Understanding and Detecting Annotation-Induced Faults of Static Analyzers

HUAIEN ZHANG, Hong Kong Polytechnic University, China and Southern University of Science and Technology, China
YU PEI, Hong Kong Polytechnic University, China
SHUYUN LIANG, Southern University of Science and Technology, China
SHIN HWEI TAN, Concordia University, Canada

Static analyzers can reason about the properties and behaviors of programs and detect various issues without executing them. Hence, they should extract the necessary information to understand the analyzed program well. Annotation has been a widely used feature for different purposes in Java since the introduction of Java 5. Annotations can change program structures and convey semantics information without awareness of static analyzers, consequently leading to imprecise analysis results. This paper presents the first comprehensive study of annotation-induced faults (AIF) by analyzing 246 issues in six open-source and popular static analyzers (i.e., PMD, SpotBugs, CheckStyle, Infer, SonarQube, and Soot). We analyzed the issues' root causes, symptoms, and fix strategies and derived ten findings and some practical guidelines for detecting and repairing annotation-induced faults. Moreover, we developed an automated testing framework called AnnaTester based on three metamorphic relations originating from the findings. AnnaTester generated new tests based on the official test suites of static analyzers and unveiled 43 new faults, 20 of which have been fixed. The results confirm the value of our study and its findings.

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

Additional Key Words and Phrases: static analyzer, annotation, software testing, empirical study

ACM Reference Format:
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 (July 2024), 23 pages. https://doi.org/10.1145/3643759

1 INTRODUCTION

Static analyzers are widely used to reason about programs and detect various issues without dynamically executing them. Since they do not need to actually run the programs under consideration, static analyzers are highly applicable in a wide range of situations and have been utilized to reason about the properties and behaviors of programs in various development stages [17]. Meanwhile, since the analyses are based on the information they extract from program code, the static analyzers must have a good understanding of the syntax, semantics, and interplay of various constructs in the programs for the analysis results to faithfully reflect the real issues. Since it is highly challenging to correctly handle all the program constructs in a static analyzer, various techniques and tools

Authors' addresses: Huaien Zhang, Hong Kong Polytechnic University, Hong Kong, China and Southern University of Science and Technology, Shenzhen, China, cshezhang@comp.polyu.edu.hk; Yu Pei, Hong Kong Polytechnic University, Hong Kong, China, yupei@polyu.edu.hk; Shuyun Liang, Southern University of Science and Technology, Shenzhen, China, liangsy2022@mail.sustech.edu.cn; Shin Hwei Tan, Concordia University, Montreal, Canada, shinhwei.tan@concordia.ca.

ACM 2994-970X/2024/7-ART33
https://doi.org/10.1145/3643759

have been developed in the past few years to detect bugs induced by incorrect handling of program constructs with predefined semantics in static analyzers [15, 34, 73, 81]. However, many modern programming languages (e.g., Java, Python, and C#) provide native support for annotations with programmer-defined semantics, which presents extra challenges to the reliability of static analyzers.

In computer programming, annotations are a form of syntactic metadata that associates additional information to various program elements. Annotations have been utilized both as a more structured way to comment on code elements and as a mechanism to support meta-programming [16]: In the former case, they do not affect the semantics of programs; In the latter case, they usually trigger special processing of the annotated elements during program compilation and/or execution, effectively extending the capabilities of the programming languages. For instance, popular frameworks like Spring, JUnit, and Lombok rely heavily on their homebrewed annotations to simplify reusing the frameworks. Nowadays, annotations have been widely utilized in practical software development. According to a previous study conducted on a large number of open-source projects hosted on GitHub, the median number of annotations per project is up to 1,707 [82].

In general, the presence of annotations poses two challenges to the reliability of static analyzers. First, annotations in programs lead to extra tokens that static analyzers need to parse, while an unprepared static analyzer may overlook or mishandle the tokens, leading to incorrect analysis results or even premature termination of the tool. For example, given the simple Java class in Figure 1 as input, the PMD static analyzer will crash because the tool is not expecting the array access operators in line 5 to be annotated. Second, annotations in programs may introduce changes to the structure or behavior of the programs at compile or execution time. Since the detailed changes are defined by annotation processors, which are programs themselves, it is impractical to fully understand the impact of all annotations without running those processors, and correspondingly, it is inevitable that static analyzers produce incorrect results if the annotations interfere with the programs' properties and behaviors being analyzed.

```
1 @Retention(RetentionPolicy.CLASS)
2 @Target({TYPE_USE})
3 @interface Anno {}
4 public class Main {
5   public <T> T[][] check(T @Anno[] @Anno [] arr) { // Trigger a crash
6     if (arr == null) {
7       throw new NullPointerException();
8     }  ...
```

Fig. 1. An annotated Java program that will cause PMD to crash.

To gain a better understanding of the extent to which annotations in programs affect the reliability of static analyzers, we conducted the first large-scale empirical study on annotation-induced faults (AIFs) of static analyzers. Although prior work has studied the usage and evolution of program annotations [43, 53, 66, 82], the maintenance of testing-related annotations [33], and the design of annotations for special purposes [10, 74], there is little to no study on AIFs in static analyzers. This work aims to fill this gap. Particularly, our study aims to answer the research questions below:

- **RQ1:** What *kinds of annotations* are more likely to induce faults? In RQ1, we study annotations that require attention when designing static analyzers.
- **RQ2:** What are the *root causes* for annotation-induced faults in static analyzers? In RQ2, we study the reasons behind annotation-induced faults to prevent them from reoccurring in the future.
- **RQ3:** What are the *symptoms* of those annotation-induced faults? In RQ3, we investigate the consequences of annotation-induced faults, which helps us assess the significance of the faults.

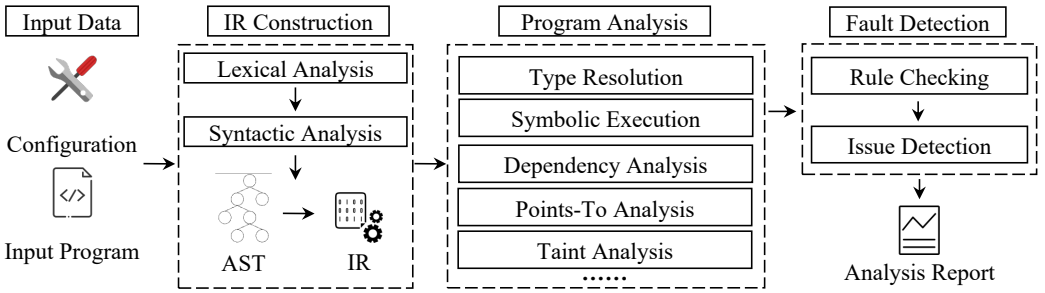

Fig. 2. The general workflow of a static analyzer

- **RQ4:** What are the *fix strategies* that developers employ when fixing the annotation-induced faults? In RQ4, we strive to establish a good understanding of viable ways to fix annotation-induced faults, which is essential for reducing debugging efforts.

To address the research questions, we manually analyzed 246 annotation-induced issues and their corresponding patches from six popular open-source static analyzers, namely PMD, SpotBugs, Infer, CheckStyle, SonarQube, and Soot. As a result, we uncovered six main reasons for the annotation-induced faults, identified four symptoms of those faults, and unveiled seven strategies developers adopted to fix the faults. We made ten major findings from the analysis results and discussed their implications for avoiding similar faults in the future. Based on our findings, we developed a framework named AnnaTester to automatically detect three types of annotation-induced faults in static analyzers via metamorphic testing. On the six aforementioned static analyzers, AnnaTester successfully detected 43 faults that were revealed for the first time. We have reported the faults to the corresponding tool developers, and 20 of them have been fixed at the time of writing, which clearly demonstrates the value of the framework, our study, and the findings.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we conducted the first empirical study on annotation-induced faults in static analyzers based on 246 issues from six popular open-source analyzers. We analyzed their root causes, symptoms, fix strategies, and types of annotations, deriving ten findings.
- Based on the findings from our study, we propose AnnaTester, a new automated testing framework that uses metamorphic testing with our customized annotated program generator to detect three types of annotation-induced faults in static analyzers.
- We evaluated AnnaTester on six static analyzers, and it was able to reveal 43 new bugs in these static analyzers, 20 of which have been confirmed and fixed. The experimental data and source code of AnnaTester are available at: https://annaresearch.github.io/.

2 BACKGROUND

2.1 Static Analyzer

Static analyzers are widely used to detect common issues without running programs. Figure 2 shows the general workflow of static analyzers based on previous work [9, 57, 78]. Using the program source code and a configuration as the input, a static analyzer first parses the program code and constructs an intermediate representation (IR). Then, it applies different program analysis techniques like dependency analysis, symbolic analysis, etc., to extract relevant semantic information from the program. Finally, it employs rule checkers to detect issues based on the extracted information and reports the detected issues as the analysis result.

2.2 Java Annotation

Java annotations offer a structured way to attach helpful information to program elements like classes, methods, variables, and types. An annotation essentially contains a (possibly empty) list of property-value pairs, with the properties specified in the annotation's definition and the values associated with the properties when each annotation declaration is used to annotate a program element. Java annotations have no effect on the program semantics in and of themselves, but a program could implement annotation processors to adjust the program's code or even behaviors based on the presence of specific annotations, effectively extending the capability of the Java programming language. For instance, an annotation *@java.lang.Override* will trigger a compile-time check on the existence of the method it annotates in the current class's superclass, and an error will be issued if the check fails.

Java programs may use *meta-annotations* (i.e., annotations applicable to other annotations) to restrict the application and effect of other annotations. For example, meta-annotation *@Target* is used to specify the types of elements (e.g., Class, Method, and Field) to which an annotation can be applied, while meta-annotation *@Retention* is used to stipulate how long an annotation should be retained during a program's lifecycle: Source level retention means an annotation will not be retained after source code is compiled into bytecode; Class retention means an annotation will be retained in bytecode but discarded when the bytecode is loaded into a JVM; Runtime retention means that an annotation is always retained and can be retrieved at runtime.

3 EMPIRICAL STUDY ON ANNOTATION-INDUCED FAULTS

3.1 Target Static Analyzers

We select target analyzers based on three criteria: (1) it must be open-source and use a public issue tracking system (GitHub or Jira) to record all its issues that have been reported and resolved so that we can identify and analyze its AIFs and corresponding fixes; (2) it should be popular and widely used so that its issues are representative of the real problems faced by users of analyzers. Particularly, we focus on analyzers with at least 2,000 stars on GitHub in this study; (3) it should support analysis of Java programs. Based on these criteria, we select six static analyzers: (1) PMD [61] is a cross-language static analyzer that detects common code smells, e.g., unused variables; (2) SpotBugs [70] is a fork of the now deprecated analyzer FindBugs that detects common bugs in Java programs via a set of code patterns; (3) CheckStyle [11] checks the conformity of Java code to a set of coding rules; (4) Infer [26] is an analyzer designed by Meta to detect bugs for Java, C, C++, and Objective-C programs; (5) SonarQube [68] is a continuous code inspection platform that detects bugs and code smells for various programming and markup languages; (6) Soot [69] is a static analysis framework that can analyze, instrument, and optimize Java and Android applications.

3.2 Data Collection

Among the six target analyzers, SonarQube uses Jira for tracking issues, while other tools use GitHub. Since the issue tracking systems of these static analyzers contain around 14,000 issues, we refrained from manually inspecting all the issues to select only issues that are related to annotation-induced faults. Instead, we first used the keyword "annotation" to search for closed issues that are likely annotation-induced. We focus only on closed issues because how an issue was resolved sheds light on its root cause and fixing strategy, and we consider a closed issue relevant to annotations if and only if the keyword "annotation" appears in the issue's title or description. The search returned 308 issues in total. Then, we manually checked these issues and excluded issues that were not associated with any fixing commits or not related to annotations. Subsequently, we have 246 faults to be analyzed in our study. Table 1 lists the total number of issues for each analyzer in its

issue tracking system (#Issue$_t$), the number of likely annotation-induced issues returned by the keyword-based search (#Issue$_s$), and the number of annotation-induced faults confirmed by the manual check and to be analyzed (#Issue$_a$). In the rest of this paper, we refer to faults using their IDs in the form TOOL-###, where TOOL denotes the name of a static analyzer, while ### denotes the corresponding issue ID on GitHub or JIRA. As all issues are confirmed by the tool developers, we did not manually reproduce the collected issues.

Table 1. The issue distribution among six static analyzers

Static Analyzer	#Issue$_t$	#Issue$_s$	#Issue$_a$
SonarQube	4370	138	128
CheckStyle	4768	60	52
PMD	2161	53	43
SpotBugs	1043	10	7
Infer	1304	8	6
Soot	1147	39	10
Total	14793	308	246

3.3 Data Labeling and Reliability Analysis

In this study, we identified the annotation-induced issues and analyzed them from three different aspects (i.e., the root cause, the symptom it exhibits, and the fix strategy). The entire study took us around six months to complete. To categorize (or label) the issues from each aspect, we followed previous work [67, 84] to adapt existing taxonomies [12, 67, 72, 84, 85] to our task via an open-coding scheme. Specifically, one author first looked through all the issue reports and pull requests of those issues to determine the issue labels in these three aspects, including adding domain-specific categories and eliminating unnecessary categories. Then, two authors independently labeled the collected issues using the previously defined categories. We use Cohen's Kappa coefficient [77] to assess the agreement between these two authors. First, the two authors labeled 5% of the issues, and Cohen's Kappa coefficient was nearly 0.69. Then, we had a training discussion and labeled 10% of the issues (including the previous 5%). At this stage, Cohen's Kappa coefficient reached 0.93. After an in-depth discussion on the issues with different labels, the two authors labeled the remaining issues in nine iterations, each covering ten more percent of the issues, and Cohen's Kappa coefficient remained greater than 0.9 in the process. In each iteration, the two authors discussed with the third author if they had any disagreement. Finally, all the issues were labeled consistently.

3.4 RQ1: AIF Prone Annotations

We collected annotations that trigger AIFs in the studied issues (we call them *AIF prone annotations*) and sorted them in descending order of occurrence. Figure 3 shows the top 30 most frequently occurred annotations in the study. The x-axis shows the names of the annotations, and the y-axis presents the number of issues caused by each annotation.

Overall, we observe that annotations that are used to specify the nullability of program elements (i.e., *@Nullable*,*@Nonnull*, *@NonNull*, *@CheckForNull*, *@NonNullApi*, and *@NotNull*) are generally AIF prone annotations. These nullability-related annotations triggered the most issues as static analyzers usually have many rules for checking the nullability of various program elements, and the implied semantics of these annotations may affect the analysis results. Meanwhile, the

annotation *@SuppressWarnings* is often used to disregard specific warnings in static analyzers, e.g., *@SuppressWarnings("WarningName")*. It caused the second most issues presumably because many static analyzers (e.g., PMD and SonarQube) support this annotation, and programmers often use this annotation for filtering out unwanted warnings. Several test-related annotations (*@Test*,*@ExtendWith* and *@VisibleForTesting*) are AIF prone annotations due to their wide usages for marking tests. Annotations *@Inject* and *@Autowired* support the automated injection of data dependence on annotated variables [2]. Understandably, static analyzers may produce incorrect analysis results if they are unaware of the implicit data flow introduced by these annotations. In Figure 3, around 23% (i.e., 7/30) of the annotations (e.g., *@Value*, *@Data* and *@Getter*) introduce changes to the original code, and failure to capture such changes may lead to bugs in static analyzers.

Finding 1: Annotations that (1) specify the nullability of program elements, (2) are widely used (for marking unit tests or suppressing undesirable warnings), and (3) alter the dependence or structure of the original code have induced the largest number of faults in static analyzers.

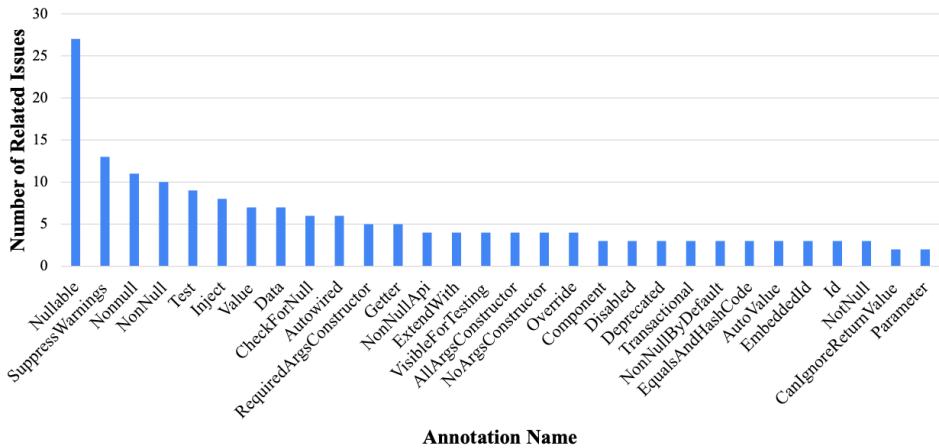

Fig. 3. The number of issues induced by each annotation from the top 30 most AIF-prone annotations.

3.5 RQ2: Root Causes

We uncovered a total of six main reasons for the annotation-induced faults. In this section, we use examples to explain the reasons in decreasing order of their total occurrences in our study. Table 2 shows the number of faults caused by each reason for each static analyzer.

3.5.1 Incomplete Semantics (IS). The most (38%) common reason for AIFs is that static analyzers usually only have incomplete knowledge about the annotations' semantics and, therefore, the semantics of the annotated programs. Understandably, if an analyzer only has access to partial information it relies on, it is bound to produce inaccurate results. Notably, no fault in CheckStyle nor in Soot is related to this root cause because both tools largely ignore the semantics of the input program (CheckStyle checks for coding styles, whereas Soot provides APIs for different analyses).

One fault induced by this root cause is SONARQUBE-3804 [51], shown in Figure 4. SonarQube has a rule stipulating that the keyword `volatile` should not be applied to non-primitive fields since when applied to a reference, the keyword makes sure that the reference itself, rather than the object it refers to, is never cached, which may cause obsolete object data to be cached and used by

Table 2. The numbers of faults due to different root causes in each analyzer.

Static Analyzer	IS	IAT	UEA	ETO	IAG	MCF	Others	**Total**
SonarQube	66	23	21	11	4	2	1	128
CheckStyle	0	39	1	0	10	1	1	52
PMD	19	9	0	4	7	4	0	43
SpotBugs	4	0	0	3	0	0	0	7
Infer	4	0	0	1	0	1	0	6
Soot	0	3	2	4	0	1	0	10
Total	93	74	24	23	21	9	2	246

IS: Incomplete Semantics; **IAT**: Improper AST Traversal; **UEA**: Unrecognized Equivalent Annotations; **ETO**: Erroneous Type Operation; **IAG**: Incorrect AST Generation; **MCF**: Misprocessing of Configuration File.

some program threads. The stipulation, however, should be disregarded when the reference type's class is annotated with *@Immutable* or *@ThreadSafe*[1] since both annotations imply that the class's objects can be safely operated in multi-thread environments. Being unaware of the semantics of the annotations, SonarQube reported violations at lines 4 and 5 of the program in Figure 4.

```
1 @javax.annotation.concurrent.Immutable  class MyImmutable {}
2 @javax.annotation.concurrent.ThreadSafe  class MyThreadSafe {}
3 class Main {
4     private volatile MyImmutable x;
5     private volatile MyThreadSafe y;
6 }
```

Fig. 4. An incomplete semantics example in SONARQUBE-3804.

Finding 2: Incomplete semantics is the most common root cause for faults in all studied analyzers, except for CheckStyle and Soot. As annotations may introduce changes to the program properties and behaviors, failing to grasp the semantics encoded by the annotations will cause static analyzers to produce inaccurate results.

3.5.2 Improper AST Traversal (IAT). After the developers of static analyzers have correctly constructed the abstract syntax tree (AST) for a program under syntactic analysis, they tend to misunderstand the impact of annotations on the ASTs and perform improper AST traversal. In CHECKSTYLE-7522 [48], the analyzer may encounter an ANNOTATION_MEMBER_VALUE_PAIR node, i.e., a type of AST nodes used to represent the key-value pairs in annotation declarations like *@Deprecated(removal=true)*, when it is not expecting one, which can cause a runtime crash. In CHECKSTYLE-9941 [64], an annotation for a method will push the nodes for the method's header comment one level down in the corresponding AST. Therefore, the static analyzer needs to access those nodes accordingly depending on the presence of annotations. Failing to do that, CheckStyle produced incorrect results when analyzing annotated methods.

Finding 3: Developers of static analyzers tend to misunderstand the impact of annotations on ASTs, causing improper AST traversal to be the second most common root cause of AIFs.

[1]Both annotations are defined in the package javax.annotation.concurrent.

3.5.3 Unrecognized Equivalent Annotations (UEA). There are many annotations with equivalent semantics, but developers of static analyzers often fail to recognize these annotations. There are two types of equivalent annotations: (1) Annotations from different libraries can have identical semantics and usage styles, but static analyzers often only recognize some of them, leading to inconsistent analysis reports. For example, the annotation *@Nullable* means that the annotated element can hold a null value, and it has been supported in many popular third-party libraries (e.g., Google Android support, MongoDB, and Spring). Rules that check for null pointers need to analyze the program element annotated with *@Nullable* to determine the nullability of the element. Moreover, with the dormition of JSR-305 [62] (Java Specification Requests that aim to develop standard annotations for Java programs to assist software defect detection tools), several new libraries (e.g., Google JSR-305 [55]) have been proposed to implement annotations in JSR-305. The annotations in these new libraries may cause UEA issues as they all comply with the same specification but have different names. For example, while SonarQube disallows variables of primitive data types to be declared as nullable in general, it reports a warning when *@android.support.annotation.Nullable* is used to mark a boolean value as nullable but fails to do so when an equivalent annotation (i.e., *@android.annotation.Nullable*) is used in the same way (lines 1 and 2 in Figure 5). (2) As a library evolves, the fully qualified names of annotations defined in the library may also evolve. For example, in SONARQUBE-3174 [27], the fully qualified package name of annotation *@Generated* changed from *javax.annotation* to *javax.annotation.processing* but SonarQube's developers were unaware of the change and failed to handle the renamed annotation correctly, causing inaccurate analysis results.

```
1 @android.support.annotation.Nullable boolean fun2() {} // report a warning
2 @android.annotation.Nullable boolean fun1() {}         // no warnings, an FN
```

Fig. 5. SONARQUBE-3536 [28]: A false negative caused by UEA

Finding 4: Most (88%) of the UEA faults were found in SonarQube. Equivalent annotations may come from (1) different libraries or (2) different versions of the same library.

3.5.4 Erroneous Type Operations (ETO). Several annotation-induced faults were due to erroneous type-related operations (e.g., missing type resolution, incorrect type casting/type checking). In SONARQUBE-3438 [29], the SonarQube developers mistakenly believed that the values stored in annotations can only be literals and incorrectly cast the AST for an annotation from ExpressionTree to LiteralTree, leading to a runtime exception. Figure 6 shows that in SONARQUBE-3045 [23], developers forgot to resolve the type of the annotation (i.e., *@MyAnnotation*) applied on the actual type parameter (i.e., *MyClass*), leaving the type parameter being annotated with an unknown type.

```
1 @Target(ElementType.TYPE_USE) @interface MyAnnotation {}
2 List<@MyAnnotation MyClass> field; // Unknown annotation type
```

Fig. 6. SONARQUBE-3045 [23]: An incorrect type resolution in SonarQube

3.5.5 Incorrect AST Generation (IAG). Static analyzers may construct incorrect ASTs at the end of the IR construction stage (Figure 2). One main reason for such problems is the grammar implemented by the static analyzers becomes obsolete after incorporating new rules about the usage of annotations into the specifications of new Java versions. For example, the obsolete grammar prevented CheckStyle from correctly parsing the annotations applied to the compact constructors

of record types in CHECKSTYLE-8734 [52]. It also causes SonarQube and CheckStyle to incorrectly handle type annotations introduced by JSR-308 [31] in SONARQUBE-1420 [59] and CHECKSTYLE-3238 [63], respectively. Meanwhile, some static analyzers may make mistakes in constructing ASTs from the tokens returned by the lexers. For example, in SONARQUBE-1167 [19], although SonarQube correctly extracted the annotations placed on type parameter declarations, it failed to store the information correctly in the corresponding AST, leading to an incorrect AST.

Finding 5: Incorrect AST generation is a common root cause, and most (85.7%) of the IAG faults were due to the obsolete grammar that static analyzers implement.

3.5.6 Misprocessing of Configuration File (MCF). As shown in Figure 2, a static analyzer usually expects as the input both the program files to be analyzed and a configuration file that specifies rules to be enabled and/or disabled, locations of the auxiliary libraries, etc. Several faults occurred because static analyzers failed to process the given configuration files correctly. For example, PMD reads from the configuration file a list of annotations to be ignored in its analysis. In PMD-2454 [46], developers forgot to trim the leading and trailing whitespaces when extracting annotation names from the configuration files, causing the failure to match "*@PreDestroy*␣" with "*@PreDestroy*".

3.5.7 Others. Two faults were highly specific to their corresponding tool implementations and cannot be attributed to any of the aforementioned reasons. In SONARQUBE-3108 [24], SonarQube crashes with an OutOfMemory exception when analyzing a method with 24 parameters, all annotated with *@Nullable*. The reason is that SonarQube creates two symbolic starting states ("NULL" and "NOT_NULL") for each nullable parameter, and it needed to create 2^{24} symbolic states for parameters of the method, which exceeded the available memory in the JVM. In CHECKSTYLE-2202 [36], *@SuppressWarnings* is utilized to suppress warnings specified by the annotation parameters, but developers ignore the parameters named in camel-case notation, leading to a false positive.

3.6 RQ3: Symptoms

In this section, we describe the four symptoms and then relate them to the six root causes to help users and developers assess the impacts of different root causes.

3.6.1 Symptom Category. All symptoms caused by annotation-induced faults are listed below:

False Positive: Symptoms in this category involve analysis reports with undesirable warnings.
False Negative: Symptoms in this category involve analysis reports that are missing warnings.
Crash/Error: Symptoms in this category involve premature terminations or compilation errors.
Other Wrong Results: While most bug reports of the studied faults contain descriptions of the symptoms caused w.r.t. the final results produced by analyzers, some reports only referred to incorrect intermediate results generated during the analyses without explaining how those intermediate results affect the overall analysis outcome. We classify those faults into this category. For instance, the issue report of CHECKSTYLE-8734 [52] explains that CheckStyle cannot parse the annotation on Java records and will construct only a partial AST for the program under analysis.

Table 3 shows the distribution of the four categories of symptoms across the six static analyzers. We observe that most of the studied faults fall into the false positive (FP) category, probably because these faults were discovered during the actual use of the analyzers, and users were more sensitive to undesirable warnings in analysis reports. The prevalence of FPs is also in line with the findings of prior studies on static analyzers [4, 5, 14, 25, 30]. Note that we identify one symptom for each fault in our study based on the bug report, which is reasonable as each bug report usually focuses on only one particular negative impact. Although it may happen in practice that a run of an analyzer exhibits multiple symptoms from different categories, we can reliably make the following finding.

Finding 6: Annotation-induced faults may cause static analyzers to produce inaccurate analysis results, to crash at runtime, or to generate incorrect intermediate results.

Table 3. The number of issues for the four categories of symptoms across the static analyzers.

Static analyzer	FP	CE	FN	OWR	**Overall**
SonarQube	99	7	12	10	128
CheckStyle	26	14	10	2	52
PMD	28	8	7	0	43
SpotBugs	4	0	3	0	7
Infer	5	0	1	0	6
Soot	0	6	0	4	10
Overall	162	35	33	16	246

FP: False Positive, **CE**: Crash/Error, **FN**: False Negative, and **OWR**: Other Wrong Results.

3.6.2 Relationship between Root Causes and Symptoms. So far, we have summarized the root causes and symptoms of annotation-induced faults. Understanding their relationship can help us better comprehend the impact of various root causes on static analysis results. Table 4 shows the relationship between the root causes and symptoms. Although IS was the most common root cause, we observed that it never triggered runtime crashes, which were mostly caused by IAG. IS only led to inaccurate analysis results, especially at the program analysis stage, while crashes often occurred at IR construction stage. We also observe that while FP results can be triggered by all root causes of AIFs, FN and CE results were never caused by incomplete semantics.

Finding 7: All identified root causes in our study led to FPs. Incomplete semantics was the most common root cause and typically led to incorrect analysis results (i.e., FP).

Table 4. Relationship between the root causes and symptoms of annotation-induced faults.

Symptom	IS	IAT	UEA	ETO	IAG	MCF	Others	**Overall**
False Positive	93	40	11	9	4	4	1	162
Crash/Error	0	8	2	8	14	2	1	35
False Negative	0	19	7	4	0	3	0	33
Wrong Intermediate Result	0	7	4	2	3	0	0	16

IS: Incomplete Semantics; **IAT**: Improper AST Traversal; **UEA**: Unrecognized Equivalent Annotations; **ETO**: Erroneous Type Operation; **IAG**: Incorrect AST Generation; **MCF**: Misprocessing of Configuration File.

3.7 RQ4: Fix Strategies

We unveiled seven common fix strategies for fixing annotation-induced faults. In this section, we first introduce each fix strategy and then relate the fix strategies to the root causes of the faults.

3.7.1 Fix Incorrect Use of Annotation Filter (FAF). As there can be many programmer-defined annotations with distinct semantics, a static analyzer often utilizes white and black lists to filter the annotations that it will or will not support. Such a list can be hard-coded into the static analyzer or fed to the static analyzer as part of a configuration file. For example, the *ignoredAnnotations* property in PMD's configuration file is used to specify the annotations to be neglected by specific rule checkers. In general, annotation filtering may suffer from two types of problems. First, a list may miss some annotations or contain undesirable annotations. For instance, in SONARQUBE-1513 [20], a rule checker was used to identify subclasses that should override the *equals* method, but it mistakenly ignored the annotation *@EqualsAndHashCode* in Lombok which, when applied to a class, will cause boilerplate implementations of *equals* and *hashCode* methods to be inserted into the class. To fix this fault, the developers added the annotation to the white list, and SonarQube will default to the class with this annotation having overridden *equals* method. Second, the utilization of the lists may be faulty. e.g., in PMD-2876 [47], a PMD user specified the list of ignored annotations in the configuration file to customize the Lombok annotations to be neglected by the tool, but the customization failed due to PMD's incorrect handling of the list.

Finding 8: Incorrect use of annotation filters was fixed by adjusting the annotation lists in 90.2% cases and correcting the mishandling of annotation lists in the remaining 9.8% cases.

3.7.2 Fix AST Node Retrieval (FAN). The ASTs of programs under analysis are essential information for static analyzers, but the process of information extraction from ASTs may suffer from two types of problems. First, the analyzers may misunderstand the structure of the ASTs, especially when the annotations used in the programs introduce changes to the ASTs. For instance, in CHECKSTYLE-10945 [65], the tool developers mistakenly neglected the *ARRAY_INIT_ARRAY* nodes as part of the annotations in the ASTs. To fix such problems, programmers need to adjust their traversal algorithms based on the actual structure of ASTs. Second, the computation performed by an analyzer when traversing an AST may be faulty. For example, PMD employs a flag variable named *hasLombok* to track in a depth-first AST traversal whether a class has an annotation from the Lombok library, and it will suppress all the *SingularField* warnings on classes where the variable value is true. In PMD-1641 [18], the traversal algorithm forgot to restore the variable's value to false after returning from the visit to an inner class, causing an unwanted *SingularField* warning. The code snippet in Figure 7 shows how the fault was fixed.

```
1 + boolean tmp = hasLombok;
2   hasLombok = hasLombokAnnotation(node);
3   Object result = super.visit(node, data);
4 + hasLombok = tmp;
```

Fig. 7. PMD-1641 [18]: Fix incorrect traversal algorithm

3.7.3 Fix Incorrect Type Operation (FIT). This strategy involves fixing erroneous type-related operations (e.g., type resolution and type casting). For example, in SONARQUBE-2205 [21], the developer mistakenly resolved the type of an annotation based on its simple name, and the fix involves replacing the simple name with the annotation's fully qualified name. In PMD-1369 [79], a runtime crash occurred due to an incorrect cast of a reference from type *ASTAnnotation* to type *ASTClassOrInterfaceType*. To fix this, a type compatibility check was added to guard the type casting.

3.7.4 Fix Grammar Issue (FGI). As shown in Figure 2, static analyzers rely on predefined grammar to perform lexical and syntactic analysis to generate intermediate representative. We classify

grammar-related fix strategies into two subcategories: (1) Fix lookahead parameter. Lookahead is often used in the lexical analysis stage. It can match the specific tokens in the source code to be analyzed. (2) Fix grammar patterns. Static analyzers can define grammar patterns to recognize corresponding syntax structures. However, these patterns may ignore annotations directly, or new usages of annotation, e.g., in CHEKCSTYLE-3238 [63], developers did not define grammar patterns to recognize annotations on variable-length parameters and failed to parse them consequently.

3.7.5 Fix Value Check (FVC). This fix strategy involves adding checks that were missing or rectifying checks that were inappropriate. For example, in CHECKSTYLE-4472 [38], a missing null value check caused a runtime crash, and the fix was to add the missing check.

3.7.6 Redesign Rule Checker Pattern (RRC). Static analyzers based on rule checkers (e.g., all evaluated tools except for Soot) use predefined patterns to detect bugs, but the patterns may be incorrect and need to be redesigned. For instance, Figure 8 shows that in PMD-1782 [1] the rule checker initially only checks if a class or interface has a package definition (ignoring annotation). In line 2, PMD checks whether a package definition exists by counting the number of occurrences of the *PackageDeclaration* node in the XPath (which represents AST as an XML-like DOM structure) but mistakenly omitted the annotation when writing the XPath. To fix this, developers redesign the rule in the XPath to check if a package declaration exists in the initial lines of a compilation unit.

```
1 - /ClassOrInterfaceDeclaration[count(preceding::PackageDeclaration)=0]
2 + CompilationUnit[not(./PackageDeclaration)]/TypeDeclaration[1]
```

Fig. 8. PMD-1782 [1]: Redesign rule pattern to recognize package declaration

3.7.7 Fix Incorrect API Usage (FIA). This fix strategy involves repairing incorrect API usages, mainly by using the correct API to retrieve elements or parse the signature of an annotation. Figure 9 shows that in SOOT-123 [35] when creating an *AnnotationTag* object, Soot incorrectly invoked the API *DexType.toSoot* (line 1) to prepare a type descriptor as the actual parameter for invoking the *AnnotationTag* constructor. Figure 10 shows that in CHECKSTYLE-2202 [36], users adopt the *@SuppressWarnings* annotation to disable a warning, but CheckStyle only recognizes rule names in lower case (line 1) and fails to detect equivalent rule names in camel case (line 2). To fix this, developers use *equalsIgnoreCase* instead of *equals* to recognize all the equivalent rule names.

```
1 - AnnotationTag aTag = new AnnotationTag(DexType.toSoot(a.getType()).toString());
2 + AnnotationTag aTag = new AnnotationTag(a.getType());
```

Fig. 9. SOOT-123 [35]: Incorrectly invoke *toSoot* to construct an *AnnotationTag*

```
1 @SuppressWarnings("checkstyle:redundantmodifier") // No warnings
2 @SuppressWarnings("checkstyle:RedundantModifier") // Report a warning, but it is an FP
```

Fig. 10. CHECKSTYLE-2202 [36]: Failing to recognize the camel case leads to an FP

3.7.8 Others. Three faults were not fixed by previously discussed fix strategies. In INFER-559 [50], Infer reported an FP because it only used method signature information to analyze the parameter properties for compiled Java programs, while no annotation is retained in the method signature information, causing the tool to miss out on all annotations on method parameters.

3.7.9 Relationship between Root Cause and Fix Strategy. Knowledge about the relationship between root causes and fix strategies is valuable for guiding the fix of annotation-induced faults. Table 5 shows the number of annotation-induced bugs caused by each root cause and fixed with each strategy. As shown in the table, FAF is the most commonly adopted fix strategy. Moreover, most faults fixed by strategy FAF were due to root causes IS or UEA, probably because it was too challenging for the static analyzers to correctly handle the semantics of the annotations involved in those faults. Therefore, the tool developers resorted to the filter-based solution as a workaround for the faults.

Table 5. Relationship between root cause and fix strategy

Root Cause	FAF	FAN	FIT	FGI	FVC	RRC	FIA	Others	Overall
Incomplete Semantics (**IS**)	83	0	0	0	3	4	0	3	93
Improper AST Traversal (**IAT**)	7	36	13	0	13	5	0	0	74
Unrecognized Equivalent Annotations (**UEA**)	21	1	0	1	1	0	0	0	24
Erroneous Type Operations (**ETO**)	1	1	16	0	3	0	2	0	23
Incorrect AST Generation (**IAG**)	0	0	1	20	0	0	0	0	21
Misprocessing of Configuration File (**MCF**)	2	0	3	1	0	1	1	1	9
Others	0	0	0	1	0	0	1	0	2
Overall	114	38	33	23	20	10	4	4	246

FAF: Fix Incorrect Use of Annotation Filter; **FAN**: Fix AST Node Retrieval; **FIT**: Fix Incorrect Type Operation; **FGI**: Fix Grammar Issue; **FVC**: Fix Value Check; **RRC**: Redesign Rule Checker Pattern; **FIA**: Fix Incorrect API Usage.

Finding 9: FAF was the most common fix strategy, especially for faults caused by IS and UEA.

FIT is also a popular fix strategy and can fix most root causes except for IS and UEA (both are fixed by FAF mostly). Most FIT related issues are due to fixing type resolution (15) and type checking (11). Sec. 3.5 states that type resolution issues are caused by incorrect auxiliary library configuration and missing identifier resolution. To fix the former type resolution issue, developers need to load proper libraries and find the correct class file to resolve the annotation. For the latter, developers should consider all possible program elements that need resolutions (e.g., the fault in Fig. 11 occurs because it fails to resolve the annotation in fully qualified name *org.foo.@MyAnnotation*, leading to an unused import FP at line 2). FGI mainly appears in one root cause (IAG) because incorrect grammar leads to parsing failure. To fix them, developers should check whether the next token from the lexical stream is a "@" symbol. Based on our study, developers often made mistakes when handling annotations on throw type, variable arguments, and generic type.

```
1 package org.foo;
2 import org.foo.bar.MyAnnotation; // report an FP
3 class A {
4   org.foo.@MyAnnotation B myB;
5 }
```

Fig. 11. SONARQUBE-2083 [22]: Fail to resolve annotation fully qualified name

Finding 10: Among all fix strategies, FIT covers the greatest number of root causes. Fixing type cast and type resolution account for the majority of issues.

4 IMPLEMENTATION OF ANNATESTER FRAMEWORK

We propose AnnaTester, a framework for automatically detecting annotation-induced faults via metamorphic testing, and it includes three checkers motivated by our study findings. Figure 12 shows the overall workflow of AnnaTester. Given a set of input programs obtained from the test suite of a static analyzer under test as the input, AnnaTester detects annotation-induced faults in the analyzer in two steps: (1) generates annotated programs by injecting annotations into the input programs; (2) checks whether the analysis reports produced by the analyzer on the input programs, both with and without annotations, satisfy the corresponding metamorphic relations. We adopt Eclipse JDT library to parse source seed files and inject annotations.

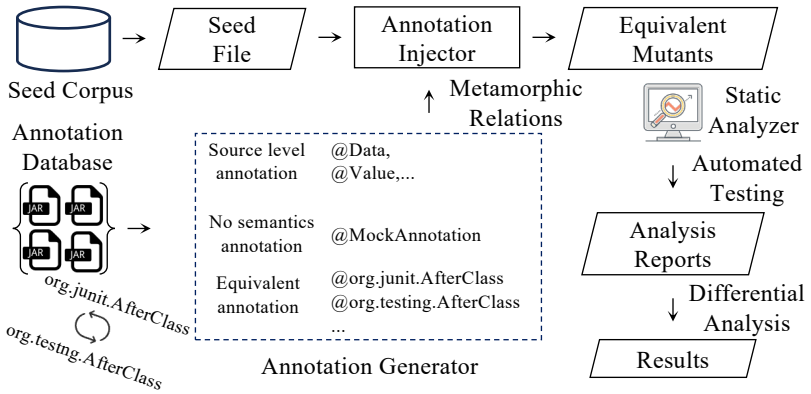

Fig. 12. Overall Workflow of AnnaTester

4.1 Checkers and Metamorphic Relations

AnnaTester essentially relies on three checkers to detect AIFs, each of which is based on a metamorphic relation concerning analysis reports on programs with and without annotations. When the metamorphic relation of a checker is violated, an AIF is detected, and the checker reports violations together with the input programs, with and without annotations, to users for further analysis. All metamorphic relations are based on the analysis equivalence relation between programs as below:

Definition 1 (Analysis Equivalence)**.** Two programs P and P' are analysis equivalent w.r.t. a static analyzer S, denoted as $P \equiv_S P'$, if and only if (1) S reports the same issues on P and P' and (2) S terminates in the same state, i.e., successful or with errors when applied on P and P'. We use $P \equiv P'$ to denote that P and P' are analysis equivalent w.r.t. any static analyzer.

In the rest of this subsection, we will use the following notations. Let P be a Java program, a be an annotation, $\mathcal{P}(P)$ denotes the resultant program produced by processing the annotations in P; $\mathcal{I}(P, a)$ denotes the set of all programs produced by applying a to appropriate elements in P.

4.1.1 Incomplete Semantics Checker (ISC). Incomplete semantics (IS) was the most common root cause for AIFs. In our study, one fault in PMD due to IS was evidenced by contrasting the analysis reports produced on two programs that were supposed to be analysis equivalent since the second program is derived by processing all the annotations in the first one [41]. Motivated by this example

and Finding 2, we design a metamorphic relation requiring that a program P should be analysis equivalent to the resultant program produced by processing the annotations in P.

Definition 2 (MR1)**.** Given a program P, P and $\mathcal{P}(P)$ should be analysis equivalent, i.e., $P \equiv \mathcal{P}(P)$.

As stated in Section 2.2, source-level annotations are not retained in the compiled code (i.e., the semantics of those annotations must be fully processed and incorporated into the program code during compilation). Hence, the differences between static analysis results of the programs before and after their source-level annotations have been processed indicate potential annotation-induced faults due to IS in the analyzers. In view of that, the incomplete semantics checker focuses on detecting IS faults caused by source-level annotations. Since IS never led to faults in CheckStyle and Soot (Finding 2), we do not apply this checker to detect faults in these two analyzers.

4.1.2 Annotation Syntax Checker (ASC). Findings 3 and 5 indicate that incorrect AST generation and traversal may cause static analyzers to produce inaccurate analysis results or even crashes. As such negative influences are independent of the semantics of the involved annotations, we implement an annotation syntax checker based on the following metamorphic relation on *dummy annotations* (i.e., annotations that mark program elements but have no impact on programs' semantics). Notably, metamorphic relation MR2 states that adding dummy annotations to a program should not affect the static analysis detection results produced on the program.

Definition 3 (MR2)**.** Given a program P and a dummy annotation d, $\forall p \in \mathcal{I}(P, d) : P \equiv p$.

4.1.3 Equivalent Annotation Checker (EAC). To identify inconsistent behaviors across equivalent annotations, we design the following metamorphic relation motivated by Finding 4:

Definition 4 (MR3)**.** Given a program P annotated with an annotation a_1 and another annotation a_2 that is equivalent to a_1, P and $P_{a_1|a_2}$ should be analysis equivalent, i.e., $P \equiv P_{a_1|a_2}$, where $P_{a_1|a_2}$ denotes the resultant program produced by replacing annotation a_1 with a_2 in P.

Our study shows that static analyzers sometimes fail to recognize all annotations with the same semantics. To address that limitation, we devise an equivalent annotation checker based on this metamorphic relation to automatically detect annotation-induced faults due to the root cause UEA.

4.2 Annotated Program Generator

We design an annotated program generator to automatically derive annotated programs from the input programs. The generated programs with annotations will be fed together with the original input programs to the static analyzers, and their analysis results will be checked by the checkers w.r.t. the aforementioned metamorphic relations. The generator has three core components: 1) an annotation database, 2) an annotation generator, and 3) an annotation injector.

4.2.1 Annotation Database. To build a database containing widely-used Java annotations, we obtain annotations from two kinds of libraries in Maven Repo [56]: (1) the top 100 popular Java libraries and (2) the top 100 popular libraries labeled as "Annotation libraries". In total, our database contains 1616 annotations from 194 Java libraries (two libraries are duplicated, and four cannot be downloaded).

4.2.2 Annotation Generator. Our generator produces three types of annotations: (1) source level annotations, (2) dummy annotations, (3) equivalent annotation tuples. They correspond to the three checkers (i.e., ISC, ASC, and EAC).

Source Level Annotations. AnnaTester automatically selects source-level annotations from the database and generates annotation declarations without explicitly specified property values. Thus, AnnaTester effectively associates all the annotations' properties to their default values.

Dummy Annotations. AnnaTester uses the dummy annotation defined in Figure 13. We set its target to include all types of program elements that can be annotated so as to test the interplay between the annotation and static analyzers' AST-related operations more thoroughly. We set its retention policy to RUNTIME so that the annotation will be retained for a longer time and hopefully can help us detect more annotation-induced faults at different stages of a static analyzer.

```
1 import java.lang.annotation.*;
2 @Target({ElementType.METHOD, ...}) // Other targets omitted for space reasons
3 @Retention(RetentionPolicy.RUNTIME)
4 public @interface MockAnnotation {}
```

Fig. 13. Definition of the dummy annotation.

Equivalent Annotation Tuples. As explained in Section 3.5.3, equivalent annotations should have similar semantics. AnnaTester conservatively considers two annotations to be equivalent if and only if they have the same name and target set. As all the annotations in a tuple are semantics equivalent and added to identical program elements, their analysis scopes are the same. In total, we have collected 132 equivalent annotation tuples. If AnnaTester were to use all 132 tuples, too many mutants could be generated (as each tuple leads to at least two annotated programs being generated). Hence, we select 24 tuples based on the top 30 AIF prone annotations identified in RQ1. All tuples have been manually verified that they are indeed equivalent tuples by two authors. Additionally, using all tuples will significantly increase running time, e.g., for PMD, the fastest among evaluated tools, tuple selection can reduce running time by 91.3% (all tuples = 69 hours, selected tuples = 6 hours) while finding the same number of bugs.

4.2.3 Annotation Injector. Given an input program P and an annotation a generated by the annotation generator, the annotation injector first analyzes the annotation to determine the set of valid targets for it, then goes through the program to collect specific locations where the annotation can be applied, and finally automatically inserts the annotation in all those locations. For example, if ElementType.METHOD is a valid target for an annotation, the annotation can be applied to annotate method declarations. The output of the annotation injector is a set of P's variants, or mutants, each with the annotation being injected in a different location. Some injected annotations may cause compilation errors (around 2%) if their corresponding properties require explicit initialization, so AnnaTester discards these syntactically invalid variants before proceeding to subsequent steps.

5 EFFECTIVENESS OF ANNATESTER

We applied AnnaTester to PMD, SpotBugs, CheckStyle, Infer, SonarQube and Soot and conducted experiments to measure the effectiveness of AnnaTester by reusing test suites from the official repositories of static analyzer as the seed corpus as prior work shows that these tests can help us cover more rule checkers to reveal more faults [83]. For static analyzers that require compilation (e.g., SpotBugs), we compile each program using Oracle JDK 17. All experiments were conducted on a machine with Intel Xeon(R) 6134 CPU 3.20GHz and 192GB RAM. For each checker and its corresponding annotations, we run AnnaTester on all analyzers in parallel until all generated mutants have been evaluated and do not set any timeout. We did not test AnnaTester on known issues as it was designed based on insights gained from these issues. Testing AnnaTester on the same issues would introduce bias. We also identify two challenges in evaluating AnnaTester on known issues: (1) it involves building old versions of analyzers from their source code, which can be quite demanding (e.g., due to the absence of required external libraries and the intricacies of the compilation process), (2) we are missing compilable input programs to reproduce some known issues, but those programs can be hard to construct manually, and AnnaTester requires them as the input.

Table 6. Effectiveness of AnnaTester

Checker	#Violations	#UniqFaults	#FP	#Fixed	Time (min, max) (hour)
ISC	258	19	8	11	(4,62)
ASC	52	8	0	4	(2,24)
EAC	123	16	0	5	(6,87)
Overall	433	43	8	20	(6,87)

Table 6 shows the experiment results. We measure the effectiveness of AnnaTester by counting the unique faults detected by each checker ("#UniqFaults" column). Specifically, we manually analyze the root causes of the identified faults and remove duplicated ones. Notably, we consider two faults duplicated if they are in (1) an identical rule checker and (2) an identical faulty location (determined by root cause diagnosis) in a static analyzer. Table 6 shows that AnnaTester found 43 bugs in evaluated static analyzers, and 20 have been fixed via merged pull requests (9 by developers and 13 by authors). Overall, AnnaTester finds the greatest number of faults using ISC. This result is consistent with Finding 2, which shows the prevalence of IS in static analyzers. The "Time" column shows the minimum and maximum total execution time for all checkers on the six static analyzers. Although different checkers use the same seed corpus, the time taken by different checkers varies because the number of annotations and the number of valid program locations to inject these annotations are different.

Fix Strategies for AnnaTester's Found Bugs. To further analyze the fixed issues, we classify the fix strategies of the 20 fixed issues. All of them fit into our taxonomy of fix strategies: 14 by FAF, 3 by FGI, 2 by FAN, and 1 by FIA. The result *illustrates the generality of our taxonomy.*

Limitations. Like other testing tools, AnnaTester also reports eight FPs ("#FP" column in Table 6). Our manual analysis of the FPs revealed that all FPs are caused by the source code changes induced by applying *MR*1 to recover annotation semantics. For example, *@NoArgsConstructor* is semantically equivalent to injecting a no-argument constructor into source code, but the constructor triggers a *UnnecessaryConstructor* warning in PMD, causing an FP (this extra warning misleads AnnaTester into thinking that the programs before and after annotation processing are not analysis equivalent). Another limitation is AnnaTester requires manual effort to verify the correctness of the 43 identified unique faults.

6 CASE STUDY

We select three faults found by AnnaTester to show AnnaTester's fault finding capability. For each fault, we present its root cause, the affected analyzer, and how AnnaTester found the issue.

A Crash in PMD [7]. Finding 5 shows that static analyzers cannot handle special annotation syntax as developers tend to neglect them. Figure 14 shows an example of crash in PMD discovered by AnnaTester. At line 7 of this example, PMD fails to process the annotation *@DummyAnnotation* placed on the class constructor reference *::new* due to the grammar issue, consequently leading to a runtime crash. The developers have fixed this issue upon receiving our report.

An FP in SonarQube [6]. Figure 15 shows a fault caused by incomplete semantics. SonarQube reports an unclosed stream warning at line 2, but it is an FP because the *@Cleanup* annotation will generate a *try-finally* statement to close *FileInputStream* in the *finally* block. This issue has been confirmed and marked as "Major" priority by the developer, indicating the importance of the fault.

An FP in PMD [44]. Figure 16 shows that PMD reported a warning against the unnecessary constructor at line 4. But that is an FP because the annotation *@Inject* uses this constructor for dependency injection. PMD does not consider the annotation "com.google.inject.Inject" in Figure 16,

```
1 import java.util.function.Function;
2 public class Main {
3   public class Inner {
4     public Inner(Object o) {}
5   }
6   public Function func(Main this) {
7     return @DummyAnnotation Main.Inner::new; // Crash
8   }
9 }
```

Fig. 14. A crash example in PMD detected by AnnaTester

```
1 public static void main(String[] args) throws IOException {
2   @Cleanup InputStream in = new FileInputStream(args[0]); // FP
3   ...
```

Fig. 15. An FP example in SonarQube detected by AnnaTester

but it has considered another equivalent annotation "javax.inject.Inject". EAC can automatically detect this FP. We have fixed this bug via a merged PR in collaboration with developers.

```
1 import com.google.inject.Inject;
2 public class Foo { private Foo() {} }
3 public class Bar extends Foo {
4   @Inject public Bar() {} // Report a warning, but it is an FP
5 }
```

Fig. 16. An FP example in PMD detected by AnnaTester

7 IMPLICATION

We discuss below the implications for developers and researchers based on our study findings:
Implication for Developers. Our study identifies the common root causes and their corresponding symptoms and fix strategies that may help developers of static analyzers to detect, understand, and repair faults caused by annotation. We also study AIF prone annotations, implying that developers should pay attention to these annotations (Finding 1). Based on this finding, we design AnnaTester to select AIF prone annotations. In the future, it is worthwhile to investigate more advanced techniques for annotation selection. Based on the two most common root causes of annotation-induced faults in our study (IS and IAT), we realized that developers of static analyzers tend to either (1) be unaware of the semantics encoded by annotations (Finding 2) or (2) neglect the impact of annotations on program ASTs (Finding 3). Hence, we hope that our study will raise awareness among developers on the impacts of Java annotations on static analyzers to improve the accuracy and correctness of static analyzers. In terms of the static analyzer workflow, our study revealed that *developers should pay careful attention to annotations when performing syntax analysis* because all studied static analyzers have annotation-induced faults in the syntactic analysis stage, especially when annotations are placed on the types such as generic type arguments and type casts since JSR-308 [31]. With the evolution of Java specification, developers should also consider annotation-induced faults when updating the grammar (Finding 5). Meanwhile, as our study also revealed that there exists a set of equivalent annotations that come from different libraries or different versions of the same libraries (Finding 4), *developers should consider these related annotation libraries when designing rule checkers to provide comprehensive support for the related annotations.*

Implication for Researchers. Our study and proposed framework lay the foundation for research in three promising directions. First, *encoding the semantics of annotations into static analyzers is essential in improving the accuracy of the analysis* because current analyzers fail to model the behavior of annotations well. Incomplete semantics is the most common root cause of AIFs in our study (Finding 2) and the greatest number of bugs detected by AnnaTester. Therefore, failing to solve this problem can affect the fault detection capability of static analyzers. Second, *detecting AIFs is necessary but yet often neglected*. For static analyzers, eliminating FPs is a worthwhile and long-term research direction [32]. As shown in Table 3, FP is the most common symptom caused by AIFs. Consequently, detecting AIFs is rewarding for reducing FPs. Metamorphic testing is a promising approach for this purpose. Researchers can produce annotated program pairs and compare their analysis reports to detect FPs (such as ISC). Third, *our fix strategies (Finding 8–10) serve as preliminary studies for future research on automated repair of AIFs*. We observe that several fix strategies in our study can be automated to reduce the effort in fixing them (e.g., Fix Annotation Filter (FAF) can fix more than half of the issues). Most of them are implemented by creating an annotation filter or extending an existing filter.

8 THREATS TO VALIDITY

External. Our study may not generalize beyond the studied analyzers and other programming languages beyond Java. To ensure the generalizability of our findings, we select six representative static analyzers (Section 3.1), and we systematically analyze 246 issues in these analyzers. The selection of whole-program analyzers may also have potential implications for the generalizability of the results. As stated in Section 3.1, we only consider static analyzers from open-source repositories with over 2,000 stars, solely choosing Soot as a representative of whole-program analyzers. However, our study shows AnnaTester can find bugs in lightweight tools (e.g., PMD), and Soot shares many root causes, symptoms, and fix strategies with lightweight tools. We leave a more comprehensive study on lightweight and whole-program static analyzers for future work.
Internal. Manually labeling annotation-induced faults may be subjective and biased. To reduce this threat, we refer to previous taxonomies [12, 67, 72, 84, 85] and adopt an open-coding scheme to adapt taxonomies to annotation-induced faults. Our code may have bugs that can affect the evaluation results. To mitigate this threat, we have made our tool and data publicly available.

9 RELATED WORK

Studies on Annotation. Several studies have investigated the common code annotation practices in Java [54, 58, 60, 66, 82]. These studies showed that annotations are widely adopted by Java developers and served as motivations for our study. Among these studies, the study of annotation-related faults and the mutation operators that mimic these faults is the closest related work [60] (e.g., the insertion of annotations in AnnaTester is similar to the ADA operator that adds an annotation to a valid target in prior work, but the ADA operator requires users to manually specify the annotation whereas we generate annotation from our annotation database [60]). Nevertheless, our study and our proposed technique are different from prior studies in several important aspects: (1) prior study only identified two categories of annotation-related faults ("misuse" and "wrong annotation parsing"), whereas our study focuses on the impacts on annotation-induced faults in static analyzers (the root cause categories of annotation-induced faults in our study is more diverse than that in prior study); (2) prior technique focuses on mutating code annotations [60] via a set of operators, but it cannot detect annotation-induced faults due to the lack of oracle, whereas AnnaTester injects annotations into input programs and uses a well-designed set of oracles to find annotated-induced faults in static analyzers. As the annotation-induced faults identified in our

study are more diverse, these faults may affect other types of software applications (beyond static analyzers). In the future, it is worthwhile to study annotation-induced faults in a broader domain.

Studies on Faults. There are several studies on faults [40, 67, 71, 75, 80, 85]. Although prior studies investigated faults for diverse types of software systems (e.g., machine learning systems [67, 75], blockchains [80], compilers [71], and static analyzers [40]), they did not cover AIFs of static analyzers. The most closely related to our study is the recent study [40] that evaluates the vulnerability detection capability of C static analyzers. Our work differs from the prior study in several aspects: (1) we conduct the first study to understand annotation-induced faults of static analyzers, covering the root cause, symptom, fix strategy, and AIF prone annotations; and (2) we propose metamorphic relations to detect annotation-induced faults, and implement a testing framework. A recent study [42] built benchmarks and evaluated the effectiveness and performance of static analyzers but did not investigate AIFs fault and failed to detect AIFs.

Testing Static Analyzers. Several techniques have been proposed for testing analyzers [3, 13, 15, 34, 39, 49, 76, 81]. Some use equivalent relations as the metamorphic relation to solve the lack of oracle problem in static analyzer testing [8, 37, 45, 83]. Similarly, we use equivalent relation (i.e., constructing three behavior equivalent properties and checking for differential analysis results of equivalent mutants) to address the oracle problem, but our equivalent properties have incorporated the characteristics of annotations to find annotation-induced faults. While prior approaches [49, 76, 83] can find other types of bugs in static analyzers, none of them focuses on AIF faults (and they could not generate annotated programs as in AnnaTester to detect AIFs).

10 CONCLUSION

We conduct the first comprehensive study which focuses on understanding and detecting annotation-induced faults of static analyzers as annotation has become a popular programming paradigm. We manually investigate 246 issues from six representative and diverse static analyzers (SonarQube, CheckStyle, PMD, SpotBugs, Infer, and Soot), identify six root causes, four symptoms, and seven fix strategies. Moreover, we also summarize ten findings. Based on these findings, we introduce a set of guidelines for AIFs detection and repair, and propose the first metamorphic testing based framework AnnaTester to automatically find AIFs in static analyzers. With our annotation synthesizer and three metamorphic relations, it can generate new tests based on official test suites and find 43 faults where 20 of them have been fixed.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their insightful comments.

DATA AVAILABILITY

The experimental data and source code are available at: https://annaresearch.github.io/.

REFERENCES

[1] adangel. 2019. *NoPackage: False Negative for enums.* https://github.com/pmd/pmd/issues/1782
[2] Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. 2009. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development.* 15–26.
[3] Cláudio A Araújo, Marcio E Delamaro, José C Maldonado, and Auri MR Vincenzi. 2016. Correlating automatic static analysis and mutation testing: towards incremental strategies. *Journal of Software Engineering Research and Development* 4, 1 (2016), 1–32.
[4] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. 241–252. https://doi.org/10.1145/1831708.1831738

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

[6] Belle. 2022. *A false positive about rule RSPEC-2095.* https://community.sonarsource.com/t/a-false-positive-about-rule-rspec-2095/67536

[7] Belle-PL. 2022. *Parse error on array type annotations.* https://github.com/pmd/pmd/issues/4152#issuecomment-1277447394

[8] Cristian Cadar and Alastair F Donaldson. 2016. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 765–768.

[9] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action.* Manning Publications Co.

[10] Walter Cazzola and Edoardo Vacchi. 2014. @ Java: Bringing a richer annotation model to Java. *Computer Languages, Systems & Structures* 40, 1 (2014), 2–18.

[11] CheckStyle. 2022. *Checkstyle.* https://checkstyle.sourceforge.io/

[12] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 339–351.

[13] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.

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

[15] 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.

[16] Krysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA.

[17] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. https://doi.org/10.1109/TSE.2020.3004525

[18] fluxroot. 2019. *False-positive with Lombok and inner classes.* https://github.com/pmd/pmd/issues/1641

[19] Michael Gumowski. 2015. *Annotations should be handled in all cases allowed by java 8.* https://sonarsource.atlassian.net/browse/SONARJAVA-1167

[20] Michael Gumowski. 2016. *Classes annotated with lombok's @EqualsAndHashCode should be ignored.* https://sonarsource.atlassian.net/browse/SONARJAVA-1513

[21] Michael Gumowski. 2017. *Annotations on type in a fully qualified name are not resolved.* https://sonarsource.atlassian.net/browse/SONARJAVA-2205

[22] Michael Gumowski. 2017. *FP on S1128 (UselessImportCheck) when annotation is used in fully qualified name.* https://sonarsource.atlassian.net/browse/SONARJAVA-2083

[23] Michael Gumowski. 2019. *Annotations are not always resolved when used in parameterized types.* https://sonarsource.atlassian.net/browse/SONARJAVA-3045

[24] Michael Gumowski. 2019. *Generating starting states make analysis crash (OutOfMemory) when too many annotated parameters.* https://sonarsource.atlassian.net/browse/SONARJAVA-3108

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

[26] Infer. 2022. *Infer Static Analyzer.* https://fbinfer.com/

[27] Quentin Jaquier. 2019. *Support Java 11 Generated annotation.* https://sonarsource.atlassian.net/browse/SONARJAVA-3174

[28] Quentin Jaquier. 2020. *Consistently support Nullable/CheckForNull/Nonnull annotations in rules.* https://sonarsource.atlassian.net/browse/SONARJAVA-3536

[29] Quentin Jaquier. 2023. *S5122: ClassCastException when annotation is defined with an identifier.* https://sonarsource.atlassian.net/browse/SONARJAVA-3438

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

[31] Josh Juneau. 2022. *JSR 308 Explained: Java Type Annotations.* https://www.oracle.com/technical-resources/articles/java/ma14-architect-annotations.html

[32] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th International Conference on Software Engineering*. 1307–1316.

[33] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Chen, and Jinqiu Yang. 2021. Studying Test Annotation Maintenance in the Wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 62–73. https://doi.org/10.1109/ICSE43902.2021.00019

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

[35] konrad jamrozik. 2013. *Regression: ERROR/dalvikvm(1854): Invalid type descriptor: 'dalvik.annotation.EnclosingClass'*. https://github.com/soot-oss/soot/issues/123

[36] krzyk. 2015. *SuppressWarnings should support CamelCase*. https://github.com/checkstyle/checkstyle/issues/2202

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

[38] lgoldstein. 2017. *EmptyBlock: NPE on annotation declaration*. https://github.com/checkstyle/checkstyle/issues/4472

[39] Junjie Li and Jinqiu Yang. 2024. Tracking the Evolution of Static Code Warnings: the State-of-the-Art and a Better Approach. *IEEE Transactions on Software Engineering* (2024).

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

[41] Dongmiao Liu. 2022. *SONARJAVA-73 add more lombok's used annotations for UnusedPrivateFieldCheck*. https://github.com/SonarSource/sonar-java/pull/102#issuecomment-87545890

[42] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 285–297. https://doi.org/10.1145/3597926.3598056

[43] Yi Liu, Yadong Yan, Chaofeng Sha, Xin Peng, Bihuan Chen, and Chong Wang. 2022. DeepAnna: Deep Learning based Java Annotation Recommendation and Misuse Detection. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 685–696.

[44] LynnBroe. 2021. *UnnecessaryConstructor: false-positive with @Inject*. https://github.com/pmd/pmd/issues/4487

[45] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 639–650.

[46] marcelmore. 2021. *UnusedPrivateMethod violation for disabled class in 6.23*. https://github.com/pmd/pmd/issues/2454

[47] martin. 2020. *UnusedPrivateField cannot override ignored annotations property*. https://github.com/pmd/pmd/issues/2876

[48] mjustin. 2020. *Exception when using SuppressWarningsHolder with @SuppressWarnings as an annotation property*. https://github.com/checkstyle/checkstyle/issues/7522

[49] 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)*. 550–562. https://doi.org/10.1109/ICSE48619.2023.00056

[50] msridhar. 2017. *Eradicate not reading annotations from class file*. https://github.com/facebook/infer/issues/559

[51] Marharyta Nedzelska. 2021. *FP in S3077 when volatile is used with @Immutable and @ThreadSafe annotations*. https://sonarsource.atlassian.net/browse/SONARJAVA-3804

[52] nrmancuso. 2020. *Compact Constructor AST is missing annotations*. https://github.com/checkstyle/checkstyle/issues/8734

[53] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. 2022. Mining Annotation Usage Rules: A Case Study with MicroProfile. In *2022 38th International Conference on Software Maintenance and Evolution, IEEE*.

[54] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. 2022. Mining Annotation Usage Rules: A Case Study with MicroProfile. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 553–562.

[55] Fernando Rodriguez Olivera. 2017. *FindBugs JSR305*. https://mvnrepository.com/artifact/com.google.code.findbugs/jsr305

[56] Fernando Rodriguez Olivera. 2023. *MvnRepository*. https://mvnrepository.com/

[57] oowekyala. 2022. *How PMD Works*. https://docs.pmd-code.org/pmd-doc-6.53.0/pmd_devdocs_how_pmd_works.html

[58] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Software Engineering* 18, 6 (2013), 1047–1089.

[59] Nicolas Peru. 2015. *Annotation on array type should be properly handled*. https://sonarsource.atlassian.net/browse/SONARJAVA-1420

[60] Pedro Pinheiro, José Carlos Viana, Márcio Ribeiro, Leo Fernandes, Fabiano Ferrari, Rohit Gheyi, and Baldoino Fonseca. 2020. Mutating code annotations: An empirical evaluation on Java and C# programs. *Science of Computer Programming* 191 (2020), 102418.
[61] PMD. 2022. *PMD An extensible cross-language static code analyzer.* https://pmd.github.io/
[62] William Pugh. 2022. *JSR 305: Annotations for Software Defect Detection.* https://jcp.org/en/jsr/detail?id=305
[63] rnveach. 2016. *Java 8 Grammar: annotations on varargs parameters.* https://github.com/checkstyle/checkstyle/issues/3238
[64] rnveach. 2021. *AtclauseOrder: Falsely ignores method with annotation.* https://github.com/checkstyle/checkstyle/issues/9941
[65] robtimus. 2021. *OperatorWrap with token ASSIGN too strict for annotations.* https://github.com/checkstyle/checkstyle/issues/10945
[66] Henrique Rocha and Marco Tulio Valente. 2011. How Annotations are Used in Java: An Empirical Study.. In *SEKE*. 426–431.
[67] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.
[68] SonarQube. 2022. *SonarQube Code Quality and Code Security.* https://www.sonarqube.org/
[69] Soot. 2023. *Soot - A framework for analyzing and transforming Java and Android applications.* http://soot-oss.github.io/soot/
[70] SpotBugs. 2022. *SpotBugs: Find bugs in Java Programs.* https://spotbugs.github.io/
[71] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.
[72] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 348–357.
[73] 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.
[74] Daniel Tang, Ales Plsek, and Jan Vitek. 2010. Static checking of safety critical Java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. 148–154.
[75] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
[76] David A Tomassi and Cindy Rubio-González. 2021. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 292–303.
[77] Susana M Vieira, Uzay Kaymak, and João MC Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *International conference on fuzzy systems*. IEEE, 1–8.
[78] Jules Villard. 2020. *Infer workflow.* https://fbinfer.com/docs/infer-workflow
[79] vovkss. 2018. *[java] Processing error (ClassCastException) if a TYPE_USE annotation is used on a base class in the "extends" clause.* https://github.com/pmd/pmd/issues/1369
[80] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 413–424.
[81] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find Bugs in Static Bug Finders. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 516–527. https://doi.org/10.1145/3377811.3380380
[82] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. 2021. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Transactions on Software Engineering* 47, 5 (2021), 969–986. https://doi.org/10.1109/TSE.2019.2910516
[83] 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*. Association for Computing Machinery (ACM).
[84] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1159–1170.
[85] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

Received 2023-09-28; accepted 2024-01-23