Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks

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

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

Shuyun Liang
Southern University of Science and Technology
Shenzhen, China
liangsy2022@mail.sustech.edu.cn

Zezhong Xing
Southern University of Science and Technology
Shenzhen, China
12232384@mail.sustech.edu.cn

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

Abstract

Static analysis frameworks (SAFs) such as Soot and WALA have provided the fundamental support in today's software analysis. They usually adopt various analysis techniques to transform programs into different representations that imply specific properties, e.g., a call graph can demonstrate the calling relationships between methods in a program, and users rely on these program representations for further analysis, like vulnerability detection and privacy leakage recognition. Hence, providing proper program representation is essential for SAFs. We conducted a systematic empirical study on program representation faults of static analysis frameworks. In our study, we first collected 141 issues from four popular SAFs and summarized their symptoms, root causes, and fix strategies, and revealed eight findings and some implications to avoid and detect program representation faults. Additionally, we implemented an automated testing framework named SAScope based on the metamorphic and differential testing motivated by findings and implications. Overall, SAScope can detect 19 program representation faults where 5 have been fixed, demonstrating its effectiveness.

CCS Concepts

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

Keywords

software testing, bug detection, empirical study

ACM Reference Format:
Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680398

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680398

1 Introduction

Nowadays, static analysis frameworks (SAFs) are playing a key role in automating tasks like vulnerability or privacy leakage detection [4, 90], malware recognition [87], and performance issue pinpoint [21]. Such frameworks usually construct various program representations [2, 42, 82] in the forms of call graphs, control flow graphs, intermediate representations, etc., that encode the properties and behaviors of the given program for further analysis. However, developers of static analysis frameworks 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 implementation of the complex call graph construction algorithms and missing support for certain programming language features are two main reasons for incorrectly constructed call graphs [56, 66].

Although ensuring the correctness of generated program representations is essential, prior studies mainly focused on (1) investigating one type of program representations (e.g., call graphs), (2) pruning of false positive edges from call graphs [44, 66], and (3) developing new call graph construction approaches that consider more specific aspects like invocations to Java libraries and implicit processes like object serialization/deserialization [1, 60]. To develop a good understanding of the reasons for, the impacts of, and the fixes to the faults in static analysis frameworks that lead to the aforementioned undesirable behaviors of SAFs, which we refer to as *program representation faults* (PRFs), we conducted the first empirical study of those faults in static analysis frameworks. Our study aims to answer the following research questions (RQs):

RQ1: Which program representations are more likely to be faulty? RQ1 aims to study the representations that are more prone to PRFs and require more attention from developers during construction.

RQ2: What symptoms can PRFs induce, and their root causes at each stage? RQ2 aims to understand the effects and causes of PRFs at each stage of the SAF workflow.
RQ3: What strategies do developers adopt when fixing PRFs? In RQ3, we aim to understand viable ways to fix PRFs, which is essential for reducing the efforts required to fix PRFs.
RQ4: How did the developers detect PRFs? In RQ4, we review the oracles developers used to determine whether a fault is a PRF to derive better designs on the automated detection of PRFs.

To address these research questions, we first manually collected PRFs and their patches from four popular static analysis frameworks, namely Soot, WALA, SootUp, and Doop. Subsequently, we identified four symptoms, analyzed their root causes at each stage of the static analysis workflow, and revealed six fix strategies for helping SAF developers debug and repair PRFs. We also made eight findings and discussed the implications of our study for developers and researchers. In particular, we found that, while it is difficult in general to check the correctness of program representations since they are often large and have distinct and complex structures, comparing the 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 proposed SAScope, a novel automated testing framework that detects PRFs in static analysis frameworks 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 SAFs, and (3) a property-based approach to group the detected faults. Inspired by Finding 8 in empirical study, our first key insight is that a static analysis framework commonly supports multiple program representations constructed using algorithms with various precision levels. Based on the precision lattice, we designed a metamorphic relation that detects faults among program representations generated by different algorithms with different precision levels in a SAF. We also observed that the algorithms with the same functionality are implemented in different SAFs. Since the computed results from the algorithms with the same functionality should be equivalent under the same inputs, we employed differential testing to find the discrepancies among them. We evaluated SAScope on the aforementioned four static analysis frameworks 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 conducted the first empirical study on program representation faults in static analysis frameworks, involving 141 issues from four popular SAFs.
- Inspired by the findings of our study, we implemented the SAScope automated testing framework to detect program representation faults based on metamorphic and differential testing. In our metamorphic testing component, we proposed a new metamorphic relation that uses the relative precision lattice of various program representation algorithms.
- We evaluated SAScope on four studied static analysis frameworks and found 19 new faults, five of which have been fixed by developers.

The source code of SAScope and the replication package for the experiments are available for download at https://sascope.github.io.

2 Background

2.1 Static Analysis Framework

Static analysis frameworks provide different APIs to reason about the properties and behaviours of programs without executing them. Based on the official tutorials and related literature [37, 41, 62], we roughly split the analysis processes of SAFs into three phases, namely data input/output, programming parsing, and core analysis, as shown in Figure 1. Given a program under analysis and the tool configuration as input, a SAF parses the program into an abstract syntax tree (AST) and/or generates an equivalent program in an intermediate language. After that, the SAF performs different static analyses like call graph construction and pointer analysis as per the input configuration to construct and output program representations in various forms. In other words, a SAF constructs code representations of the input program in the first stage and graphical representations of the input program in the second stage.

2.2 Program Representation

Static analysis frameworks encode the knowledge they derive about the program under consideration in representations of various forms. We reviewed existing literature [7, 13, 25, 37, 41, 76, 77] to obtain a set of eight types of program representations commonly supported by mainstream static analysis frameworks, 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 analysis framework usually parses the input program into an AST or converts the program into an intermediate language, with both forms fully encoding the semantics of the input program. For instance, Soot utilizes the Jimple [77] intermediate language and performs optimization on Jimple code, while Doop [7] uses the Shimple intermediate language, which is essentially the static single assignment (SSA) variant of Jimple. On the other hand, a static analysis framework 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 the 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 of 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.

3 Study of Program Representation Faults

3.1 Static Analysis Frameworks Selection

We select static analysis frameworks for study based on the criteria below: (1) it should be popular and widely used so that its issues are representative of practical problems faced by the users of frameworks. Particularly, we focus on frameworks with at least 100 stars

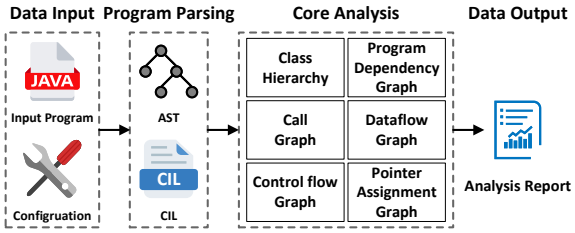

Figure 1: General Workflow of a Static Analysis Framework

on GitHub and appeared in related researches [44, 50, 56, 60, 75]; (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 analysis frameworks: (1) WALA [40] can analyze Java, Android, and JavaScript programs using many standard static program analysis techniques. (2) Soot [41] is a framework that analyzes, instruments, and optimizes Java and Android applications. (3) SootUp [37] is a new version of Soot with a completely overhauled architecture. (4) Doop [7] is a framework for Java pointer analysis.

Table 1: Issue distribution for four static analysis frameworks

SAF	#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

3.2 Issue Collection

Among the studied static analysis frameworks, 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 1 lists, for each SAF, 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 paper, we refer to the PRFs using their IDs in the form Tool-###, where Tool denotes the name of a static analysis framework, whereas ### represents the corresponding issue ID on BitBucket or GitHub. As developers have confirmed and fixed all the PRFs, we did not try to reproduce them manually.

3.3 Issue Labeling and Reliability Analysis

This study focuses on issues related to program representation faults and analyzes them from three aspects, i.e., the symptoms it exhibits, root causes at each stage, and fix strategies. The entire study took us around six months to complete. To categorize (or label) the issues from each aspect, we followed taxonomies of previous work [11, 63, 67, 86, 88, 89] and adapted them to our task. Specifically, one author first looked through all the issue reports and pull requests to determine the labels in 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 use Cohen's Kappa coefficient [78] to assess the agreement between these two authors. First, the two authors labeled 5% of the issues, and Cohen's Kappa coefficient was nearly 0.65. Then, they had a training discussion and labeled 10% of the issues (including the previous 5%). At this stage, 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 greater than 0.9 in 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.

3.4 RQ1: Bug-Prone Program Representations

In this RQ, we focus on understanding which program representations are more prone to bugs. Table 2 shows, for each SAF, the total number of PRFs we inspected and the breakdown of that number to different types of program representations.

Table 2: The total number of PRFs we inspected and the breakdown of that number to different types of program representations.

SAF	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 2 confirms that static analysis frameworks have bugs that affect all of the studied program representations. The call graph is the most bug-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. Intermediate representation is the second common bug-prone representation, especially in issues related to Soot and SootUp. The number of issues related to AST is the lowest because static analysis frameworks usually adopt class files as the input and convert them into IR for further analysis, only WALA supports mature source code frontend among four evaluated SAFs.

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

3.5 RQ2: Symptoms and Root Causes

In this section, we attempt to understand the symptoms caused by the analysed issues, their distribution across different stages of the SAF workflow, and their root causes. Overall, we summarize four symptoms. Table 3 shows the symptom distribution in the workflow of static analysis frameworks.

Table 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.

*3.5.1 Missing Elements in Program Representation (**MEPR**).* Table 3 shows that missing elements in program representation is the most popular symptom. This symptom category involves the program representations that are missing elements. Table 3 shows that this symptom may be observed in all phases, and it occurred most frequently in the core analysis stage (40/52, 76.9%). Most MEPR issues at this stage were due to improper handling of (implicit) invocations to special methods during call graph construction (23/40, 57.5%). For instance, Figure 2 shows a piece of code that reveals SootUp-459 [27]. 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 that. Besides, the developers may understand program representation construction algorithm incorrectly. In SootUp-456 [29], Rapid Type Analysis (RTA) algorithm [5] wrongly set the processed methods as completed after the first-round process as 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 3 shows SootUp-456 [29] that 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 not having any effect. In general, to construct correct call graphs, developers should (1) take note of special methods like *clinit* and concrete methods in an abstract class, (2) consider the inheritance relationships between classes, and (3) traverse the graphs to obtain information of all classes.

```
1 class A {
2   static { System.out.println("A.<clinit>"); }
3   public void foo(){
4     System.out.println("foo");
5   }
6 }
```

missing

Figure 2: SootUp incorrectly processes *clinit* methods

```
1 class B { public void target() {} }
2 class C extends B { public void target() {} }
3 class D extends B { public void target() {} }
4 class A {
5   public void foo() {
6     process(new D());
7     second();
8   }
9   public void process(B b) { b.target(); }
10  public void second() { process(new C()); }
11}
```

missing

Figure 3: SootUp wrongly implemented RTA algorithm

Meanwhile, class hierarchy and control flow graph are other common representations triggering MEPR issues at core analysis stage. For instance, in WALA-322 [51], 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 [70], Soot tried to convert 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 4. 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*.

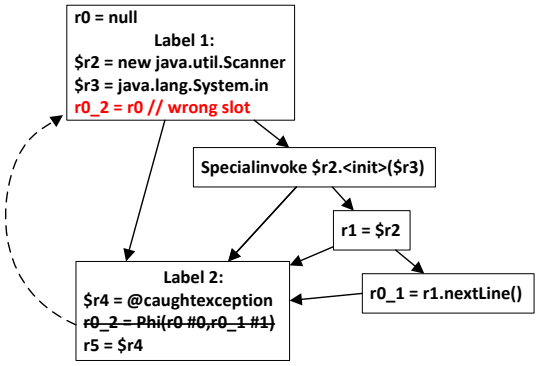

Figure 4: Soot-385: Incorrect control flow graph construction

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 [22], developers did not reset the variable that configures 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.

*3.5.2 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 program parsing stage, most issues are due to lack of consideration of specific language features. Figure 5 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 SAFs (except for Doop) have issues parsing lambda expressions.

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

Figure 5: Wrong IR construction for lambda expression

The second root cause triggering FPRG is incorrect intermediate representation (IR) optimization. SAFs include many optimization algorithms to optimize IR (e.g., dead assignment and unused local variable elimination [41]), but these algorithms may have implementation errors. For example, in Soot-358 [6], 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, SAFs adopt an operand stack for simulation. Figure 6 shows SootUp-326 [35] that developers first initialized the operand stack before block 1 to simulate the IR construction of the left path (1->2->4). However, they did 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.

Finding 3: Three common root causes for failed program representation generation at program parsing stage are (1) missing consideration of specific program elements, (2) incorrect IR optimization, (3) wrong operand stack for simulating 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. Class hierarchy is often accessed and modified by different threads. Figure 7 shows Soot-1189 [53] that different threads simultaneously invoked *getOrMakeFastHierarchy*, leading to runtime error due to concurrent hierarchy construction. Besides, inadequate handling of edge representation is the second most popular root cause. In Soot-416 [26], Soot threw an runtime exception when handling a graph where the dominator of a block is the same as the block itself.

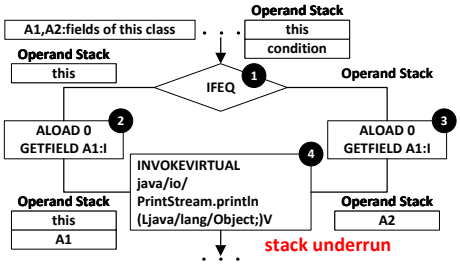

Figure 6: SootUp-326 [35]: An operand stack underrun issue

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

Figure 7: The concurrency bug in class hierarchy of Soot

*3.5.3 Incorrect Elements in Program Representation (**IEPR**).* IEPR is the third common symptom (29/141, 20.6%). This category involves program representations with incorrect or redundant program elements. 62.1% IEPR issues occurred at core analysis stage. For example, in SootUp-495 [30], developers fixed the RTA implementation by only considering `new()` expression as the instantiated class rather than *init* method in bytecode. Otherwise, RTA may wrongly regard statements like *super();* as the class instantiation, leading to incorrect edges in the call graph. In SootUp-715 [32], developers did not add the essential libraries from the configuration for specifying dependency (used to provide knowledge for analyzing input programs) to the call graph, and SAFs should not perform core analyses to the libraries. The remaining 37.9% issues are related to program parsing stage, including two root causes: (1) incorrect type assignment and (2) miscompilation in compiler-synthetic methods. Figure 8 shows an example of the incorrect type assignment in SootUp-103 [9] where the developer incorrectly resolved boolean expression at line 1 to integer type because SootUp did not support boolean type and used integer 1 or 0 to represent true or false, resulting in an incorrect return statement at line 8. Figure 9 shows an example of miscompilation in compiler-synthetic

```
1 public boolean logicalOr(boolean a,boolean b){ return a||b; }
    1 r0 := @this:BinaryOperations
    2 $z0 := @parameter0: boolean
    3 $z1 := @parameter1: boolean
    4 if $z0 ==0 goto $i0 = $z1
    5 $i0 = 1
    6 goto [?= return $i0]
    7 $i0 =$z1
    8 return $i0
```

Figure 8: Incorrect boolean expression resolution in SootUp

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 9: Wrong *return* of a compiler-synthetic method

Finding 4: IEPR mainly appeared at core analysis and program parsing stages. Developers should take note of (1) type resolution and (2) IR generation of compiler-synthetic methods to avoid it.

*3.5.4 Inefficient Program Representation Generation (**IPRG**).* This symptom category occurs when the time cost for the program representation exceeds the expectations of developers or users. Although this symptom occurs less frequently (11/141, 7.8%), it still affects the usability of SAFs. 72.7% (8/11) IPRG issues arose during core analysis phase. This phase involves three common root causes. First, developers implemented inefficient core analysis algorithms (e.g., in SootUp-728 [31], CHA is invoked twice to resolve call sites for the inter-procedural CFG construction, which is time-consuming). The second root cause is using immutable data structures. In SootUp-281 [68], developers used Guava library [54] to implement the graph structures but Guava only provided immutable structures, and each operation on them involved costly deep copying of objects. The third root cause is the lack of a cache for storing program representation. In WALA-9 [14], developers repeatedly constructed the IR and def-use chain (two time-consuming operations) of a method. To avoid the performance downgrading, developers cached the IR and def-use chain and adopted *WeakReference* to relieve GC release problem. 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 cache to store results of time-consuming operations.

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

3.6 Fix Strategy

We uncovered six fix strategies for fixing PRFs. In this section, we first summarize each strategy and show fix patterns in each strategy. Table 4 shows the fix strategy distribution among the four SAFs.

*3.6.1 Fix Improper Program Representation Construction Algorithm Design (**FAD**).* Table 4 shows that FAD is the most popular fix strategy (49/141). This strategy fixes design flaws like misunderstanding the construction algorithm or neglecting specific program elements that affect the results. FAD involves three patterns: (1) *adding functionality to handle specific program elements*, e.g., concrete methods

Table 4: Distribution of fix strategy among SAFs

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

in abstract class or interface. In Soot-514 [79], developers forgot to consider all types of concrete methods when computing the local pointer assignment graphs (PAG). To fix the bug, developers change the conditions to filter these unprocessed methods. (2) *fixing incorrect logic of algorithm implementation.* Figure 10 shows Soot-486 [48] that developers inserted *goto* instructions to implement control flow jumps from current instruction S to target instruction T. However, due to the restriction of jump distance, one *goto* cannot fulfill 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 a binary search to determine the farthest slot a *goto* instruction can reach and insert it, closing the gap to T. (3) *avoiding redundant computational operations.* In SootUp-728 [31], developers constructed call graphs twice to build the inter-procedural CFG. Hence, they removed the second CG construction and reused previous results.

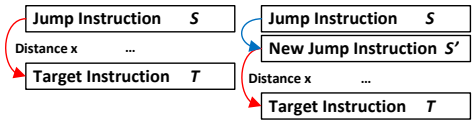

Figure 10: Incorrect *goto* instruction insertion

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

*3.6.2 Fix Incorrect Program Element Resolution (**FIR**).* The second common fix strategy is FIR (38/141). SAFs rely on the resolution to understand the type and property of a specific program element (e.g., for a call site, the resolution analyzes the symbolic information attached to the call site and finds the actual method to be invoked). Fixing the call site resolution of *invokedynamic* instruction introduced by JSR-292 [59] is a popular pattern in FIR. In WALA-285 [20], developers misused *invokedynamic* handler to resolve the statement *new LambdaMetaFactory*, causing a crash. To fix this, they changed the condition of resolving the *invokedynamic* instruction to filter *<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 the potential targets when the call site target defined in the superclass

does not have specific implementation. In SootUp-499 [28], SootUp did not find the concrete method implementation in the current class, and the fix involves 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 [24], *WeakObjectType* is a subclass of *RefType* which 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 [73], 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 site, repairing expression type assignment, and accomplishing type resolutions due to inheritance.

*3.6.3 Fix Incorrect Program Representation Traversal (**FIT**).* Fixing incorrect program representation traversal is the third 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, e.g., in Soot-875 [52], developers did not visit and resolve the inner class so the fix involves rounding out the traversal. The second pattern is fixing incorrect setting 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 [36]. Developers added a state label to mark the visited statement. Figure 6 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 stack instead of directly using the operand stack attached in the entry block.

*3.6.4 Fix Incorrect Processing of Configuration (**FPC**).* FPC includes nine issues that fixes incorrect configuration. We divide them into two categories. The first category is fixing incorrect logic for processing configurations. For example, in Doop-4 [74], 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 to avoid segmentation. Another category is adding new options to offer more analysis methods for users to choose. For instance, Figure 11 shows Soot-109 [64] that the pedantic throw analysis added an edge from each statement in the try branch to catch handler (namely statements at lines 3–7 to handler 8–10). However, the edge from line 6 to 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.

*3.6.5 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:

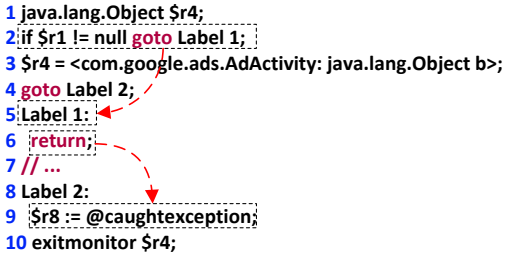

Figure 11: Pedantic throw analysis leads to verification error

(1) using thread-safe data structures (e.g., in SootUp-591 [10], as *HashMap* cannot support concurrent operations, developers replaced it with *Cache* from Guava); (2) use synchronized statements (e.g., in Soot-1125 [33], different threads simultaneously visited the hierarchy triggering concurrent modification exception, and developers added *synchronized* to the read(), write(), and clear() methods of 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.

*3.6.6 Fix Neglected Language Feature (**FLF**).* Due to frequent updates of Java/Android versions, static analysis frameworks may not meet the specification requirements. For instance, in Soot-35 [69], developers considered the neglected annotations in Dalvik bytecode that Jimple converts.

3.6.7 Others. Two faults were not fixed by previously discussed fix strategies. In Soot-502 [16], developers wrongly used decrement operators (*index--* instead of *--index*), causing out-of-bounds read error. In Soot-1874 [45], *this* reference was missing on the object leading to *edges.remove(edges)* which should be *this.edges.removeedges*.

3.7 RQ4: Oracle Design

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

Most issues of missing or incorrect elements in program representation are found by the analysis results violating the expectations of SAF users or the unit tests manually crafted by the developers. These two approaches rely on the prior knowledge of developers and users. Hence, they cannot be directly applied to detect program representation faults automatically. Another two approaches are based on the comparison between analysis results. The first approach compares the analysis results between algorithms with the same purpose but different precision [17, 19], and the second approach compares the analysis results from different tools with the same target (e.g., Doop-1 [73] 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 were recognized by exceptions that led to crashes. Inefficient program representation generation issues were usually 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 [81]). Hence, failure to achieve results within a conservative

time limit can be an oracle for detecting performance issues. The others 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.

4 Methodology of SAScope

We proposed and implemented SAScope, an automated testing framework to detect program representation faults via two-dimensional testing. Figure 12 shows the overall workflow of SAScope. It involved two main parts: metamorphic testing and differential testing components, which are inspired by our study Finding 8. First, we give a formal definition 4.1 of program representation based on existing work [2, 42, 82]. For instance, a node V in the call graph is a method representing caller or callee, and an edge E represents the calling relationship between two methods.

Definition 4.1 (Program Representation). Program representations contain different forms of modeling programs, including graphical representations and instruction representations. For a graphical representation, we adopt a directed graph $\mathcal{G} =< V, E >$ to describe it formally. 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 the instruction representation, we use a list $\mathcal{L} = [l_1, \ldots, l_n](n \geq 1)$ to define it, and each element $l_i (1 \leq i \leq n)$ in $\mathcal{L}$ represents an instruction.

Algorithm 1 shows more details of these two components, and currently we focus on testing graphical program representation. Given an input program p in *Progs*, SAScope first invoke a static analysis framework S via the invocation template (lines 4–5) to get program representations via 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 (Line 6–8). For the program representations generated by identical SAF with different algorithms, SAScope leverages metamorphic testing at lines 9–16 to reveal potential faults. Then, the analysis results will be appended to R. In lines 18–25, SAScope uses differential testing to inspect program representations from different SAFs with the same input program and analysis algorithm. It records the inconsistencies between them as potential faults.

4.0.1 Invocation Template. For input programs to be analyzed, we designed a template to invoke various analysis modules in evaluated static analysis frameworks (SAFs). For a SAF, the template will retain all configurations except for input programs before starting an analysis. Figure 13 shows a template example used to activate RTA analysis in Soot. For configurations not involved in the templates, we reuse the default values of the evaluated SAFs.

4.1 Testing Approaches and Oracle Design

In general, SAScope depends on two testing components to recognize program representation faults. Both of them rely on the

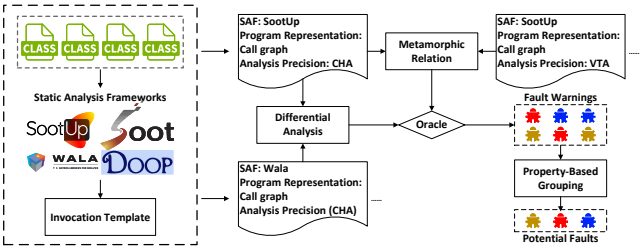

Figure 12: Overall Workflow of SAScope

Algorithm 1: Overall SAScope Testing Approach

Input: Input programs *Progs*, a set of static analysis frameworks *SAFs*, timeout *timeL*
Output: A set of potential issues in static analysis frameworks I

```
1  I ← ∅
2  for P ∈ Progs do
3      R ← [ ]
4      for S ∈ SAFs 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 ← ∅
       /* Ψ denotes algorithms supported by S, sorted by the
          precision in ascending order                          */
29     while execTime < timeL do
30         for ψ ∈ Ψ do
               /* Invoke S on P to get analysis results        */
31             = S.invoke(P, ψ)
32             ResList.append(Result)
33     return ResList
```

```
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 13: A template example for Soot RTA algorithm

analysis results of the static analysis frameworks. In the rest of this section, we adopt ϕ denotes a program representation and δ denotes its generation algorithm.

4.1.1 Metamorphic Testing. Our key insight is that static analysis frameworks 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 [25] includes the proofs for many algorithms in the precision lattice, we extract and focus only on the related algorithms supported by evaluated SAFs. Figure 14 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 14 shows that RTA is more precise than CHA as it prunes nodes where internal types are never instantiated [5]. Currently, our metamorphic relation only conservatively supports algorithms for call graphs and pointer assignment graphs that have been proven in prior work [25]. In the future, it is worthwhile to include other program representations with different precision levels (e.g., IR with different precision [58]) to detect more faults. Def. 4.3 presents the metamorphic relation used in our metamorphic testing.

Definition 4.2 (Less Precise Operator ($\preccurlyeq$)). Given two program representation construction algorithms δ_1 and δ_2, we denote $\delta_1 \preccurlyeq \delta_2$ if and only if δ_1 is less precise than δ_2 based on the precision lattice in Figure 14.

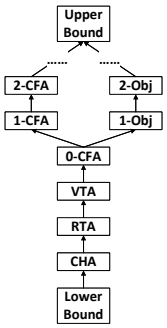

Figure 14: Relative precision lattice of tested algorithms

Definition 4.3 (Metamorphic Relation). Given the program representation ϕ_1 and ϕ_2 generated by δ_1 and δ_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 4.3, $ResList_{i+1}.V$ (line 10 in Algorithm 1) should be a subset of $ResList_i.V$ as 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 4.3 will be regarded as a fault.

4.1.2 Differential Testing.

Definition 4.4 (Equivalent Program Representation). Two program representation ϕ_1 and ϕ_2 are equivalent if and only if (1) $\mathcal{G}_1 = \mathcal{G}_2$ or $\mathcal{L}_1 = \mathcal{L}_2$; (2) ϕ_1 and ϕ_2 are generated by the same algorithm.

We observe that different static analysis frameworks usually implement the same analysis algorithms, and they should produce analysis reports with equivalent program representation (Def. 4.4) 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 $R_{i+1}.v$, which are generated by the same algorithm in different static analysis frameworks (lines 18–19 in Algorithm 1), 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.

4.2 Property-Based Grouping

As too many warnings are reported from previous steps, we need to group them based on the underlying root causes to reduce the manual efforts in examining these warnings. As observed from Findings 3–4 and 7, most program representation faults have their own syntax and type features. Hence, we consider several characteristics to classify the detected warnings: (1) the testing approach; (2) types of related static analysis frameworks;(3) types of related generation algorithms; (4) fine-grained properties of related program elements (e.g., for a call graph bug, we consider the invocation instruction type and the access modifiers of the caller and callee). Figure 15 shows a missing element in program representation fault found by SAScope, the left is the minimized code example, and the right is the group that includes this fault. Incorrect implementation of CHA algorithm in WALA caused this fault, leading to a missing call edge from constructor *Test()* to *foo()*. We identified it via metamorphic testing as 0-CFA algorithm correctly constructed the call graph.

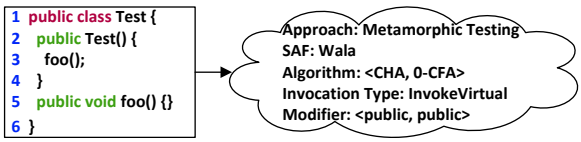

Figure 15: A MEPR fault in WALA and its group

5 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 experimental questions below:

Q1: How many unique bugs can SAScope find?
Q2: What is the effectiveness of the property-based grouping?

Input Program Selection. We selected the top 200 popular projects (ranked by the number of usages) in Maven Central [23] as the input programs because they are popular real-world projects. We did not reuse benchmarks from prior testing approaches of static analysis tools [56, 65, 80, 85] because: (1) benchmarks in prior work [34, 57] are designed for specific tasks, whereas JCG [56] was designed for evaluating the recall of call graphs and only used top 50 popularity

projects in Maven, Defects4J was used for debugging and program repair with only 17 projects. Both of them are too small to reveal bugs in static analysis frameworks. (2) some approaches [80, 85] used the official test suites to test static analysis tools, but the test suites of SAFs usually 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 deep state of SAFs 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 ran SAScope on all analyzers in parallel and set the timeout *timeL* as 5 hours. Note that we did not test SAScope on analyzed issues as it was designed based on insights from these issues. Testing SAScope on the same issues would introduce bias. There are two challenges in evaluating SAScope on known issues: (1) it involves building old versions of frameworks 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 SAFs are 10 (SootUp), 32 (WALA), 35 (Soot), and 37 (Doop) hours.

5.1 Q1: Evaluating Effectiveness of SAScope

Table 5 shows the experiment 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 4.2) to group warnings and consider all warnings within the same group as one unique bug. In total, SAScope found 19 faults in evaluated SAFs, 5 have been fixed via merged pull requests ("#Fixed" column).

Table 5: Effectiveness of SAScope

SAFs	#Warnings	#Groups	#UniqFaults	#Fixed
WALA	26951	10	8	1
SootUp	31734	11	7	4
Soot	21051	6	3	0
Doop	12896	4	1	0
Overall	92632	31	19	5

5.1.1 Case Study. We select two faults found by SAScope to show SAScope's fault detection capability. For each fault, we present its root cause, the affected analyzer, and how SAScope found the issue.

A Missing Elements in Program Representation Fault in SootUp [71]. Figure 16 shows a fault detected by SAScope where the call graph misses an edge from *m1* to *m2*. The fault occurs due to the incorrect resolving of the lambda invocation at lines 3–4. As WALA currently supports *MethodHandle* resolving and has this edge in the call graph, SAScope can detect this via differential testing. We submitted this fault and SootUp's developer replied 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 [38]. Figure 17 shows a fault caused by the incorrect implementation of RTA algorithm in Soot, causing an incorrect call edge

```
1 public class Test {
2  public void m1(){
3   Runnable runnable = this::m2;
4   runnable.run();
5  }
6  public void m2(){ System.out.println("m2"); }
7 }
```
missing

Figure 16: A MEPR fault in SootUp

from *main* to *Thread2.run()*. SAScope found it via metamorphic testing as the CHA algorithm correctly constructed the call graph.

```
1 class Thread1 extends Thread { public void run() {}}
2 class Thread2 extends Thread { public void run() {}}
3 public class Main {
4  public static void main(String[] args){
5   Thread t = new Thread1();
6   t.start();
7  }
8 }
```

Figure 17: 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 the minor difference among SAFs. The first FP is due to Soot adding single quotes ("'") around reserved words that appear in signatures. For instance, Soot use *'with'* when the method name is *with*, leading to the signature discrepancy with other SAFs. Another FP is related to the *bridge* method handling. The *bridge* method is generated by the compiler to fill the gap between the subclass methods with different erasure signatures and their superclass methods. WALA connect them via compiler-synthetic bridge methods, but Soot and SootUp directly add a CG edge connecting methods in subclass and superclass. Both FPs can be removed by configuring SAScope to consider these special cases.

5.2 Q2: Evaluating Effectiveness of Grouping

Table 5 also shows the grouping results of evaluated SAFs. The "#Warnings" column represents the number of overall fault warnings and the"#Group" column shows the number of groups generated by the approach in Section 4.2. In total, the property-based grouping approach can refine 92632 warnings down to 31 groups. After investigation, we found some groups represent duplicated bugs and 19 of them are unique faults. We randomly sampled 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 identical root cause. This indicates that our property-based grouping helps SAScope identify unique bugs and save time in manually checking the 92632 warnings.

6 Implication

Based on our study and findings, we discuss below implications:

Implication for Developers. Our study identifies the common symptoms and their corresponding root causes at each stage and fix strategies that may help developers of static analysis frameworks to

detect, understand, and fix program representation faults. We also study program representations that are bug-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 realized that developers of static analysis frameworks tend to either (1) neglect specific program elements or (2) misunderstand the construction or optimization algorithms (Finding 2–3). We hope that our study will raise awareness among developers on the importance of generating correct program representations to improve the accuracy of SAFs. In terms of the workflow, our study revealed that *developers should pay careful attention to the core analysis stage* because all studied SAFs have the greatest number of PRF faults at this stage, especially when constructing a call graph and class hierarchy. With the evolution of Java specification, developers should also consider program representation faults when updating the grammar (Finding 3,7). Meanwhile, as our study also revealed that redundant analysis operations, immutable structures, and the lack of cache can lead to performance issues, *developers should consider these scenarios when designing SAFs to improve the efficiency of the analysis.*

Implication for Researchers. Our study and proposed framework establish a basis for research in two promising directions. First, *automated detection of program representation faults is important but yet often neglected.* For a static analysis framework, completing missing or eliminating incorrect elements in various forms of program representations is a worthwhile and long-term research direction [44, 65, 66]. As shown in Table 3, MEPR and IEPR account for 57.4% in all analyzed issues, which means detecting PRFs is rewarding for fixing MEPR and IEPR faults. Finding 8 indicates that metamorphic testing is a promising approach for detecting PRFs. Researchers can produce 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 in fixing them. (e.g., Fix Incorrect Algorithm Design (FAD) can fix 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.

7 Threats to Validity

External. Our study may not generalize beyond the studied frameworks and other programming languages beyond Java. To ensure the generalizability of our study, we select four representative static analysis frameworks based on our selection criteria in Section 3.1, and we systematically examine issues in these frameworks.

Internal. Manually categorizing PRFs may be biased. To reduce this threat, we refer to prior literature to obtain program representations commonly used by target SAFs, and we adopt existing taxonomies before conducting our empirical study using an open-coding scheme. Our code and scripts may have bugs that can affect the results. To mitigate this, we open-sourced our tool and data.

8 Related Work

Studies related to Program Representation Faults. Several prior studies focus on evaluating and improving unsoundness and recall of call graphs [44, 56, 65, 66]. Our work differs from the prior study in several key aspects: (1) we conduct the first study to understand program representation faults of static analysis frameworks, covering the bug-prone stages of the analysis workflow, symptoms, root causes, and fix strategies; and (2) we propose metamorphic relations and differential testing to automatically detect program representation faults (which includes several call graph algorithms and pointer assignment graphs). Several techniques [18, 47, 55, 61] proposed novel program representation algorithms to support new language features or improve its performance, but they did not focus on program representation faults.

Testing Static Analysis Techniques. Several techniques have been proposed for testing static analysis tools [3, 12, 15, 39, 46, 50, 72, 80] or compiler fuzzing [83, 84]. The most related work to us is ECSTATIC [50], which can identify partial order relation bugs of configuration in static analyzers. However, most program representation faults are not triggered by configurations (only 6%, 9/141 issues in our study in Section 3.6.4) are related to configuration), and SAScope can perform differential testing to detect faults across different SAFs (but ECSTATIC only focuses on configuration faults without differential testing). Several techniques use metamorphic relation to solve the lack of oracle problem in testing static analyzer tools [8, 43, 49, 85, 86]. Similarly, we use metamorphic relation to address the oracle problem, but our metamorphic relation uses the relative precision lattice of analysis algorithms. Meanwhile, prior approaches [72, 85] specialized in finding other types of faults in static analysis tools, whereas our technique focuses on finding program representation faults.

9 Conclusion

We conducted the first empirical study which focuses on characterizing and detecting program representation faults of static analysis frameworks (SAFs) as program representation is the core of SAFs. We studied 141 issues from four representative and diverse SAFs (Soot, WALA, SootUp, and Doop), identify four symptoms and six fix strategies. Moreover, we also summarized eight findings. Based on these findings, we introduced a set of guidelines for PRF detection and repair. We also proposed SAScope, the first metamorphic and differential testing framework to automatically find PRFs in SAFs. With a two-dimensional testing approach, SAScope can automatically inspect the generated program representations based on top the 200 popular projects in Maven Central and find 19 faults. Currently, SAScope uses program representations generated by real-world projects to detect PRFs. In the future, we can improve SAScope by designing mutation operators specialized in generating diverse program representations to detect more PRFs.

Acknowledgments

We thank all anonymous reviewers for their insightful comments. We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for their funding support for this work. Yu Pei's work is supported in part by the Hong Kong Polytechnic University Fund under Grants P0051205 and P0051074.

References

[1] Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming*. Springer, 688–712.

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

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

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[5] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 324–341.

[6] bergerbd. 2015. *Exception in UnitThrowAnalysis.* https://github.com/soot-oss/soot/issues/358

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.

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

[9] cbruegg. 2019. *Source code frontend assigns boolean to int.* https://github.com/soot-oss/SootUp/issues/103

[10] ceclin. 2023. *How to parallelize?* https://github.com/soot-oss/SootUp/issues/591

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

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

[13] WALA Community. 2006. *T.J. Watson Libraries for Analysis.* https://github.com/wala/WALA/

[14] cos. 2013. *Performance improvement when caching the IR and DefUse within CGNode.* https://github.com/wala/WALA/issues/9

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

[16] elephantom. 2015. *java.lang.ArrayIndexOutOfBoundsException in DavaFlowSet when decompiling an APK.* https://github.com/soot-oss/soot/issues/502

[17] ericbodden. 2014. *Incomplete call graph in Spark's RTA mode.* https://github.com/soot-oss/soot/issues/297

[18] Zhiyu Fan, Shin Hwei Tan, and Abhik Roychoudhury. 2023. Concept-Based Automated Grading of CS-1 Programming Assignments. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 199–210.

[19] flankerhqd. 2021. *SPARK missing support for collection element type.* https://github.com/soot-oss/soot/issues/1755

[20] florian kuebler. 2018. *java.lang.ClassCastException when using SubtypesEntrypoint.* https://github.com/wala/WALA/issues/285

[21] foundry. 2018. *Surprising Uses of Static Analysis: Performance Optimization.* https://www.grammatech.com/learn/surprising-uses-of-static-analysis-performance-optimization/

[22] fripSide. 2016. *Reset ClassPath for Android.* https://github.com/soot-oss/soot/issues/524

[23] frodriguez. 2024. *Maven Repository: Top Projects.* https://mvnrepository.com/popular

[24] giltho. 2021. *WeakObjectTypes should (maybe?) keep track of the SootClass.* https://github.com/soot-oss/soot/issues/1739

[25] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[26] GUIpsp. 2015. *java.lang.RuntimeException: Assertion failed. at soot.toolkits.graph.SimpleDominatorsFinder.getImmediateDominator.* https://github.com/soot-oss/soot/issues/416

[27] JonasKlauke. 2022. *Callgraph Generation ignores <clinit>.* https://github.com/soot-oss/SootUp/issues/459

[28] JonasKlauke. 2022. *Callgraph: search for the concrete dispatch if the method is not implemented.* https://github.com/soot-oss/SootUp/issues/499

[29] JonasKlauke. 2022. *RTA ignores new instantiated classes at a later method call.* https://github.com/soot-oss/SootUp/issues/456

[30] JonasKlauke. 2022. *RTA should only consider Classes as instantiated with new.* https://github.com/soot-oss/SootUp/issues/495

[31] JonasKlauke. 2023. *The ICFG DotExporter should use a call graph to decide which methods are called.* https://github.com/soot-oss/SootUp/issues/728

[32] JonasKlauke. 2023. *Sources marked as Library shouldn't be further analyzed in the call graph generation.* https://github.com/soot-oss/SootUp/issues/715

[33] jpstotz. 2019. *Unit test LambdaMetaFactoryAdaptTest: ConcurrentModificationException.* https://github.com/soot-oss/soot/issues/1125

[34] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[35] kadirayk. 2020. *stack underrun: AsmMethodSource.convertBinopInsn.* https://github.com/soot-oss/SootUp/issues/326

[36] kadirayk. 2024. *Fix LocalSplitter.* https://github.com/soot-oss/SootUp/issues/798

[37] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 229–247.

[38] kitty 1998. 2024. *Soot reports a false positive edge in call graph.* https://github.com/soot-oss/soot/issues/2061

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

[40] Rahul Krishna, Raju Pavuluri, Saurabh Sinha, Divya Sankar, Julian Dolby, and Rangeet Pan. 2023. Towards Supporting Universal Static Analysis using WALA. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

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

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

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

[44] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. 2022. 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*. 520–532.

[45] learncat163. 2022. *CallGraph's removeEdges method maybe has a bug.* https://github.com/soot-oss/soot/issues/1874

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

[47] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–40.

[48] madberries. 2015. *Infinite loop occurs in DexPrinter.findLongJumps() while processing android framework.* https://github.com/soot-oss/soot/issues/502

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

[50] Austin Mordahl. 2023. Automatic Testing and Benchmarking for Configurable Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1532–1536.

[51] msridhar. 2018. *Allow WALA to support missing superclasses?* https://github.com/wala/WALA/issues/322

[52] Naplues. 2018. *java.lang.RuntimeException when ...* https://github.com/soot-oss/soot/issues/875

[53] nrosner. 2019. *Nondeterministic crashes from PackManager.retrieveAllBodies().* https://github.com/soot-oss/soot/issues/1189

[54] Chris Povirk. 2024. *Guava.* https://guava.dev/

[55] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 474–486.

[56] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. 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*. 251–261.

[57] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 107–112.

[58] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. 2020. 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*. 2–7.

[59] John Rose. 2011. *JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform.* https://jcp.org/en/jsr/detail?id=292

[60] Joanna Santos, Mehdi Mirakhorli, and Ali Shokri. 2023. Sound Call Graph Construction for Java Object Deserialization. *arXiv preprint arXiv:2311.00943* (2023).
[61] Joanna CS Santos, Reese A Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. 2021. Serialization-aware call graph construction. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 37–42.
[62] Joanna Cecilia Da Silva Santos and Julian Dolby. 2022. Program Analysis using WALA. In *ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[63] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.
[64] StevenArzt. 2013. *Pedantic validation rejects code that is arguably ok.* https://github.com/soot-oss/soot/issues/109
[65] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. 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*. Springer, 69–88.
[66] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1049–1060.
[67] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 348–357.
[68] swissiety. 2020. *adapt CastAndReturnInliner to use Stmt Graph.* https://github.com/soot-oss/SootUp/issues/281
[69] ThomasGP. 2013. *toDex: "check-cast on non-reference in v0" with k9mail APK.* https://github.com/soot-oss/soot/issues/35
[70] tim hoffman. 2015. *JVM VerifyError after transformation from .class via shimple.* https://github.com/soot-oss/soot/issues/385
[71] tisble. 2024. *Support MethodHandle Resolve in Callgraph Algorithm.* https://github.com/soot-oss/SootUp/issues/906
[72] David A Tomassi and Cindy Rubio-González. 2021. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 292–303.
[73] Former user. 2019. *Very different results of the same analysis with different engines.* https://bitbucket.org/yanniss/doop/issues/1
[74] Former user. 2020. *DOOP doesn't handle names/directories with spaces.* https://bitbucket.org/yanniss/doop/issues/4/doop-doesnt-handle-names-directories-with
[75] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a balance: pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*. 2043–2055.
[76] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[77] Raja Vallee-Rai and Laurie J Hendren. 1998. *Jimple: Simplifying Java bytecode for analyses and transformations.* Technical Report. Technical report, McGill University.
[78] Susana M Vieira, Uzay Kaymak, and João MC Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *International conference on fuzzy systems*. IEEE, 1–8.
[79] vitaliiavdiienko. 2015. *Exception while building RTA CallGraph.* https://github.com/soot-oss/soot/issues/514
[80] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find Bugs in Static Bug Finders. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 516–527. https://doi.org/10.1145/3377811.3380380
[81] will leeson. 2023. *Callgraph taking a while to build.* https://github.com/soot-oss/SootUp/issues/558
[82] Xiao Xiao. 2013. On the Importance of Program Representations in Static Analysis. (2013).
[83] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1127–1139.
[84] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*. 435–450.
[85] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery (ACM).
[86] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 722–744.
[87] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1105–1116.
[88] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1159–1170.
[89] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.
[90] Yunhui Zheng and Xiangyu Zhang. 2013. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 652–661.

Received 2024-04-12; accepted 2024-07-03