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

HAN CUI, MENGLEI XIE, and TING SU, East China Normal University, China
CHENGYU ZHANG, ETH Zurich, Switzerland
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 the 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 350 historical issues of FNs/FPs from three popular static code analyzers (*i.e.*, PMD, SpotBugs, and SonarQube). 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 14 new issues of FNs/FPs, 11 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 four 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*.

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

1 INTRODUCTION

Static code analyzers (*or* static checkers) [55] are commonly used to help find program flaws at the early stages of software development (after code compilation and before testing) [37, 58, 64, 68]. These analyzers usually target various types of program flaws including best practice violations, code design issues, common programming mistakes and security vulnerabilities [43, 68, 76]. Otherwise, these flaws might be costly and difficult to find by manual code reviews or testing [35, 48, 82]. 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, it is known that the effectiveness and usability of these analyzers could be affected by the problems of *false negatives* (*FN*s for short), *i.e.*, missing *true* program flaws [46, 71, 72], and *false positives* (*FP*s for short), *i.e.*, the reported warnings are *spurious* [48, 51, 77]. To investigate FNs and FPs, most of existing studies [46, 51–54, 59, 71–73, 77] evaluate the fault detection abilities or the usability of these analyzers against known faults. For example, Habib *et al.* [46] use the faults of Defects4J to investigate FNs and find that these analyzers could *only* find 4.5% of the field defects; Wedyan *et al.* [77] use the known coding faults from some open-source projects to investigate FPs and find that 96% of the warnings reported by these analyzers 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 field defects are missed because these defects are not targeted by existing rules in the analyzers or these defects are domain-specific errors [46, 52, 54, 71, 72].

Authors' addresses: Han Cui; Menglei Xie; Ting Su, East China Normal University, China; Chengyu Zhang, ETH Zurich, Switzerland; Shin Hwei Tan, Concordia University, Canada.

Different from these prior studies, 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 of these issues manifested as a FN or FP of these analyzers and has been confirmed and fixed by developers. Exploring this perspective has one key benefit — we can inspect the fixing patches and the implementations (which however the prior studies usually do not care about) of the analyzers to obtain the fine-grained insights on why FNs and FPs are induced. These insights could be useful for the analyzers' developers to mitigate FNs/FPs at the root. To our knowledge, *no* prior work *systematically* studies such issues.

To fill the gap, we studied a broad range of 350 historical issues (corresponding to 80 FNs and 270 FPs), which were collected from the issue repositories of three popular, open-source static code analyzers (*i.e.*, PMD [20], SpotBugs [34], and SonarQube [39]). All these issues are valid and representative because they have been confirmed *and* already fixed by the developers. Specifically, to obtain a general and in-depth understanding of FNs and FPs, we study how these issues are induced (*i.e.*, *root causes*), and which characteristics of input programs could lead to these issues (*i.e.*, *input characteristics*). 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 [75, 80] in detail in Section 8.

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? 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 3). This RQ identifies the reasons why the FNs/FPs are induced and helps developers avoid or mitigate such issues.
- **RQ2 (Input Characteristics)**: What are the characteristics of input programs leading to these historical issues of FNs and FPs of the static code 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 4). This RQ identifies which characteristics of the input programs are pivot for inducing FNs or FPs and helps developers design better test programs or testing strategies for static code analyzers.

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 [46, 51–54, 59, 71–73, 77]. Through answering **RQ1** and **RQ2**, we obtained several new interesting findings and implications that shed light on mitigating FNs/FPs (Section 5). Therefore, we aim to investigate the usefulness of these findings:

- **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 validate 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 14 new issues (12 FNs and 2 FPs) of the studied analyzers, 11 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 four 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:

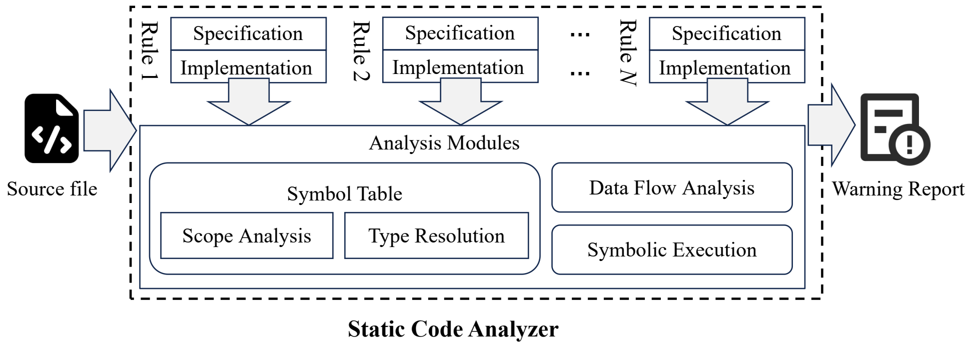


Fig. 1. Typical workflow and architecture of a static code analyzer.

- 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 350 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 350 historical issues of FNs and FPs to investigate their root causes and input characteristics and identify several new interesting findings.
- We discuss the implications of our study to shed light on mitigating FNs and FPs. We also demonstrate the usefulness of our study's findings by two proof-of-concept demonstrations.
- We have made all the artifacts (datasets and tools) publicly available at *https://zenodo.org/doi/10.5281/zenodo.11525129*.

2 STUDY METHODOLOGY

This section details the methodology of our study. Specifically, Section 2.1 gives some background knowledge on the static code analyzers, Section 2.2 introduces the selected analyzers for our study, Section 2.3 presents how we collect the historical issues of FNs and FPs from the issue repositories of the studied analyzers, and Section 2.4 explains how we manually analyze these issues.

2.1 Background on Static Code Analyzers

Figure 1 shows the typical workflow and architecture of classic static code analyzers like PMD [20], SpotBugs [34], and SonarQube [39]. Given the source files under check, an analyzer converts it to some form of intermediate representation, *e.g.*, abstract syntax tree (AST), bytecode or control-/data-flow graph (CFG/DFG), and searches for the code snippets violating the rules (outputted as a warning report). Specifically, an analyzer usually implements a number of rules, each of which has its own specification and implementation (*e.g.*, PMD's rules [17], SpotBugs's rules [6], SonarQube's rules [18]). Each rule is designed to detect one specific type of program flaws, *e.g.*, *best practice violations*, *code design issues*, *common programming mistakes* and *security vulnerabilities*. In detail, each rule is implemented based on some form of static analysis (chosen by the developers of the analyzers), *e.g.*, AST-based syntactic pattern matching, data-flow analysis and symbolic execution. Note that the major components of different analyzers may not align with each other. For example, all the classic analyzers like PMD, SpotBugs and SonarQube implement a number of checking rules (associated with the corresponding rule specifications), and all the analyzers would construct the symbol tables in their analysis modules. But only SonarQube uses symbolic execution for static analysis. PMD only implements reaching definition analysis (one specific data-flow analysis), while SpotBugs implements data-flow analysis more generally supporting forward and backward analyses. In addition, PMD and SonarQube's analysis module works at the source code level, while SpotBugs's works at the bytecode level.

For example, *AbstractClassWithoutAbstractMethod* [5] is one of the rules implemented in PMD to enforce generally accepted best practices. It warns any abstract class which does not contain any abstract methods because an abstract class suggests an incomplete implementation, which is to be completed by subclasses implementing the abstract methods. This rule is implemented by syntactic pattern matching based on AST. For another example, *S2259 (Null pointers should not be dereferenced)* [28] is one of the rules in SonarQube to detect programming bugs. It checks that a reference to `null` should never be dereferenced or accessed because doing so will cause a `NullPointerException`. This rule is implemented upon SonarQube's symbolic execution engine.

2.2 Static Code Analyzers Selected for Our Study

In this work, 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 three representative static code analyzers for Java, *i.e.*, PMD [20], SpotBugs [34], and SonarQube [39], as the subjects for our study based on the following reasons. First, these tools 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 [47], has been used by Google; SonarQube has been applied at the CI pipelines by many companies. Moreover, these tools have been widely studied by many prior work [41, 46, 51, 57, 65, 75, 77]. Second, these tools are open-sourced. It eases the issue collection and analysis. We can inspect the fixing patches and tool implementations to investigate FNs and FPs. Third, these tools adopt different analysis strategies, including syntactic pattern matching, data-flow analysis and symbolic execution. These characteristics can help us gain more overall understanding on FNs and FPs. There are other static code analyzers for Java [63] like FindBugs [47], ErrorProne [15], Infer [16] and CheckStyle [12], but we did not select these tools 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 and ErrorProne do not maintain the issues of FNs and FPs (*e.g.*, without 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, and Checkstyle focuses more on coding standards rather than program flaws.

Figure 2 shows our study's workflow including collecting and analyzing the historical issues of FNs and FPs from the issue repositories of the studied analyzers. We explain the process in Section 2.3 and Section 2.4.

2.3 Collecting Historical Issues of FNs and FPs

To obtain a broad range of issues, we collected *all* the reported issues before the time of our study (which was started in Oct. 2022). We focus on the FNs/FPs that were reported on checking Java programs supported by all the three studied analyzers. Moreover, we require that these FNs/FPs have been confirmed *and* fixed by the developers because the confirmed bug information, the committed patches, the discussions between the developers and the implementations of the rules provide us informative details to study the root causes and the input characteristics of these issues.

We detail the issue collection process below. Note that we focus on those issues of FNs and FPs which have been confirmed *and* fixed. 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 and SonarQube, 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 the fixing commits.

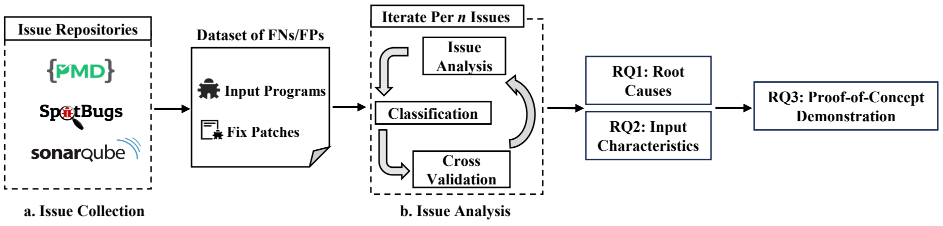


Fig. 2. Workflow of our study.

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	2017/2/17	2022/10/8	226(155/71)
SpotBugs	3K	468	2016/12/7	2022/10/27	21(15/6)
SonarQube	7.7K	618	2018/6/11	2022/10/30	103(100/3)

- **PMD**. We collected all the *closed* issues with the labels *"a:false-positive"* and *"a:false-negative"* from PMD's issue repository on GitHub. 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 80 FNs and 228 FPs.
- **SpotBugs**. SpotBugs's issues are not well-classified with explicit issue labels. Thus, we checked whether the issue title or body contains the keywords *"FP"*, *"FN"*, *"bug"*, *"error"*, *"false positive"*, *"false alarm"*, or *"false negative"* to filter issues. Finally, we obtained 104 unclassified issues.
- **SonarQube**. SonarQube's issues are maintained on the community website. We use the APIs of Sonar community [33] to collect the issues with the labels *"Clean Code"*, *"Report False-positive / False-negative"*, *"java"*, *"answered"*, and *"closed"*. We obtained 229 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 those mislabeled 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 350 FNs/FPs, including 71 FNs/155 FPs of PMD, 6 FNs/15 FPs of SpotBugs, and 3 FNs/100 FPs of SonarQube. Each issue is associated with at least one piece of issue-triggering program 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, or the issue-triggering programs were deemed as unrealistic (not likely written by humans). The issues which were closed but not confirmed were also excluded from our study.

2.4 Analyzing Historical Issues of FNs and FPs

To answer RQ1 and RQ2, we investigated 350 FNs/FPs in the dataset to understand the root causes and input characteristics. Specifically, we inspected the documentation (*e.g.*, rule specifications), the discussions in the issue reports, the issue-triggering program and the fixing patches. In this process, some issue labels (*e.g.*, *dataflow analysis*) or some keywords (*e.g.*, *anonymous class*) in the issue titles could indicate the likely root causes or the input characteristics leading to the issue. 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. During the analysis process, we may inspect the AST of the issue-triggering program and mutate the program to locate which code parts are pivot for triggering FNs/FPs.

Specifically, to build the taxonomies, we adopted the open card sorting approach [69] 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 (with 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 the discussion to help make the final decision. Such an iteration was repeated twelve times until all 350 issues were analyzed. We observed that this iterative process converged on the categories after the first 3 rounds. This manual analysis process requires considerable knowledge of Java and the implementations of static code analyzers, which took us around six person months. During this manual analysis process, we computed the Cohen's Kappa coefficient [36] to evaluate the inter-rater agreement [**?**] between the two co-authors. 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 ensured that 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 the discussions between them in each round, they eventually achieved a perfect agreement with a Kappa 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, affected versions, latency, and fixability) because these other dimensions are difficult to be analyzed or may bring additional threats to validity. For example, none of PMD, SpotBugs and SonarQube maintains the labels of severity because how to tell the severity of a FN or a FP is unclear. Investigating the versions of the analyzers affected by the issues of FNs and FPs may not offer interesting insights because the FNs/FPs were reported for the checking rules. Instead, examining how many times of the rules were affected by FNs/FPs might be more interesting (which we have investigated in Section 6.2). 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.

3 RQ1: ROOT CAUSES

In this section, we focus on the 350 issues in the three analyzers to study the root causes. We identified the 7 major root causes from 350 FNs/FPs. To be concise, we brief these root causes in Table 2. Specifically, we categorized the issues into the disjoint groups of root causes according to the major components of a static code analyzer (see Figure 1) in which the issues may happen. We explain and illustrate these root causes as follows.

3.1 Flawed Rule Specification

In this category, the rule specification itself (documented in natural language) is flawed, thus leading to FNs or FPs. For example, PMD's rule *DoNotUseThreads* [14] 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 the use of `Runnable` because the developers misunderstand

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

Category	Description	Example	#Issues	Ratio
○ ○ **Flawed Rule Specification**	Some design flaws in the rule specification.	-	21	6%
◐ ○ **Inconsistent Rule Implementation**	The rule specification is correct but the implementation is inconsistent with the specification.	-	9	2.6%
Unhandled Language Features or Libraries	Unhandled Language Features or Libraries	-	101	28.9%
○ ◐ Unhandled Language Features	Some language features (*e.g.* lambda expressions) are not handled.	Figure 3a	74	21.1%
○ ○ Unhandled Java Libraries	Some Java libraries (*e.g.* java.util.Optional) are not handled.	-	27	7.7%
Missing Cases	Missing Cases	-	125	35.7%
○ ○ Missing cases that should be whitelisted	Some objects that should be added to the whitelist are missing.	Figure 3b	38	10.9%
◐ ○ Missing cases which should be similarly handled	Missing cases or objects which should be similarly handled.	-	30	8.6%
◐ ○ Missing specific cases	Missing specific cases that are not covered by the two categories above	-	57	16.3%
○ ◐ **Mishandling Intermediate Representations**	Matching the incorrect nodes during the traversal of the IR (*e.g.* AST)	-	24	6.9%
Analysis Module Error or Limitation	Analysis Module Error or Limitation	-	51	14.6%
○ ○ Scope analysis error or limitation	Wrong resolving of the variable scopes.	Figure 3c	6	1.7%
○ ○ Type resolution error or limitation	Incorrect or limited type resolution.	Figure 3d	38	10.9%
○ ○ Dataflow analysis error or limitation	Errors in the dataflow analysis or limited use of dataflow analysis	Figure 3e	3	0.9%
○ ○ Symbolic execution error or limitation	Errors or limitations in the symbolic execution engine.	Figure 3f	4	1.1%
○ ○ **General Programming Error**	General errors which are not unique for static code analyzers.	-	11	3.1%
Miscellaneous	Minor issues that affect only a few FNs/FPs.	-	8	2.3%

The symbols"◐" and "○", "◐" and "○" denote the differences between our study and the two most relevant work from Wang *et al.* [75] and Zhang *et al.* [80], respectively, in terms of the results of issue analysis. We discuss the differences in detail in Section 8.

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 be warned. Due to this flawed specification, this rule reports a FP (PMD's Issue #1627) when the input program uses `Runnable`. To resolve this issue, the developer fixed the rule specification and implementation[1].

Finding 1 & Implications: *Flawed Rule Specification* could lead to both FNs and FPs. To avoid such rule specification issues, the analyzers' developers should carefully inspect and fully understand the language specifications (*e.g.*, Java language) as well as the language's idioms and the best practices when designing the rules.

3.2 Inconsistent Rule Implementation

In this category, the rule specification is correct but the implementation is *inconsistent*. For example, PMD's rule *AvoidThrowingNullPointerException* is specified to warn manually throwing

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

```
1  public class Outerclass {
2      private int[] arr;
3      public int[] getArr() {return arr;}  // true positive
4      public static class Innerclass{
5            private int[] arr2;
6            public int[] getArr() {return arr2;} // FN
7      }
8  }
```

(a) Unhandled language features: PMD #1738

```
// should be whitelisted
@javax.annotation.concurrent.Immutable
 class MyImmutable{...}

// should be whitelisted
@javax.annotation.concurrent.ThreadSafe
 class MyThreadSafe{...}

 class Main
 {
    private volatile MyImmutable x;    // FP
    private volatile MyThreadSafe y;   // FP
 }
```

(b) Missing cases that should be whitelisted: SonarJava-3804

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

(c) Scope analysis limitation: PMD #3284

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

(d) Type resolution limitation: PMD #2976

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

(e) Dataflow analysis error: PMD #1749

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

(f) Symbolic execution error: SonarJava-3619

Fig. 3. Illustrative examples for explaining root causes (the code snippets are simplified).

`NullPointerExceptions`. However, this rule's implementation is inconsistent with the specification. It simply warns every occurrence of `NullPointerException` without checking whether an exception throwing occurs. As a result, this rule leads to a FP (PMD's Issue #2580): for example, the rule will warn this code line `Exception e = new NullPointerException("Test message")`, however this code line does not throw an exception but only creates an exception object.

3.3 Unhandled Language Features or Libraries

3.3.1 Unhandled Language Features. A rule may lead to FNs or FPs if some language features (*e.g.*, lambda expressions, nested classes) are not handled. For example, Figure 3a showcases a FN of PMD's rule *MethodReturnsInternalArray* induced by nested classes. This rule warns the methods that return internal arrays. In this case, `arr` at line 2 and `arr2` at line 5 are internal arrays. The two methods `getArr()`s at line 3 and 6, return the internal arrays `arr` and `arr2`, respectively. They violate the rule. However, the rule did not handle the nested class, and did not warn at line 6.

3.3.2 Unhandled Java Libraries. Some Java libraries were not handled, leading to FNs or FPs. For example, Java 8 introduces the class `Optional` to handle optional values. Specifically, `Optional` provides `Optional.isPresent()` to check if a value is present (*i.e.*, `non-null`). Later, Java 11 introduced a new method `Optional.isEmpty()`. This method allows to check whether the `Optional` value is `null`. However, SonarQube failed to timely support `Optional.isEmpty()`, leading to a FP (see SonarQube's issue SonarJava-3087).

Finding 2 & Implications: *Unhandled language features or libraries* affects 101 of the 350 issues (28.9%), which is one major root cause. It indicates that the analyzers' developers should *timely* check the rules when some new Java language features or new Java libraries are introduced, and update the rule implementations if necessary.

3.4 Missing Cases

3.4.1 Missing cases that should be whitelisted. The analyzers commonly use whitelists (precluding checking on specific code elements) to avoid spurious warnings. However, these analyzers may miss the cases which should be whitelisted. For example, Figure 3b showcases a FP of SonarQube's rule *S3077* due to failing to whitelist some Java annotations. This rule warns the non-primitive fields modified by `volatile`. In this example, the classes `MyImmutable` and `MyThreadSafe` are annotated as immutable (line 2) and thread-safe (line 6), respectively. It means the immutability and the thread-safety are already ensured by the users. Thus, the non-primitive fields (*i.e.*, `x` at line 11 and `y` at line 12) should not be warned. The developers fixed this FP by adding the annotations `@ThreadSafe` and `@Immutable` into the whitelist of this rule. *Note that the issues of this category all lead to FPs.*

3.4.2 Missing cases which should be similarly handled. Some cases are functionally equivalent *w.r.t.* the rule specification. However, sometimes analyzers may only handle some cases but fail to similarly handle others. For example, SonarQube's issue SonarJava-3586 reported a FP because SonarQube only handles the annotation `org.springframework.lang.Nullable` but not the annotation `reactor.util.annotation.Nullable`. Both of these annotations have identical semantics, *i.e.*, indicating the annotated elements can be `null` in some circumstances. To fix this issue, the developers added `reactor.util.annotation.Nullable`. Note that this issue does not belong to the category of unhandled language features (discussed in Section 3.3.1) because the rule can correctly handle the @Nullable annotations but missed `reactor.util.annotation.Nullable`.

3.4.3 Missing specific cases. The analyzers may miss other specific cases when doing rule checking. For example, PMD's Issue #2275 showcases a FP of the rule *AppendCharacterWithChar*. For an object of class `StringBuffer`, say `sb`, this rule recommends converting `sb.append("a")` to `sb.append('a')` to improve performance. However, for the case of `sb.append("a".repeat(length))`, it is valid and should not be warned by this rule. But PMD assumes that `append()`'s argument should always be string literals. It does not consider the case of using method calls like `repeat()`.

Finding 3 & Implications: *Missing cases* affects 125 of the 350 issues (35.7%), which is the most common root cause. It indicates that the analyzers' developers should (1) carefully decide which cases need to be whitelisted and similarly handled, and (2) design different diverse test programs to validate the rule implementations.

3.5 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.

3.6 Analysis Module Error or Limitation

3.6.1 Scope analysis error or limitation. Java adopts the static scoping rules to analyze the scopes of symbols. However, some issues are caused by imprecise scope analysis. Figure 3c showcases a FP of PMD's rule *InvalidLogMessageFormat*. 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

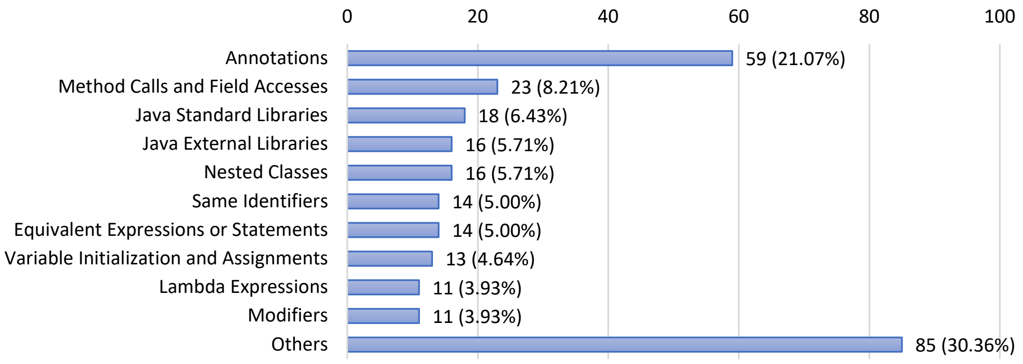


Fig. 4. Input characteristics leading to FNs/FPs.

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).

3.6.2 Type resolution error or limitation. Type resolution is a crucial ability of analyzers to decide the types of symbols. However, the implementation of type resolution may have some limitations. Figure 3d showcases a FP of PMD's rule *CompareObjectsWithEquals* due to the incorrect type resolution. This rule requires using "`equals()`" instead of "==" to compare object references. However, at line 2, `a1.length` and `a2.length` are the type of integers and it is fine to use "==". PMD's incorrect type resolution leads to a FP. We observe that these three analyzers do not utilize Java compilers for precise type resolution, but implement their own type resolutions.

3.6.3 Dataflow analysis error or limitation. Some dataflow analysis errors could lead to FNs or FPs. For example, Figure 3e showcases a FP of PMD's rule *DataflowAnomalyAnalysis*. 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).

3.6.4 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. For example, Figure 3f showcases a FP of SonarQube's rule *S2589*. This rule warns 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".

Finding 4 & Implication: *Type resolution errors or limitations* is the most common root cause among the static analysis modules. It indicates that improving the abilities and precision of such static analysis is important to mitigate FNs/FPs.

3.7 General Programming Error

Some FNs or FPs are caused by general programming errors, *e.g.*, mistaking the logic expression `(A || B)` as `(A && B)`, where `A` and `B` are boolean conditions.

```
1   @Value
2   public class CustomException extends RuntimeException{
3       String customValue; // FP: the final modifier should be added
4   }
```

(a) False positive with *annotations*: SonarJava-3320

```
1   import static java.util.Objects.*;
2   public void bar(Connection conn, String sql) {
3       PreparedStatement lPreparedStmt = null; // FP: not closed after use
4       try {
5           lPreparedStmt = conn.prepareStatement(sql);
6       } catch (SQLException ex) {...
7       } finally {
8           if (nonNull(lPreparedStmt)) {     // nonnull() is not recognized
9           // if (lPreparedStmt != null){    // works fine
10              try {
11                  lPreparedStmt.close();
12              } catch (SQLException pEx) {...}
13  }}}
```

(b) False positive with *Java standard libraries*: PMD #3148

```
1    // correct case
2   public class Case1 {
3       public void foo() {doSomething();}
4       private static void doSomething() {}    // no FP
5   }
6   // FPcase
7   public class Case2 {
8       public void foo() {
9           InnerClass.doSomething();   // doSomething() is used here
10  }
11      static class InnerClass {
12        private static void doSomething(){ // FP: unused private method
13        }
14      }
15  }
```

(c) False positive with *nested classes*: PMD #3468

```
1   public class Test {
2       public double energy(int x) {    return 0.0;   }
        // if the array energy is renamed as energy2,  no FP
3       private void f(double[] energy) {
4           energy[0] = energy(1);
5   }}
```

(d) False positive with *same identifiers*: PMD #1474

```
1   public class Foo {
2       public void bar(int a) {
3           if (a > 3 + 5) {    // FN: avoid literal in if condition
4           // if (a > 8)       // can be correctly warned
5       }}
6   }
```

(e) False negative with *complex expressions*: PMD #2140

```
1   public int case() {
2       int start;      // FP: unused variable start
3       if (true)    start = 1;
4       else    start = 2;
5       return start;
6   }
```

(f) False positive with *initialization and assignments*: PMD #3114

```
1   Runnable someAction = () -> {
2       var foo = new ArrayList<String>(5); // FP: use diamond
    operator
3       System.err.println(foo);
4   };
```

(g) False positive with *lambda expressions*: PMD #1723

```
1   public boolean func() {
2       String s1 = "str1";
3       final String s2 = "str2";
4       return s1 == s2;     // FN: strings are compared by "=="
5   }
```

(h) False negative with *modifiers*: SpotBugs #1764

Fig. 5. Illustrative examples for explaining input characteristics (the code snippets are simplified)

3.8 Miscellaneous

This category includes some miscellaneous reasons. For example, some issues are caused by the unhandled whitespace in rule properties, some are caused by missing setting the property value of target type, and some are the limitations of libraries used by the analyzers (*e.g.* limitations of XPath version 1.0 used by PMD). These cases are not relevant to the core functionality of the analyzers in performing rule checking.

4 RQ2: INPUT CHARACTERISTICS

This section investigates the input characteristics leading to FNs and FPs of the studied analyzers. From the 350 issues, we excluded 70 issues because these issues are not relevant to the input programs (*e.g.*, those issues caused by *flawed rule specification*, *inconsistent rule implementation*, *general programming errors*). Thus, we analyzed the remaining 280 issues and identified 10 major input characteristics. Figure 4 shows their proportions. We illustrate these input characteristics from the most to the least common and also discuss their correlations with the root causes.

Annotations. Java annotations are a form of metadata that provides additional information of Java programs. They are usually placed above the declarations of code segments (*e.g.*, classes, methods and fields) to provide compile-time or runtime information. Figure 5a showcases this characteristic which leads to a FP of SonarQube's rule *S1165*. This rule requires that the fields of exception classes should be `final`. When `@Value` (from Lombok) annotates the class `CustomException` (line 1), all the fields (*e.g.*, `customValue` at line 3) in the class are made `final` by default. However, the rule does not handle `@Value`, and thus warns that a `final` modifier should be added (a FP). Note that this category of "annotations" are mainly triggered by the root causes of *missing cases that should be whitelisted* (Section 3.4.1) and *unhandled Java libraries* (Section 3.3.2).

Finding 5 & Implication: *Annotations* is the most common input characteristics of programs to trigger FNs/FPs. It is mainly related with the root cause of *missing cases that should be whitelisted* and affects all the three studied analyzers. The analyzers' developers should carefully model these annotations in the rule implementations.

Method Calls and Field Accesses. Method calls or field accesses may induce FNs or FPs. This characteristic is mainly related with the root cause of *type resolution errors or limitations* (Section 3.6.2). For example, PMD rule *UseEqualsToCompareStrings* warns comparing strings by using "==" or "`!=`". In PMD issue #3004, this rule gives a FP on the expression `s.charAt(0) == s.charAt(1)`, where `s` is a `string` variable and `s.charAt(i)` is a method call returning the character at index `i` of `s`. This FP is caused by the incorrect type resolution on the return value of `charAt()`. Figure 3e shows another example of FP with the characteristics of accessing the field `length` of an `array`.

Java Standard Libraries. Static code analyzers may fail to handle the classes in Java standard library (*e.g.*, `java.lang.*`). This characteristic is mainly related with the root cause of *unhandled java libraries* (Section 3.3.2). For example, Figure 5b shows a FP of PMD's rule *CloseResource* when handling `java.util.Objects.nonNull()`. This rule warns unclosed resources. In Figure 5b, the resource `lPreparedStmt` is created (line 5) and properly closed (line 11). However, PMD reports a FP as it fails to handle the semantic of `nonNull()`, thus believes `lPreparedStmt.close()` is not reachable. Replacing `nonNull()` with `lPreparedStmt!=null` eliminates the FP.

Nested Classes. Some rules may fail to support or handle nested classes. This characteristic is mainly related with the root cause of *unhandled language features* (Section 3.3.1). Figure 5c illustrates a FP of PMD's rule *UnusedPrivateMethod.* This rule warns unused private methods. For the private method `doSomething()` in class `Case1`, PMD does not report a warning at line 4. However, when `doSomething()` is declared in a nested class `InnerClass` (line 12) but used in `foo` (line 9), a FP (line 12) occurs.

Same Identifiers. In Java programs, variables, fields, methods may have the same symbol names. This characteristic is mainly related with the root causes of *type resolution errors or limitations* (Section 3.6.1) and *type resolution errors or limitations* (Section 3.6.2). Because the same identifiers may complicate the symbol table construction which requires scope analysis and type resolution. Figure 5d shows an input program that triggers a FP of PMD's rule *ArrayIsStoredDirectly.* This rule warns that the constructors and methods receiving arrays should clone objects and store the copy. Because it can prevent future changes from the users affecting the original array. However, in Figure 5d, when the double type array `energy` (line 3) has the same symbol name as the method `energy` (line 2), this rule mistakes the method call `energy` (on the right hand side, line 4) as an array, and thus reports a spurious warning (FP).

Complex Expressions or Statements. Some complex expressions or statements may induce FNs or FPs. This characteristic is mainly related with the root causes of *mishandling intermediate representations* (Section 3.5) and *missing specific cases* (Section 3.4.3). For example, complex arithmetic operations (*e.g.* changing `a>3` to `a>1+2`) or complex boolean operations (*e.g.* changing `false` to `false||false`), addition of no side-effect expressions (*e.g.*, encapsulating statements by `if(true)` or expressions by `{}`), and addition of `this.` to non-static field access (*e.g.* changing `a` to `this.a`). Figure 5e shows a FN due to the complex arithmetic operation. While the rule successfully detects simple expression `a>8`, it fails to accurately analyze complex but equivalent expression `a>3+5`.

Finding 6 & Implication: Deliberately complicating expressions or statements could be a useful strategy to stress-testing of the rule implementations and manifest the issues caused by *mishandling intermediate representations* and *missing specific cases.*

Java External Libraries. Java external libraries are prepackaged modules in JAR files, offering versatile functions, *e.g.*, the Spring Framework, JUnit, or Google Guava. However, inaccurate or late support for these libraries may lead to inaccurate analysis results. This characteristic is mainly related with the root cause of *unhandled java libraries* (Section 3.3.2).

Variable Initialization and Assignments. Variable initializations and assignments may induce FPs or FNs. This characteristic is mainly related with the root causes of *missing specific cases* (Section 3.4.3) and *mishandling intermediate representations* (Section 3.5). For example, direct initialization (variables are assigned during initialization, *e.g.*, `int a = 1;`), delayed assignment (variables are assigned after declaration, *e.g.*, `int a; ...; a = 1;`), assignment within expressions (the assignments are within other expressions, *e.g.*, `if(condition = var == 3)`), nontrivial assignments (assignments involving other variables instead of only literals, *e.g.*, `a = b`), and uninitialized variables can induce FPs or FNs. Figure 5f shows a FP of PMD's rule *UnusedAssignment.* This rule warns unused assignments and variables. The variable `start` is declared (line 2) and used (lines 3 and 4). However, the rule fails to handle the delayed variable assignment and incorrectly warns `start` is not used (line 2).

Lambda Expressions. Lambda expression was introduced in Java 8. Failing to handle lambda expressions may induce issues. This characteristic is mainly related with the root causes of *unhandled language features* (Section 3.3.1). For example, Figure 5g illustrates a FP of PMD's rule *UseDiamondOperator* induced by a lambda expression. This rule avoids the duplicate declarations of type parameters in the diamond operator. At line 2, the variable `foo` does not get explicitly typed, so the type declaration `string` in the diamond operator is not duplicated. However, this rule gives a FP only inside the lambda expression.

Modifiers. Java modifiers (*e.g.*, `public`, `private`, `protected`, `static` and `final`) control the accessibility of classes, constructors, fields or attributes. Analyzers may fail to deal with these modifiers. Figure 5h shows a FN of SpotBugs rule *ES_COMPARING_STRINGS_WITH_EQ* triggered by the `final` modifier. This rule warns when the strings are compared with ==. In this case, if `s2` is annotated with `final`, no warning is reported at line 5, although there is a comparison with ==, which is a FN. Deleting the `final` modifier makes the rule work correctly. This characteristic is mainly related with the root causes of *unhandled language features* (Section 3.3.1).

Others. We find that a few minor characteristics (*e.g.*, `extend`, `enum`, anonymous classes) that may also induce the issues of FNs/FPs.

5 IMPLICATIONS AND DISCUSSIONS

This section discusses the implications distilled from the findings of **RQ1** and **RQ2** to shed light on what the developers and researchers could do to tackle FNs/FPs. We will also discuss other aspects of our work.

Avoid issues caused by common root causes or input characteristics. Our study finds that *unhandled language features* is one of the most common root causes for *all* the three studied analyzers (**RQ1**'s Finding 2). Thus, developers should *timely* check the rule specifications when some new Java language features are introduced, and update the rule implementations if necessary. According to the statistics in Table 2, timely supporting new language features could avoid 21.1% of the issues. *Missing cases* is another common root cause (**RQ1**'s Finding 3). To counter this, given a rule, the analyzers' developers should consider different input characteristics (summarized **RQ2**) when designing its test programs. The code under check may be written in different specific ways. In this regard, researchers could devise automated testing techniques to generate diverse test programs. To show the feasibility, in Section 6.1, we designed such a proof-of-concept testing strategy to generate new test programs by mutating existing ones. On the other hand, *annotations*, *method calls and field accesses*, and *Java standard libraries* are the major input characteristics affecting all the three studied analyzers (see **RQ2**). Thus, developers should pay more attention to model the semantics of annotations and Java standard libraries when implementing the rules. According to the statistics in Figure 4, properly modeling *annotations* could avoid 21% of the issues.

Improving the underlying static analysis modules. Static code analyzers usually adopt some form of static analysis, *e.g.*, AST-based syntactic pattern matching, data-flow analysis and symbolic execution. We find that *improving the abilities of static analysis modules* in general is important for mitigating FNs/FPs. For example, PMD is mainly affected by *type resolution error or limitation* and *scope analysis error or limitation* (**RQ1**'s Finding 4). As a result, PMD may incur FNs/FPs when handling *same identifiers* (revealed by **RQ2**) which requires proper type resolution or scope analysis. Indeed, we note that PMD has recently significantly rewritten its type resolution for the new version v7.0. SonarQube may incur FNs/FPs due to the errors or limitations of its symbolic execution engine. The developers of SonarQube are also continuing the improvement of the engine.

However, we observe that in some cases developers may not immediately improve the static analysis modules, although they would confirm the reported FNs/FPs. This is because they need to trade off many factors like the investment cost, the development plans, the tool performance and the analysis precision (or soundness) after the improvement. For example, SonarQube's Issue #90849 [2] is an FN caused by the limitation of the symbolic execution engine. The developers confirmed this FN but do not plan to fix it immediately considering too much investment is required. The developers disclosed that the current engine is purposely designed in this way to avoid raising FPs at the cost of inducing FNs. But the developers commented 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 (this new engine is already running on SonarCloud and some versions of SonarQube)*". For another example, PMD's developers confirmed several issues like #4127 [1] (which is a FP) which were caused by the limitations of type resolution. But it took the PMD's developers more than one year to improve the type resolution module and resolve all these issues.

Additionally, we observe that *choosing the appropriate form of static analysis for implementing the rules* is also important. For example, data-flow analysis can provide more precise information than AST-based syntactic pattern matching, thus reducing potential FNs/FPs. In Section 6.2, we will investigate the weaknesses of the static analysis modules in the studied analyzers, and show the consequences and trade-offs of choosing the forms of static analysis.

Following best practices when building static code analyzers. During our study, we observed some best practices to tackle FNs/FPs by inspecting the fixing patches. We explicitly summarize these best practices to inspire (new) developers. (1) *Enforcing modularity when designing rules.* Some rules may have similar analysis procedures. In such scenarios, developers should consider moving these procedures into a common utility class. In this way, fixing the FNs or FPs induced by this utility class could benefit all the relevant rules (no separated fixes are needed anymore). In the PMD's PR #2899, one developer commented "*using a common utility class to share logic between rules, or to store procedures that are not really rule-specific*". PMD's developers commented in its roadmap "*In general, a rule should use TR (type resolution) when it can, and fall back on non-TR approach otherwise. No need for separating rules for TR and non-TR.*" [26]. These comments confirm the importance of modularity design. (2) *Avoiding workaround fixes of FNs or FPs.* The workaround

Table 3. 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 equivalent statements or expressions.	`- j++;` `+ ++j;` (1) `-------------------------------` `- if(false){...}` `+ if(false || false){...}` (2) `-------------------------------` `class foo{` `int a;` `void foo(){` `-         bar(a);` `+         bar(this.a);` `}}` (3)

fixes may introduce some adverse side effects — fixing a FP but introducing new FNs, or visa versa. Thus, developers should always try to fix the true root causes of FNs/FPs, and carefully evaluate the effect of their patches. During fixing, avoiding introducing FPs is also important because the number of reported FPs could affect the usability of the analyzers [67]. For example, SonarQube's developers commented that "*SonarQube is not warning against any good practice, rather the goal of each rule is to favor good practices while keeping the false positive rates as low as possible (ideally zero)*" [38]. Researchers could devise effective techniques to help developers find or avoid FPs.

6 RQ3: PROOF-OF-CONCEPT DEMONSTRATION OF OUR FINDINGS

This section demonstrates how our findings can help identify issues in analyzers and reveal the weaknesses of static analysis modules.

6.1 Finding the Analyzers' FNs/FPs by Automatically Generating 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 equivalent input programs from existing ones to stress test the analyzers.

Equivalent Input Program Mutation. One insight obtained from our study is that we can perform equivalent input program mutations to help find FNs or FPs. Specifically, given an original input program of an analyzer, we can generate some equivalent program variants based on some program mutation operators. Here, equivalent program variants means that these variants are expected to have the same analysis results with the original program when checked by the analyzer. If the analysis results (*e.g.*, the reported warnings) are different between the original program and its variants, some FNs or FPs are likely found. Based on the findings of **RQ2**, we selected four kinds of input characteristics to design the program mutation operators (see Table 3). Specifically, based on the findings of **RQ1**, the mutation operators (a) and (b) target the root cause of *unhandled language features*, (c) targets the root cause of *type resolution and scope analysis error or limitation*, and (d) targets the root cause of *missing specific cases.*

(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.
(d) *Replacing with equivalent statements or expressions.* We support three forms of 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`.

Implementation. We use JavaParser [19] to parse the input programs into abstract syntax trees (ASTs), manipulate the tree nodes according to the mutation operators and generate 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 three studied analyzers at the time of our study (PMD v7.0.0-rc3, Spotbugs v4.7.4 and SonarQube v10.0.0.68432). We collected all the test programs released by these tools as the original input programs: 3,754 programs from PMD, 572 programs from SonarQube, and 1,227 programs from Spotbugs. 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, and 1 hour for Spotbugs.

Results and Analysis. Table 4 gives the issues of FNs and FPs (with Issue IDs from 1~14) found by our testing strategy. We found 12 FNs and 2 FPs from the three static code analyzers. We reported all these issues to the developers. Up to now, 11 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 12 different rules from PMD and SpotBugs. We note that most issues were found by the mutation operator (b), which found 6 issues, and (d) (the three forms of equivalent expression) found 1, 1 and 5 issues, respectively. The mutation (c) did not find new issues in the analyzers. The reason may be that the latest version of PMD v7.0 significantly improved its type resolution and scope analysis modules. Thus, PMD may avoid many potential type resolution issues, while the other two analyzers are robust.

To our knowledge, Wang *et al.* [75] 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 two different analyzers to find issues. Different from their work, our testing strategy is one form of metamorphic testing [40] which does not require a reference analyzer. Indeed, 10 issues found by us which cannot be found by [75]. Because the related 8 buggy rules in our experiment do not have similar rules in other analyzers. For example, for the rule *"UseIOStreamsWithApacheCommonsFileItem"* in PMD, no similar rule in SonarQube exists. Thus, the issue (with ID: 6 in Table 4) could be missed by differential testing. Thus, our proof-of-concept testing strategy

Table 4. 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	Sonarqube	—	FN	Java:S2589	Fixed
16	Sonarqube	—	Spec.	Java:S2384	Fixed
17	PMD	—	FP	CloseResource	Confirmed
18	Sonarqube	—	FP	Java:S2384	Confirmed
19	Sonarqube	—	FN	Java:S2259	Confirmed

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, the weaknesses of the static analysis modules in the analyzers may lead to FNs/FPs (see **RQ1**'s Finding 4). 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 buggy rules from PMD, SpotBugs and SonarQube for careful inspection. Our insight is that investigating such "buggy" rules are more likely to reveal the weaknesses of static analysis modules. Table 5 lists these selected rules from the three analyzers. In the column of "Rule Name (#Fixed Issues)", the number in the parenthesis following the rule name is the number of historical fixed issues. The column "Reference" gives the rule specification.

During our investigation, for each rule in Table 5, 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**: (i) the common input characteristics leading to FNs/FPs. From the top ten input characteristics triggering FPs/FNs identified in Section 4 (also see Fig. 5), 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 analyzers. We did not select the other input characteristics from the top ten because some input characteristics (*e.g.*, *nested classes*, *same identifiers*) are suitable for automatically

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

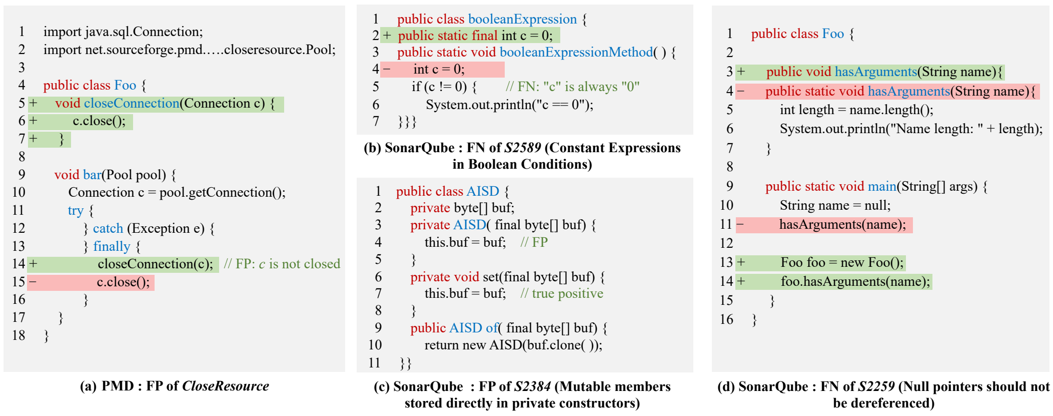
Tool	Rule Name (#Fixed Issues)	Reference
PMD	ImmutableField (12)	[22]
	UnnecessaryFullyQualifiedName (11)	[23]
	CloseResource (11)	[21]
	UnusedPrivateField (8)	[25]
	UnusedImports (8)	[24]
SpotBugs	DMI_RANDOM_USED_ONLY_ONCE (2)	[7]
	NP_NONNULL_PARAM_VIOLATION (2)	[8]
	RV_RETURN_VALUE_IGNORED (2)	[10]
	RCN_REDUNDANT_NULLCHECK_ OF_NONNULL_VALUE (2)	[9]
	UPM_UNCALLED_PRIVATE_METHOD (2)	[11]
SonarQube	S2589: Boolean expressions should not be gratuitous (3)	[30]
	S3749: Members of Spring components should be injected (3)	[32]
	S2384: Mutable collection or array members should not be stored or returned directly (3)	[29]
	S2259: Null pointers should not be dereferenced (3)	[27]
	S2695: "PreparedStatement" and "ResultSet" methods should be called with valid indices (2)	[31]

generating 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. (ii) The correlations between the root causes and the input characteristics. For example, mutating *method calls and field accesses* may affect or require *type resolution* and *control/data-flow analysis*, and mutating *variable initialization and assignments* and *complex expressions or statements* may affect or require *scope analysis*. Note that these code mutations do not guarantee the mutated program is equivalent to the rule's original test program (like what we did in Section 6.1).

Table 4 shows the five issues (with Issue IDs from 15~19) 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. We illustrate these issues below.

Fail to handle basic method call flows. Figure 6a 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 [13]).

Fail to analyze variable scopes. Figure 6b shows a FN of SonarQube's rule *S2589*. This rule enforces that boolean expressions should not be gratuitous (if a boolean expression does not change the evaluation of the condition, it is redundant and should be removed). In this example, we moved



```
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 booleanExpression {
2 +   public static final int c = 0;
3     public static void booleanExpressionMethod( ) {
4 −     int c = 0;
5       if (c != 0) {       // FN: "c" is always "0"
6         System.out.println("c == 0");
7     }}}
```

(b) SonarQube : FN of *S2589* (Constant Expressions in Boolean Conditions)

```
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   }}
```

(c) 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  }
```

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

Fig. 6. Simplified code examples illustrating the weaknesses of static analysis modules in the studied analyzers.

the declaration of the local variable `c` (line 4) outside of the method `booleanExpressionMethod` and declared `c` as a static final class field (line 2). However, SonarQube failed to report the rule violation. We reported this issue to SonarQube. The developer confirmed that this is a valid FN and commented that "*Indeed our engine is failing to evaluate constants outside the method's scope*". This issue was fixed [4]. This case shows SonarQube's weakness in analyzing variable scope.

Fail to choose the appropriate form of static analysis. Figure 6c 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 4) which has been fixed [3].

Fail to track runtime types in symbolic execution. Figure 6d shows a FN of SonarQube's rule *S2259*. This rule warns about null pointer dereference. For the original input program, this rule can correctly warn 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 support 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.

7 THREATS TO VALIDITY

Internal Validity. Our study requires manual analysis and the human expertise of static code analyzers to answer RQ1 and RQ2. Thus, we may introduce some threats to the categories of root causes and input characteristics and their percentages. To counter this, in Sections 3 and 4, two of the co-authors independently inspected the issues, cross-checked their results, and discussed them with the other co-authors to reach a consensus. They tried their best to reduce potential threats.

We studied 350 issues which include 270 FPs and 80 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 disparity might be difficult to correct and it does not mean that FPs are more important than FNs. Because (1) the users are more likely to report FPs than FNs, and (2) many FNs are not reported because no checking rules exist [46, 52, 54, 71] (such FNs may not be considered valid issues of the analyzers).

External Validity. We collected and investigated 350 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 the generability of our findings can be justified by the following reason. We have collected *all* the historical, fixed issues of FNs and FPs from the three representative PMD, SpotBugs and SonarQube 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). Compared to the prior relevant work, our study investigates the largest number of issues of FNs and FPs and their patches. For example, Thung *et al.* [71] and Habib *et al.* [46] only investigate 19 and 20 FNs respectively, Wang *et al.* [75] only investigate 46 issues (38 FNs and 8 FPs) and Zhang *et al.* [80] only investigate 79 issues of FNs and FPs. We have looked into these FNs/FPs which were analyzed by these 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 deciding the 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 [75]. 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 by our findings (which we will discuss in detail in Section 8), and thus our findings should be general.

In addition, PMD has more historical issues than the other two analyzers. It may affect the generalizability of our findings. But we find the root causes of PMD's issues are similar to those of SpotBugs's and SonarQube's issues. Thus, we believe the distilled categories of root causes should be general. Our study only considers three static code analyzers. So our conclusions may not generalize beyond these studied analyzers. However, these three 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 [63]). 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.

8 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 [41, 44, 46, 51–54, 59, 71–73, 77]. All these studies reveal that static code analyzers suffer from FNs. To analyze the reasons of FNs, for example, Thung *et al.* [71, 72] and Habib *et al.* [46] 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 [52, 54] — 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 [48, 51, 66, 77], thus undermining the usability [56, 63, 74]. 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 [45] 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.* [75] and Zhang *et al.* [80] 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 350 developer-confirmed *and* -fixed FNs/FPs. Wang *et al.* [75] and Zhang *et al.* [80] 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 [75]) summarized by Wang *et al.* and 5 "root causes" (see Section 5.2 in [80]) 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 [75]), 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 24.3%(≈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 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 10 out of the 14 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 10 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. It would be interesting in the future to further compare the differential testing strategy and our metamorphic testing strategy in a more systematic way. But in general it is difficult to decide which testing strategy is more effective in finding FNs/FPs because different factors like the diversity of input programs and the number/strength of identified metamorphic relations may affect the results. 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 3, (d)-(2)) used by our testing strategy is overlapped with those of Zhang *et al.* (see Table 2 in [80]). The remaining five mutation operators (see Table 3, (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 4) out of 14 FNs/FPs found by us. Last but not least, our work identifies additional four FNs/FPs caused by four 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.

In the literature, there are also other works 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 [42], alias analysis [78], data-flow analysis [70], the configurations of static analysis [61, 62], and annotation-introduced faults [81].

There are some work validating the correctness of more sophisticated program analyzers like abstract interpreters [60], symbolic executors [49], model checkers [79] and compilers [50]. But these work does not target the analyzers we studied.

9 CONCLUSION

We present the first systematic study on 350 historical issues of FNs/FPs from three representative static code analyzers. 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.

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] 2023. *SONARJAVA-4475 Add visitIdentifier to DivisionByZeroCheck.PostStatementVisitor to evaluate constants outside method's scope.* https://github.com/SonarSource/sonar-java/pull/4382 Accessed: 2024-01-30.
[5] 2024. *AbstractClassWithoutAbstractMethod - PMD Java Best Practices.* https://docs.pmd-code.org/latest/pmd_rules_java_bestpractices.html#abstractclasswithoutabstractmethod Accessed: 2024-01-30.
[6] 2024. *Bug Descriptions - SpotBugs Documentation.* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html Accessed: 2024-01-30.

[7] 2024. *Bug Descriptions: DMI: Random object created and used only once (DMI_RANDOM_USED_ONLY_ONCE).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#dmi-random-object-created-and-used-only-once-dmi-random-used-only-once Accessed: 2024-01-30.

[8] 2024. *Bug Descriptions: NP: 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.

[9] 2024. *Bug Descriptions: RCN: Redundant nullcheck of value known to be non-null (RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#rcn-redundant-nullcheck-of-value-known-to-be-non-null-rcn-redundant-nullcheck-of-nonnull-value Accessed: 2024-01-30.

[10] 2024. *Bug Descriptions: RV: Method ignores return value (RV_RETURN_VALUE_IGNORED).* https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#rv-method-ignores-return-value-rv-return-value-ignored Accessed: 2024-01-30.

[11] 2024. *Bug Descriptions: UPM: 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.

[12] 2024. *Checkstyle.* https://github.com/checkstyle/checkstyle Accessed: 2024-01-30.

[13] 2024. *DataflowPass | PMD.* https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/internal/DataflowPass.java

[14] 2024. *DoNotUseThreads - PMD Rules for Java Multithreading.* https://docs.pmd-code.org/latest/pmd_rules_java_multithreading.html#donotusethreads Accessed: 2024-01-30.

[15] 2024. *Error Prone.* https://github.com/google/error-prone Accessed: 2024-01-30.

[16] 2024. *Infer.* https://github.com/facebook/infer Accessed: 2024-01-30.

[17] 2024. *Java Rules - PMD Source Code Analyzer.* https://docs.pmd-code.org/latest/pmd_rules_java.html Accessed: 2024-01-30.

[18] 2024. *Java Rules - SonarSource Static Code Analysis.* https://rules.sonarsource.com/java/ Accessed: 2024-01-30.

[19] 2024. *JavaParser.* https://github.com/javaparser/javaparser

[20] 2024. *PMD.* https://pmd.github.io/

[21] 2024. *PMD Rules: CloseResource.* https://docs.pmd-code.org/latest/pmd_rules_java_errorprone.html#closeresource Accessed: 2024-01-30.

[22] 2024. *PMD Rules: ImmutableField.* https://docs.pmd-code.org/latest/pmd_rules_java_design.html#immutablefield Accessed: 2024-01-30.

[23] 2024. *PMD Rules: UnnecessaryFullyQualifiedName.* https://docs.pmd-code.org/latest/pmd_rules_java_codestyle.html#unnecessaryfullyqualifiedname Accessed: 2024-01-30.

[24] 2024. *PMD Rules: UnusedImports.* https://pmd.sourceforge.io/pmd-6.55.0/pmd_rules_java_bestpractices.html#unusedimports Accessed: 2024-01-30.

[25] 2024. *PMD Rules: UnusedPrivateField.* https://docs.pmd-code.org/latest/pmd_rules_java_bestpractices.html#unusedprivatefield Accessed: 2024-01-30.

[26] 2024. *Roadmap | PMD Source Code Analyzer.* https://docs.pmd-code.org/latest/pmd_devdocs_roadmap.html

[27] 2024. *RSPEC-2259.* https://sonarsource.atlassian.net/browse/RSPEC-2259 Accessed: 2024-01-30.

[28] 2024. *RSPEC-2259: Null pointers should not be dereferenced - SonarSource Java Rules.* https://rules.sonarsource.com/java/tag/symbolic-execution/rspec-2259/ Accessed: 2024-01-30.

[29] 2024. *RSPEC-2384.* https://sonarsource.atlassian.net/browse/RSPEC-2384 Accessed: 2024-01-30.

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

[31] 2024. *RSPEC-2695.* https://sonarsource.atlassian.net/browse/RSPEC-2695 Accessed: 2024-01-30.

[32] 2024. *RSPEC-3749.* https://sonarsource.atlassian.net/browse/RSPEC-3749 Accessed: 2024-01-30.

[33] 2024. *Sonar Community.* https://community.sonarsource.com/

[34] 2024. *SpotBugs.* https://spotbugs.github.io/

[35] 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

[36] Arie Ben-David. 2008. Comparison of classification accuracy using Cohen's Weighted Kappa. *Expert Systems with Applications* 34, 2 (2008), 825–832.

[37] 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

[38] Buono, Angelo. 2023. *Java S2384 False negative when the mutable member is not private.* Retrieved 2023-05 from https://community.sonarsource.com/t/java-s2384-false-negative-when-the-mutable-member-is-not-private/90349/6.

[39] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action.*

[40] 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.

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

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

[51] 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

[52] 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

[53] 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

[54] 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

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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

[66] 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

[67] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[68] 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

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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

[75] 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

[76] 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

[77] 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

[78] 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

[79] 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

[80] 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

[81] 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

[82] 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