# EFFECTIVE FAULT DETECTION FOR STATIC ANALYZERS VIA AUTOMATED TESTING

HUAIEN ZHANG

PhD

The Hong Kong Polytechnic University

2025

The Hong Kong Polytechnic University

Department of Computing

# Effective Fault Detection for Static Analyzers via Automated Testing

Huaien Zhang

A thesis submitted in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

March 2025

# CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

Signature: _____

Name of Student: Huaien Zhang

# Abstract

Static analyzers comprehend and analyze input programs without dynamically executing them to gather insights into and detect flaws in their properties and behaviors. These tools are indispensable for ensuring software quality and supporting various software engineering tasks, including vulnerability detection, privacy leakage identification, and malware analysis. Despite their widespread adoption in real-world software development and maintenance, static analyzers, like other computer programs, are susceptible to implementation faults, and it is a common practice for static analyzers to detect such faults via testing. Manually creating test cases for static analyzers, however, is highly time-consuming and labor-intensive because both constructing input programs to trigger specific analyses and deriving the correct analysis results for the input programs are non-trivial tasks. Meanwhile, existing research efforts to automatically generate test cases and uncover faults in static analyzers suffer from three important limitations that restrict their applicability. These efforts depend on dedicated oracles designed for specific programming languages or particular sets of static analyzers, have limited support for certain program elements, or overlook bugs reflected in only the intermediate representations constructed by the static analyzers but not the warnings they report.

To address these limitations, we develop three novel techniques, namely Statfier, AnnaTester, and SAScope. The Statfier technique leverages semantics-preserving program transformations to derive valid variants from existing test input programs for static analyzers, and it discovers faults in the static analyzers via metamorphic testing. We systemati-

cally investigate the impact of program annotations on static analyzers and propose another metamorphic testing technique, AnnaTester, to automatically identify annotation-induced faults. Furthermore, we comprehensively study the root causes of program representation faults and their fix strategies and develop the SAScope technique to detect relevant faults via automated testing.

We have implemented the techniques into three testing frameworks with the same names. Using the testing frameworks, we identify 141 faults in popular static analyzers. We have reported all identified faults to the respective developers via issue tracking systems, with 72 of them confirmed or fixed.

# Publications Arising from the Thesis

(* indicates the corresponding author.)

1. **Huaien Zhang**, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan, "Characterizing and Detecting Program Representation Faults of Static analyzers", in *the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*.

2. **Huaien Zhang**, Yu Pei, Shuyun Liang, and Shin Hwei Tan, "Understanding and Detecting Annotation-Induced Faults of Static Analyzers", in *the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)*.

3. **Huaien Zhang**, Yu Pei, Junjie Chen, and Shin Hwei Tan*. 2023, "Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations", in *the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*.

4. Ying Li, Haibo Wang, **Huaien Zhang**, Shin Hwei Tan, "Classifying Code Comments via Pre-trained Programming Language Model", in the 2nd ICSE Workshop on NL-based Software Engineering (NLBSE 2023).

5. Haibo Wang, Zhuolin Xu, **Huaien Zhang***, Shin Hwei Tan, "An Empirical Study of Refactoring Engine Bugs", submitted to TOSEM, under revision.

6. Xiaowen Zhang, **Huaien Zhang***, Shin Hwei Tan, "Symbiotic Code and Test Reuse for Redesigned Projects", submitted to TOSEM, under review.

# Acknowledgments

First and foremost, I feel profound gratitude towards my supervisors, Prof. Yu Pei and Prof. Shin Hwei Tan, for all their meticulous efforts through every stage of my study. Learning and working under their supervision is a precious experience of a lifetime. They are always optimistic, energetic, and inspiring, which profoundly affects me. I have learned a lot from our countless discussions; they are always full of fascinating ideas that can turn a difficult problem I encountered into a chance, and they always welcome and attentively listen to my immature research thoughts. Looking back on my PhD journey, choosing them as my supervisors was, without a doubt, the absolute best decision I could have made.

I would like to thank my co-supervisor, Prof. Yuqun Zhang. His meticulous approach to research serves as an exemplary paradigm for an outstanding scholar. I also learned a great deal from him, encompassing writing skills and research strategies.

I am hugely appreciative of Prof. Junjie Chen, who gave me much valuable advice on my first research paper. His generous assistance and insightful ideas made our achievements possible, making the process significantly easier than it would have been otherwise. I am also deeply grateful to Prof. Minxue Pan, Prof. Yepang Liu, Prof. Tian Zhang, Prof. Geng Hong, Dr. Liushan Chen, Dr. Sen Yang, Dr. Mingyuan Wu, and Dr. Xin Tan for their precious help and sincere suggestions on my career choice.

At last, I want to thank my parents, colleagues, and friends, for supporting me with their love and selfless help during my PhD study.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Static analyzers comprehend and analyze the input programs without dynamically executing them to gather insights into their properties and behaviors. The insights can be utilized to facilitate a wide range of downstream software engineering tasks, including, e.g., bug detection [7, 14, 16, 17], privacy leakage recognition [69, 145, 218], and vulnerability identification [84, 216].

Popular static analyzers typically go through a three-step process in analyzing the input programs and producing the analysis reports [53, 76, 124, 206]. In particular, given the program code and relevant configurations as input, a static analyzer first parses the code via lexical and syntactic analyses and then constructs the corresponding abstract syntax trees (ASTs) and intermediate representations (IRs) for the program; next, it builds more sophisticated program representations like class hierarchy, control flow graph, and call graph, as models of the program to support the following analyses; finally, when applicable, it invokes integrated rule checkers to identify flaws based on the constructed program representations and warns the users against them. Figure 1.1 illustrates the process.

Although static analyzers are widely used in practice to help enhance software quality, they sometimes produce erroneous results due to faults mistakenly introduced by their developers during tool design and/or implementation. For instance, prior research [182] showed

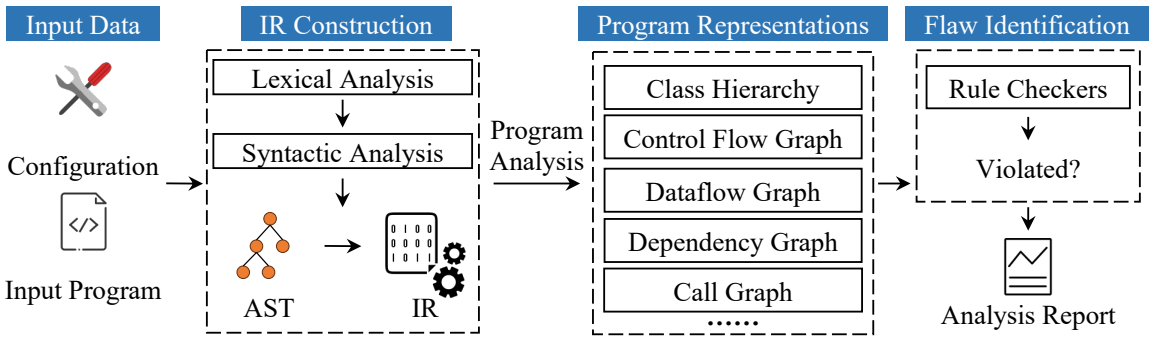


Figure 1.1: General workflow of a static analyzer

that the buggy implementations of call graph construction algorithms and the lack of support for certain programming language features were two leading root causes of incomplete or inaccurate call graphs [173, 191]. So far, testing is the most widely applied technique to effectively discover the faults in static analyzers, and a considerable amount of test cases in static analyzer testing were crafted manually [208]. Such a manual approach to test preparation, however, is highly insufficient because the number of viable ways to implement specific functionalities by combining different language constructs is typically vast, while the number of combinations covered by manual tests is often very limited.

In view of that, researchers have proposed various techniques to automatically test the static analyzers in the past few years. The state-of-the-art automatic testing techniques for static analyzers can be classified into three main categories, namely techniques based on differential testing, random testing, and fault injection. For instance, Wang et al. [208] use a natural language processing approach to identify equivalent rule checkers between two static analyzers and reveal inconsistencies among these detectors via differential testing. Cuoq et al. [85] propose an approach that generates mutants and designs specific oracles for internal analyses in Frama-C. SolidiFI [97] mutates programs written in the Solidity programming language for developing smart contracts by injecting seven types of bugs and evaluates the effectiveness of smart contract analysis tools based on how often they can detect the injected faults. Overall, we identify three major limitations in existing static analyzer testing techniques that restrict their effectiveness and applicability. First, they rely

on dedicated oracles designed for a specific programming language (e.g., Solidity) or static analyzers with specific properties (e.g., those implementing rule checkers with equivalent functionalities). Second, although certain language constructs (e.g., annotation) can considerably influence the programs' semantics, static analyzers often have limited support for such constructs. Third, they solely focus on the warnings reported in the analysis results and essentially ignore the other problems with the representations that the static analyzers construct and pass on for downstream tools to consume without causing any warnings.

## 1.1 Main Contributions

This thesis focuses on developing automated software testing techniques to (partially) overcome the aforementioned limitations and effectively detect faults in static analyzers. We started by proposing a testing framework called Statfier to detect generic faults in static analyzers. In our experimental evaluation of Statfier, we identified two dominant root causes for the detected faults. First, static analyzers may fail to handle certain language features correctly and effectively. Second, due to the inherent complexity in understanding and modeling the intricate relationships among program elements, static analyzers may misrepresent the programs under analysis. Based on these findings, we conducted empirical studies on faults due to these root causes and proposed two automated testing approaches, namely AnnaTester and SAScope, to uncover these types of faults in static analyzers.

More concretely, the main contributions of this thesis include the following.

- Statfier*: testing static analyzers via semantics-preserving program transformations.* Testing static analyzers in a general and automated manner begins with addressing the test oracle problem, a significant challenge. This problem is ubiquitous in the automated testing of various software systems. One widely used technique to partially address this challenge is differential testing, which relies on the behaviors of similar applications or different implementations of the same application as cross-

references. However, differential testing is not always viable for testing static analyzers, particularly their checkers (e.g., flaw detectors), because these checkers often verify different properties and detect various types of bugs [108, 208], making their analysis reports seldom directly comparable.

Another challenge is the automated preparation of high-quality input programs for static analyzers. Given the property being checked by the analyzers, effective and efficient testing requires input programs that contain essential elements to activate the corresponding checker, while remaining minimal to facilitate easy comprehension of the analysis reports. These requirements make the preparation of high-quality test inputs for static analyzers particularly demanding.

To address the two challenges, we propose the STATFIER technique that can effectively enhance the flaw detection capability of test suites in static analyzers. STATFIER leverages semantics-preserving transformations and metamorphic testing to generate new input programs, thereby overcoming the oracle challenge. To improve test generation effectiveness, STATFIER employs two novel heuristics: analysis report guided location selection and structure diversity driven variant selection. Additionally, we reuse the official test suites and documentation to extract seed input programs and achieve a high rule coverage to solve the input program challenge. To validate the effectiveness of our approach, we apply STATFIER to five popular static analyzers, identifying 79 faults, 46 of which have been fixed or confirmed by developers.

- ANNATESTER*: understanding and detecting annotation-induced faults of static analyzers.* In recent years, annotations have been widely adopted in practical software development, providing a structured way to attach helpful information to program elements such as classes, methods, variables, and types. Popular frameworks like Spring, JUnit, and Lombok rely heavily on their homebrewed annotations to simplify reusing the frameworks.

  In general, the presence of annotations poses two challenges to ensuring the correctness and reliability of static analyzers. First, annotations in programs introduce

additional tokens that analyzers need to parse. An unprepared analyzer may overlook or mishandle these tokens, resulting in incorrect analysis results or even premature runtime termination. For example, given the simple Java class in Figure 1.2 as input, PMD will crash because it is not designed to handle the annotated array access operators in line 6. Second, annotations can alter the structure or behavior of 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. Consequently, static analyzers may produce incorrect results if annotations interfere with the properties or behaviors being analyzed.

```
1  @Retention(RetentionPolicy.CLASS)
2  @Target({TYPE_USE})
3  @interface Anno {}
4  public class Main {
5    // Causes PMD to crash
6    public <T> T[][] check(T @Anno[] @Anno [] arr) {
7      if (arr == null) {
8        throw new NullPointerException();
9      }
10     return arr;
11   }
12 }
```

Figure 1.2: An annotated Java program causing PMD to crash

We conduct the first empirical study on annotation-induced faults in static analyzers based on 238 issues from six popular open-source analyzers. We analyze the root causes, symptoms, and fix strategies of these issues, leading to ten key findings and several practical guidelines for detecting and repairing annotation-induced faults. Additionally, we develop an automated testing framework called AnnaTester based on three metamorphic relations derived from our findings. AnnaTester generates new tests from the official test suites of static analyzers and uncovers 43 new faults, 20 of which have been fixed. The results confirm the value of our study and its findings.

- SAScope*: characterizing and detecting program representation faults of static analyzers.* Static analyzers typically construct various program representations, such as call graphs, control flow graphs, and intermediate representations, which encode the properties and behaviors of the given program for further analysis [65, 140, 209]. However, developers of static analyzers may make mistakes when implementing different analysis algorithms, resulting in incomplete/inefficient analysis processes and/or incorrect program representations. For instance, prior research shows that buggy implementations of complex call graph construction algorithms and missing support for certain programming language features are two main reasons for incorrectly constructed call graphs [173, 191].

  We conduct the first empirical study on program representation faults in static analyzers and develop SAScope, an automated testing framework that detects these faults using metamorphic and differential testing. In our metamorphic testing component, we propose a new metamorphic relation based on the relative precision lattice of various program representation algorithms. In summary, SAScope can identify 19 program representation faults, 5 of which have been fixed by developers.

## 1.2 Terminology

In this thesis, we use the term "fault" to refer to issues identified by our proposed testing approaches, and the term "flaw" to refer to bugs, vulnerabilities, and code smells detected by static analyzers.

## 1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 reviews existing research work closely related to the automatic testing of static analyzers. Chapters 3, 4, and 5 elaborate

on our efforts to address the limitations in existing approaches to automated testing of static analyzers, respectively. Chapter 6 concludes the thesis and discusses possible future work.

# Chapter 2

# Related Work

Given the vital importance of static analyzers for software development and maintenance, extensive research has been conducted to understand and improve the quality of static analyzers from various aspects. For instance, previous studies have assessed the efficiency of static analysis methods [70,71,83,108,120] and proposed techniques to prioritize the warnings they report and therefore improve their user friendliness [112,127,184]. In contrast to these studies, Our work evaluates the effectiveness of static analyzers and develops novel methods for detecting faults in static analyzers through automated testing. In this chapter, we systematically review related work in the areas of static analyzer testing and compiler testing.

## 2.1 Static Analyzer Testing

Several approaches have been proposed for testing static analyzers [85, 135, 196, 208], including those based on differential testing, random testing, and fault injection.

### 2.1.1 Differential Testing

To identify potential bugs in complex software systems such as static analyzers and compilers, McKeeman [153] proposes the differential testing method. This approach is a software testing technique that identifies bugs by providing the same input to a series of equivalent software systems and observing inconsistencies during the program execution. It has been successfully used to detect semantic bugs successfully in diverse real-world systems like web application firewalls [67], security policies for APIs [189], and antivirus software [116, 168]. Wang et al. [208] adopt a natural language processing approach to recognizing equivalent bug checkers between two static analyzers and revealing inconsistencies among them. However, static analyzers often have different objectives and may not have common equivalent checkers to fulfill the requirements of differential testing. In contrast, we combine semantics-preserving program transformations and metamorphic testing to generate variants that can test frameworks generally.

Taneja et al. [196] propose the SMT solver based differential testing approach. The authors first select some typical static analyses deployed in the LLVM compiler. Then, they implement these analyses via an SMT solver, and performed differential analysis on the results provided by LLVM and SMT solver, aiming to identify precision and soundness issues from their inconsistencies. In contrast, we use transformed programs as variants for the test generation, which is more efficient because utilizing an SMT solver is often time-consuming and requires extensive processing to resolve complex constraints. The experimental results also confirm that our test generation approach is practical and can generally find bugs in multiple static analyzers.

Kapus and Cadar combine fuzzing with a differential testing approach to find bugs in symbolic execution engines [123]. They create three models of programs, including concrete mode, single-path mode, and multi-path mode. Besides, they also design three differential testing based oracles via executor crash, output, and function call chain perspectives. Unlike our approach, this work specifically targets symbolic execution engines and compares

them to randomly generated programs which have proved ineffective in detecting bugs in static analyzers.

Karine et al. [95] introduce GrayC, an open-source greybox fuzzing and differential testing tool, designed to detect faults in C program analyzers. GrayC generates valid program variants in statically-typed languages through semantics-aware mutation operators, capable of triggering deep-seated faults in static analyzers. This approach combines the effectiveness of syntax-based blackbox fuzzing with the targeted search capabilities of greybox methods. GrayC performs mutations at the abstract syntax tree (AST) level, including modifications to individual programs and combinations of multiple program elements. However, applying GrayC to static analyzers still needs a couple of static analyzers to compare the analysis results. Otherwise, it can only detect crash bugs without identifying soundness and precision issues.

### 2.1.2 Random Testing

Random testing is a black-box software testing approach in which programs are generated randomly without relying on specific inputs. The output results are evaluated against the software specifications to determine whether the test output meets the pass or fail criteria. In the absence of specifications, language exceptions are utilized as a criterion; specifically, if an exception is raised during test execution, it indicates the presence of a fault in the program. This approach also helps mitigate the risk of biased testing.

Frama-C [128] can perform static analysis on C programs. Cuoq et al. [85] reuse Csmith, an automatic program generator [211], to randomly create C programs and design some specific oracles to detect bugs in the static analyses of Frama-C. Since the value analysis in Frama-C is an over-approximation, they attempt to make Frama-C function as an interpreter and compare its interpretation results with those of the compiled execution, revealing precision issues where the analyzer loses information that should not be omitted. In addition to comparing with actual execution results, they insert assertions (containing variable

range information derived from value analysis, e.g., $x > 3$ and $x < 7$) into the source code and verify correctness via runtime execution at a finer granularity. For constant propagation, they compare the checksums of all global variables in the transformed and original programs.

In contrast to these studies, we implement more general oracles to identify potential faults across various static analyses, without being limited by their specific types. Moreover, while random testing is highly effective at revealing crash bugs, it is challenging to construct programs that identify logic bugs, such as precision and soundness faults.

### 2.1.3 Bug Injection

Several previous studies have adopted the bug injection approach to testing static analysis tools [97, 113, 148]. This approach typically injects known bugs (which can be detected by the static code analyzers) into source programs. Subsequently, the corresponding analysis reports from static analyzers are examined to verify whether the bugs are identified. This process allows for the direct observation of soundness and precision faults in static analyzers.

SolidiFI [97] evaluates the performance of smart contract analysis tools by injecting seven types of well-known code security bugs to generate mutants. As the injected bugs are predefined, SolidiFI can infer the analysis results based on the types of inserted bugs. However, it is not applicable to general static analyzers, as the injected bug types are determined by the evaluated tools, and it is not possible to predict whether these tools can detect the bugs without manual analysis. Furthermore, the injected bugs in SolidiFI are specifically designed for smart contracts. Hu et al. [113] use an automated bug injection approach to construct a baseline dataset for evaluating the effectiveness of the popular symbolic execution engine KLEE. However, their work lacks generality and does not extend to other similar tools.

Parveen et al. [166] propose a mutation-based approach for injecting tainted data flow into

seed applications and evaluating taint analysis tools. They design operators that inject tainted data flow into the original input programs and mutate the relevant program slices. However, these operators are tailored for specific IoT applications and cannot be generalized to other types of applications. For example, they mutate string literals in APIs like *sendPush*.

In contrast to these bug injection approaches targeting domain-specific applications, our approach focuses on general-purpose static analyzers and can effectively detect potential faults.

## 2.2 Compiler Testing

Since compiler front-ends, just like static analyzers, also need to statically analyze the programs to compile and report the detected flaws, techniques for testing static analyzers and compilers often share common purposes, designs, and properties.

### 2.2.1 Differential Testing

Differential testing for compilers generally requires at least two compilers designed and implemented based on equivalent specifications. The compiled results from these compilers are then inspected to identify potential bugs. In summary, existing research methodologies can be categorized into three approaches: cross-compiler, cross-optimization, and cross-version. The cross-compiler approach detects compiler bugs by performing differential analysis on the results produced by different compilers. This is the most general approach in differential testing for compilers. The cross-optimization approach focuses on detecting compiler bugs by comparing the results produced by different optimization levels of a single compiler. The cross-version approach requires multiple versions of an official compiler.

Sheridan [186] tests a C99 compiler, specifically the PalmSource Cobalt ARM C/C++ em-

bedded cross-compiler, using the cross-compiler strategy. In this study, Sheridan examines the output differences between the compiler under test and existing tools to identify compiler bugs. Specifically, the GNU C Compiler in C99 mode and the ARM ADS assembler are employed as reference tools. The rationale for employing this strategy hinges on three key observations: First, compilers for the same programming language should ideally generate identical outputs for the same input. Second, when a bug is triggered by the input, different compilers rarely exhibit the same bug or produce the same erroneous output, given the disparities in their implementations. Third, if two compilers produce different outputs for the same input, it is likely that one of them contains a bug.

Ofenbeck et al. [161] introduce RandIR, a method for detecting compiler bugs by using random instances of an intermediate representation (IR) as inputs for differential testing. Their study focuses on vanilla Scala code. To use RandIR, users must provide the grammar of the code represented by the IR, which consists of a collection of typed functions and operations. Specifically, RandIR randomly generates these operations and records them in a typed dependency graph. It then translates the generated operations into regular Scala functions. For differential testing, an additional regular Scala program is required, which can be derived either from the typed dependency graph or through another compiler pipeline that converts IRs into Scala functions. In essence, the cross-compiler approach is applied for differential testing.

Yang et al. [211] develop another popular compiler testing tool Csmith, which adopts the cross-compiler strategy to identify bugs in compilers. It generates mutant programs by applying eight features, ensuring both well-formed structure and maximum expressiveness. Mendoza et al. [94] design CsmithEdge that removes some constraints of Csmith, and the generated programs may contain undefined behaviors. They investigate the effects of Csmith's UB-free constraints on finding compiler bugs. However, these differential testing based approaches necessitate a pair of equivalent compilers, such as Clang and GCC, to verify the consistency among them. Consequently, if such conditions are not met, they can only detect faults with more obvious symptoms, such as runtime crashes, thereby imposing

significant limitations on the tested tools.

Differential testing is also extensively applied to Java virtual machines (JVMs) [81,82]. As JVMs from different providers should comply with one specification [147], ClassFuzz [82] adopted the mutation-based approach to generating test cases and verifying the consistency of behaviors across different JVMs. Classming [81] builds upon ClassFuzz, enhancing it by deliberately disrupting variable dependencies to produce more valid seeds and reveal flawed data flow patterns. Similar to the introduced JVM testing approaches, our research also generates Java programs. However, unlike these approaches, which use various JVM implementations for differential testing, we apply metamorphic testing to static analyzers, with our test generation heuristics designed to detect bugs in static analyzers rather than JVM implementations.

Le et al. [143] introduce Proteus, a tool designed to evaluate the link-time optimization (LTO) of compilers using a cross-optimization approach. Proteus employs three distinct compilation methods to assess the compiler's behavior. First, the test program is compiled directly by the target compiler without enabling LTO. Next, the same program is compiled with LTO, and additional optimizations are activated. Finally, the test program is divided into multiple compilation units, each compiled separately with various optimizations, and then linked with LTO enabled. Consistency across the results of these three methods is expected; any discrepancies indicate the presence of a compiler bug.

Liu et al. [151] implement Tzer which can set different values of optimization levels, and compare the results of a single tensor compiler to find inconsistency bugs. Kitaura [129] leverages differential testing to identify performance degradation. They propose a hybrid approach, combining static and dynamic based code comparison strategies. In the static step, it differentiates the assembly codes generated from given test programs under two different compilers or versions to detect an inconsistency bug. It then reduces the test program to isolate the difference. In the dynamic step, it runs the codes produced from the reduced test programs under two different compilers or versions to compare their actual execution time. Sharma et al. [183] propose RustSmith, the first Rust differential testing

tool for end-to-end unveiling bugs of Rust compilers. RustSmith constructs valid mutants that not only adhere to Rust's advanced type system but also respect borrowing and lifetime rules, thereby guaranteeing well-defined outputs. Consequently, RustSmith is highly well-suited for differential testing among compilers and optimization levels. When applied to multiple versions of the official Rust compiler, RustSmith can uncover elusive historical bugs that remained undetected for a long period. Although these approaches are effective in testing compilers, static analyzers have completely different configurations with compilers. Hence, the cross-optimization based testing approach cannot be applied to static analyzers directly.

### 2.2.2 Metamorphic Testing

Metamorphic testing [80] is another popular approach for addressing the test-oracle problem. The core challenge to implement metamorphic testing is constructing metamorphic relations, which specify how particular changes to the input of the project under test would change the output. For instance, when testing the sine function, it is too hard to infer the expected value of $sin(1)$. However, the mathematical property of the sine function, i.e., $sin(x) = sin(2\pi + x)$, can help test the implementations of the sine function. In essence, we can test whether $sin(1) = sin(2\pi + 1)$ to perform the testing of the sine function. There are two common types of metamorphic relations in the compiler testing domain: (1) general equivalence and (2) equivalence under a specific set of testing inputs. Many research methodologies rely on the general equivalent relation as the metamorphic relation for addressing the lack of oracle problem in compiler testing [92, 155, 198].

Donaldson and Lascu [93] pioneer the application of metamorphic testing in the evaluation of OpenGL compilers. Their research introduces a novel approach wherein dead code is strategically injected into existing test programs to generate functionally equivalent program variants. According to their methodology, both the original test program and its derived equivalents are expected to yield identical results. Any divergence in the outcomes

thus serves as a definitive indicator of a compiler bug. Subsequently, Donaldson et al. [92] propose a metamorphic testing based approach called GLFuzz to mining faults in OpenGL compilers. This approach adopts domain-specific program transformations to automatically construct families of semantically equivalent mutant shaders from existing valuable seed shader programs. While the authors conduct various effectiveness experiments over seven GPU providers the effectiveness of their technique by detecting more than 60 shader compiler faults, GLFuzz is specifically designed for GPU programs, unable to apply to static analyzers.

Nakamura and Ishiura [155] develop a testing framework Orange4 that generates mutant programs through equivalent transformations to test C programming language compilers. However, the designed program transformations are not systematic, and the authors did not discuss the effectiveness of Orange4, e.g., the number of detected bugs. Similar to these approaches, we also rely on equivalent relation (i.e., applying semantics-preserving program transformations, and checking for differential analysis results of transformed programs) to address the lack of an oracle problem, but our set of transformations is more diverse (e.g., the class-level transformations), which helps to test the effectiveness of analyzers in analyzing programs with different AST structures. Moreover, we have designed techniques to reduce locations to be transformed and select variants that are more likely to represent distinct bugs in static analyzers.

Tao et al. [198] introduce Mettoc, a metamorphic testing framework designed to evaluate compilers by leveraging the concept of equivalence-preservation as the metamorphic relation. Mettoc systematically generates pairs of equivalent test programs, which are subsequently compiled using the target compiler to produce executable binaries. Discrepancies in the output results of these executables indicate the presence of a compiler bug. To generate equivalent test programs, Mettoc employs techniques such as constructing equivalent expressions, assignment blocks, and submodules. Here, the term “assignment block” denotes a compound statement comprising a sequence of assignments, while “submodule” refers to a compound statement that may include conditional structures. Specifically, Met-

toc begins by constructing a general control flow graph, where each block node represents a submodule. It then populates each block node with equivalent statements or expressions. Finally, Mettoc traverses the populated graph to synthesize the equivalent test programs. Although Mettoc can efficiently test compilers, the

The most representative metamorphic testing approach related to the equivalence under a specific set of testing inputs is equivalence modulo inputs (EMI) [141, 142]. The key motivation of EMI is to leverage the interplay between dynamically running a program $P$ on a subset of inputs and compiling the program to work on the entire input data. To clarify, given a program $p$ and a set of input values $I$ from the input space, the input set $I$ produces a natural set of programs $C$ such that every program $q \in C$ is equivalent to $p$ modulo $I : \forall i \in I, Q(i) = P(i)$. Then, the set $C$ can be used to perform differential testing of any compiler $Comp$ : If $Comp(P)(i) \neq Comp(Q)(i)$ for $i \in I$ and $Q \in C$, $Comp$ has a potential bug.

Le et al. [141] propose the first EMI-based research work to test C programming language compilers. Orion [141] generates the variant programs by removing the dead code segments under a specific sect of test inputs. Athena [142] not only removes the dead regions but also inserts code into unreachable code regions, consequently producing more diverse mutants to reveal potential bugs. It also adopts a statistical learning based algorithm to optimize the mutant generation process when performing the transformations. However, these two implemented EMI testing tools usually need dynamic analysis (e.g., profilers) of input programs and rely on the specific input test data, contradicted with the rationale of static analysis techniques. Besides, our approach can generate variant programs statically, independent of any execution information. Besides, they can only transform part of the source code, while our approach can directly mutate any valid program element in the whole input program.

Samet [178–180] introduces a novel compiler testing methodology that leverages metamorphic testing to detect compiler bugs by assessing the equivalence between a source program and its compiled object program. This approach relies on intermediate repre-

sentations (IRs) to establish the metamorphic relations needed to identify discrepancies indicative of potential bugs. Specifically, the methodology follows a multi-step process: First, the source program is translated into an IR to capture its semantic essence. Then, the object program generated by the compiler under test is also converted into an IR, ensuring both representations can be directly compared. Finally, the equivalence between these two IRs is rigorously verified through symbolic interpretation, which allows for precise analysis of program semantics. This method is particularly innovative in its use of metamorphic relations, which are specifically designed to mutate high-level programs while preserving their semantic integrity. By comparing the execution results of the original and mutated programs, the framework can effectively uncover erroneous compilations that might otherwise go undetected. Notably, the symbolic interpretation process used to derive the IR of the object program is crucial for maintaining the fidelity of the comparison, ensuring that any detected differences reflect genuine compiler bugs rather than artifacts of the testing process. Despite this, Samet's approach represents a significant advancement in the systematic evaluation of compiler correctness, it faces limitations when dealing with the diverse properties described by static analyzers. The linear IR cannot adequately capture their semantics. For example, graphical structures are typically used to characterize the properties of function calling chains and control flow.

# Chapter 3

# STATFIER: Testing Static Analyzers via Semantics-Preserving Program Transformations

Within this chapter, we present the STATFIER technique, a novel approach designed to proficiently identify faults in static analyzers. Our technique accomplishes this by performing semantics-preserving transformations on existing test suites and subsequently comparing the analyzer's reports generated for these variant programs. When we refer to "semantics-preserving", we emphasize that these transformations do not alter the program's behavior.

We identify two challenges in the automated testing of static analyzers: (**C1**) the lack of oracles for automated testing; (**C2**) the automated preparation of high-quality input programs. To address the first challenge, i.e., the test oracle problem, STATFIER employs metamorphic testing. Metamorphic testing leverages metamorphic relations, i.e., relationships between the expected outputs from multiple system executions, to determine whether the systems have executed correctly, and it has been successfully applied to verify compilers [92,141,198], machine learning systems [91,210,215], and interactive debuggers [200]. Specifically, given an initial input program for a static analyzer, STATFIER iteratively ap-

plies semantics-preserving transformations to the program to generate a group of variant programs, runs the static analyzer on both the input program and the relevant variant programs, and compares the produced analysis reports to determine whether the static analyzer contains any faults. The rationale here is that, since the transformations are semantics-preserving, the input and variant programs should have the same behavior, and the analysis reports produced by the static analyzer on the programs should also be equivalent. Therefore, if the analysis reports for a pair of input and variant programs differ, an issue has been found in the static analyzer. Furthermore, non-semantics-preserving transformations will produce variant programs that are distinct from the input programs, almost inevitably violating the metamorphic relation on which Statfier is built. Based on this property, our transformations help ensure that the semantic errors in the original programs are preserved after variant generation.

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

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

We implement the Statfier technique into a tool with the same name. The tool supports

12 types of semantics-preserving program transformations gathered from existing literature [78, 90, 92, 146, 155, 192], and it reports a fault to the user if the results produced by a static analyzer on an input program and a corresponding variant program turn out to be different. To empirically evaluate the effectiveness of Statfier and its heuristics, we apply Statfier to detect faults in five popular static analyzers: PMD, SpotBugs, Infer, SonarQube, and CheckStyle. PMD [14] is a cross-language static analyzer that finds common programming falws (e.g., unused variables). Rules in PMD are written either in Java or XPath. SpotBugs [17] is a fork of FindBugs (which is now deprecated) that detects common falws in Java code via a set of patterns; SonarQube [16] is a platform developed by SonarSource for continuous code inspection; CheckStyle [1] is used for checking if Java code adheres to a coding standard; Infer [7] is a static analyzer designed by Meta that detects flaws for Java, C, C++, and Objective-C programs.

We select these tools because they (1) are widely used open-source tools for Java programs, (2) have been used in prior studies on static analyzers [70, 71, 83, 108, 120, 187], and (3) are representative of static analyzers adopted for different purposes (e.g., SonarQube supports automated code review, whereas CheckStyle checks coding style) and by different companies (e.g., SpotBugs is used in Google [70], whereas Infer is used in Meta [12]). Statfier successfully detected 79 faults, of which 46 have been confirmed by the developers of the corresponding static analyzers, and 26 have been fixed by the developers. Moreover, the variant programs generated by Statfier for detecting 26 faults have been incorporated into the official test suites of the analyzers. The experimental results suggest that both heuristics are essential for the effectiveness of Statfier.

Overall, this work makes the following contributions:

- We propose the Statfier technique to effectively enhance the fault detection capability of test suites in static analyzers by generating new input programs using semantics-preserving transformations and metamorphic testing. To improve the effectiveness of test generation, Statfier employs two novel heuristics: analysis report guided location selection and structure diversity driven variant selection;

- We empirically evaluate and confirm the appropriateness of using existing test input programs to drive automated testing of static analyzers with Statfier. Input programs from the official test suites and examples in documentation of static analyzers can activate the checking of a high percentage of rules (e.g., up to 100% for PMD and CheckStyle) during testing, and they are small in size;

- We implement the Statfier technique into a tool with the same name and experimentally evaluate the tool by applying it to detect faults in five popular static analyzers. The experimental results show that both Statfier and its heuristics are effective.

## 3.1 Illustrative Example

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

**Definition 1** (Structurally Diverse)**.** We construct a new test program $P'$ by applying a sequence of transformations $T_1, \ldots, T_i$ to the input program with program contexts $C_1, \ldots, C_i$ (a program context $C$ denotes the type and location information of program element to be transformed) where $P'$ is represented by $\{C_1 : T_1, \ldots, C_i : T_i\}$. We consider $P'$ to be a *structurally diverse* program if the sequence $\{C_1 : T_1, \ldots, C_i : T_i\}$ applied to the original program $P$ to generate $P'$ is distinct among all generated variants, namely for any pair $C_i : T_i$ and $C_j : T_j$ $(i \neq j)$ in the sequence are different.

**Definition 2** (Differential Analysis Results)**.** Given a program $P$, and a variant program $P'$ (where $P'$ is obtained via a semantics-preserving transformation to $P$), we consider a static analyzer $S$ generates *differential analysis results* if $o_p$=invoke($S$, $P$) and $o_{p'}$=invoke($S$, $P'$) where $|o_p| \neq |o_{p'}|$. Here, $|o_p|$ represents the type and number of warnings in the report generated by running $P$ on a static analyzer.

We illustrate the general workflow of Statfier using an example fault found by Statfier in PMD. Given a static analyzer under test $S$, Statfier first extracts input programs from

either the test suite of $S$ or the documentation of $S$. Listing 3.1 shows a program $P$ extracted from the official test cases in PMD [45]. PMD developers designed the program $P$ to test the "HardCodedCryptoKey" rule (i.e., checks if hardcoded values are used for cryptographic keys).

**Prior Approaches.** Existing approaches that test static analyzers [85, 196, 208] fail to find this fault because they rely on either static analysis rule pairs [208] or specialized oracles [85, 196]. Specifically, the "HardCodedCryptoKey" rule cannot be tested by a prior approach [208] that relies on differential testing of static analysis rule pairs, because only PMD supports this rule, while SonarQube does not have a matching rule [15]. Meanwhile, an approach that randomly generates and selects variants [85] wastes time in evaluating variants that do not help discover new faults in static analyzers.

To address the limitations of prior approaches, we propose two key heuristics, including (1) analysis report guided location selection (AL) and (2) structurally diverse variant selection (SS), which we will explain using the example in Listing 3.1.

**Analysis Report Guided Location Selection (AL).** Given an input program $P$ for a specific rule (e.g., "*HardCodedCryptoKey*" checked by PMD), Statfier invokes $S$ to generate an analysis report for $P$. Each analysis report includes the location where the rule violation occurs (e.g., line 4 in Listing 3.1). After obtaining the line $L$ in which the violation occurs and all statements within the backward slice of $L$ (no other statement is data- or control-dependent on $L$ in this example), Statfier systematically explores all semantics-preserving program transformations that are valid at line $L$ (i.e., those that fulfill the prerequisite for a given transformation). Listing 3.2 shows one such valid transformation (program $P'$) that modifies line 4 in Listing 3.1 by extracting a local variable *str*. For each variant program that reports the same rule violation as the original program, we keep it in a queue for further modifications. In this example, program $P'$ causes the same rule violation as the one reported in $P$, so Statfier further modifies it to $P''$ by moving the assignment at line 4 (Listing 3.3). Since Statfier only introduces semantics-preserving program transformations, the static analyzer $S$ should report the same rule violation for Listing 3.3. However,

Listing 3.1: Original input program $P$ triggering a *Hardcoded-CryptoKey* warning in PMD

```
1 import javax.crypto.spec.SecretKeySpec;
2 public class Foo {
3  void encrypt() {
4   SecretKeySpec keySpec =
5   new SecretKeySpec("Hardcoded Crypto Key".getBytes(),"AES"); // warning
6   }
7 }
```

Listing 3.2: Transformed program $P'$ generated by StatFier

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

Listing 3.3: Generated program by transforming $P'$ to $P''$

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

when STATFIER runs PMD for $P''$, it reports differential analysis results, so we consider the missing rule violation as a false negative (FN). We reported this fault to PMD developers, who confirmed and fixed it. According to the developers' feedback of the submitted issue, the problem occurs because "The rule currently only checks the initializer of the variable. Since the variable *str* is not initialized at all, the rule does not see a problem" [89].

**Structurally Diverse Variant Selection (SS).** As stated in Definition 1, this approach represents each variant as $P' = \{C_1 : T_1, \ldots, C_i : T_i\}$, where $C_1, \ldots, C_i$ refer to the program contexts, and $T_1, \ldots, T_i$ denote the sequence of transformation types applied to the original input program $P$. Specifically, we represent the variant in Listing 3.2 as $\{C_1 : T_1\}$ = {StringLiteral, Line=4, Column=[52,72], "Extract local variable"}. To guide the search toward structurally diverse variants, this approach avoids selecting a new variant $k$ where the context and the selected type of semantics-preserving transformation are the same as in Listing 3.2 (e.g., $C_k$=$C_1$ and $T_k$=$T_1$). Compared to a random approach that searches through 112 variants, the SS approach is more efficient as it can find the same fault in Listing 3.3 after iterating through only 51 variants.

## 3.2 Methodology

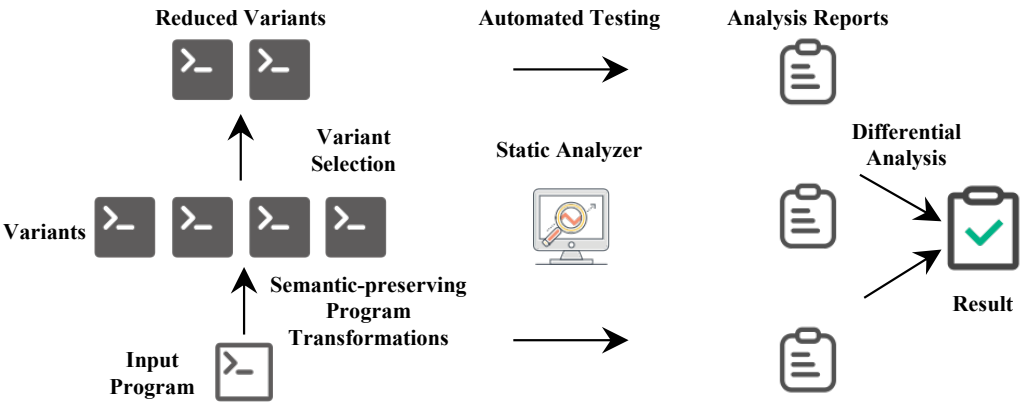


Figure 3.1: Overview of STATFIER

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

**Input:** Input programs *Progs* for a rule $R$ in a static analyzer $S$, a set of transformations *Trans*, timeout *timeL*

**Output:** A set *errP* containing test programs that produce differential analysis results

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

Figure 3.1 provides an overview of StatFier. Given an input program, StatFier applies semantics-preserving transformations and uses variant selection to obtain a reduced set of variants, which are run against the static analyzers to check for differential analysis results. After obtaining variants that produce differential analysis results, StatFier uses them as input programs to trigger faults in static analyzers under test. Algorithm 1 shows our test generation algorithm. Our algorithm consists of four components: (1) variant generation, (2) selection of program locations for transformation, (3) variant selection, and (4) feedback-driven exploration. Given a set of input programs $Progs$ for a rule $R$ in a static analyzer $S$, a set of semantics-preserving transformations $Trans$, and a timeout $timeL$, our test generation algorithm produces a set of programs that indicate faults for the rule $R$ in the static analyzer $S$. Specifically, StatFier initializes the algorithm parameters at lines 1–4 of the algorithm, whereas lines 5–22 constitute the core logic of the algorithm. After initializing the output set $errP$ (line 1), the normal variant set $normP$ (line 2), and a set $disSeq$ for variant selection (line 3), StatFier retrieves the first element of $Q$ as the program $currP$ to be transformed (line 8), and runs a static analyzer on $currP$ to obtain the number, type, and location of warnings (line 9). After that, it performs backward slicing to obtain all related locations $locs$ and gets the corresponding node type $nodeType$ at lines 10–12. Finally, it iteratively applies each transformation $t$ in $Trans$ to $loc$ and performs structure diversity driven variant selection at lines 13–16. At lines 17–20, the isDifferential function compares the analysis reports of $currP$ and $newP$ to recognize inconsistencies as the potential faults. StatFier requires manual analysis of the detected inconsistencies to classify the fault types. If no faults are found, $newP$ is added into $Q$ (line 21). Note that, instead of generating an extensive test suite to check all rules in a static analyzer, our test generation algorithm produces test programs for each rule separately as each rule has its own set of input programs. Mixing input programs from different rules would make it difficult to isolate the fault.

Table 3.1: Rule coverage of input programs for all evaluated static analyzers

| Static Analyzer | Tests within the test suite | | Documentation | | Combined Rule |
|---|---|---|---|---|---|
| | $\frac{\#of\ covered\ rules}{Total\ \#\ of\ rules}$ | Rule Coverage (%) | $\frac{\#of\ covered\ rules}{Total\ \#\ of\ rules}$ | Rule Coverage (%) | Coverage (%) |
| PMD | $\frac{274}{274}$ | 100% | $\frac{273}{274}$ | 99.6% | 100% |
| SpotBugs | $\frac{360}{458}$ | 78.6% | $\frac{52}{458}$ | 11.4% | 78.6% |
| SonarQube | $\frac{581}{617}$ | 94.2% | $\frac{587}{617}$ | 95.1% | 95.3% |
| CheckStyle | $\frac{183}{183}$ | 100% | $\frac{159}{183}$ | 86.9% | 100% |
| Infer | $\frac{67}{92}$ | 72.8% | $\frac{30}{92}$ | 32.6% | 77.2% |
| Total | $\frac{1465}{1624}$ | 90.2% | $\frac{1101}{1624}$ | 67.8% | 92.2% |

### 3.2.1 Selection of Input Programs

As stated in **C2**, before testing static analyzers, we need to obtain input programs with high rule coverage to serve as the initial set for transformation. Static analyzers rely on rule checkers to detect faults in a given input program. To help users understand the reported rule violations, static analyzers typically include code examples that explain the problematic code. Hence, our key insight to solve **C2** is *the input programs designed by the developers of static analyzers can address the challenge of constructing input programs which can trigger more rule violations*. Based on this insight, we extract input programs from two sources: (1) the official test suite and (2) code examples of each rule given in the documentation of the static analyzer. When extracting input programs from the provided test suite, we first obtain the folder storing the input programs of a static analyzer (e.g., src/test/resources/net/sourceforge/pmd/lang/java/rules for PMD), and then automatically crawl the directories to obtain the input programs for each rule. To reuse code examples in the documentation, we manually create complete input programs from the code snippets in documentation by adding the missing (1) variable/method/class declarations and (2) import statements. Our goal is to obtain as many input programs as possible to maximize *rule coverage* (the percentage of rules checked by a static analyzer that contains

at least one input program) and to test different behaviors of the covered rule checker. After obtaining the input programs, we run them against the static analyzer (from which the input programs originated) to measure the quality of these initial input programs. Furthermore, 86% of the extracted input programs have more than one violation, 8% have one violation, and the remaining 6% have no violations.

Since the input programs are from two sources, i.e., the official documentation and test suites, we study the rule coverage achieved by input programs from different sources both separately and as a whole. Particularly, we manually analyzed the official documentation of each static analyzer and counted the rules mentioned in the documentation as the total number of rules supported by each tool. Meanwhile, we ran each static analyzer on all the input programs extracted for it and considered a rule to be activated, or covered, if and only if a flaw was reported because a violation of the rule was detected in an input program. Table 3.1 shows the rule coverage information for all evaluated static analyzers. The second and the third columns denote the ratio of covered rules and the rule coverage given by the input programs within the test suite, whereas the fourth and fifth columns show similar measurements given by the documentation of each rule. Given the set $T$ of the rules covered by the test suite and the set $D$ of the rule covered by the documentation, the "Combined Rule Coverage (%)" column presents the overall rule percentage of $T \cup D$.

Instead of obtaining input programs from a test suite, an alternative approach is to extract them from real-world projects. However, a prior evaluation [208] that uses input programs from 2,728 projects shows that this alternative approach can only achieve 74%–89% rule coverage for four static analyzers. Another alternative is to adopt Defects4J [122], a widely used dataset with faulty Java programs, while prior studies [109] reveals that the rule coverage achieved by static analyzers in Defects4J is only around 2%–4%. Compared to these alternatives, our study shows that *reusing programs from official test suites of static analyzers can cover more rules and increase coverage*. As shown in the third column of Table 3.1, rule coverage achieved by the test suites of static analyzers ranges from 72.8%–100%, specifically all rules in PMD and CheckStyle can be covered. Our study

of the test suite of analyzers also gives evidence for the hypothesis that most bugs have small counter-examples [114] because we observed that test input programs in analyzers are small, i.e., they have an average of 1.25 classes (max=30), 3.16 methods (max=106), 1.33 fields (max=1000), and 49.42 lines of code (max=65544).

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

### 3.2.2 Variant Generation via Program Transformations

Program transformations have been successfully applied to enhance many software engineering tasks, including producing simulated source code plagiarism [78], improving testability (i.e., transforming a program to make it easier for a given test generation method to generate test data) [110, 111], enhancing the generalizability of neural program models [172], and testing refactoring engines [90]. Given an input program $P$, STATFIER produces variants by applying a set of program transformations (line 16 in Algorithm 1), and these transformations should be semantics-preserving. To obtain a comprehensive set of transformations, we refer to existing literature on semantics-preserving program transformations [78, 90, 92, 146, 155, 192]. We use the following design principles when selecting our set of program transformations:

**Multi-Level Transformations.** Based on the program elements to be transformed, we divide the space of transformations into five levels: variable, expression, statement, method, and class. We adopt the transformations from existing literature [78, 90, 146, 192] for the first four levels (i.e., variable, expression, statement, method). Since no prior approaches focus on class-level semantics-preserving program transformations, we refer to GitHub

Table 3.2: Semantic-preserving program transformations supported in STATFIER

| Level | Transformation | Example | Source |
|---|---|---|---|
| Variable-level | Extract local variable | `- invokeMethod("StringLiteral");`<br>`+ String str="StringLiteral";`<br>`+ invokeMethod(str);` | [78, 155] |
| | Move assignment | `- int var=10;`<br>`+ int var;`<br>`+ var=10;` | [78] |
| Expression-level | Equivalent boolean expression:<br>Add $\|\|false$ or $\&\&true$ expression<br>Swap symmetrical elements, e.g., $a == b \rightarrow b == a$ | `- boolean tag=true;`<br>`+ boolean tag=true\|\|false;` | [92, 99] |
| | Equivalent arithmetic expression:<br>Add +0, -0, or +1-1 expression | `- int value=10;`<br>`+ int value=10 + 0;` | [66, 92] |
| | Add parenthesis to expression | `- int value=10;`<br>`+ int value=(10);` | [78, 90] |
| Statement-level | Equivalent statement conversion:<br>Convert to equivalent for/while/do-while/lambda | `- for(i = 0; i<1; i++) {}`<br>`+ i = 0;`<br>`+ while(i++ < 1) {}` | [78] |
| | Statement wrapping:<br>Wrap statements with if/while/for/do-while | `- target_statement;`<br>`+ if(true) { target_statement; }` | [92, 192] |
| | Dead code injection:<br>Insert dead if/while/for statement | `  target_statement;`<br>`+ for(int i=0; i<0; ) {`<br>`+  target_statement;`<br>`+ }` | [146, 155] |
| Method-level | Encapsulate field | `- SecretKeySpec("Hardcode");`<br>`+ String getHardcode() {`<br>`+  return "Hardcode";`<br>`+ }`<br>`+ SecretKeySpec(getHardcode());` | [78, 90] |
| Class-level | Nested class wrapping | `- original_program;`<br>`+ class NestClass {`<br>`+  original_program;`<br>`+ }` | [86] |
| | Anonymous class wrapping | `- original_program;`<br>`+ Object c = new Object() {`<br>`+  original_program;`<br>`+ };` | [87] |
| | Enum wrapping | `- original_program;`<br>`+ enum enumClass {`<br>`+  original_program;`<br>`+ }` | [8] |

issues of static analyzers to derive the class-level transformations [8, 86, 87].

**Dead and Live Code Injections.** We include two types of transformations: (1) transformations that inject dead code into the original program (e.g. "Unreachable code injection") [146, 192] and (2) transformations that inject live code into the original program (e.g., "Statement wrapping") [92, 155].

**Incorporating Analysis Capability of Static Analyzers.** Considering the trade-off between accuracy and efficiency, static analyzers may support different levels of analysis capabilities for producing more precise or faster analysis. For example, Infer and SonarQube support inter-procedural analysis, while PMD and SpotBugs only support intra-procedural analysis. For analyzers that do not support inter-procedural analysis, variants generated through method-level program transformation (i.e., *Encapsulate field*) are not meaningful as detecting these variants exceeds their analysis capability. Our design integrates the analysis capability of each studied analyzer by excluding transformations that are beyond its analysis capability.

**No Style-Related Transformations.** We exclude program transformations that involve changes in coding style (e.g., changing comments or identifier names) because (1) these transformations may trigger inaccurate differential analysis results in tools like CheckStyle that check for coding standards, and (2) transformations such as renaming requires encoding Java naming conventions into the transformation rules, which is beyond the scope of this work. Specifically, we exclude all transformations in "Level 1–Changes to Comments&Indentation" and "Level 2–Changes to Identifiers" from prior work [78].

Table 3.2 shows the 12 types of semantics-preserving program transformations supported in Statfier, each with an example. The "Source" column indicates the relevant work from which we derived the corresponding transformation.

In the context of static analyzers, randomized program transformation has been used for testing [85]. Meanwhile, a prior work [156] has formalized the impact of program transformations on the results of static analysis by setting up a mathematical framework. Al-

though we derive our set of semantics-preserving transformations from existing literature [78, 90, 155], our set of transformations is more comprehensive because it includes transformations of program elements at different levels, and our set of transformations is specifically used for testing static analyzers rather than for other applications. The most closely related to our transformation mechanism is the recent work [204] which uses program transformations to resolve false positives of static analyzers. Our approach differs from prior work [204] in several aspects: (1) we generate input programs for testing static analyzers, and file analysis reports to indicate bugs, whereas prior work improves static analyzers from the perspective of the analysis user (no reports are filed as a result); (2) we use different transformations (we use semantics-preserving transformations that are general instead of using rewrite templates that are tool-specific in prior work [204]); (3) our research can identify various types of faults (including false negatives, false positives, and runtime crashes) to improve the overall reliability of static analyzers, whereas prior work primarily focuses on resolving false positives.

### 3.2.3 Heuristic-Based Testing Process

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

**Analysis Report Guided Location Selection.** The program locations in which we choose to apply the transformations affect the effectiveness of variant generation and selection. Our main insight is that *we can select program locations for transformations based on the locations listed in the analysis reports generated by static analyzers*. Specifically, given an input program $P$, Statfier applies a static analyzer to $P$ to obtain (1) the number of reported warnings and (2) the program locations where the analyzer reports violations (line

9 in Algorithm 1). Instead of relying solely on the reported program locations which may be incomplete, our goal is to obtain the set of locations that have either control or data dependencies with respect to each program location stated in the analysis report. Hence, after executing the program against the static analyzer under test, STATFIER obtains the backward slice of the program starting from each program location stated in the analysis report. Specifically, the GETBACKWARDSLICE(*currP*, $S$, $loc_b$) function takes as input (1) the input program *currP*, (2) the static analyzer under test $S$, and (3) the program locations $loc_b$ for the reported violations to produce a set of locations *locs* that include all locations stated in the analysis report, together with their backward slices (line 10 in Algorithm 1). This step produces a set of program locations where we apply the set of semantics-preserving transformations. Since we use an analysis report to guide the selection of program locations, we refer to this step *analysis report guided location selection*.

**Structure Diversity Driven Variant Selection.** Given that STATFIER can easily generate many more variants than can be checked in a reasonable amount of time, we propose structure diversity driven variant selection to prioritize variants based on the heuristic that variants with unique structures are more likely to trigger distinct faults and cover overlooked corner cases. This heuristic is based on the rationale that, static analyzers detect flaws by looking for specific syntactic patterns, hence structurally diversified variants are more likely to be matched with different patterns than the structurally similar ones. The rationale can be exemplified by real-world GitHub issues. For instance, PMD issues #5046 [98] and #5049 [96] were marked as "duplicate" by developers because both involve *try-catch* statements analyzed by the same rule checker.

Specifically, given an original program $P$, we represent each variant $P' = \{C_1 : T_1, \ldots, C_i : T_i\}$ by checking (1) the program context $C_1, \ldots, C_i$ under which each transformation has been applied to, and (2) the transformation types $T_1, \ldots, T_i$ that have been applied starting from $P$. Given a variant program location *loc*, we determine the program context by checking the AST node type as follows: (1) if the AST node at *loc* is a leaf node, we use its AST node type (e.g., operand) as the program context; (2) if the AST node at *loc* is a non-leaf

node, we use the AST node type of the root of the subtree (e.g., if-statement) to represent the program context.

As shown at lines 11–16, STATFIER checks if the current program context (represented by $nodeType$) and transformation type (represented by $t$) exist in the previously encountered sequence $disSeq$. This selection aims to eliminate variants with the same program context and use the same transformation type as previously chosen variants. Subsequently, the selected variants are structurally diverse (see Definition 1).

**Feedback-Driven Exploration.** The test generation algorithm of RANDOOP avoids extending illegal method sequences (e.g., those that lead to exceptional behavior) based on the feedback obtained from executing test inputs during test generation [164]. Inspired by RANDOOP's test generation algorithm, we use a feedback-driven approach for our test generation by avoiding further exploration of input programs that lead to differential analysis results (see Definition 2), essentially incorporating feedback obtained from running a static analyzer. At lines 17–21 in Algorithm 1, we store the newly generated variant program *newP* in the *normP* set for further extension of *legal program sequences* (those that do not lead to differential analysis results) and return all programs in the *errP* set, which contains programs that indicate faults in a static analyzer.

STATFIER inspects differential analysis results in the isDIFFERENTIAL function in Algorithm 1. The function (1) runs the variant program on the given static analyzer (line 24), and (2) compares the number and type of warnings before and after applying the transformation and returns true if the number of warnings differs (line 25). In general, the differential analysis results represent two cases: (1) a false negative (FN) in $S$ if the original program $P$ produces a warning in $S$ but the variant program $P'$ does not give a warning (a warning is missing), and (2) a false positive (FP) in $S$ if the original program $P$ does not produce a warning but the variant program $P'$ leads to a warning (indicating a false warning). Furthermore, we add a filter that checks for newly added types of reported warnings to remove false positives, as stated in Section 3.3.2.

## 3.3 Evaluation

We applied STATFIER to five static analyzers, including PMD, SpotBugs, CheckStyle, SonarQube, and Infer, to address the following research questions:

**RQ 3.1:** How many unique faults can STATFIER detect, and what are the characteristics of these faults?

**RQ 3.2:** Can the proposed heuristics in STATFIER reduce the number of variants while preserving its fault-finding capability?

**RQ 3.3:** How many faults can each transformation detect?

### 3.3.1 Experimental Setup

We implement semantics-preserving program transformations and backward slicing using the Eclipse JDT library with over 7000 lines of Java code. We do not use existing slicers because (1) using the same library for program transformations and slicing can simplify traversing and matching the AST, and (2) the Eclipse JDT library is widely used for semantics-preserving transformations (e.g., it supports refactoring operations [2]). Specifically, STATFIER constructs the program dependency graph based on Eclipse JDT and extracts the control and data dependency information. Since most static analyzers cannot perform inter-procedural analysis well, we construct only intra-procedural backward slicing without alias analysis. Based on the appropriate values for parameters selected through our experiment in Figure 3.2, we set the time limit *timeL* of each static analyzer to six hours (which includes transforming and checking all input programs for an analyzer) for running STATFIER in RQ 3.1 and RQ 3.2. For each rule in each static analyzer, we reuse the configuration specified in the test case if available; otherwise, we use the default configuration recommended by the static analyzer. For static analyzers that require compilation (e.g., SpotBugs), we compile each program using Oracle JDK 8 and JDK 11. All experiments

were conducted on a machine with an Intel Xeon(R) 6134 CPU (3.20GHz) and 192GB RAM.

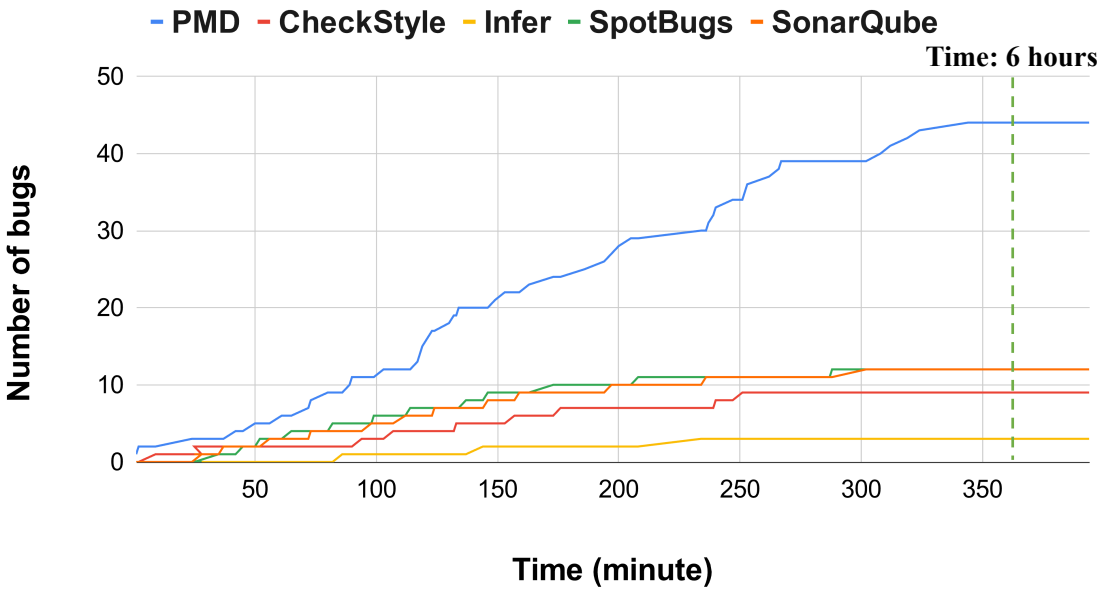


Figure 3.2: Effects of execution time on the number of found faults by each static analyzer

### 3.3.2 RQ 3.1: Assessing Effectiveness of STATFIER

We use the number of discovered unique faults to evaluate the effectiveness of proposed approaches. Notably, we consider a fault to be a ***unique fault*** if it is in (1) different rule checkers triggered by various transformations, (2) different faulty locations (determined by root cause diagnosis) in a static analyzer, and (3) not the nine false positives discussed in Section 3.3.3. We use this definition of *unique fault* because developers of static analyzers adopt a similar definition when checking for duplicate faults [5, 6], and usually repair faults for different rules in different program locations for their corresponding rule checkers. Specifically, we adopt the type of rule checker and the applied transformation to automatically cluster variants which lead to discrepant analysis results. We consider variants in the same cluster as equivalent and randomly pick one representative variant from

each cluster. After automated clustering differential analysis results based on our definition of a unique fault, STATFIER can find 79 faults that span across all of the evaluated static analyzers.

**Status of Reported Faults.** We have reported the 79 unique faults to developers of static analyzers. Table 3.3 shows the current status of our submitted reports. We classify them into four categories based on the responses that we have received from developers of static analyzers so far. The categories are listed below:

**Fixed:** The fault report was fixed by a merged pull request.

**Confirmed:** The fault was confirmed by the developer, but has not been fixed so far.

**Pending:** We have not received a response from developers.

**Won't fix:** The developer acknowledged that the fault is a limitation of the static analyzer, but will not fix it.

Table 3.3: Status of submitted fault reports

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

Among all the evaluated static analyzers, we observe that STATFIER finds the most significant number of faults in PMD. This is because PMD has the highest rule coverage with the greatest number of input programs (as shown in Table 3.1). With more input programs, our approach can further transform these programs, making it more likely to discover new

faults in PMD. For each fault found by STATFIER, we manually inspect it to filter FPs, and report to developers only after checking for its validity. As illustrated in Table 3.3, except for "Won't Fix" where developers have decided not to fix the faults, all our reports are perceived positively by developers (i.e., we did not have any rejected fault reports), and nearly half of them have been fixed or confirmed. It is worthwhile to note that STATFIER *generated variants for 26 of the submitted faults have been integrated into official test suites* by the developers of static analyzers, demonstrating the importance of input program generation for testing static analyzers. Moreover, as SonarQube is the only evaluated analyzer that uses Jira for issue tracking, we manually checked the assigned severity of the reported faults. Among the 10 confirmed faults in SonarQube, 7 are marked as major, and 3 are minor.

We also compare STATFIER with a mutation testing based approach [99] that compares the effectiveness of static analyzers. It considers a static analyzer that kills a mutant when the number of warnings or errors increases with mutation and adopts the Universal Mutator (**UM**) [100] to finish the mutation process. As the tool for prior approach [99] is not publicly available, we re-implement it by (1) using the open-source Universal Mutator [100], and (2) reproducing the oracle in their paper (i.e., the number of warnings or errors increases). To ensure a fair comparison, we measure its effectiveness in detecting faults in static analyzers using the same timeout as STATFIER. Table 3.4 demonstrates the comparison result. Among the 472637 mutants generated for all static analyzers, it only finds one real fault in PMD and produces two false positives in SpotBugs. This result indicates that the prior mutation testing approach is less effective than STATFIER in finding faults in static analyzers.

Table 3.4: Number of detected faults

| Approach | PMD | SpotBugs | SonarQube | CheckStyle | Infer | Total |
|---|---|---|---|---|---|---|
| STATFIER | 44 | 12 | 12 | 8 | 3 | 79 |
| UM | 1 | 0 | 0 | 0 | 0 | 1 |

**Limitations.** We notice two limitations of our approach during the manual analysis of the identified faults. Firstly, as common in a testing tool, our approach may produce FPs (the discovered faults are not real faults). For example, when applying the "Statement wrapping" transformations to wrap code with if-statement, SpotBugs may report one more *DB_DUPLICATE_BRANCHES* warning, but this extra warning is triggered by our applied transformation (not a real fault) because this rule detects duplicate if-else branches. However, in our experiment, we only find nine FPs and they are excluded from Table 3.3 and Table 3.6. Our filter can remove these FPs automatically. The second limitation is that the tested static analyzers need to be able to generate an analysis report, and the report should include the warning types and locations. In the future, it is worthwhile to investigate how to extend our technique to support other types of analysis (e.g., we can perform differential analysis on the reports of info-flow analysis between the input program and its variants as their behaviors should be equivalent).

Table 3.5: Statistics for root causes of faults in static analyzers

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

**Root Causes of Detected Faults.** We manually analyze the root causes behind the found faults and summarize them into five categories. Table 3.5 shows the numbers of each root cause for these faults, this table is sorted in descending order by the number of discovered faults. We discuss representative examples of each category below.

**Variable Declaration.** When analyzing variables, rules in static analyzers either (1) only check for direct initialization, or (2) fail to analyze declarations at the global level. For

example, Listing 3.3 shows that PMD fails to detect the hardcoded key of the assignment statement at line 6 as it does not analyze the usage for the *str* variable. StatFier also found FNs due to incomplete analysis of global variables (fields). Figure 3.3 shows an example [3] where Infer fails to report the null pointer dereference for the field at line 6 but can detect the dereference if *color* is a local variable.

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

Figure 3.3: A null dereference fault in Infer

**Complex Class Structure.** Static analyzers may need to retrieve methods or fields within classes, but the retrieval can be incomplete if the given program's class structure is complex, e.g., when there are nested classes. Figure 3.4 shows such an example in SpotBugs for the *MS_EXPOSE_REP* rule (this rule detects a security flaw that occurs when a public static method returns a reference to an array that is part of the static state of the class) [10]. Given the original input program, SpotBugs can detect the rule where the method *faultMethod* leaks the private field *key*. However, after StatFier transforms the program via "Nested class wrapping", the SpotBugs detector "FindReturnRef" for this rule can no longer detect the reference in the nested class. The developer has prepared a fix for the fault soon after reporting.

```
1 public class Bug1397 {
2   private static String[] key;
3 + static class NestedClass {
4     public static String[] faultMethod() {
5       return key; // should report a warning
6     }
7 + }
8 }
```

Figure 3.4: An FN for the rule *MS_EXPOSE_REP* in SpotBugs

**Control Flow Structure.** Our manual analysis shows that programs with complex control

flow structures can lead to unexpected results in static analyzers. For example, the PMD rule *UseStringBufferForStringAppends* recognizes the use of the += operator for appending strings and warns that the operator causes the JVM to use an internal *StringBuffer*, which is inefficient. Figure 3.5 shows an example [52] where PMD reports one warning for this rule at line 3 and two duplicate warnings (warnings that are exactly the same) for the same rule at line 5. Although line 3 and line 5 are equivalent, the duplicate warnings at line 5 show two problems in PMD: (1) an FP caused by the statements, and (2) the failure to filter duplicate warnings. Moreover, our analysis also shows that some rule checkers in static analyzers may fail to support different control flow structures (e.g., SonarQube can detect infinite *for* and *while* loops, but does not consider *do-while* loop [9], and the developer has fixed the fault upon receiving our report).

```
1 public void bar() {
2   String x = "foo";
3   x += "bar" + x; // report one warning
4 + if (false) {
5 +   x += "bar" + x; // report duplicate warnings
6 + }
7 }
```

Figure 3.5: An FP for the PMD rule *UseStringBufferForStringAppends*

**Compound Expression.** When analyzing compound expressions such as parenthesized expression and binary expression, static analyzers like PMD may fail to check the subexpression either due to (1) incomplete AST node representation (the Java AST library in PMD does not model *ParenthesizedExpression* as an AST node type but its JavaScript AST library does not have this problem) or (2) fail to traverse the subexpression. Figure 3.6 shows a false positive example [4] for the PMD rule *RedundantFieldInitializer* that detects a redundant initialization (assigning a field to its default values). The FP occurs because PMD fails to check the subexpression by traversing only the first operand (value `0` is the default value for a *char*) in the binary expression, and mistakenly reports the field *c* to be a redundant initialization.

**Java Version and New Features.** With the release of new Java versions, new faults may

```
1 class A {
2 - char c = 1;
3 + char c = 0 + 1; // should not report a warning
4 }
```

Figure 3.6: An FP for the PMD rule *RedundantFieldInitializer*

occur in static analyzers either due to (1) the differences in the generated Java bytecodes or (2) insufficient support of new language features such as lambda expression and annotation. When communicating with developers of static analyzers, they noted that the fault failed to reproduce in different Java versions, pinpointing the root causes to be the different Java versions used, e.g., *DMI_INVOKING_TOSTRING_ON_ARRAY* rule in SpotBugs can detect the issue shown in Figure 3.7 when we compile with Java 8 but SpotBugs fails to detect the issue when using newer Java versions (Java 11, 16, 17) [11]. In our experiment, we only tested input programs with Java 8 and 11, it is worthwhile to study a differential testing approach that checks the input programs against different Java versions in the future, especially for analyzers like SpotBugs that act on bytecodes. For the new language feature example, the CheckStyle rule *ParameterAssignment* that detects assignment to parameters fails to recognize parameters in the lambda expression *list.forEach((i)→{i*=10;});* [13]. We reported this fault and it has been fixed by the CheckStyle developer.

```
1 + final String[] gargs = new String[] {"1", "2"};
2   public void print() {
3 -   final String[] gargs = new String[]{"1", "2"};
4     System.out.println(""+gargs); //should give a warning
5   }
```

Figure 3.7: An FN caused by Java version in SpotBugs

### 3.3.3 RQ 3.2: Assessing Effectiveness of Heuristics

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

**Random Location (RL):** An approach that randomly selects program locations to transform.

**Analysis Report Guided Location Selection (AL):** An approach that uses analysis reports for selecting program locations (see Section 3.2.3).

**Random Variant Selection (RS):** An approach that randomly selects variants generated via semantics-preserving transformations.

**Structurally Diverse Variant Selection (SS):** An approach that selects structurally diverse variants (see Section 3.2.3).

Although there are several prior approaches that test static analyzers [85, 135, 196, 208], most of them do not generate variants [196, 208] so we exclude them from comparison. To ensure a fair comparison with prior work that uses *Csmith* for generating variants for C programs via random mutations [85], we emulate prior work using the baseline **RL*RS** that reuses the same set of transformations and the same oracle (differential analysis results) as Statfier but randomly selects variants and locations to transform. When generating new variants by **RL**, we remove duplicate variants and set the number of transformations equal to **AL** to ensure fairness. As all approaches (except for Statfier that uses AL*SS) rely on randomized algorithms that may produce different results across different runs, we re-run each randomized approach five times with different random seeds.

Table 3.6 shows the effectiveness of each evaluated approach where each cell represents the (minimum, maximum, median) number of detected faults by the four approaches (note that Statfier that uses AL and SS is deterministic, so we do not need to rerun it five times). We observe that all approaches that use "Random location (RL)" fail to detect any fault in all evaluated static analyzers, including **RL*RS** that emulates prior work [85]. Indeed, as the RL approaches can skip the static analysis steps (i.e., analysis report generation and backward slicing), they perform transformations on many randomly selected program locations rather quickly, causing rapid growth of variants and consuming too much time to analyze all variants. This indicates that *our proposed analysis report guided location*

Table 3.6: Number of detected bugs across five seeds

| Static Analyzer | RL*RS [85] | AL*RS | RL*SS | AL*SS (STATFIER) |
|---|---|---|---|---|
| PMD | (0, 0, 0) | (36, 40, 38) | (0, 0, 0) | 44 |
| SpotBugs | (0, 0, 0) | (11, 12, 12) | (0, 0, 0) | 12 |
| SonarQube | (0, 0, 0) | (11, 12, 12) | (0, 0 ,0) | 12 |
| CheckStyle | (0, 0, 0) | (7, 8, 7) | (0, 0, 0) | 8 |
| Infer | (0, 0, 0) | (3, 3, 3) | (0, 0 ,0) | 3 |
| Total/Avg | (0, 0, 0) | (68, 75, 72) | (0, 0, 0) | 79 |

*selection heuristic plays an essential role in reducing the number of locations* selected for modifications, subsequently guiding the test generation towards producing more valuable variants. Compared to STATFIER that discovers 79 faults, the AL*RS approach that uses random variant selection only finds 68–75 faults across the five runs. As we designed the structure diversity driven variant selection heuristic to avoid evaluating variants that trigger similar faults, *the fact that* STATFIER *outperforms the AL*RS approach shows that this heuristic guides* STATFIER *in finding distinct faults*.

During the test generation, the number of variants can proliferate based on the number of valid transformations. Too many variants take a considerable amount of time. Hence, we propose structurally diverse variant selection and compare this technique with other baseline approaches. As shown in Table 3.6, RL*RS and RL*SS approaches fail to find any fault in static analyzers. Hence, we only compare the number of generated variants between AL*RS and AL*SS. We define *variant selection percentage* below:

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

Based on the above equation, a *lower number for the variant selection percentage denotes a better approach* (with greater selection power) that only needs to evaluate fewer variants (e.g., for all evaluated analyzers, STATFIER needs to run fewer variants compared to AL*RS to find the same number of faults in SpotBugs). Table 3.7 depicts the variant

selection percentage across the five seeds; each cell is of the form "number of generated variants(variant selection percentage)". On average, the AL*SS heuristic in STATFIER selects 40.28%–41.27% variants compared to AL*RS. Based on Table 3.6 and Table 3.7, although the AL*SS heuristic evaluates fewer variants, it still finds more unique faults than AL*RS. Hence, we believe that the *proposed AL*SS heuristic is effective in reducing the number of variants while preserving its fault finding capability*.

Table 3.7: Variant selection percentage for AL*SS versus AL*RS

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

### 3.3.4 RQ 3.3: Assessing Effectiveness of Transformations

We further analyze the effectiveness of each supported semantics-preserving program transformation. Figure 3.8 shows the number of faults found using different types of program transformations. According to the fault distribution, each supported program transformation can find at least one fault in the evaluated analyzers. This means *all of the implemented transformations are effective*. Moreover, we observe that transformations that involve extracting a variable (e.g., "Extract local variable" and "Move assignment") are more likely to find faults in the evaluated analyzers.

**Comparison with Prior Mutation Testing Technique.** Prior work [66] mainly provides evidence on the correspondence between mutations and some types of static warnings. However, it does not focus on detecting and reporting faults in static analyzers. After comparing the transformations used in this work, there are three transformations IOR (Over-

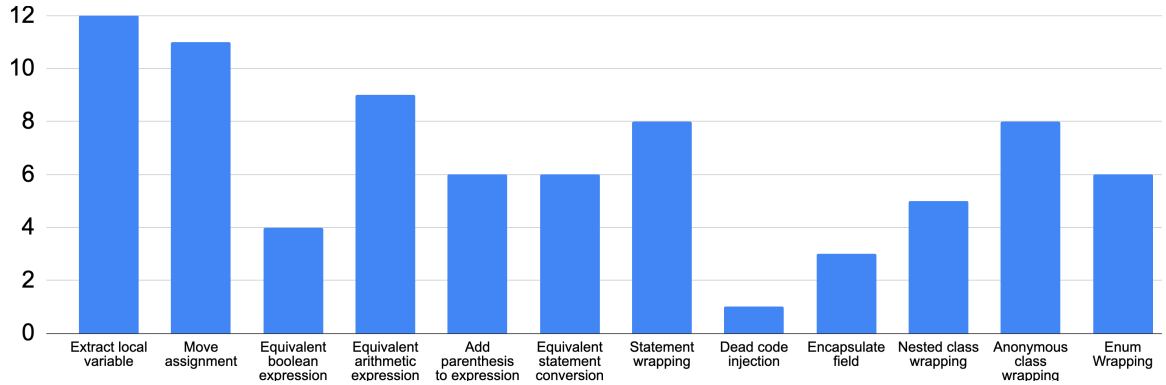


Figure 3.8: Number of faults detected by each transformation in STATFIER

ridden method rename), AOR (Arithmetic operator replacement), and AOI (Arithmetic operator insertion) that are semantics-preserving and related to STATFIER. However, as stated in section 4.2, we exclude style-related transformations like changing identifier names because it may lead to inaccurate differential analysis results. Hence, we do not consider the transformation IOR. The remaining AOI and AOR are similar to the transformation "Equivalent arithmetic expression". Overall, as shown in Figure 3.8, this transformation can find nine faults, which is less than our findings.

**Comparison with Prior Compiler Testing Technique.** In a related prior approach for compiler testing [192], the Hermes tool synthesizes predicates representing known boolean values in control flow statements by executing the input programs to obtain runtime information, whereas STATFIER does not need to execute the input programs which is also the characteristic of static analyzers. Besides, the Hermes tool is input sensitive as it relies on profiling analysis which is determined by the input, whereas our approach only requires static information and is more general. Additionally, we provide more transformations to make variants diverse. As mentioned in the design principle (Section 3.2.2), our design incorporates the analysis capability of static analyzers by excluding transformations that are beyond the analysis capability of existing analyzers. Hence, we adapted the prior compiler testing technique [192] for testing static analyzers by representing it with the "statement wrapping" transformation, which uses literal boolean values as predicates. Figure 3.8

shows that the "statement wrapping" transformation can only detect eight faults in total, which is less effective than Statfier.

## 3.4 Summary

We present Statfier, a heuristic-based testing approach that automatically generates input programs via semantics-preserving program transformations for discovering faults in static analyzers. Statfier relies on two key heuristics: analysis report guided location selection and structure diversity driven variant selection. Our experiments show that Statfier outperforms the evaluated baselines by finding more faults yet iterating through fewer variants. Overall, Statfier has discovered 79 faults, of which 46 have been confirmed. Our results suggest that developers of static analyzers can incorporate our approach into their test suites to further improve the fault detection capability of existing test suites. In fact, 26 of the input programs we generated have been integrated into the official test suites of the evaluated static analyzers. While we focus on faults that lead to differential analysis results in this work, it would be worthwhile to study other faults in static analyzers in the future (e.g., inconsistencies between documentation and the behavior of static analyzers [195]). Another direction is to improve the effectiveness of Statfier by tuning configurations of static analyzers as prior work shows that configurable software such as program verification tools can be tuned [136].

# Chapter 4

# AnnaTester: Understanding and Detecting Annotation-Induced Faults of Static Analyzers

Currently, annotations are widely used in practical software development. According to a previous study on a large number of open-source projects hosted on GitHub, the median number of annotations per project is up to 1,707 [212]. An annotation essentially contains a (possibly empty) list of property-value pairs, where 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 themselves do not affect program semantics, but programs can implement annotation processors to adjust the program's code or even behavior based on specific annotations, thereby effectively extending the capabilities of the Java programming language. For instance, the annotation *@java.lang.Override* triggers a compile-time check for the existence of the method it annotates in the current class's superclass, and an error is issued if the check fails.

Java programs can use *meta-annotations* (i.e., annotations that are applicable to other annotations) to restrict the application and effect of other annotations. For example, the meta-

annotation *@Target* specifies the types of elements (e.g., Class, Method, and Field) to which an annotation can be applied, while meta-annotation *@Retention* determines how long an annotation should be retained during a program's lifecycle:

- **Source retention:** The annotation is not retained after the source code is compiled into bytecode.
- **Class retention:** The annotation is retained in the bytecode but discarded when the bytecode is loaded into the JVM.
- **Runtime retention:** The annotation is always retained and can be retrieved at runtime.

Before delving into the details of research work, we first provide the definition of annotation-induced faults. An Annotation-Induced Fault (AIF) occurs when a static analyzer fails to correctly handle program annotations, leading to incorrect analysis results or unexpected runtime failures. The presence of annotations poses two challenges to the correctness of static analyzers from both syntactic and semantic perspectives. To better understand the impact of annotations on the reliability of static analyzers, we conducted the first large-scale empirical study on annotation-induced faults (AIFs) in static analyzers. For instance, in SONARQUBE-3438 [119], the development team overlooked the potential initialization of annotations through constant identifiers, ultimately resulting in a runtime crash as shown in Figure 4.1.

```
1 final static String STAR = "*";
2 @CrossOrigin(MyClass.STAR)
3 @CrossOrigin(STAR)
```

Figure 4.1: SONARQUBE-3438 [119]: A crash due to an unsupported initialization pattern

Although prior work has studied the usage and evolution of program annotations [152, 159, 176, 212], the maintenance of testing-related annotations [126], the design of annotations for special purposes [77, 197], and the faults for diverse types of software systems (e.g.,

machine learning systems [185, 199], blockchains [207], compilers [193], and static analyzers [149]), research on AIFs in static analyzers remains limited.

Existing work related to AIFs primarily focuses on common code annotation practices in Java [160, 165, 169, 176, 212]. These studies show that annotations are widely adopted by Java developers and have served as motivations for our study. Among these studies, the work on annotation-related faults and the mutation operators that mimic these faults is the most closely related [169]. For example, the insertion of annotations in AnnaTester is similar to the ADA operator, which adds an annotation to a valid target in prior work. However, the ADA operator requires users to manually specify the annotation, whereas we generate annotations from our annotation database. Nevertheless, our study and proposed technique differ from prior studies in several important aspects: (1) Prior studies have identified only two categories of annotation-related faults ("misuse" and "wrong annotation parsing"), whereas our study focuses on the impacts of annotation-induced faults in static analyzers. The root cause categories of annotation-induced faults in our study are more diverse than those in prior studies. (2) Prior techniques focus on mutating code annotations [169] via a set of operators but cannot detect annotation-induced faults due to the lack of oracles. In contrast, AnnaTester injects annotations into input programs and uses a well-designed set of oracles to reveal annotation-induced faults in static analyzers.

Consequently, our study aims to fill this gap by addressing the following research questions:

**RQ 4.1:** What *kinds of annotations* are more likely to trigger AIFs? In RQ 4.1, we examine annotations that require attention when designing static analyzers.

**RQ 4.2:** What are the *root causes* of annotation-induced faults in static analyzers? In RQ 4.2, we study the reasons behind annotation-induced faults to prevent them from recurrence in the future.

**RQ 4.3:** What are the *symptoms* of annotation-induced faults? In RQ 4.3, we investigate the consequences of annotation-induced faults, which helps us assess their significance.

**RQ 4.4:** What are the *fix strategies* that developers employ when fixing AIFs? In RQ 4.4, we strive to establish a comprehensive understanding of viable ways to fix annotation-induced faults, which is essential for reducing debugging efforts.

To address the aforementioned research questions, we manually analyze 238 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 uncover seven main reasons for the annotation-induced faults, identify four symptoms of those faults, and unveil six strategies that developers adopted to fix the faults. We summarize nine major findings from the analysis results and discuss their implications for avoiding similar faults in the future. Based on our findings, we develop a metamorphic testing based framework named AnnaTester to automatically detect three types of annotation-induced faults in static analyzers. On the six aforementioned static analyzers, AnnaTester can successfully detect 43 faults that were revealed for the first time. We have reported the faults to the corresponding developers, and 20 of them have been fixed, which clearly demonstrates the value of our framework, study, and findings.

In summary, this work makes the following contributions:

- To the best of our knowledge, we conduct the first empirical study on annotation-induced faults in static analyzers based on 238 issues from six popular open-source analyzers. We analyze their root causes, symptoms, fix strategies, and types of annotations, deriving nine findings.

- Based on our findings, we propose AnnaTester, a novel automated testing framework that uses metamorphic testing with a customized annotated program generator to detect three types of annotation-induced faults in static analyzers.

- We evaluate AnnaTester on six static analyzers, and it can reveal 43 new faults in these analyzers, 20 of which have been confirmed and fixed.

## 4.1 Empirical Study of Annotation-Induced Faults

### 4.1.1 Target Static Analyzers

We select target static analyzers based on the following three criteria: (1) The analyzer 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 annotation-induced faults (AIFs) and corresponding fixes; (2) The analyzer should be popular and widely used so that its issues are representative of the real problems faced by users of analyzers. Specifically, we focus on analyzers with at least 2,000 stars on GitHub in this study; (3) The analyzer should support the analysis of Java programs. Based on these criteria, we select six static analyzers: (1) PMD [14] is a cross-language static analyzer that detects common code smells, e.g., unused variables; (2) SpotBugs [17] is a fork of the now deprecated analyzer FindBugs that detects common flaws in Java programs via a set of code patterns; (3) CheckStyle [1] checks the conformity of Java code to a set of coding rules; (4) Infer [7] is an analyzer designed by Meta to detect flaws for Java, C, C++, and Objective-C programs; (5) SonarQube [16] is a continuous code inspection platform that detects bugs and code smells for various programming and markup languages; (6) Soot [58] is a static analyzer that can analyze, instrument, and optimize Java and Android applications.

### 4.1.2 Data Collection

Among the six target analyzers, SonarQube uses Jira for tracking issues, while the other tools use GitHub. Since the issue tracking systems of these static analyzers contain around 14,000 issues, we refrain from manually inspecting all the issues to select only those related to annotation-induced faults. Instead, we first used the keyword “annotation” to search for closed issues that are likely annotation-induced. We only focus 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 269 issues in total. Then, we manually checked these issues and excluded those that were not associated with any fixing commits or not related to annotations. Subsequently, we collected 238 faults to be analyzed in our study. Table 4.1 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 remaining part, we refer to faults using their IDs in the form TOOL-###, where TOOL denotes the name of a static analyzer, and ### denotes the corresponding issue ID on GitHub or JIRA. As all collected issues were confirmed by the developers, we do not manually reproduce them again.

Table 4.1: 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 | 269 | 238 |

### 4.1.3 Issue Labeling and Reliability Analysis

In this study, we identify annotation-induced issues and analyze them from three aspects: the root cause, the symptom exhibited, and the fix strategy. The entire study takes around six months to complete. To categorize (or label) the issues from each aspect, we follow previous work [185, 217] to adapt existing taxonomies [79, 185, 194, 217, 219] to our task via an open-coding scheme. Specifically, one author first reviewed all the reports and pull

requests of those issues to determine the labels for these three aspects, including adding domain-specific categories and eliminating unnecessary ones. Then, two authors independently labeled the collected issues using the predefined categories. We used Cohen's Kappa coefficient [205] to assess the agreement between the two authors. First, the two authors labeled 5% of the issues, and the Cohen's Kappa coefficient was nearly 0.69. Then, we conducted a training discussion and labeled 10% of the issues (including the previous 5%). At this stage, the 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 an additional 10% of the issues, with the Cohen's Kappa coefficient remaining above 0.9 throughout the process. In each iteration, the two authors discussed with the third author if they had any disagreement. Finally, all issues were labeled consistently.

### 4.1.4 RQ 4.1: AIF-Prone Annotation

We collect annotations that trigger AIFs in the studied issues (we refer to them as *AIF-prone annotations*) and sort them in descending order of occurrence. Figure 4.2 shows the top 30 most frequently occurring annotations in our study. The x-axis displays the names of the annotations, and the y-axis presents the number of issues caused by each annotation.

Overall, we observe that annotations used to specify the nullability of program elements (i.e., @*Nullable*, @*Nonnull*, @*NonNull*, @*CheckForNull*, @*NonNullApi*, and @*NotNull*) are generally AIF-prone. These nullability-related annotations trigger the most issues as static analyzers typically 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 suppress specific warnings in static analyzers, e.g., @*SuppressWarnings("WarningName")*. It causes the second most issues, presumably because many static analyzers, such as PMD and SonarQube, support this annotation, and programmers frequently use it to filter out unwanted warnings Several test-related annotations (@*Test*, @*ExtendWith*, and @*VisibleForTesting*) are AIF-prone due

to their widespread use for marking tests. Annotations *@Inject* and *@Autowired* support the automated injection of data dependencies on annotated variables [63]. Understandably, static analyzers may produce incorrect analysis results if they are unaware of the implicit data flow introduced by these annotations. In Figure 4.2, around 23% (i.e., 7/30) of the annotations (such as *@Value*, *@Data*, and *@Getter*) can modify the original code, and failure to capture such changes can lead to faults 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 data dependence or structure of the original code induce the largest number of faults in static analyzers.

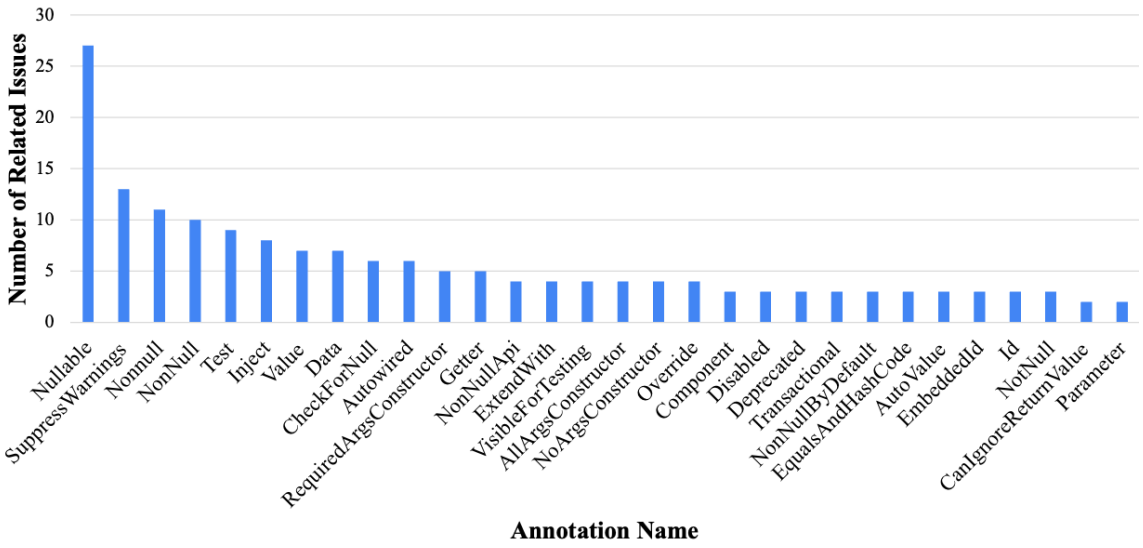


Figure 4.2: Number of issues induced by top 30 most AIF-prone annotations

### 4.1.5 RQ 4.2: Root Cause

We uncover a total of seven main reasons for the annotation-induced faults. In this section, we explain these reasons using examples, in decreasing order of their total occurrences in our study. Table 4.2 shows the number of faults caused by each reason for each static

analyzer.

Table 4.2: Number of faults due to different root causes in each static 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 | 238 |

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

### Root Cause 1: Incomplete Semantics (IS)

The most (38%) common reason for AIFs is that static analyzers usually only have incomplete knowledge about the semantics of annotations 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, neither CheckStyle nor 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 [157], shown in Figure 4.3. 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 ensures 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 some program threads. However, this stipulation 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 these annotations, SonarQube reported violations at lines 4 and 5 of the program in Figure 4.3.

```
1 @javax.annotation.concurrent.Immutable
2 class MyImmutable {}
3 @javax.annotation.concurrent.ThreadSafe
4 class MyThreadSafe {}
5 class Main {
6     private volatile MyImmutable x;
7     private volatile MyThreadSafe y;
8 }
```

Figure 4.3: An incomplete semantics example in SONARQUBE-3804 [157]

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

### Root Cause 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 AST and perform improper AST traversal. In CHECKSTYLE-7522 [40], the analyzer may encounter an *ANNOTATION_MEMBER_VALUE_PAIR* node, i.e., a type of AST node used to represent the key-value pairs in annotation declarations like @*Deprecated(removal=true)*, when it is not expecting one, causing a runtime crash. In CHECKSTYLE-9941 [43], an annotation for a method pushes the nodes for the method's header comment one level down in the corresponding AST. Therefore, the static analyzer

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

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.

**Root Cause 3: Unrecognized Equivalent Annotations (UEA)**

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 [171] (Java Specification Requests that aim to develop standard annotations for Java programs to assist software fault detection tools), several new libraries (e.g., Google JSR-305 [162]) 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 report a warning when an equivalent annotation (i.e., *@android.annotation.Nullable*) is used in the same way (lines 1 and 2 in Figure 4.4). (2) As a library evolves, the fully qualified names of annotations defined in the library may also evolve. For example, in SONARQUBE-3174 [117], the fully qualified package name of annotation *@Generated* changed from *javax.annotation* to *javax.annotation.processing* but SonarQube's develop-

ers were unaware of the change and failed to handle the renamed annotation correctly, causing inaccurate analysis results.

```
1 @android.support.annotation.Nullable
2 boolean fun1() {} // Report a warning
3 @android.annotation.Nullable
4 boolean fun2() {} // No warning reported, an FN
```

Figure 4.4: SONARQUBE-3536 [118]: A false negative caused by UEA

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

#### Root Cause 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 [119], developers of SonarQube mistakenly believed that the values stored in annotations could only be literals and incorrectly cast the AST of an annotation from ExpressionTree to LiteralTree, resulting in a runtime exception. Figure 4.5 shows that in SONARQUBE-3045 [106], developers forgot to resolve the type of the annotation (i.e., *@MyAnnotation*) applied to the actual type parameter (i.e., *MyClass*), leaving the type parameter annotated with an unknown type.

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

Figure 4.5: SONARQUBE-3045 [106]: An incorrect type resolution in SonarQube

**Root Cause 5: Incorrect AST Generation (IAG)**

Static analyzers may construct incorrect ASTs at the end of the intermediate representation (IR) construction stage (Figure 1.1). 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 [38]. Similarly, SonarQube and CheckStyle incorrectly handled type annotations introduced by JSR-308 [121] in SONARQUBE-1420 [167] and CHECKSTYLE-3238 [27]. Meanwhile, some static analyzers may make errors in constructing ASTs from the tokens returned by the lexers. For example, in SONARQUBE-1167 [102], 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 are due to the obsolete grammar that static analyzers implement.

**Root Cause 6: Misprocessing of Configuration File (MCF)**

As shown in Figure 1.1, a static analyzer usually expects as 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 [48], 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*".

**Root Cause 7: Others**

The "Others" category comprises issues with high variability and limited quantity, and we deliberately avoided creating new taxonomic categories for them. Two faults were highly specific to their corresponding tool implementations and cannot be attributed to any of the aforementioned reasons. In SONARQUBE-3108 [107], SonarQube crashed with an OutOfMemory exception when analyzing a method with 24 parameters, all annotated with *@Nullable*. The reason is that SonarQube created two symbolic starting states ("NULL" and "NOT_NULL") for each nullable parameter, requiring the creation of $2^{24}$ symbolic states for parameters of the method, which exceeded the available memory in the JVM. In CHECKSTYLE-2202 [26], *@SuppressWarnings* is utilized to suppress warnings specified by the annotation parameters. However, developers ignored the parameters named in camel-case notation, leading to a false positive.

### 4.1.6 RQ 4.3: Symptom

In this section, we describe the four symptoms and then relate them to the seven root causes to help users and developers assess the impacts of different root causes. 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 reports of the studied faults contain descriptions of the symptoms caused w.r.t. the final results produced by analyzers, some reports only

refer to incorrect intermediate results generated during the analyses without explaining how these intermediate results affect the overall analysis outcome. We classify those faults into this category. For instance, the issue report of CHECKSTYLE-8734 [38] explains that CheckStyle cannot parse the annotation on Java records and constructs only a partial AST for the program under analysis.

Table 4.3 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 are discovered during the actual use of the analyzers, and users are 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 [70,71,83,108,120]. Note that we identify one symptom for each fault in our study based on the issue report, which is reasonable because each issue 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.

### 4.1.7 Correlation Analysis between Root Cause and Symptom

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 4.4 shows the relationship between the root causes and symptoms. Although IS is the most common root cause, we observe that it never triggers runtime crashes, which are mostly caused by IAG. IS only leads to inaccurate analysis results, especially at the program analysis stage, while crashes often occur at the IR construction stage. We also observe that while FP results can be triggered by all root causes of AIFs, FN and CE results are never caused by incomplete semantics.

Table 4.3: Number of issues for the four categories of symptoms across the static analyzers

| Static Analyzer | FP | CE | FN | OWR | Total |
|---|---|---|---|---|---|
| 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 |
| Total | 162 | 35 | 33 | 16 | 238 |

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

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

Table 4.4: Correlation analysis between root cause and symptom

| Symptom | IS | IAT | UEA | ETO | IAG | MCF | Others | Total |
|---|---|---|---|---|---|---|---|---|
| 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.

### 4.1.8 RQ 4.4: Fix Strategy

We unveil six 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.

**Fix Strategy 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 [103], a rule checker was used to identify subclasses that should override the *equals* method, but it mistakenly ignored the annotation *@EqualsAndHashCode* in Lombok which. Applying this annotation to a class causes boilerplate implementations of the *equals* and the *hashCode* methods to be inserted. To fix this fault, the developers added the annotation to the white list, and SonarQube defaults to considering classes with this annotation as having overridden the *equals* method. Second, the utilization of the lists may be faulty. For example, in PMD-2876 [42], 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 can be fixed by adjusting the annotation lists in 90.2% instances and correcting the mishandling of annotation lists in the remaining 9.8% cases.

**Fix Strategy 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 [44], tool developers mistakenly neglected *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 the 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, suppressing all the *SingularField* warnings on classes where the variable value is true. In PMD-1641 [158], the traversal algorithm failed to restore the variable's value to false after visiting an inner class, causing an unwanted *SingularField* warning. The code snippet in Figure 4.6 illustrates how the fault was fixed.

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

Figure 4.6: PMD-1641 [158]: Fixing incorrect traversal algorithm

**Fix Strategy 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 [104], developers mistakenly resolved the type of an annotation based on its simple name, and the fix involved replacing the simple name with the annotation's fully qualified name. In PMD-1369 [31], 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.

**Fix Strategy 4: Fix Grammar Issue (FGI)**

As shown in Figure 1.1, static analyzers rely on predefined grammar to perform lexical and syntactic analysis to generate intermediate representations. 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 CHECKSTYLE-3238 [27], developers did not define grammar patterns to recognize annotations on variable-length parameters and failed to parse them consequently.

**Fix Strategy 5: Fix Value Check (FVC)**

This fix strategy involves adding checks that are missing or rectifying checks that are inappropriate. For example, in CHECKSTYLE-4472 [29], a missing null value check caused a runtime crash, and the fix was to add the missing check.

**Fix Strategy 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 these patterns may be incorrect and need to be redesigned. For instance, Figure 4.7 shows that in PMD-1782 [88] the rule checker initially only checked if a class or interface has a package definition (ignoring annotations). At line 2, PMD checked whether a package definition exists by counting the number of occurrences of the *PackageDeclaration* node in the XPath (which represents the AST as an XML-like

DOM structure) but mistakenly omitted annotations when writing the XPath. To fix this issue, the developers redesigned 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]
```

Figure 4.7: PMD-1782 [88]: Redesigning rule pattern to recognize package declaration

**Fix Strategy 7: Fix Incorrect API Usage (FIA)**

This fix strategy involves repairing incorrect API usages, primarily by using the correct API to retrieve elements or parse the signature of an annotation. Figure 4.8 shows that in SOOT-123 [115] 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 4.9 shows that in CHECKSTYLE-2202 [26], users adopted the *@SuppressWarnings* annotation to disable a warning, but CheckStyle only recognized rule names in lower case (line 1) and failed to detect equivalent rule names in camel case (line 2). To fix this, developers replaced *equals* with *equalsIgnoreCase* to recognize all equivalent rule names.

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

Figure 4.8: SOOT-123 [115]: Incorrect invocation of *toSoot* to construct an *AnnotationTag*

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

Figure 4.9: CHECKSTYLE-2202 [26]: Failing to recognize the camel case leads to an FP

**Fix Strategy 8: Others**

Three faults were not fixed by the fix strategies discussed previously. In INFER-559 [30], Infer reported an FP because it only used method signature information to analyze the parameter properties for compiled Java programs. Since no annotation is retained in the method signature information, the analyzer missed all annotations on method parameters.

### 4.1.9 Correlation Analysis 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 4.5 shows the number of annotation-induced faults 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 are due to root causes IS or UEA, probably because it is too challenging for the static analyzers to correctly handle the semantics of the annotations involved in those faults. Therefore, tool developers resorted to the filter-based solution as a workaround for the faults.

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

FIT is also a popular fix strategy and can address most root causes except for IS and UEA (both are primarily fixed by FAF). Most FIT-related issues stem from fixing type resolution (15) and type checking (11). Section 4.1.5 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 the proper libraries and find the correct class file to resolve the annotation. For the latter, developers should consider all possible program elements that need resolution (e.g., the fault in Figure 4.10 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

Table 4.5: Correlation analysis between root cause and fix strategy

| Root Cause | FAF | FAN | FIT | FGI | FVC | RRC | FIA | Others | Total |
|---|---|---|---|---|---|---|---|---|---|
| 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 |
| Total | 114 | 38 | 33 | 23 | 20 | 10 | 4 | 4 | 238 |

**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.

to parsing failure. To fix these issues, 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 types, variable arguments, and generic types.

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

Figure 4.10: SONARQUBE-2083 [105]: Failing to resolve the fully qualified name of an annotation

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

## 4.2 Implementation of AnnaTester Framework

We propose AnnaTester, an automated testing framework to detect annotation-induced faults through metamorphic testing, and it includes three checkers motivated by our study findings. Figure 4.11 shows the overall workflow of AnnaTester. Given a set of input programs obtained from the test suite of a static analyzer under test, AnnaTester detects annotation-induced faults in the analyzer in two steps: (1) It generates annotated programs by injecting annotations into the input programs; (2) It 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 the Eclipse JDT library to parse source seed files and inject annotations.

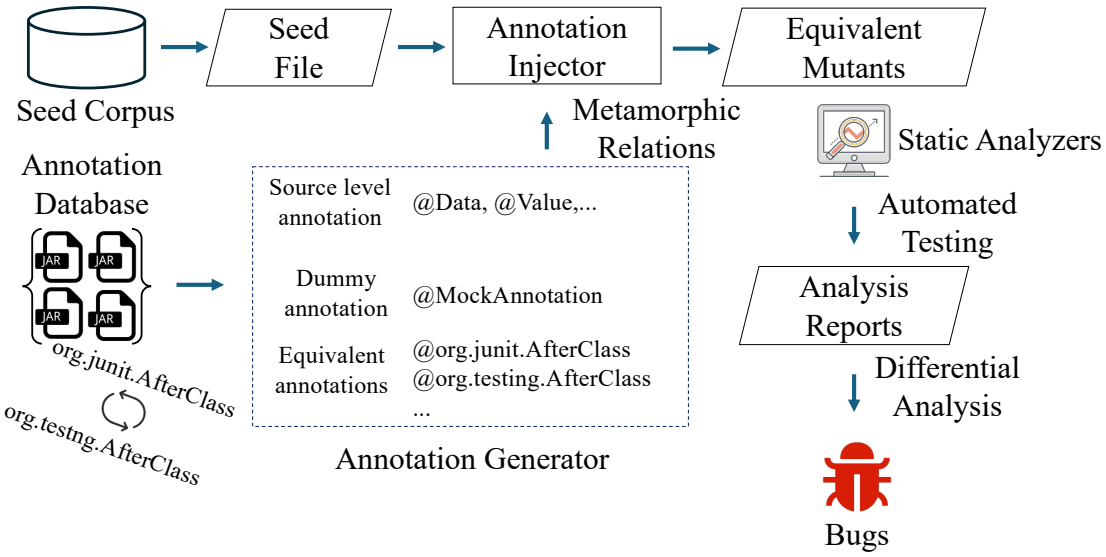


Figure 4.11: Overall workflow of AnnaTester

### 4.2.1 Issue 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 the violations together with the input programs (both with and without annotations) to users for further analysis. All metamorphic relations are based on the analysis of equivalence relations between programs as below:

**Definition 3** (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 to $P$ 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 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 $P$.

#### Incomplete Semantics Checker (ISC)

Incomplete semantics (IS) is 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 was derived by processing all the annotations in the first one [150]. 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 4** (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, 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 discrepancies 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 this, 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.

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

#### Equivalent Annotation Checker (EAC)

To identify inconsistent behaviors across equivalent annotations, we design the following metamorphic relation motivated by Finding 4:

**Definition 6** (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 this limitation, we devise an equivalent annotation checker

based on this metamorphic relation to automatically detect annotation-induced faults due to the UEA root cause.

### 4.2.2 Annotated Program Synthesizer

We design and implement an annotated program synthesizer 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 with respect to the aforementioned metamorphic relations. The synthesizer has three core components: (1) an annotation database, (2) an annotation generator, and (3) an annotation injector.

#### Annotation Database

To build a database containing widely-used Java annotations, we obtain annotations from two kinds of libraries in Maven Repo [163]: (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).

#### Annotation Generator

Our generator produces three types of annotations: (1) source-level annotations, (2) dummy annotations, and (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 4.12. We set its target to include all types of program elements that can be annotated 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 DummyAnnotation {}
```

Figure 4.12: Definition of the dummy annotation

**Equivalent Annotation Tuples.** As explained in Section 4.1.5, 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 semantically equivalent and applied 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 (each tuple generates at least two annotated programs). Hence, we select 24 tuples based on the top 30 AIF-prone annotations identified in RQ 4.1. All tuples have been manually verified that they are indeed equivalent tuples by two authors. Additionally, using all tuples will significantly increase the running time. For example, in PMD (the fastest among evaluated tools), tuple selection can reduce the running time by 91.3% (all tuples = 69 hours, selected tuples = 6 hours) while finding the same number of faults.

### 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 ap-

plied, 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. Therefore, AnnaTester discards these syntactically invalid variants before proceeding to subsequent steps.

## 4.3 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 analyzers as the seed corpus as prior work shows that these tests can help us cover more rule checkers to reveal more faults [213]. All experiments were conducted on a machine with Intel Xeon(R) 6134 CPU 3.20GHz and 192GB RAM. For static analyzers that require compilation (e.g., SpotBugs), we compile each program using Oracle JDK 17. 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 do not test AnnaTester on known issues as it is 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 4.6 shows the experiment results. We measure the effectiveness of AnnaTester by counting the unique faults detected by each checker ("#UniqFaults" column). Specifically,

Table 4.6: Effectiveness of StatFier

| 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) |
| Total | 433 | 43 | 8 | 20 | (6,87) |

we manually analyze the root causes of the identified faults and remove duplicates. 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 4.6 shows that AnnaTester found 43 AIFs in evaluated static analyzers, and 20 have been fixed via merged pull requests (9 by developers and 11 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 AIFs Found by AnnaTester.** To further analyze the fixed faults, we classify the fix strategies of the 20 fixed faults. 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 nine FPs ("#FP" column in Table 4.6). Our manual analysis of the FPs revealed that all FPs are caused by the source code changes induced by applying $MR1$ 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.

## 4.4 Case Study

We select three faults detected 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 [54].** Finding 5 shows that static analyzers cannot handle special annotation syntax as developers tend to neglect them. Figure 4.13 shows a crash example 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.

```
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 }
```

Figure 4.13: A crash in PMD detected by AnnaTester

**An FP in SonarQube [51].** Figure 4.14 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.

```
1 public static void main(String[] args) throws IOException {
2   @Cleanup
3   InputStream in = new FileInputStream(args[0]); // FP
4   ...
```

Figure 4.14: An FP in SonarQube detected by AnnaTester

**An FP in PMD [47].** Figure 4.15 shows that PMD reports a warning against the unnecessary constructor at line 4. However, this is an FP because the annotation *@Inject* uses this constructor for dependency injection. In Figure 4.15, PMD does not consider the annotation "com.google.inject.Inject", but it has considered another equivalent annotation, "javax.inject.Inject". EAC can automatically detect this FP. We have fixed this fault via a merged PR in collaboration with the 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 }
```

Figure 4.15: An FP in PMD detected by AnnaTester

## 4.5 Implication

We discuss the implications for developers and researchers based on our study findings in the subsequent section.

### 4.5.1 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 [121]. 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.

### 4.5.2 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 faults 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, elimi-

nating FPs is a worthwhile and long-term research direction [125]. As shown in Table 4.3, 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 the 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.

## 4.6 Summary

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 238 issues from six representative and diverse static analyzers (SonarQube, CheckStyle, PMD, SpotBugs, Infer, and Soot), identify seven root causes, four symptoms, and six fix strategies. Moreover, we summarize nine findings in the study. Based on these findings, we introduce a set of guidelines for AIF detection and repair, and propose AnnaTester, the first metamorphic testing based framework to automatically identify 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.

# Chapter 5

# SAScope: Characterizing and Detecting Program Representation Faults of Static Analyzers

Users often invoke program analyzers to construct various program representations [65, 140, 209] such as call graphs, control flow graphs, and intermediate representations, which encode the properties and behaviors of the given program, to support further analysis. However, developers of static analyzers may make mistakes when implementing different analysis algorithms, resulting in incomplete or inefficient analysis processes and incorrect program representations. For instance, prior research shows that buggy implementations of complex call graph construction algorithms and missing support for certain programming language features are two main reasons for incorrectly constructed call graphs [173, 191].

Although ensuring the correctness of generated program representations is essential, prior studies have primarily focused on (1) investigating a single type of program representation (e.g., call graphs), (2) pruning false positive edges from call graphs [144, 191], and (3) developing new call graph construction approaches that consider more specific aspects, such as invocations to Java libraries and implicit processes like object serialization/deserializa-

tion [64, 181]. To develop a comprehensive understanding of the reasons for, the impacts of, and the fixes for the faults in static analyzers that lead to the aforementioned undesirable behaviors, which we refer to as *program representation faults* (PRFs), we conduct the first empirical study of those faults in static analyzers. Our study aims to answer the following research questions (RQs):

**RQ 5.1:** Which program representations are more likely to be faulty? This RQ aims to study the representations that are more prone to PRFs and require more attention from developers during construction.

**RQ 5.2:** What symptoms can PRFs induce, and what are their root causes at each stage? This RQ aims to understand the effects and causes of PRFs at each stage of the static analyzer workflow.

**RQ 5.3:** What strategies do developers adopt when fixing PRFs? This RQ aims to understand viable ways to fix PRFs, which is essential for reducing the effort required to repair them.

**RQ 5.4:** How do developers detect PRFs? This RQ reviews the oracles developers used to determine whether a fault is a PRF, with the goal of deriving better designs for automated PRF detection.

To address these research questions, we first manually collect PRFs and their patches from four popular static analyzers: Soot, WALA, SootUp, and Doop. Subsequently, we identify four symptoms, analyze their root causes at each stage of the static analysis workflow, and reveal six fix strategies to help analyzer developers debug and repair PRFs. We also make eight findings and discuss the implications of our study for developers and researchers. In particular, we find that, while it is generally difficult to check the correctness of program representations since they are often large and have distinct and complex structures, comparing program representations based on their corresponding analysis precision and

functionality is a promising approach to automatically detecting certain types of PRFs.

Based on the findings of our study, we propose SAScope, a novel automated testing framework that detects PRFs in static analyzers using (1) a new metamorphic relation defined over different program representations and algorithms, (2) differential testing to verify the correctness of the same program representation across different static analyzers, and (3) a property-based approach to group detected faults. Inspired by Finding 8 in the empirical study, our first key insight is that a static analyzer commonly supports multiple program representations constructed using algorithms with various precision levels. Based on the precision lattice, we design a metamorphic relation that detects faults among program representations generated by different algorithms with different precision levels in a static analyzer. We also observe that algorithms with the same functionality are implemented in different static analyzers. Since the computed results from algorithms with the same functionality should be equivalent under the same inputs, we employ differential testing to find the discrepancies among them. We evaluate SAScope on the aforementioned four static analyzers, identifying 19 new faults. All of them have been submitted to the corresponding developers, and five have been fixed.

In summary, our work makes the following contributions:

- To the best of our knowledge, we conduct the first empirical study on program representation faults in static analyzers, involving 141 issues from four popular static analyzers.

- Inspired by the findings of our study, we implement the SAScope automated testing framework to detect program representation faults based on metamorphic and differential testing. In our metamorphic testing component, we propose a new metamorphic relation that uses the relative precision lattice of various program representation algorithms.

- We evaluate SAScope on four studied static analyzers and identify 19 new faults, five of which have been fixed by developers.

## 5.1 Empirical Study of Program Representation Faults

### 5.1.1 Tool Selection

We select static analyzers for study based on the following criteria: (1) It should be popular and widely used so that its issues are representative of practical problems faced by the users of analyzers. Particularly, we focus on static analyzers with at least 100 stars on GitHub and that have appeared in related researches [144, 154, 173, 181, 201]; (2) It must be open-source and use a public issue tracking system to record all issues that have been reported and resolved so that we can identify and analyze their program representation faults and the corresponding fixes; (3) It should provide APIs for users to access fundamental program representations, such as different intermediate representations and analysis graphs. Otherwise, it is difficult for us to identify PRFs in collected issues. Based on these criteria, we selected four static analyzers: (1) WALA [137] can analyze Java, Android, and JavaScript programs using many standard static program analysis techniques. (2) Soot [139] is a static analyzer that analyzes, instruments, and optimizes Java and Android applications. (3) SootUp [124] is a new version of Soot with a completely overhauled architecture. (4) Doop [75] is a static analyzer specifically designed for different Java pointer analysis algorithms.

Table 5.1: Issue distribution for four static analyzers

| Static Analyzer | #Star | #Issue$_c$ | #Issue$_k$ | #Issue$_a$ |
|---|---|---|---|---|
| Soot | 2782 | 773 | 168 | 64 |
| Wala | 724 | 321 | 119 | 30 |
| SootUp | 486 | 319 | 79 | 38 |
| Doop | 129 | 45 | 26 | 9 |
| Total | 4121 | 1458 | 392 | 141 |

### 5.1.2 Issue Collection and Labeling

Among the studied static analyzers, Doop uses BitBucket for issue tracking, while the others use GitHub instead. Since there are around 1458 closed issues in their issue tracking systems, we filtered out the irrelevant ones to keep the manual inspection of the issues manageable. In particular, we used keywords representing various forms of program representations, including "IR", "AST", "hierarchy", and "graph", to search for relevant closed issues. Overall, we collected 392 issues based on the keywords. Then, we manually reviewed the issues and removed the ones that are not faults, have no fixing commits, or are irrelevant to program representations. Table 5.1 lists, for each static analyzer, the number of stars in its open-source repository, the total number of closed issues in its issue tracking system (#Issue$_c$), the number of program representation issues returned by the keyword-based search (#Issue$_k$), and the number of PRFs confirmed by the manual analysis (#Issue$_a$). In the rest of this work, we refer to the PRFs using their IDs in the form Tool-###, where Tool denotes the name of a static analyzer and ### represents the corresponding issue ID on BitBucket or GitHub. Since the developers have confirmed and fixed all the PRFs, we did not attempt to reproduce them manually.

**Reliability Analysis.** This study focuses on issues related to program representation faults and analyzes them from three aspects: the symptoms exhibited, root causes at each stage, and fix strategies. The entire study took around six months to complete. To categorize (or label) the issues from each aspect, we followed the taxonomies of previous work [79, 185, 194, 214, 219] and adapted them to our task. Specifically, one author first looked through all the issue reports and pull requests to determine the labels for these three aspects, including adding domain-specific categories and eliminating unnecessary categories. Then, two authors independently labeled these issues using the previously defined categories. We used Cohen's Kappa coefficient [205] to assess the agreement between the two authors. First, the two authors labeled 5% of the issues, and Cohen's Kappa coefficient was nearly 0.65. Then, they conducted a training discussion and labeled 10% of the issues (including the previous 5%). At this stage, the Cohen's Kappa coefficient reached 0.92. 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. Cohen's Kappa coefficient remained above 0.9 throughout the process, and the two authors discussed with a third author to settle any disagreement between them in each iteration. Finally, all issues were labeled consistently.

### 5.1.3 RQ 5.1: Fault-Prone Program Representations

In this section, we focus on understanding which program representations are more prone to faults. We reviewed existing literature [18, 75, 101, 124, 139, 202, 203] to obtain a set of eight types of program representations commonly supported by mainstream static analyzers, including abstract syntax tree (AST), intermediate representation (IR, i.e., code in an intermediate language), call graph (CG), control flow graph (CFG), data flow graphs (DG), class hierarchy (CH), pointer assignment graph (PAG), and program dependency graph (PDG). On the one hand, a static analyzer usually parses the input program into an AST or converts it into an intermediate language, both of which fully encode the semantics of the input program. For instance, Soot utilizes the Jimple intermediate language [203] and performs optimization on Jimple code, while Doop [75] uses the Shimple intermediate language, which is essentially the static single assignment (SSA) variant of Jimple. On the other hand, a static analyzer may also construct one or more graphical representations of the input program, each focusing on one aspect of the program's semantics. A call graph represents the calling relationships between different methods within a program. A control flow graph adopts graph notation to model all paths that might be exercised during a program's execution, A data flow graph represents the set of values defined and used in calculations at various locations in a program, whereas a pointer assignment graph is a directed graph showing the viable types that each variable can point to. A class hierarchy models the inheritance relationships between program classes. Table 5.2 shows, for each static analyzer, the total number of PRFs we inspected and the breakdown of that number to different types of program representations.

Table 5.2: Number of PRFs we inspected and the breakdown of that number to different types of program representations

| Static Analyzer | CG | IR | CFG | CH | DG | PAG | PDG | AST | Total |
|---|---|---|---|---|---|---|---|---|---|
| Soot | 14 | 17 | 11 | 11 | 5 | 3 | 1 | 2 | 64 |
| SootUp | 9 | 12 | 9 | 5 | 3 | 0 | 0 | 0 | 38 |
| WALA | 16 | 1 | 2 | 5 | 3 | 1 | 1 | 1 | 30 |
| Doop | 4 | 1 | 0 | 1 | 0 | 1 | 2 | 0 | 9 |
| Total | 43 | 31 | 22 | 22 | 11 | 5 | 4 | 3 | 141 |

**IR**: Intermediate Representation, **CG**: Call Graph, **CFG**: Control Flow Graph, **DG**: Dataflow Graph, **CH**: Class Hierarchy, **PAG**: Pointer Assignment Graph, **PDG**: Program Dependency Graph, **AST**: Abstract Syntax Tree.

Table 5.2 confirms that static analyzers have faults that affect all of the studied program representations. The call graph is the most fault-prone program representation among the eight program representations partly because call graphs can be rather complex and hard to get all right and partly because they provide the foundation for many other analyses and, therefore, are more thoroughly tested. The intermediate representation is the second common fault-prone representation, especially in issues related to Soot and SootUp. The number of issues related to AST is the lowest because static analyzers usually adopt class files as input and convert them into IR for further analysis, only WALA supports a mature source code frontend among four evaluated static analyzers.

**Finding 1:** The top two most fault-prone program representations are call graphs and intermediate representations, accounting for 52.5% of the studied issues.

### 5.1.4 RQ 5.2: Symptom and Root Cause

In this section, we attempt to understand the symptoms (**S**) caused by the analyzed issues, their distribution across different stages of the static analyzer workflow, and their root causes. Overall, we summarize four symptoms. Table 5.3 shows the symptom distribution in the workflow of static analyzers.

Table 5.3: Distribution of symptoms at each phase of workflow

| Symptom | Core Analysis | Program Parse | Data Input/Output | Total |
|---|---|---|---|---|
| MEPR | 40 | 7 | 5 | 52 |
| FPRG | 21 | 26 | 2 | 49 |
| IEPR | 18 | 11 | 0 | 29 |
| IPRG | 8 | 3 | 0 | 11 |
| Total | 87 | 47 | 7 | 141 |

**MEPR**: Missing Elements in Program Representation, **FPRG**: Failed Program Representation Generation, **IEPR**: Incorrect Elements in Program Representation, **IPRG**: Inefficient Program Representation Generation.

#### Symptom 1: Missing Elements in Program Representation (MEPR)

Table 5.3 shows that missing elements in program representation is the most popular symptom. This symptom category involves program representations that are missing elements. Table 5.3 shows that this symptom may be observed in all phases and occur most frequently in the core analysis stage (40/52, 76.9%). Most MEPR issues at this stage are due to improper handling of (implicit) invocations to special methods during call graph construction (23/40, 57.5%). For instance, Figure 5.1 shows a piece of code that reveals SootUp-459 [130]. SootUp should be able to identify an invocation from *A.foo* to the class initialization method *A.clinit* in the code if method *foo* is set as an entry method during call graph construction, but it failed to do so. Besides, developers may misunderstand the pro-

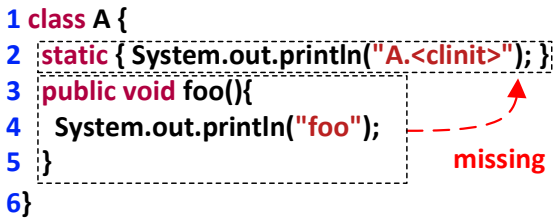


Figure 5.1: SootUp incorrectly processes *clinit* methods

gram representation construction algorithm. In SootUp-456 [132], the Rapid Type Analysis (RTA) algorithm [72] wrongly set the processed methods as completed after the first-round process as the RTA algorithm is implemented via a worklist, which can traverse the methods iteratively to find more caller and callee targets. Hence, the RTA algorithm may process each method multiple times rather than once. Figure 5.2 shows that in SootUp-456 [132], the call edge from *A.process* to *C.target* should be added to the call graph, but *A.process* method was labeled as "completed" after the first visit at line 6, causing the re-visiting at line 10 to have no effects. In general, to construct correct call graphs, developers should (1) take note of special methods like *clinit* and concrete methods in abstract classes, (2) consider the inheritance relationships between classes, and (3) traverse the graphs to obtain information about all classes.

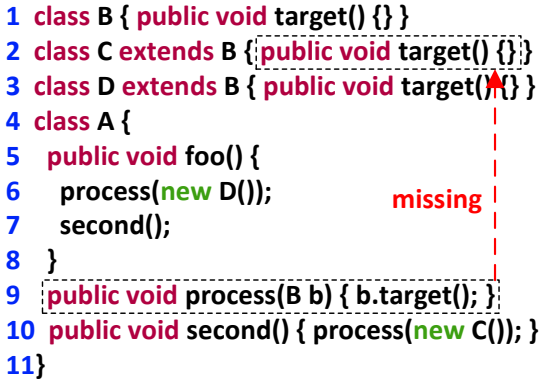


Figure 5.2: SootUp wrongly implemented the RTA algorithm

Meanwhile, class hierarchies and control flow graphs are other common representations that trigger MEPR issues at the core analysis stage. For instance, in WALA-322 [188],

developers failed to add unresolved superclasses to the class hierarchy graph. However, unresolved superclasses may not affect some analysis functionalities (e.g., building IR), and developers provided the phantom class to handle it. In Soot-385 [25], Soot tried to convert the Shimple-based control flow graph to Jimple-based, both of which are intermediate representations. To achieve that, Soot had to eliminate the Phi() function in Label 2 of Figure 5.3. However, it incorrectly put the instruction *r0_2=r0* in the last slot of Label 1 block as this block is a try handler of a trap statement, and preceding statements may lead to uninitialization errors of *r0_2*.

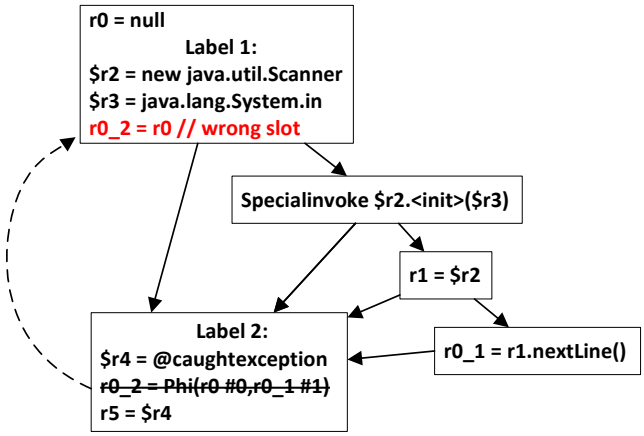


Figure 5.3: Soot-385 [25]: Incorrect control flow graph construction

Data input/output is also a common phase for MEPR issues (5/52) due to the wrong process for setting input/output data related to program representations. For instance, in Soot-524 [28], developers did not reset the variable that configures the input class path, leading to incomplete elements in analysis results.

**Finding 2:** MEPR is the most common symptom of PRFs, and many such PRFs occur at the core analysis phase. Two common root causes of MEPR are (1) neglecting certain methods and (2) misunderstanding of graph construction algorithms.

**Symptom 2: Failed Program Representation Generation (FPRG)**

Failed program representation generation is the second most popular symptom (49/141, 34.7%), especially for the program parsing stage (26/49, 53%). This refers to the scenario where the program representation generation terminates unexpectedly. At the program parsing stage, most issues are due to a lack of consideration of specific language features. Figure 5.4 shows an example where WALA used *LambdaMetaFactory* to parse a lambda invocation expression (represented by *invokedynamic* instruction in bytecode) and get the callee target (*println*) method at line 1. However, developers wrongly used this function to parse the new *LambdaMetaFactory* statement at line 2 as they mistakenly treated this call site as being generated by the compiler to handle lambda invocations, and actually this statement is an *invokevirtual* instruction in bytecode. Our study revealed that all evaluated static analyzers (except for Doop) have issues parsing lambda expressions.

```
1 new Thread(() -> System.out.println("Lambda")).start();
2 LambdaMetaFactory factory = new LambdaMetaFactory();
```

Figure 5.4: Wrong IR construction for lambda expression

The second root cause triggering FPRG is incorrect intermediate representation (IR) optimization. Static analyzers include many optimization algorithms to optimize IR (e.g., dead assignment and unused local variable elimination [139]), but these algorithms may have implementation errors. For example, in Soot-358 [73], developers mistakenly marked the code in the trap to be removed as reachable when performing dead code elimination. The third root cause is related to the operand stack. As JVM is a stack-based virtual machine, static analyzers adopt an operand stack for simulation. Figure 5.5 shows SootUp-326 [41] that developers first initialize the operand stack before block 1 to simulate the IR construction of the left path (1- 2- 4). However, they do not recover the operand stack before the IR construction of the right path. Hence, the operand stack has insufficient elements when basic block 4 is re-processed, causing a stack underrun error.

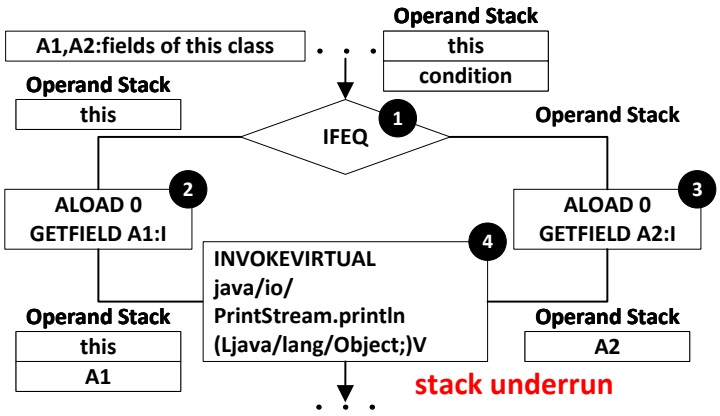


Figure 5.5: SootUp-326 [41]: An operand stack underrun issue

**Finding 3:** Three common root causes for failed program representation generation at the program parsing stage are (1) missing consideration of specific program elements, (2) incorrect IR optimization, and (3) wrong operand stack for simulating the JVM stack.

The second FPRG-prone stage is core analysis (21/49, 42.9%). Two leading root causes are (1) concurrency bugs in class hierarchy and (2) inadequate handling of edge representation. The class hierarchy is often accessed and modified by different threads. Figure 5.6 shows that in Soot-1189 [33], different threads simultaneously invoked *getOrMakeFastHierarchy*, leading to a runtime error due to concurrent hierarchy construction. Besides, inadequate handling of edge representation is the second most popular root cause. In Soot-416 [24], Soot threw a runtime exception when handling a graph where the dominator of a block is the same as the block itself.

```
1 public FastHierarchy getOrMakeFastHierarchy() {
2   if (!hasFastHierarchy()) { setFastHierarchy(new FastHierarchy()); }
3   return getFastHierarchy();
4 }
```

Figure 5.6: A concurrency bug in class hierarchy of Soot

### Symptom 3: Incorrect Elements in Program Representation (IEPR)

IEPR is the third most common symptom (29/141, 20.6%). This category involves program representations containing incorrect or redundant program elements. 62.1% of IEPR issues occur at the core analysis stage. For example, in SootUp-495 [133], developers fix the RTA implementation by only considering the *new()* expression as the instantiated class rather than *init* method in bytecode. Otherwise, the RTA algorithm may mistakenly regard statements like *super();* as class instantiations, leading to incorrect edges in the call graph. In SootUp-715 [134], developers fail to add the essential libraries from the configuration for specifying dependencies (used to provide knowledge for analyzing input programs) to the call graph, and static analyzers should not perform core analyses on the libraries. The remaining 37.9% of issues are related to the program parsing stage, including two root causes: (1) incorrect type assignment and (2) miscompilation in compiler-synthetic methods. Figure 5.7 shows an example of incorrect type assignment in SootUp-103 [34] where developers incorrectly resolve a boolean expression at line **1** to the integer type because SootUp does not support boolean type and used integer values 1 or 0 to represent true or false, resulting in an incorrect return statement at line **8**. Figure 5.8 shows an example of

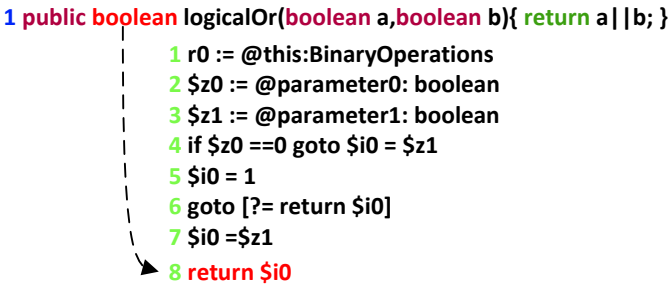


Figure 5.7: Incorrect boolean expression resolution in SootUp

miscompilation in compiler-synthetic methods where the compiler would generate a *get* method for the *Supplier<Object>* at line **1** to obtain the inner object, but Soot fails to consider the return type of *new* at line **1** when parsing the generated *get* method, causing an incorrect return statement with void type at line **7**.

```
1 public Supplier<Object> constructorRefereturn(){ return Object::new; }
1 public java.lang.Object get() {
2   A$init__1 $r0;
3   java.lang.Object $r1;
4   $r0 := @this: A$init__1;
5   $r1 = new java.lang.Object;
6   specialinvoke $r1.<java.lang.Object: void <init>()>();
7   return;
8 }
```

Figure 5.8: Wrong *return* of a compiler-synthetic method

**Finding 4:** IEPR mainly appears at the core analysis and program parsing stages. To avoid it, developers should take note of (1) type resolution and (2) IR generation of compiler-synthetic methods.

### Symptom 4: Inefficient Program Representation Generation (IPRG)

This symptom category occurs when the time cost for generating program representations exceeds the expectations of developers or users. Although this symptom is less frequent (11/141, 7.8%), it still affects the usability of static analyzers. 72.7% (8/11) of IPRG issues arose during the core analysis phase. This phase involves three common root causes. First, developers implemented inefficient core analysis algorithms. For example, in SootUp-728 [57], CHA is invoked twice to resolve call sites for inter-procedural CFG construction, which is time-consuming. The second root cause is using immutable data structures. In SootUp-281 [37], developers use the Guava library [170] to implement the graph structures but Guava only provides immutable structures, and each operation on them involves costly deep copying of objects. Third, the lack of caching for program representations can lead to inefficiencies. In WALA-9 [19], developers repeatedly construct the IR and def-use chain (both time-consuming operations) of a method. To avoid performance degradation, developers cache the IR and def-use chain and adopt *WeakReference* to relieve the garbage collection issues. Overall, we suggest developers to: (1) avoid redundant analysis operations (e.g., resolving call sites when constructing ICFG); (2) avoid using immutable data structures to store analysis results; (3) use the cache to store the results of time-consuming

operations.

**Finding 5:** Most inefficient program representation generation issues take place at the core analysis stage (8/11) due to algorithm implementation, immutable structure, and cache missing.

### 5.1.5 RQ 5.3: Fix Strategy

We uncovered six fix strategies for fixing PRFs. In this section, we first summarize each strategy and illustrate fix patterns associated with each strategy. Table 5.4 shows the distribution of fix strategies among the four static analyzers.

Table 5.4: Distribution of fix strategy among static analyzers

| Symptom | FAD | FIR | FIT | FPC | FLF | FCB | Others | Total |
|---|---|---|---|---|---|---|---|---|
| Soot | 24 | 17 | 8 | 4 | 4 | 5 | 2 | 64 |
| SootUp | 18 | 9 | 6 | 0 | 4 | 1 | 0 | 38 |
| WALA | 7 | 9 | 12 | 1 | 1 | 0 | 0 | 30 |
| Doop | 0 | 3 | 2 | 4 | 0 | 0 | 0 | 9 |
| Total | 49 | 38 | 28 | 9 | 9 | 6 | 2 | 141 |

**Fix Strategy 1: Fix Improper Program Representation Construction Algorithm Design (FAD)**

Table 5.4 shows that FAD is the most popular fix strategy (49/141). This strategy fixes design flaws such as misunderstandings of the construction algorithm or neglect of specific program elements, that affect the results. FAD involves three patterns: (1) *Adding functionality to handle specific program elements*, such as concrete methods in abstract classes or interfaces. In Soot-514 [21], developers forgot to consider all types of concrete methods

when computing the local pointer assignment graphs (PAG). To fix the issue, developers changed the conditions to filter these unprocessed methods. (2) *Fixing incorrect logic of algorithm implementation*. Figure 5.9 shows that in Soot-486 [22], developers inserted *goto* instructions to implement control flow jumps from the current instruction $S$ to the target instruction $T$. However, due to the restriction of jump distance, one *goto* cannot bridge the gap between $S$ and $T$. Developers used multiple *goto* and inserted one $S'$ in the next slot, which cannot reduce the actual distance to $T$ as it was also moved to the next slot, causing endless *goto* insertion and an infinite loop. To fix this, developers used binary search to determine the farthest slot a *goto* instruction can reach and insert it, thereby closing the gap to $T$. (3) *Avoiding redundant computational operations*. In SootUp-728 [57], developers constructed call graphs twice to build the inter-procedural control flow graph. To optimize this, they removed the second call graph construction and reused the previous results.

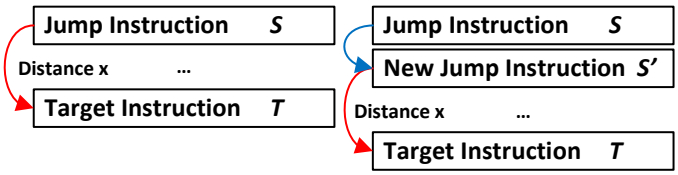


Figure 5.9: Incorrect *goto* instruction insertion

**Finding 6:** FAD is the most frequently used fix strategy (34.8%), covering three fix patterns: (1) fixing missing functionalities to handle specific program elements, (2) fixing incorrect logic of algorithm implementation, and (3) avoiding redundant computational operations.

### Fix Strategy 2: Fix Incorrect Program Element Resolution (FIR)

The second most common fix strategy is FIR (38/141). Static analyzers rely on resolution to understand the type and property of a specific program element (e.g., for a call site, resolution analyzes the symbolic information attached to the call site and identifies the actual method to be invoked). Fixing the call site resolution of the *invokedynamic* instruction,

introduced by JSR-292 [177], is a popular pattern in FIR. In WALA-285 [138], developers misused the *invokedynamic* handler to resolve the statement *new LambdaMetaFactory*, causing a crash. To fix this, they changed the condition for resolving the *invokedynamic* instruction to filter out <*init*> methods. Type dispatch via class hierarchy is a common practice when resolving classes or methods. Type dispatch analyzes program elements from two perspectives: (1) searching for unimplemented methods or fields in the superclass, e.g., *constructor*; (2) traversing down the hierarchy to find potential targets when the call site target defined in the superclass does not have a specific implementation. In SootUp-499 [131], SootUp did not find the concrete method implementation in the current class, and the fix involved getting it from the superclass. Another common strategy is fixing incorrect type resolution as developers either ignore the type resolution or confuse the types with similar ones. In Soot-1739 [49], *WeakObjectType* is a subclass of *RefType* that includes a *SootClass*-type field. However, Soot only resolves the name of *WeakObjectType*-defined variables without the information saved in *SootClass* like internal fields or methods. Developers added the missing resolution for the *SootClass*-type field to fix the issue. The last sub-strategy of FIR is fixing incorrect dependent libraries, which involves the original symbolic information for type resolution. Developers usually added missing libraries or updated dependent libraries. In Doop-1 [36], developers updated the version of Souffle in Doop to fix the fault.

**Finding 7:** FIR is the second most common fix strategy (27%), which includes fixing incorrect dispatch or resolution of *dynamicinvoke* call sites, repairing expression type assignment, and accomplishing type resolutions due to inheritance.

#### Fix Strategy 3: Fix Incorrect Program Representation Traversal (FIT)

Fixing incorrect program representation traversal is the third most common fix strategy (28/141). Traversal is used to obtain information from program representation. We find two main fix patterns of this strategy. The first pattern is fixing incomplete traversal. For

example, in Soot-875 [32], developers did not visit and resolve the inner class, so the fix involved rounding out the traversal. The second pattern is fixing the incorrect settings of state variables. Soot's developers did not realize that the successor of a statement may include itself and failed to mark the visited statements, leading to a dead loop fault in SootUp-798 [59]. Developers added a state label to mark the visited statement. Figure 5.5 shows another example: when traversing the two control flow paths, SootUp did not recover the operand stack after traversing the first path. To fix this, developers used a deep copy of the stack instead of directly using the operand stack attached to the entry block.

**Fix Strategy 4: Fix Incorrect Processing of Configuration (FPC)**

FPC includes nine issues that address incorrect configurations. We divide them into two categories. The first category is fixing incorrect logic for processing configurations. For example, in Doop-4 [39], Doop gave incorrect results when given an input name with spaces, as it splits input commands by spaces, and the name was incorrectly divided into multiple tokens. To fix the issue, developers used quotation marks to isolate the file path and avoid segmentation. Another category is adding new options to offer more analysis methods for users to choose. For instance, Figure 5.10 shows Soot-109 [68] that the pedantic throw analysis added an edge from each statement in the try branch to the catch handler (namely, statements at lines 3–7 to handler 8–10). However, the edge from line 6 to the catch handler is incorrect, as the path (1-2) →6→(9-10) led to an uninitialization error of *r4* at line 10. Hence, developers added an option for users to select the throw analysis mode to avoid applying pedantic throw analysis to such input programs.

**Fix Strategy 5: Fix Concurrency Bug (FCB)**

We find six issues due to incorrect concurrency operations. Five of them stem from concurrent access to the hierarchy. Developers adopted two sub-strategies: (1) using thread-safe data structures (e.g., in SootUp-591 [56], as *HashMap* cannot support concurrent opera-

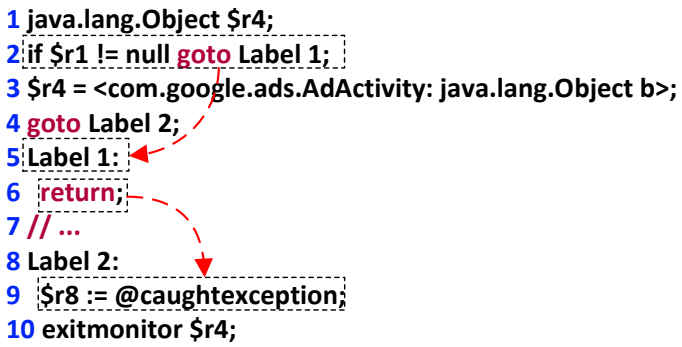


Figure 5.10: Pedantic throw analysis leads to verification error

tions, developers replaced it with *Cache* from Guava); (2) using synchronized statements (e.g., in Soot-1125 [35], different threads simultaneously visited the hierarchy, triggering concurrent modification exception, and developers added *synchronized* to the read(), write(), and clear() methods of the hierarchy). The remaining issue is due to the fields defined in a transformation class used by multiple threads, and developers changed these fields to local variables.

**Fix Strategy 6: Fix Neglected Language Feature (FLF)**

Due to frequent updates to Java/Android versions, static analyzers may not meet the specification requirements. For instance, in Soot-35 [20], developers considered the neglected annotations in the Dalvik bytecode that Jimple converts.

**Fix Strategy 7: Others**

Two faults were not fixed by the previously discussed fix strategies. In Soot-502 [23], developers wrongly used decrement operators (*index*-- instead of --*index*), causing an out-of-bounds read error. In Soot-1874 [50], the *this* reference was missing on the object, leading to *edges.remove(edges)* instead of *this.edges.removeedges*.

### 5.1.6 RQ 5.4: Oracle Design

In this section, we attempt to understand how developers identify an issue as a PRF and the oracles they use. This research question helps us design oracles for automated PRF detection.

Most issues involving missing or incorrect elements in program representation are identified when the analysis results violating the expectations of static analyzer users or the unit tests manually crafted by the developers. These two approaches rely on prior knowledge of developers and users, and thus cannot be directly applied to automatically detect program representation faults. Another two approaches are based on comparing analysis results. The first approach compares analysis results between algorithms with the same purpose but different precision [46,74], and the second approach compares the analysis results from different tools with the same target (e.g., Doop-1 [36] compared the results from different datalog engines). This leads us to a differential testing approach that can be automated. All failed program representation generation issues are recognized by exceptions that cause crashes. Inefficient program representation generation issues are typically identified by measuring the actual waiting time of the generation. Users may perceive long waiting time as a performance issue, especially when the input program is relatively small (e.g., 10 minutes in SootUp-558 [55]). Hence, failure to achieve results within a conservative time limit can serve as an oracle for detecting performance issues. The remaining issues are spotted by developers as they proactively found redundant computational operations.

**Finding 8:** Comparing the analysis results based on algorithm precision and functionality is a promising approach for MEPR and FPRG detection. For FPRG faults, inspecting the exception stack trace can recognize them.

## 5.2 Methodology of SAScope

We propose and implement SAScope, an automated testing framework for detecting program representation faults via two-dimensional testing. Figure 5.11 shows the overall workflow of SAScope. It involves two main components: metamorphic testing and differential testing components, which are inspired by the Finding 8 in our study. First, we give a formal definition 7 of program representation based on existing work [65, 140, 209]. For instance, a node $V$ in the call graph represents a method (caller or callee), and an edge $E$ represents the calling relationship between two methods.

**Definition 7** (Program Representation)**.** Program representations include different forms of modeling programs, such as graphical representations and instruction representations. For a graphical representation, we formally describe it using a directed graph $\mathcal{G} =< V, E >$. Here, $V$ involves all program elements and $E =< v_i, v_j >$ depicts the relationships between program elements $v_i$ and $v_j$ in $V$. For a instruction representation, we define it using a list $\mathcal{L} = [l_1, \ldots, l_n](n \geq 1)$, and each element $l_i(1 \leq i \leq n)$ in $\mathcal{L}$ represents an instruction.

Algorithm 2 shows more details of these two components, and currently, we focus on testing graphical program representations. Given an input program $p$ in $Progs$, SAScope first invokes a static analyzer $S$ via the invocation template (lines 4–5) to obtain program representations using different algorithms at lines 27–33. *ResList* stores the program representations of $S$, sorted by the precision in ascending order. If $InvokeTemplate$ terminates unexpectedly or cannot obtain any analysis result within the *timeL*, $ResList$ will be empty, and SAScope records a potential crash (lines 6–8). For the program representations generated by identical static analyzers with different algorithms, SAScope leverages metamorphic testing at lines 9–16 to reveal potential faults. Then, the analysis results are appended to $R$. In lines 18–25, SAScope uses differential testing to inspect program representations from different static analyzers with the same input program and analysis algorithm. It records the inconsistencies between them as potential faults.

**Invocation Template.** To analyze input programs, we design a template to invoke various

**Algorithm 2:** Overall SAScope Testing Approach

**Input:** Input programs *Progs*, a set of static analyzers *SAs*, timeout *timeL*

**Output:** A set of potential faults in static analyzers *I*

```
1  I ← ∅
2  for P ∈ Progs do
3      R ← []
4      for S ∈ SAs do
5          ResList = InvokeTemplate(P, S, timeL)
6          if ResList == null then
7              I = I ∪ {P, S}
8              continue
9          for i = 1 → |ResList| − 1 do
10             if ResList_i.V ⊉ ResList_{i+1}.V then
11                 I = I ∪ {ResList_i.V, ResList_{i+1}.V}
12             for v ∈ ResList_i.V ∩ ResList_{i+1}.V do
13                 E_1 = AdjacentEdge(ResList_i, v)
14                 E_2 = AdjacentEdge(ResList_{i+1}, v)
15                 if |E_1| ⊉ |E_2| then
16                     I = I ∪ {E_1, E_2}
17         R.append(ResList)
18     for i = 1 → |R| − 1 do
19         if R_i.v ≠ R_{i+1}.v then
20             I = I ∪ {R_i.V, R_{i+1}.V}
21         for v ∈ R_i.V ∩ R_{i+1}.V do
22             E_1 = AdjacentEdge(R_i, v)
23             E_2 = AdjacentEdge(R_{i+1}, v)
24             if E_1 ≠ E_2 then
25                 I = I ∪ {E_1, E_2}
26 return I
27 Func InvokeTemplate (P, S, timeL):
28     ResList ← ∅
       /* sorted by the precision in ascending order                    */
29     while execTime < timeL do
           /* Ψ denotes algorithms supported by S                       */
30         for ψ ∈ Ψ do
               /* Invoke S on P to get analysis results                 */
31             Result = S.invoke(P, ψ)
32             ResList.append(Result)
33     return ResList
```

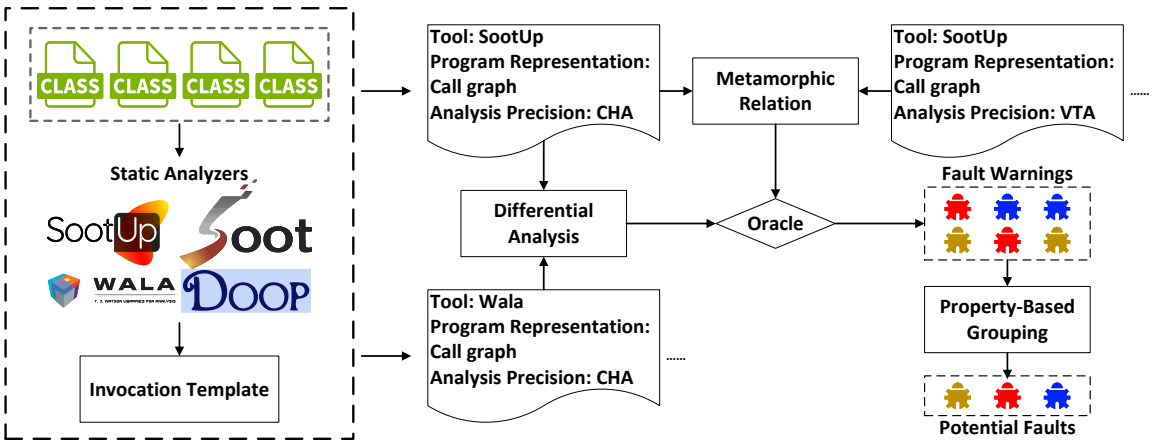


Figure 5.11: Overall workflow of SAScope

analysis modules in the evaluated static analyzers. For each static analyzer, the template retains all configurations except for the input programs before starting an analysis. Figure 5.12 shows a template example used to activate the RTA analysis in Soot. For configurations not involved in the template, we reuse the default values of the evaluated static analyzers.

```
1 AnalysisInputLocation input = new JavaClassPathAnalysisInputLocation(
2                                   "INPUT_PATH", SourceType.Application);
3 JavaView view = new JavaView(new ArrayList<>() {{add(input);}});
4 CallGraphAlgorithm rta = new RapidTypeAnalysisAlgorithm(view);
5 CallGraph cg = rta.initialize(entryMethods);
```

Figure 5.12: A template example for the Soot RTA algorithm

### 5.2.1 Testing Approaches and Oracle Design

In general, SAScope depends on two testing approaches to recognizing program representation faults. Both approaches rely on the analysis results of the static analyzers. In the rest of this section, we use $\phi$ denotes a program representation and $\delta$ to denote its generation algorithm.

**Metamorphic Testing.** Our key insight is that static analyzers usually support multiple

program representations constructed using different algorithms with various precision levels (where one representation is more precise than the other). Based on this insight, we design SAScope that adopts a new metamorphic relation. Specifically, the metamorphic relation leverages the total order relation of different program representation algorithms in the precision lattice. As prior work [101] includes the proofs for many algorithms in the precision lattice, we extract and focus only on the related algorithms supported by evaluated static analyzers. Figure 5.13 shows the precision lattice of tested algorithms in our work. "CFA" means call site sensitive, "Obj" denotes object sensitive, and the precision increases gradually from the bottom-up direction in the lattice. For instance, rapid type analysis (RTA) and class hierarchy analysis (CHA) are popular call graph generation algorithms. Figure 5.13 shows that the RTA algorithm is more precise than CHA, as it prunes nodes where internal types are never instantiated [72]. Currently, our metamorphic relation only conservatively supports algorithms for call graphs and pointer assignment graphs that have been proven in prior work [101]. In the future, it is worthwhile to include other program representations with different precision levels (e.g., IR with different precision [175]) to detect more faults. Definition 9 presents the metamorphic relation used in our metamorphic testing.

**Definition 8** (Less Precise Operator ($\preccurlyeq$))**.** Given two program representation construction algorithms $\delta_1$ and $\delta_2$, we denote $\delta_1 \preccurlyeq \delta_2$ if and only if $\delta_1$ is less precise than $\delta_2$ based on the precision lattice in Figure 5.13.

**Definition 9** (Metamorphic Relation)**.** Given the program representations $\phi_1$ and $\phi_2$ generated by $\delta_1$ and $\delta_2$ under the same input program, they should possess the property $\phi_1 \supseteq \phi_2$ if $\delta_1 \preccurlyeq \delta_2$.

According to Definition 9, $ResList_{i+1}.V$ (line 10 in Algorithm 2) should be a subset of $ResList_i.V$ since the former is more precise than the latter. Hence, we first check the node set relationship at lines 10–11. Then, we compute the intersection of node sets and compare the adjacent edges of each common node at lines 12–16. Any violation of the Definition 9

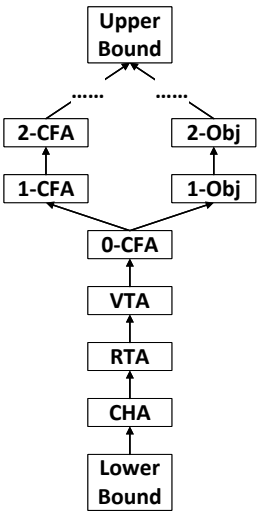


Figure 5.13: Relative precision lattice of tested algorithms

will be regarded as a fault.

**Definition 10** (Equivalent Program Representation)**.** Two program representation $\phi_1$ and $\phi_2$ are equivalent if and only if (1) $\mathcal{G}_1 = \mathcal{G}_2$ or $\mathcal{L}_1 = \mathcal{L}_2$; (2) $\phi_1$ and $\phi_2$ are generated by the same algorithm.

**Differential Testing.** We observe that different static analyzers usually implement the same analysis algorithms, and should produce analysis reports with equivalent program representation (Definition 10) given the same input program. Hence, differential testing is naturally well-suited for this scenario. Specifically, SAScope first selects two analysis reports $R_i.v$ and $R_{i+1}.v$, generated by the same algorithm in different static analyzers (lines 18–19 in Algorithm 2), and performs differential analysis on these reports. SAScope compares the node sets (lines 19–20) and adjacent edges (lines 21–25). Then, it reports all discrepancies as potential faults.

### 5.2.2 Property-Based Grouping

As too many warnings are reported from previous steps, we need to group them based on their underlying root causes to reduce the manual efforts required to examine these warnings. As observed from Findings 3, 4, and 7, most program representation faults have distinct syntax and type features. Hence, we consider several characteristics to classify the detected warnings: (1) the testing approach; (2) types of related static analyzers; (3) types of related generation algorithms; (4) fine-grained properties of related program elements (e.g., for a fault in call graph, we consider the invocation instruction type and the access modifiers of the caller and callee). Figure 5.14 shows a missing element in the program representation fault found by SAScope, where the left side shows the minimized code example and the right side shows the group that includes this fault. The fault is caused by the incorrect implementation of the CHA algorithm in WALA, leading to a missing call edge from the constructor *Test()* to *foo()*. We identify this fault via metamorphic testing, as the 0-CFA algorithm correctly constructed the call graph.

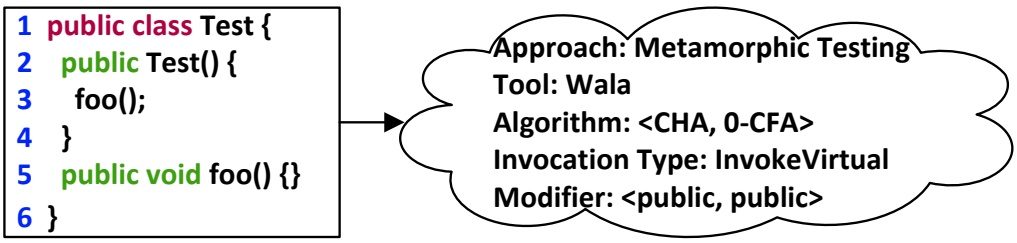


Figure 5.14: A MEPR fault in WALA and its group

## 5.3 Effectiveness of SAScope

We applied SAScope to Soot 4.4.2, WALA 1.6.2, SootUp 1.1.2, and Doop 4.24.10 and conducted experiments to evaluate the effectiveness of SAScope based on the following experimental questions:

**Q1:** How many unique PRFs can SAScope identify?

**Q2:** What is the effectiveness of the property-based grouping?

**Input Program Selection.** We select the top 200 popular projects (ranked by the number of usages) in Maven Central [60] as the input programs because they are popular real-world projects. We do not reuse benchmarks from prior testing approaches of static analysis tools [173, 190, 208, 213] because: (1) benchmarks in prior work [122, 174] are designed for specific tasks, whereas JCG [173] is designed for evaluating the recall of call graphs and only uses top 50 popularity projects in Maven, Defects4J is used for debugging and program repair with only 17 projects. Both are too small to reveal faults in static analyzers. (2) some approaches [208, 213] use the official test suites to test static analyzers, but these suites only include small programs for unit testing which may not contain complex structures (e.g., programs with multiple methods are required for call graph constructions) for reaching the deep state of static analyzers and revealing faults.

All experiments were conducted on a server with Intel Xeon(R) CPU 3.20GHz and 192GB RAM. For each input library, we run SAScope on all analyzers in parallel and set the timeout *timeL* to 5 hours. Note that we do not test SAScope on analyzed issues, as it is designed based on insights from these issues. On the one hand, testing SAScope on the same issues can introduce bias. On the other hand, there are two challenges in evaluating SAScope on known issues: (1) it involves building old versions of static 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) and (2) we are missing compilable input programs to reproduce some of the known issues, but these programs can be hard to construct manually, and SAScope requires them as input. Overall, the testing times for static analyzers are 10 hours (SootUp), 32 hours (WALA), 35 hours (Soot), and 37 (Doop) hours.

### 5.3.1 Q1: Evaluating Effectiveness of SAScope

Table 5.5 shows the experimental results. We measure the effectiveness of SAScope by counting the unique faults detected by each approach ("#UniqFaults" column). We use a property-based approach (Section 5.2.2) to group warnings and consider all warnings within the same group as one unique fault. In total, SAScope can identify 19 faults in the evaluated static analyzers, five have been fixed via merged pull requests ("#Fixed" column).

Table 5.5: Effectiveness of SAScope

| Static Analyzer | #Warnings | #Groups | #UniqFaults | #Fixed |
|---|---|---|---|---|
| WALA | 26951 | 10 | 8 | 1 |
| SootUp | 31734 | 11 | 7 | 4 |
| Soot | 21051 | 6 | 3 | 0 |
| Doop | 12896 | 4 | 1 | 0 |
| Total | 92632 | 31 | 19 | 5 |

We select two faults found by SAScope to demonstrate its fault detection capability. For each fault, we present its root cause, the affected static analyzer, and how SAScope found the issue.

**A Missing Elements in Program Representation Fault in SootUp [62].** Figure 5.15 shows a fault detected by SAScope, where the call graph misses an edge from *m1* to *m2*. The fault occurs due to the incorrect resolution of the lambda invocation at lines 3–4. As WALA currently supports *MethodHandle* resolving and includes this edge in the call graph, SAScope can detect this fault via differential testing. We submit this fault and SootUp's developer replies to us saying that *it is an indirect edge which is currently not covered* as *MethodHandle* resolving is currently not supported in SootUp.

**An Incorrect Elements in Program Representation Fault in Soot [61].** Figure 5.16 shows a fault caused by the incorrect implementation of the RTA algorithm in Soot, re-

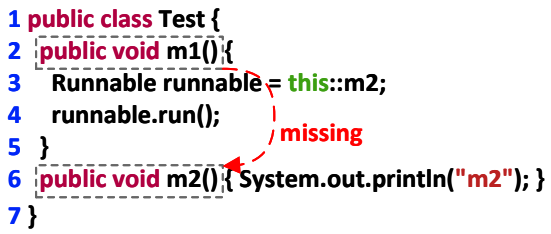


Figure 5.15: A MEPR fault in SootUp

sulting in an incorrect call edge from *main* to *Thread2.run()*. SAScope identifies it via metamorphic testing, as the CHA algorithm correctly constructed the call graph.

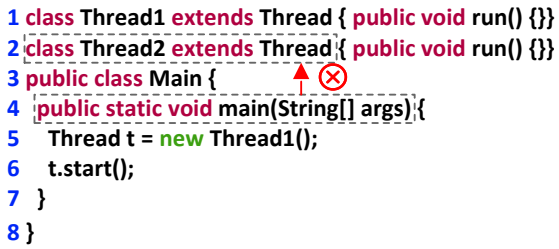


Figure 5.16: An IEPR fault in Soot

**Limitations.** Similar to other testing tools, SAScope may report false positives. Our manual analysis shows that it only reported two FPs. Both FPs are caused by minor differences among static analyzers. The first FP is due to Soot adding single quotes (‘’) around reserved words that appear in signatures. For instance, Soot uses *‘with’* when the method name is *with*, leading to the signature discrepancy with other static analyzers. The second FP is related to the handling of the *bridge* method. The *bridge* method is generated by the compiler to bridge the gap between the subclass methods with different erasure signatures and their superclass methods. WALA connects them via compiler-synthetic bridge methods, whereas Soot and SootUp directly add a call graph edge connecting methods in the subclass and superclass. Both FPs can be mitigated by configuring SAScope to consider these special cases.

### 5.3.2 Q2: Evaluating Effectiveness of Property-Based Grouping

Table 5.5 also shows the grouping results of the evaluated analyzers. The "#Warnings" column represents the total number of fault warnings and the "#Group" column shows the number of groups generated by the approach described in Section 5.2.2. In total, the property-based grouping approach can refine 92,632 warnings down to 31 distinct groups. After investigation, we find some groups represent duplicated faults and 19 of them are unique faults. We randomly sample 30 warnings from each group (930 warnings in total) to validate the uniqueness of each group; all warnings in the same group are caused by an identical root cause. This indicates that our property-based grouping helps SAScope identify unique faults and significantly reduces the time required for manual inspection of the 92,632 warnings.

## 5.4 Implication

Based on our study and findings, we discuss the implications from the perspectives of researchers and developers.

### 5.4.1 Implication for Developers

Our study identifies common symptoms and their corresponding root causes at each stage and fix strategies that may help developers of static analyzers detect, understand, and fix program representation faults. We also highlight program representations that are particularly fault-prone, implying that developers should pay attention to these representations (Finding 1). Based on the two most common symptoms of program representation faults in our study (MEPR and FPRG), we realize that developers of static analyzers tend to either (1) neglect specific program elements or (2) misunderstand the construction or optimization algorithms (Findings 2, 3). We hope our study raises awareness among developers on

the importance of generating correct program representations to improve the accuracy of analyzers. In terms of the workflow, our study reveals that *developers should pay careful attention to the core analysis stage* because all studied analyzers have the greatest number of PRFs at this stage, especially when constructing call graphs and class hierarchies. With the evolution of the Java specification, developers should also consider program representation faults when updating the grammar (Findings 3, 7). Meanwhile, our study also shows that redundant analysis operations, immutable structures, and the lack of caching can lead to performance issues. *Developers should consider these scenarios when designing static analyzers to improve analysis efficiency.*

### 5.4.2 Implication for Researchers

Our study and proposed testing framework establish a basis for research in two promising directions. First, *automated detection of program representation faults is important yet often neglected.* For a static analyzer, completing missing or eliminating incorrect elements in various forms of program representations is a worthwhile and long-term research direction [144, 190, 191]. As shown in Table 5.3, MEPR and IEPR account for 57.4% of all analyzed issues, indicating that detecting PRFs is rewarding for fixing MEPR and IEPR faults. Finding 8 suggests that metamorphic testing is a promising approach for detecting PRFs. Researchers can generate program representations in various precision levels and perform differential analysis based on their metamorphic relations to detect PRFs. Second, *our fix strategies (Findings 6, 7) serve as preliminary studies for future research on automated repair of PRFs*. We observe that several fix strategies in our study can be automated to reduce the effort required to address these issues. For example, Fix Incorrect Algorithm Design (FAD) can resolve over one-third of the issues. Most of them are implemented using similar patterns (e.g., considering a particular element in program representation) or fixing the logic errors in the generation algorithms.

## 5.5 Summary

We conduct the first empirical study focusing on characterizing and detecting program representation faults of static analyzers, as program representation lies at the core of static analysis. We study 141 issues from four representative and diverse static analyzers (Soot, WALA, SootUp, and Doop), identify four symptoms and six fix strategies. Moreover, we also summarize eight findings. Based on these findings, we introduce a set of guidelines for PRF detection and repair. We also propose SAScope, the first metamorphic and differential testing framework to automatically identify PRFs in analyzers. With a two-dimensional testing approach, SAScope can automatically inspect the generated program representations based on the top 200 popular projects in Maven Central and find 19 faults. In the experiment section, SAScope uses only program representations originating from real-world projects to detect PRFs. Future improvements to SAScope can focus on constructing new metamorphic relations that are independent of existing algorithm families. For instance, we can verify whether inserting a function call necessarily introduces a new edge in the call graph.

# Chapter 6

# Conclusion and Future Work

This chapter summarizes our contributions to static analyzer testing as presented in this thesis and outlines potential directions for future research. Section 6.1 recapitulates the major contributions, and Section 6.2 describes future work.

## 6.1 Conclusion

In this thesis, we first introduce an overview and discuss the necessity of testing static analyzers. Then, we investigate the limitations of existing work and summarize three emerging research questions affecting the reliability and usability of static analyzers. After that, we illustrate the main results and contributions of our three research projects, which address the proposed open research problems.

In the first research work, we present Statfier, a heuristic-based testing technique that automatically generates input programs via semantics-preserving program transformations to discover faults in static analyzers. Statfier relies on two key heuristics: analysis report guided location selection and structure diversity driven variant selection. Our experiments confirm that Statfier outperforms the evaluated baselines by finding more faults yet iter-

ating through fewer variants. Overall, STATFIER has discovered 79 faults, of which 46 have been confirmed.

In the second research work, we conduct the first comprehensive study focusing on understanding and detecting annotation-induced faults in static analyzers. We manually investigate 238 issues from five representative and popular static analyzers (SonarQube, CheckStyle, PMD, SpotBugs, and Infer), identify seven root causes, four symptoms, and six fix strategies. Moreover, we also summarize nine findings. Based on these findings, we introduce a set of guidelines for annotation-induced fault detection and repair, and propose the first metamorphic testing based framework ANNATESTER to automatically identify annotation-induced faults in static analyzers. Overall, our framework can detect 43 faults where 20 of them have been fixed.

In the third research work, we focus on the program representation faults of static analyzers. To understand the characteristics of this type of fault, we first conduct an empirical study on their symptoms, root causes, and fix strategies. Next, we design an automated testing approach SASCOPE, combining differential and metamorphic testing, to mining the potential faults in static analyzers. In summary, SASCOPE can identify 19 new faults and five of them have been fixed by developers.

## 6.2 Future Work

We identify three promising research directions that are worth exploring in the future.

*Understanding and detecting faults caused by various program elements in static analyzers.* Although we evaluate the impacts of annotations on static analyzers and introduce three metamorphic relations in our framework ANNATESTER to identify annotation-induced faults, the work has limitations. For instance, the framework only enables static analyzers to better handle one additional type of syntactic element, i.e., annotations. Still, there are many other types of elements, e.g., assertions, extensively used in programs. In many

popular programming languages, developers write assertions to express their expectations about the runtime states or behaviors of the programs, and when an assertion is violated, an exception is thrown and the normal execution is suspended till the exception is handled. Given that such properties are critical enough for the developers' to spend extra effort to explicitly specify, we believe such properties, if utilized properly, can help static analyzers better understand the program behaviors and improve the quality of their analysis results. We are interested in detecting two types of assertion-related static analyzer faults in the future. First, faults that prevent static analyzers from effectively identifying assertions that may not hold or will never hold. Second, faults that cause static analyzers to overlook/misuse the program properties explicitly specified in assertions and therefore, to produce false-positive or false-negative results.

*Expanding automated testing techniques to advanced static analysis techniques.* Our research work AnnaTester and SAScope focus on testing fundamental analysis techniques in static analyzers. In future research, it is worth exploring how to extend our techniques to support other advanced static analysis methods, making them more general and comprehensive. For example, taint analysis is a popular program analysis technique in the security domain, particularly for detecting privacy leaks [69]. It tracks how private information flows through a program and determines whether it may leak to public observers. Generally, this technique traces sensitive "tainted" data by starting at a predefined source (e.g., an API method returning a user identifier) and following the data flow until it reaches a sink (e.g., a method sending the information via SMS), providing precise details about potential data leaks. To test taint analysis, we can apply semantics-preserving program transformations to the source code of taint analysis tools, ensuring that the sensitive data flow remains unchanged. Then, we should compare the analysis reports before and after the transformation to identify inconsistencies. However, we face two main challenges: 1) existing program transformations may not suit the new systems under test, and the generated variants may fail to trigger potential faults; 2) the system under testing may not report warning locations and types in the analysis reports (e.g., some tools only output the infor-

mation flow of programs, which constitutes the intermediate data used for taint analysis), making our feedback-driven heuristic for differential testing ineffective. To address these challenges, we need to devise more transformation operations and summarize their characteristics. Additionally, we should develop a method to analyze a broader range of report types, beyond just those containing warning locations and types.

*Enhancing the effectiveness of testing static analyzers using learning-based techniques.* At present, artificial intelligence techniques such as large language models (LLMs) have gained significant traction. Some complex testing tasks are challenging to address with traditional methods, but substantial progress can be achieved using learning-based techniques. First, the AnnaTester testing framework cannot handle runtime-level annotations, which constitute a substantial portion of the existing annotations, because it cannot leverage the corresponding annotation processors to recover the semantics of runtime-level annotations. Incomplete semantics is also the most common root cause in static analyzers, as identified in Finding 1 of our second study. Thus, failing to analyze their semantics can impair the fault detection capability of static analyzers. To address this, we need to accurately model the behaviors of runtime-level annotations. Developers typically implement runtime-level annotations in corresponding third-party libraries (TPLs), which often have large and complex codebases. Therefore, understanding the semantics of runtime annotations and injecting them into source code via traditional approaches is difficult. LLMs, trained on massive amounts of data, possess strong semantic understanding capabilities. Generating mutants with runtime annotations using LLMs is a promising approach. Second, we use an enumeration-based strategy to mutate program elements in the source code when generating variants, which may produce many ineffective variants and consume significant time. To improve performance, we can adopt a learning-based algorithm to reduce the search space and execution time. The main challenges include constructing a high-quality training set and training an effective learning model. Third, annotations are highly configurable and can provide different functionalities by setting parameters. However, due to the infinite search space of annotation parameters, we only add annotations with default

configurations in the second study. Using LLMs or reinforcement learning-based methods to configure annotation parameters and generate diverse mutants for fault detection holds significant potential.

# References


[1] Checkstyle. URL: https://checkstyle.sourceforge.io/.

[2] Eclipse java development tools. URL: https://www.eclipse.org/jdt/.

[3] A false negative about null_dereference. URL: https://github.com/facebook/infer/issues/1628.

[4] A false positive about the rule redundantfieldinitializer. URL: https://github.com/pmd/pmd/issues/4070.

[5] How to contribute to pmd. URL: https://github.com/pmd/pmd/blob/49d35d0973d91dcd6526f67433ce701f2e291644/CONTRIBUTING.md.

[6] How to report an issue? URL: https://checkstyle.sourceforge.io/report_issue.html.

[7] Infer static analyzer. URL: https://fbinfer.com/.

[8] [java] compareobjectswithequalsrule: False positive with enums. URL: https://github.com/pmd/pmd/issues/2716.

[9] [java] rspec-2189 should consider do-while loop. URL: https://community.sonarsource.com/t/java-rspec-2189-should-consider-do-while-loop/55080.

[10] Ms_expose_rep cannot detect nested class. URL: https://github.com/spotbugs/spotbugs/issues/2042.

[11] New java versions lead to fn about the rule dmi_invoking_tostring_on_array. URL: https://github.com/spotbugs/spotbugs/issues/1874.

[12] Open-sourcing facebook infer: Identify bugs before you ship. URL: https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

[13] Parameterassignment does not detect problem for lambda parameters. URL: https://github.com/checkstyle/checkstyle/issues/11038.

[14] Pmd an extensible cross-language static code analyzer. URL: https://pmd.github.io/.

[15] Sonarqube and pmd. URL: https://github.com/wuchiuwong/Diff-Testing-01/blob/master/RulePair/Sonarqube%20and%20PMD.csv.

[16] Sonarqube code quality and code security. URL: https://www.sonarqube.org/.

[17] Spotbugs find bugs in java programs. URL: https://spotbugs.github.io/.

[18] T.j. watson libraries for analysis, 2006. URL: https://github.com/wala/WALA/.

[19] Performance improvement when caching the ir and defuse within cgnode, 2013. URL: https://github.com/wala/WALA/issues/9.

[20] todex: "check-cast on non-reference in v0" with k9mail apk, 2013. URL: https://github.com/soot-oss/soot/issues/35.

[21] Exception while building rta callgraph, 2015. URL: https://github.com/soot-oss/soot/issues/514.

[22] Infinite loop occurs in dexprinter.findlongjumps() while processing android framework, 2015. URL: https://github.com/soot-oss/soot/issues/502.

[23] java.lang.arrayindexoutofboundsexception in davaflowset when decompiling an apk, 2015. URL: https://github.com/soot-oss/soot/issues/502.

[24] java.lang.runtimeexception: Assertion failed. at soot.toolkits.graph.simpledominatorsfinder.getimmediatedominator, 2015. URL: https://github.com/soot-oss/soot/issues/416.

[25] Jvm verifyerror after transformation from .class via shimple, 2015. URL: https://github.com/soot-oss/soot/issues/385.

[26] Suppresswarnings should support camelcase, 2015. URL: https://github.com/checkstyle/checkstyle/issues/2202.

[27] Java 8 grammar: annotations on varargs parameters, 2016. URL: https://github.com/checkstyle/checkstyle/issues/3238.

[28] Reset classpath for android, 2016. URL: https://github.com/soot-oss/soot/issues/524.

[29] Emptyblock: Npe on annotation declaration, 2017. URL: https://github.com/checkstyle/checkstyle/issues/4472.

[30] Eradicate not reading annotations from class file, 2017. URL: https://github.com/facebook/infer/issues/559.

[31] [java] processing error (classcastexception) if a type_use annotation is used on a base class in the ”extends” clause, 2018. URL: https://github.com/pmd/pmd/issues/1369.

[32] java.lang.runtimeexception when ..., 2018. URL: https://github.com/soot-oss/soot/issues/875.

[33] Nondeterministic crashes from packmanager.retrieveallbodies(), 2019. URL: https://github.com/soot-oss/soot/issues/1189.

[34] Source code frontend assigns boolean to int, 2019. URL: https://github.com/soot-oss/SootUp/issues/103.

[35] Unit test lambdametafactoryadapttest: Concurrentmodificationexception, 2019. URL: https://github.com/soot-oss/soot/issues/1125.

[36] Very different results of the same analysis with different engines, 2019. URL: https://bitbucket.org/yanniss/doop/issues/1.

[37] adapt castandreturninliner to use stmt graph, 2020. URL: https://github.com/soot-oss/SootUp/issues/281.

[38] Compact constructor ast is missing annotations, 2020. URL: https://github.com/checkstyle/checkstyle/issues/8734.

[39] Doop doesn't handle names/directories with spaces, 2020. URL: https://bitbucket.org/yanniss/doop/issues/4/doop-doesnt-handle-names-directories-with.

[40] Exception when using suppresswarningsholder with @suppresswarnings as an annotation property, 2020. URL: https://github.com/checkstyle/checkstyle/issues/7522.

[41] stack underrun: Asmmethodsource.convertbinopinsn, 2020. URL: https://github.com/soot-oss/SootUp/issues/326.

[42] Unusedprivatefield cannot override ignored annotations property, 2020. URL: https://github.com/pmd/pmd/issues/2876.

[43] Atclauseorder: Falsely ignores method with annotation, 2021. URL: https://github.com/checkstyle/checkstyle/issues/9941.

[44] Operatorwrap with token assign too strict for annotations, 2021. URL: https://github.com/checkstyle/checkstyle/issues/10945.

[45] Pmd, 2021. URL: https://github.com/pmd/pmd/blob/ac26d3dc6d7c121de72e6e7ddc1769caf8986e85/pmd-java/src/test/resources/net/sourceforge/pmd/lang/java/rule/security/xml/HardCodedCryptoKey.xml.

[46] Spark missing support for collection element type, 2021. URL: https://github.com/soot-oss/soot/issues/1755.

[47] Unnecessaryconstructor: false-positive with @inject, 2021. URL: https://github.com/pmd/pmd/issues/4487.

[48] Unusedprivatemethod violation for disabled class in 6.23, 2021. URL: https://github.com/pmd/pmd/issues/2454.

[49] Weakobjecttypes should (maybe?) keep track of the sootclass, 2021. URL: https://github.com/soot-oss/soot/issues/1739.

[50] Callgraph's removeedges method maybe has a bug, 2022. URL: https://github.com/soot-oss/soot/issues/1874.

[51] A false positive about rule rspec-2095, 2022. URL: https://community.sonarsource.com/t/a-false-positive-about-rule-rspec-2095/67536.

[52] False positive about the rule usestringbufferforstringappends, 2022. URL: https://github.com/pmd/pmd/issues/4078.

[53] How pmd works, 2022. URL: https://docs.pmd-code.org/pmd-doc-6.53.0/pmd_devdocs_how_pmd_works.html.

[54] Parse error on array type annotations, 2022. URL: https://github.com/pmd/pmd/issues/4152#issuecomment-1277447394.

[55] Callgraph taking a while to build, 2023. URL: https://github.com/soot-oss/SootUp/issues/558.

[56] How to parallelize?, 2023. URL: https://github.com/soot-oss/SootUp/issues/591.

[57] The icfg dotexporter should use a call graph to decide which methods are called, 2023. URL: https://github.com/soot-oss/SootUp/issues/728.

[58] Soot - a framework for analyzing and transforming java and android applications, 2023. URL: http://soot-oss.github.io/soot/.

[59] Fix localsplitter, 2024. URL: https://github.com/soot-oss/SootUp/issues/798.

[60] Maven repository: Top projects, 2024. URL: https://mvnrepository.com/popular.

[61] Soot reports a false positive edge in call graph, 2024. URL: https://github.com/soot-oss/soot/issues/2061.

[62] Support methodhandle resolve in callgraph algorithm, 2024. URL: https://github.com/soot-oss/SootUp/issues/906.

[63] Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 15–26, 2009.

[64] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, page 688–712, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_30.

[65] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

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

[67] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1690–1701, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978383.

[68] Steven Arzt. Pedantic validation rejects code that is arguably ok, 2013. URL: https://github.com/soot-oss/soot/issues/109.

[69] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594299.

[70] Nathaniel Ayewah and William Pugh. The google findbugs fixit. pages 241–252, 01 2010. doi:10.1145/1831708.1831738.

[71] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 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*, PASTE '07, page 1–8, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1251535.1251536.

[72] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, oct 1996. doi:10.1145/236338.236371.

[73] Bernhard J. Berger. Exception in unitthrowanalysis, 2015. URL: https://github.com/soot-oss/soot/issues/358.

[74] Eric Bodden. Incomplete call graph in spark's rta mode, 2014. URL: https://github.com/soot-oss/soot/issues/297.

[75] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 243–262, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640108.

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

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

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

[79] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. Understanding exception-related bugs in large-scale cloud systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 339–351. IEEE Press, 2019. doi:10.1109/ASE.2019.00040.

[80] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1), jan 2018. doi:10.1145/3143561.

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

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

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

[84] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[85] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, page 120–125, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-28891-3_12.

[86] Andreas Dangel. [java] arrayisstoreddirectly doesn't consider nested classes. URL: https://github.com/pmd/pmd/issues/3613.

[87] Andreas Dangel. [java] unusedformalparameter doesn't consider anonymous classes. URL: https://github.com/pmd/pmd/issues/3618.

[88] Andreas Dangel. Nopackage: False negative for enums, 2019. URL: https://github.com/pmd/pmd/issues/1782.

[89] Andreas Dangel. [java] hardcodedcryptokey false negative with variable assignments, 2021. URL: https://github.com/pmd/pmd/issues/3368#issuecomment-872794683.

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

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

[92] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi:10.1145/3133917.

[93] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*, pages 44–47, 2016. doi:10.1145/2896971.2896978.

[94] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. Csmithedge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Softw. Engg.*, 27(6), November 2022. doi:10.1007/s10664-022-10146-1.

[95] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1219–1231, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598130.

[96] Clément Fournier. [java] localvariablecouldbefinal false positive in exception handling, 2024. URL: https://github.com/pmd/pmd/issues/5049.

[97] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Pro-*

*ceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020.

[98] Philip Graf. [java] localvariablecouldbefinal false positive with try/catch, 2024. URL: https://github.com/pmd/pmd/issues/5046.

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

[100] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 25–28, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183440.3183485.

[101] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, nov 2001. doi:10.1145/506315.506316.

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

[103] Michael Gumowski. Classes annotated with lombok's @equalsandhashcode should be ignored, 2016. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-1513.

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

[105] Michael Gumowski. Fp on s1128 (uselessimportcheck) when annotation is used in fully qualified name, 2017. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-2083.

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

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

[108] A. Habib and M. Pradel. 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)*, pages 317–328, 2018. doi:10.1145/3238147.3238213.

[109] Andrew Habib and Michael Pradel. 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)*, pages 317–328. IEEE, 2018.

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

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

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

[113] Yu Hu, Zekun Shen, and Brendan Dolan-Gavitt. Characterizing and improving bug-finders with synthetic bugs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 971–982. IEEE, 2022.

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

[115] Konrad Jamrozik. Regression: Error/dalvikvm(1854): Invalid type descriptor: 'dalvik.annotation.enclosingclass', 2013. URL: https://github.com/soot-oss/soot/issues/123.

[116] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94, 2012. doi:10.1109/SP.2012.15.

[117] Quentin Jaquier. Support java 11 generated annotation, 2019. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-3174.

[118] Quentin Jaquier. Consistently support nullable/checkfornull/nonnull annotations in rules, 2020. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-3536.

[119] Quentin Jaquier. S5122: Classcastexception when annotation is defined with an identifier, 2023. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-3438.

[120] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi:10.1109/ICSE.2013.6606613.

[121] Josh Juneau. Jsr 308 explained: Java type annotations, 2022. URL: https://www.oracle.com/technical-resources/articles/java/ma14-architect-annotations.html.

[122] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2610384.2628055.

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

[124] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–247. Springer, 2024. doi:10.1007/978-3-031-57246-3_13.

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

[126] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Chen, and Jinqiu Yang. Studying test annotation maintenance in the wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 62–73, 2021. doi:10.1109/ICSE43902.2021.00019.

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

[128] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal aspects of computing*, 27(3):573–609, 2015.

[129] Kota Kitaura and Nagisa Ishiura. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 38–44, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3278186.3278192.

[130] Jonas Klauke. Callgraph generation ignores ¡clinit¿, 2022. URL: https://github.com/soot-oss/SootUp/issues/459.

[131] Jonas Klauke. Callgraph: search for the concrete dispatch if the method is not implemented, 2022. URL: https://github.com/soot-oss/SootUp/issues/499.

[132] Jonas Klauke. Rta ignores new instantiated classes at a later method call, 2022. URL: https://github.com/soot-oss/SootUp/issues/456.

[133] Jonas Klauke. Rta should only consider classes as instantiated with new, 2022. URL: https://github.com/soot-oss/SootUp/issues/495.

[134] Jonas Klauke. Sources marked as library shouldn't be further analyzed in the call graph generation, 2023. URL: https://github.com/soot-oss/SootUp/issues/715.

[135] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 239–250, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3330553.

[136] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S Foster, and Adam A Porter. Satune: a study-driven auto-tuning approach for configurable software verification tools. In

*2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 330–342. IEEE, 2021.

[137] Rahul Krishna, Raju Pavuluri, Saurabh Sinha, Divya Sankar, Julian Dolby, and Rangeet Pan. Towards supporting universal static analysis using wala. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.

[138] Florian Kübler. java.lang.classcastexception when using subtypesentrypoint, 2018. URL: `https://github.com/wala/WALA/issues/285`.

[139] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[140] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. `doi:10.1109/CGO.2004.1281665`.

[141] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2594291.2594334`.

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

[143] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 327–337, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2771783.2771785`.

[144] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D. Le, and Quyet Thang Huynh. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 520–532, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3540250.3549175.

[145] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

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

[147] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-wesley, 2013.

[148] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 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*, pages 544–555, 2022.

[149] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 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*, pages 544–555, 2022.

[150] Dongmiao Liu. Sonarjava-73 add more lombok's used annotations for unusedprivatefieldcheck, 2022. URL: https://github.com/SonarSource/sonar-java/pull/102#issuecomment-87545890.

[151] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi:10.1145/3527317.

[152] Yi Liu, Yadong Yan, Chaofeng Sha, Xin Peng, Bihuan Chen, and Chong Wang. Deepanna: Deep learning based java annotation recommendation and misuse detection. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 685–696. IEEE, 2022.

[153] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[154] Austin Mordahl. Automatic testing and benchmarking for configurable static analysis tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1532–1536, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3605232.

[155] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of c compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 676–679. IEEE, 2016.

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

[157] Marharyta Nedzelska. Fp in s3077 when volatile is used with @immutable and @threadsafe annotations, 2021. URL: https://sonarsource.atlassian.net/browse/SONARJAVA-3804.

[158] Phokham Nonava. False-positive with lombok and inner classes, 2019. URL: https://github.com/pmd/pmd/issues/1641.

[159] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. Mining annotation usage rules: A case study with microprofile. In

*2022 38th International Conference on Software Maintenance and Evolution, IEEE*, 2022.

[160] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. Mining annotation usage rules: A case study with microprofile. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 553–562. IEEE, 2022.

[161] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. Randir: differential testing for embedded compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, page 21–30, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2998392.2998397.

[162] Fernando Rodriguez Olivera. Findbugs jsr305, 2017. URL: https://mvnrepository.com/artifact/com.google.code.findbugs/jsr305.

[163] Fernando Rodriguez Olivera. Mvnrepository, 2023. URL: https://mvnrepository.com/.

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

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

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

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

[168] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017. doi:10.1109/SP.2017.27.

[169] Pedro Pinheiro, José Carlos Viana, Márcio Ribeiro, Leo Fernandes, Fabiano Ferrari, Rohit Gheyi, and Baldoino Fonseca. Mutating code annotations: An empirical evaluation on java and c# programs. *Science of Computer Programming*, 191:102418, 2020.

[170] Chris Povirk. Guava, 2024. URL: https://guava.dev/.

[171] William Pugh. Jsr 305: Annotations for software defect detection, 2022. URL: https://jcp.org/en/jsr/detail?id=305.

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

[173] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 251–261, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3330555.

[174] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, page 107–112, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236454.3236503.

[175] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. Tacai: an intermediate representation based on abstract interpretation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2020, page 2–7, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3394451.3397204.

[176] Henrique Rocha and Marco Tulio Valente. How annotations are used in java: An empirical study. In *SEKE*, pages 426–431, 2011.

[177] John Rose. Jsr 292: Supporting dynamically typed languages on the javatm platform, 2011. URL: https://jcp.org/en/jsr/detail?id=292.

[178] H. Samet. A machine description facility for compiler testing. *IEEE Transactions on Software Engineering*, SE-3(5):343–351, 1977. doi:10.1109/TSE.1977.231159.

[179] Hanan Samet. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 Annual Conference*, ACM '76, page 492–497, New York, NY, USA, 1976. Association for Computing Machinery. doi:10.1145/800191.805648.

[180] Hanan Samet. A normal form for compiler testing. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, page 155–162, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800228.806945.

[181] Joanna Santos, Mehdi Mirakhorli, and Ali Shokri. Sound call graph construction for java object deserialization. *arXiv preprint arXiv:2311.00943*, 2023.

[182] Joanna C. S. Santos, Mehdi Mirakhorli, and Ali Shokri. Seneca: Taint-based call graph construction for java object deserialization. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi:10.1145/3649851.

[183] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. Rustsmith: Random differential compiler testing for rust. In *Proceedings of the 32nd ACM SIGSOFT*

*International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1483–1486, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3604919.

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

[185] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 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*, ESEC/FSE 2021, page 968–980, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3468591.

[186] Flash Sheridan. Practical testing of a c99 compiler using output comparison. *Softw. Pract. Exper.*, 37(14):1475–1488, November 2007.

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

[188] Manu Sridharan. Allow wala to support missing superclasses?, 2018. URL: https://github.com/wala/WALA/issues/322.

[189] Varun Srivastava, Michael D. Bond, Kathryn S. McKinley, and Vitaly Shmatikov. A security policy oracle: detecting security holes using multiple api implementations. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 343–354, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993539.

[190] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation. In *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*, pages 69–88. Springer, 2018. URL: https://doi.org/10.1007/978-3-030-02768-1_4.

[191] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1049–1060, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380441.

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

[193] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2931037.2931074.

[194] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 348–357, 2017. doi:10.1109/APSEC.2017.41.

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

[196] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 81–93, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368826.3377927.

[197] Daniel Tang, Ales Plsek, and Jan Vitek. Static checking of safety critical java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 148–154, 2010.

[198] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*, pages 270–279, 2010. doi:10.1109/APSEC.2010.39.

[199] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012.

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

[201] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: Pruning false-positives from static call graphs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2043–2055, 2022. doi:10.1145/3510003.3510166.

[202] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[203] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Technical report, McGill University, 1998.

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

[205] Susana M. Vieira, Uzay Kaymak, and João M. C. Sousa. Cohen's kappa coefficient as a performance measure for feature selection. In *International Conference on Fuzzy Systems*, pages 1–8, 2010. doi:10.1109/FUZZY.2010.5584447.

[206] Jules Villard. Infer workflow, 2020. URL: https://fbinfer.com/docs/infer-workflow.

[207] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 413–424. IEEE, 2017.

[208] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. Find bugs in static bug finders. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 516–527, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524610.3527899.

[209] Xiao Xiao. On the importance of program representations in static analysis. 2013.

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

[211] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference*

*on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993532.

[212] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, 47(5):969–986, 2021. doi:10.1109/TSE.2019.2910516.

[213] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 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 2023, page 237–249, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3616272.

[214] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. Understanding and detecting annotation-induced faults of static analyzers. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024. doi:10.1145/3643759.

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

[216] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1105–1116, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660267.2660359.

[217] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs, 2020.

[218] Ruide Zhang, Ning Zhang, Assad Moini, Wenjing Lou, and Y. Thomas Hou. Privacyscope: Automatic analysis of private data leakage in tee-protected applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44, 2020. doi:10.1109/ICDCS47774.2020.00013.

[219] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 129–140, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213866.