. We focus on the historical issues triggered by Java programs (Java is one of the most popular languages supported by existing static code analyzers [88]). As a result, some of our findings may be specific to Java and may not be generalized for other programming languages. Therefore, we plan to study the historical issues of FNs/FPs from other programming languages in the future to mitigate this potential threat.

In addition, SonarQube and PMD have more historical issues than the other two analyzers. It may affect the generalizability of our findings. But we find the root causes of SonarQube's and PMD's issues are similar to those of Spotbugs's and ErrorProne's issues. Thus, we believe the distilled categories of root causes should be general.

Our first proof-of-concept demonstration, automatically generating analytically equivalent programs, performs mutation on regression test suites of static analyzers. However, regression test suites are inherently limited by developers' existing knowledge of potential issues, as they mainly encode previously identified bugs and may miss unknown behaviors and corner cases. To mitigate this limitation, we introduce a second proof-of-concept demonstration, where we manually review the implementations and issue patches of historically error-prone rules and construct targeted program variants to uncover more diverse analyzer issues. Nevertheless, we acknowledge that applying mutation to real-world programs could potentially uncover more diverse issues and provide greater practical value, which we leave as an important direction for future work.

Construct Validity. Our study requires manual analysis and the human expertise of static code analyzers to answer RQ1 and RQ2. To ensure construct validity, we took measures to enhance the reliability of our labeling process. In particular, we relied on Cohen's Kappa coefficient to evaluate the inter-rater agreement when labeling FN/FP issues, thereby reducing subjectivity and measurement bias. The labeling was guided by detailed guidelines and independently performed by multiple raters, followed by a reconciliation process to resolve disagreements. Although some degree of subjectivity is inevitable in manual analysis, the use of inter-rater agreement metrics provides evidence that our labeling is reasonably consistent and reliable.

9 RELATED WORK

This section discusses two strands of related work on studying static code analyzers: (1) evaluating the effectiveness and usability, and (2) finding and studying the FNs and FPs.

Evaluating Static Code Analyzers. In the literature, many studies exist in evaluating the effectiveness (*i.e.*, the fault detection abilities) of the static code analyzers [62, 66, 68, 74, 75, 77, 78, 84, 99–101, 105]. All these studies reveal that static code analyzers suffer from FNs. To analyze the reasons of FNs, for example, Thung *et al.* [99, 100]

Just Accepted

and Habib *et al.* [68] respectively manually examine 19 and 20 missed field defects and their corresponding program. They find that *almost* all these defects were missed because they are not targeted by existing rules in the analyzers or they are domain-specific errors. This insight is also shared by more recent and comprehensive studies [75, 78] — the insufficient rules *w.r.t.* field defects and the inability to handle logical (domain-specific) errors are the main reasons of FNs. On the other hand, several studies reveal that static code analyzers are also affected by FPs [71, 74, 92, 105], thus undermining the usability [81, 88, 102]. Different from these prior studies, our work studies the analyzers from a new perspective, *i.e.*, examining the historical (fixed) issues to understand FNs and FPs. Moreover, our study inspects the implementations of static code analyzers, and the fixing patches to analyze FNs/FPs. Thus, many of our findings are fine-grained and have not been identified by these prior studies. A recent comprehensive survey [67] shows that, to mitigate FPs, most work develops post-processing techniques (*e.g.*, statistical analysis, machine learning) to classify or rank the static analysis warnings. These work usually does not care about the implementation of static code analyzers. In contrast, our work inspects the implementations to find insights which could mitigate the FPs at the root.

Finding and Studying FNs/FPs. To our knowledge, the studies of Wang *et al.* [103] and Zhang *et al.* [109] are most relevant to ours. However, our study have several major differences with these two prior studies. *First*, the research goals of our study and these two studies are different. Our work aims to conduct a systematic study on understanding the historical issues of FNs/FPs. Thus, we investigate a broad range of 1257 developer-confirmed *and* -fixed FNs/FPs. Wang *et al.* [103] and Zhang *et al.* [109] mainly focus on designing some testing techniques to find FNs/FPs of the analyzers. Although Wang *et al.* and Zhang *et al.* inspect the FNs/FPs *found by their techniques* — Wang *et al.* inspect 46 found FNs/FPs (only 19 were confirmed *and* fixed) and Zhang *et al.* inspect 79 found FNs/FPs (only 26 were confirmed *and* fixed) — their conclusions could be biased by the limited diversity of their found issues. Moreover, they do not examine the fixing patches and the implementations of the analyzers when inspecting FNs/FPs.

Second, due to the differences between research goals and the datasets, our findings are more systematic and in-depth. For example, the 13 "bug patterns" and 3 "typical faults" (see Section 3.2 and 3.3 in [103]) summarized by Wang *et al.* and 5 "root causes" (see Section 5.2 in [109]) summarized by Zhang *et al.* are all included in our identified root causes and input characteristics. Specifically, in Table 2, the root causes annotated by "◐" and "◐" are only partially identified by Wang *et al.* and Zhang *et al.* respectively, while the root causes annotated by "○" and "○" are missed by Wang *et al.* and Zhang *et al.* respectively.

Third, Wang *et al.* use a differential testing strategy to find FNs/FPs, while we use a metamorphic testing strategy to find FNs/FPs (in Section 6.1). These two testing strategies have their own strengths and weaknesses and can complement each other in finding FNs/FPs. The differential testing strategy might be limited by the number of paired rules with similar functionality between two different analyzers. Take PMD and SonarQube as an example, according to the statistics reported by Wang *et al.* (Section 3.1 in [103]), PMD and SonarQube respectively have 304 and 545 rules in total, but they only have 74 paired rules. Therefore, the differential testing strategy can only test 24.3%(≈74/304) and 13.6%(≈74/545) of all the rules of PMD and SonarQube, respectively, while the metamorphic testing strategy do not require the paired rules. On the other hand, the metamorphic testing strategy is limited to the number and strength of the identified metamorphic relations (*i.e.*, the program mutation operators for generating analytically equivalent program variants). In our case study, the issue finding results (discussed in Section 6.1) indeed show that our metamorphic testing strategy can complement the differential testing strategy because 11 out of the 15 FNs/FPs found by our metamorphic testing strategy cannot be found by the differential testing strategy. The reason is that the rules affected by these 11 FNs/FPs do not have the paired rules, and thus cannot be tested by differential testing. But our metamorphic testing strategy does not have such limitations. On the other hand, the rules affected by the remaining 4 FNs/FPs could be found by the differential testing strategy because these rules have their paired rules. In general, different factors such as the diversity of

Just Accepted

input programs and the number and strength of identified metamorphic relations may influence the effectiveness of differential and metamorphic testing in finding FNs/FPs. A promising direction for future work is to conduct a more systematic comparison between the two techniques. Zhang *et al.* use metamorphic testing like us, and propose 13 mutation operators (3 operators are inspired from historical issues). However, only one mutation operator (see Table 6, (d)-(2)) used by our testing strategy is overlapped with those of Zhang *et al.* (see Table 2 in [109]). The remaining five mutation operators (see Table 6, (a), (b), (c), (d)-(1), (d)-(3)) from our work are new and have not been identified by Zhang *et al.*. This difference also reflects that our study's findings are more systematic. As a result, the testing technique proposed by Zhang *et al.* could find only one issue (*i.e.*, Issue 3 in Table 7) out of 15 FNs/FPs found by us. Last but not least, our work identifies additional three FNs/FPs caused by three types of weaknesses of the static analysis modules, while Wang *et al.* and Zhang *et al.* do not perform such deep analysis on the static analysis modules.

There is existing work that focuses on specific types of issues or static analysis tools targeting other programming languages. For example, Zhang *et al.* [110] investigated annotation-induced faults, one type of input characteristic (*i.e.*, *Annotations*) in our study, by analyzing 246 issues in six static analyzers. Our study provides a comprehensive examination of all types of FN/FP issues, yielding a more complete set of issues and root causes. As a result, our findings can better improve the design and testing of static analyzers. He *et al.* [69] leveraged metamorphic testing and dynamic program execution as oracles to find the FN/FP issues in three C/C++ static analyzers. They summarized eight categories of different defects, some of which (*e.g.*, *Implementation Errors*, *Mishandling Language Features*) overlap with the root causes identified in our work. This overlap indicates that the conclusions drawn from our study are, to some extent, also generalizable to other analyzers and different programming languages.

In the literature, there is also other work focused on finding defects in static code analyzers. These works focus on either specific static analysis modules or specific input program characteristics, *e.g.*, value analysis and constant propagation [63], alias analysis [106], data-flow analysis [98], and the configurations of static analysis [86, 87].

There is some work validating the correctness of more sophisticated program analyzers like abstract interpreters [85], symbolic executors [72], model checkers [108] and compilers [73]. But these work does not target the analyzers we studied.

10 CONCLUSION

We present the first systematic study on 1257 historical issues of FNs/FPs from four representative rule-based static code analyzers for Java. We investigated the root causes and input characteristics, which help developers and researchers to understand FNs/FPs. Our study yields some new interesting findings and implications to improve the static code analyzers. Additionally, we conduct two proof-of-concept demonstrations to show the usefulness of our findings: (1) finding FNs and FPs, and (2) investigating the weaknesses of the static analysis modules. We have made our artifacts publicly available at *https://zenodo.org/doi/10.5281/zenodo.11525129* to benefit the community.

Just Accepted

Acknowledgements

This work was supported by the NSFC (Project No. 62572192), the Fundamental Research Funds for the Central Universities, and the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2024-04301.

References

[1] 2022. *[java] UnusedAssignment false-positive with try-with-resources.* https://github.com/pmd/pmd/issues/4127

[2] 2023. *java:S2259 False Negative: Fails to Report Null Pointer Dereferences in Non-static Methods.* https://community.sonarsource.com/t/java-s2259-false-negative-fails-to-report-null-pointer-dereferences-in-non-static-methods/90849

[3] 2023. *Modify rule S2384: Update metadata to clarify the rule scope.* https://github.com/SonarSource/rspec/pull/1943/commits/9f2479f437bf27709f11287ed4a69b8d2ac374ec Accessed: 2024-01-30.

[4] 2024. *artho.com, Jlint.* http://artho.com/jlint/

[5] 2024. *Bug Descriptions - SpotBugs Documentation.* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html Accessed: 2024-01-30.

[6] 2024. *Bug Descriptions: Method call passes null to a non-null parameter (NP_NONNULL_PARAM_VIOLATION).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-call-passes-null-to-a-non-null-parameter-np-nonnull-param-violation Accessed: 2024-01-30.

[7] 2024. *Bug Descriptions: Possible null pointer dereference due to return value of called method (NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-possible-null-pointer-dereference-due-to-return-value-of-called-method-np-null-on-some-path-from-return-value Accessed: 2024-01-30.

[8] 2024. *Bug Descriptions: Private method is never called (UPM_UNCALLED_PRIVATE_METHOD).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#upm-private-method-is-never-called-upm-uncalled-private-method Accessed: 2024-01-30.

[9] 2024. *Checkstyle.* https://github.com/checkstyle/checkstyle Accessed: 2024-01-30.

[10] 2024. *DataflowPass | PMD.* https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/internal/DataflowPass.java

[11] 2024. *DoNotUseThreads - PMD Rules for Java Multithreading.* https://docs.pmd-code.org/latest/pmd_rules_java_multithreading.html#donotusethreads Accessed: 2024-01-30.

[12] 2024. *Infer.* https://github.com/facebook/infer Accessed: 2024-01-30.

[13] 2024. *Java Rules - PMD Source Code Analyzer.* https://docs.pmd-code.org/latest/pmd_rules_java.html Accessed: 2024-01-30.

[14] 2024. *Java Rules - SonarSource Static Code Analysis.* https://rules.sonarsource.com/java/ Accessed: 2024-01-30.

[15] 2024. *JavaParser.* https://github.com/javaparser/javaparser

[16] 2024. *JetBrains Co. IntelliJ® IDEA 7.0.* http://www.jetbrains.com/idea/

[17] 2024. *PMD.* https://pmd.github.io/

[18] 2024. *RSPEC-1144.* https://sonarsource.atlassian.net/browse/RSPEC-1144 Accessed: 2024-01-30.

[19] 2024. *RSPEC-1612.* https://sonarsource.atlassian.net/browse/RSPEC-1612 Accessed: 2024-01-30.

[20] 2024. *RSPEC-1905.* https://sonarsource.atlassian.net/browse/RSPEC-1905 Accessed: 2024-01-30.

[21] 2024. *RSPEC-1948.* https://sonarsource.atlassian.net/browse/RSPEC-1948 Accessed: 2024-01-30.

[22] 2024. *RSPEC-2095.* https://sonarsource.atlassian.net/browse/RSPEC-2095 Accessed: 2024-01-30.

[23] 2024. *RSPEC-2187.* https://sonarsource.atlassian.net/browse/RSPEC-2187 Accessed: 2024-01-30.

[24] 2024. *RSPEC-2201.* https://sonarsource.atlassian.net/browse/RSPEC-2201 Accessed: 2024-01-30.

[25] 2024. *RSPEC-2259.* https://sonarsource.atlassian.net/browse/RSPEC-2259 Accessed: 2024-01-30.

[26] 2024. *RSPEC-2384.* https://sonarsource.atlassian.net/browse/RSPEC-2384 Accessed: 2024-01-30.

[27] 2024. *RSPEC-2386.* https://sonarsource.atlassian.net/browse/RSPEC-2386 Accessed: 2024-01-30.

[28] 2024. *RSPEC-2589.* https://sonarsource.atlassian.net/browse/RSPEC-2589 Accessed: 2024-01-30.

[29] 2024. *RSPEC-2637.* https://sonarsource.atlassian.net/browse/RSPEC-2627 Accessed: 2024-01-30.

[30] 2024. *RSPEC-2699.* https://sonarsource.atlassian.net/browse/RSPEC-2699 Accessed: 2024-01-30.

[31] 2024. *RSPEC-2755.* https://sonarsource.atlassian.net/browse/RSPEC-2755 Accessed: 2024-01-30.

[32] 2024. *RSPEC-3457.* https://sonarsource.atlassian.net/browse/RSPEC-3457 Accessed: 2024-01-30.

[33] 2024. *RSPEC-3518.* https://sonarsource.atlassian.net/browse/RSPEC-3518 Accessed: 2024-01-30.

[34] 2024. *RSPEC-5996.* https://sonarsource.atlassian.net/browse/RSPEC-5996 Accessed: 2024-01-30.

[35] 2024. *Sonar Community.* https://community.sonarsource.com/

[36] 2024. *SpotBugs.* https://spotbugs.github.io/

[37] 2025. *ErrorProne.* https://errorprone.info/

[38] 2025. *ErrorProne GitHub Issues.* https://github.com/google/error-prone/issues Accessed: 2025-08-29.

[39] 2025. *ErrorProne Rule PreferJavaTimeOverload.* https://errorprone.info/bugpattern/PreferJavaTimeOverload Accessed: 2025-08-29.

[40] 2025. *ErrorProne Rule UnusedVariable.* https://errorprone.info/bugpattern/UnusedVariable Accessed: 2025-08-29.

[41] 2025. *Java Rules - ErrorProne Source Code Analyzer.* https://pmd.github.io/pmd/pmd_rules_java_errorprone.html Accessed: 2025-08-29.

[42] 2025. *PMD GitHub Issues.* https://github.com/pmd/pmd/issues Accessed: 2025-08-29.

[43] 2025. *PMD Rules: CloseResource.* https://docs.pmd-code.org/latest/pmd_rules_java_errorprone.html#closeresource Accessed: 2025-08-29.

[44] 2025. *PMD Rules: CompareObjectsWithEquals.* https://pmd.github.io/pmd/pmd_rules_java_errorprone.html#compareobjectswithequals Accessed: 2025-08-29.

Just Accepted

[45] 2025. *PMD Rules: DataflowAnomalyAnalysis.* https://docs.pmd-code.org/latest/pmd_rules_java_errorprone.html#dataflowanomalyanalysis Accessed: 2025-08-29.
[46] 2025. *PMD Rules: ImmutableField.* https://docs.pmd-code.org/latest/pmd_rules_java_design.html#immutablefield Accessed: 2025-08-29.
[47] 2025. *PMD Rules: InvalidLogMessageFormat.* https://docs.pmd-code.org/latest/pmd_rules_java_errorprone.html#invalidlogmessageformat Accessed: 2025-08-29.
[48] 2025. *PMD Rules: MethodArgumentCouldBeFinal.* https://docs.pmd-code.org/latest/pmd_rules_java_codestyle.html#methodargumentcouldbefinal Accessed: 2025-08-29.
[49] 2025. *PMD Rules: SimplifiedTernary.* https://docs.pmd-code.org/latest/pmd_rules_java_design.html#simplifiedternary Accessed: 2025-08-29.
[50] 2025. *PMD Rules: UnnecessaryFullyQualifiedName.* https://docs.pmd-code.org/latest/pmd_rules_java_codestyle.html#unnecessaryfullyqualifiedname Accessed: 2025-08-29.
[51] 2025. *PMD Rules: UnnecessaryImport.* https://docs.pmd-code.org/latest/pmd_rules_java_codestyle.html#unnecessaryimport Accessed: 2025-08-29.
[52] 2025. *PMD Rules: UnusedPrivateField.* https://docs.pmd-code.org/latest/pmd_rules_java_bestpractices.html#unusedprivatefield Accessed: 2025-08-29.
[53] 2025. *PMD Rules: UnusedPrivateMethod.* https://docs.pmd-code.org/latest/pmd_rules_java_bestpractices.html#unusedprivatemethod Accessed: 2025-08-29.
[54] 2025. *SonarJava Jira Issues.* https://sonarsource.atlassian.net/jira/software/c/projects/SONARJAVA/issues?jql=project%20%3D%20%22SONARJAVA%22%20ORDER%20BY%20created%20DESC Accessed: 2025-08-29.
[55] 2025. *SpotBugs GitHub Issues.* https://github.com/spotbugs/spotbugs/issues Accessed: 2025-08-29.
[56] 2025. *wikipedia: Inter-rater reliability.* https://en.wikipedia.org/wiki/Inter-rater_reliability Accessed: 2025-08-29.
[57] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.* 45, 3 (2019), 261–284. https://doi.org/10.1109/TSE.2017.2776152
[58] Arie Ben-David. 2008. Comparison of classification accuracy using Cohen's Weighted Kappa. *Expert Systems with Applications* 34, 2 (2008), 825–832.
[59] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75. https://doi.org/10.1145/1646353.1646374
[60] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action.*
[61] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 2020. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology.
[62] César Couto, João Eduardo Montandon, Christofer Silva, and Marco Túlio Valente. 2013. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Softw. Qual. J.* 21, 2 (2013), 241–257. https://doi.org/10.1007/s11219-011-9172-5
[63] 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. https://doi.org/10.1007/978-3-642-28891-3_12
[64] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2490–2520.
[65] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 234–245. https://doi.org/10.1145/512529.512558
[66] 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. https://doi.org/10.1109/QRS54544.2021.00032
[67] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Trans. Software Eng.* 49, 12 (2023), 5154–5188. https://doi.org/10.1109/TSE.2023.3329667
[68] 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).* ACM, 317–328. https://doi.org/10.1145/3238147.3238213
[69] 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. 1, FSE (2024). https://doi.org/10.1145/3660781
[70] David Hovemeyer and William W. Pugh. 2004. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 132–136. https://doi.org/10.1145/1028664.1028717

Just Accepted

[71] 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 *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[72] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 590–600. https://doi.org/10.1109/ASE.2017.8115669

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

[74] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are sonarqube rules inducing bugs?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 501–511. https://doi.org/10.1109/SANER48275.2020.9054821

[75] 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)*. ACM, 921–933. https://doi.org/10.1145/3611643.3616262

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

[77] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 544–555. https://doi.org/10.1145/3533767.3534380

[78] 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. https://doi.org/10.1145/3597926.3598056

[79] Zixi Liu, Yang Feng, Yunbo Ni, Shaohua Li, Xizhe Yin, Qingkai Shi, Baowen Xu, and Zhendong Su. 2025. An Empirical Study of Bugs in the `rustc` Compiler. In *Proceedings of the ACM on Programming Languages, OOPSLA (SPLASH)*. Singapore, to appear.

[80] Panagiotis Louridas. 2006. Static Code Analysis. *IEEE Softw.* 23, 4 (2006), 58–61. https://doi.org/10.1109/MS.2006.114

[81] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *Proceedings of the 27th International Conference on Program Comprehension*. 209–219. https://doi.org/10.1109/ICPC.2019.00040

[82] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Dias Canedo, Welder Pinheiro Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed?: a closer look at realistic usage of SonarQube. In *Proceedings of the 27th International Conference on Program Comprehension, (ICPC)*. 209–219. https://doi.org/10.1109/ICPC.2019.00040

[83] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *J. Syst. Softw.* 168 (2020), 110671. https://doi.org/10.1016/j.jss.2020.110671

[84] Sahar Mehrpour and Thomas D. LaToza. 2023. Can static analysis tools find more defects? *Empir. Softw. Eng.* 28, 1 (2023), 5. https://doi.org/10.1007/s10664-022-10232-4

[85] Jan Midtgaard and Anders Møller. 2015. QuickChecking Static Analysis Properties. In *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102603

[86] Austin Mordahl and Shiyi Wei. 2021. The impact of tool configuration spaces on the evaluation of configurable taint analysis for Android. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 466–477. https://doi.org/10.1145/3460319.3464823

[87] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 550–562. https://doi.org/10.1109/ICSE48619.2023.00056

[88] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 532–543. https://doi.org/10.1145/3533767.3534374

[89] Louis-Philippe Querel and Peter C. Rigby. 2021. Warning-Introducing Commits vs Bug-Introducing Commits: A tool, statistical models, and a preliminary user study. In *29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 433–443. https://doi.org/10.1109/ICPC52881.2021.00051

[90] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing Static Bug Finders and Statistical Prediction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 424–434. https://doi.org/10.1145/2568225.2568269

[91] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. 2020. Empirical standards for software engineering research. *arXiv preprint arXiv:2010.03525* (2020).

Just Accepted

[92] Zachary P. Reynolds, Abhinandan B. Jayanth, Ugur Koc, Adam A. Porter, Rajeev R. Raje, and James H. Hill. 2017. Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. In *4th IEEE/ACM International Workshop on Software Engineering Research and Industrial Practice*. IEEE, 55–61. https://doi.org/10.1109/SER-IP.2017..20

[93] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An empirical study of bugs in webassembly compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.

[94] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[95] Justin Smith, Brittany Johnson, Emerson R. Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Trans. Software Eng.* 45, 9 (2019), 877–897. https://doi.org/10.1109/TSE.2018.2810116

[96] Donna Spencer and T Warfel. 2004. Card Sorting. *Boxes and arrows* 7 (2004).

[97] Fausto Spoto. 2010. The nullness analyser of Julia. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 405–424.

[98] 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. https://doi.org/10.1145/3368826.3377927

[99] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. https://doi.org/10.1145/2351676.2351685

[100] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Autom. Softw. Eng.* 22, 4 (2015), 561–602. https://doi.org/10.1007/s10515-014-0169-8

[101] David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 292–303. https://doi.org/10.1109/ASE51524.2021.9678535

[102] 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. https://doi.org/10.1007/s10664-019-09750-5

[103] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find bugs in static bug finders. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 516–527. https://doi.org/10.1145/3524610.3527899

[104] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a "golden" feature set for static warning identification?: an experimental evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 17:1–17:10. https://doi.org/10.1145/3239235.3239523

[105] Fadi Wedyan, Dalal Alrmuny, and James M. Bieman. 2009. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction. In *Second International Conference on Software Testing Verification and Validation (ICST)*. 141–150. https://doi.org/10.1109/ICST.2009.21

[106] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective dynamic detection of alias analysis errors. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 279–289. https://doi.org/10.1145/2491411.2491439

[107] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 1319–1331.

[108] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 763–773. https://doi.org/10.1145/3338906.3338932

[109] 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)*. ACM, 237–249. https://doi.org/10.1145/3611643.3616272

[110] 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, Article 33 (jul 2024), 23 pages. https://doi.org/10.1145/3643759

[111] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting Code Knowledge Graph for Bug Localization via Bi-directional Attention. In *ICPC '20: 28th International Conference on Program Comprehension*. 219–229. https://doi.org/10.1145/3387904.3389281

Just Accepted