2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

An Extensive Empirical Study of Nondeterministic Behavior in Static Analysis Tools

Miao Miao
University of Texas at Dallas
Richardson, TX, USA
mmiao@utdallas.edu

Austin Mordahl
University of Texas at Dallas
Richardson, TX, USA
austin.mordahl@utdallas.edu

Dakota Soles
University of Texas at Dallas
Richardson, TX, USA
dakota.soles.j@gmail.com

Alice Beideck
University of Texas at Dallas
Richardson, TX, USA
alice.beideck@utdallas.edu

Shiyi Wei
University of Texas at Dallas
Richardson, TX, USA
swei@utdallas.edu

***Abstract*—Recent research has studied the importance and identified causes of nondeterminism in software. Static analysis tools exhibit many risk factors for nondeterministic behavior, but no work has analyzed the occurrence of such behavior in these tools. To bridge this gap, we perform an extensive empirical study aiming to understand past and ongoing nondeterminism in 12 popular, open-source static analysis tools that target 5 types of projects. We first conduct a qualitative study to understand the extent to which nondeterministic behavior has been found and addressed within the tools under study, and find results in 7 tool repositories. After classifying the issues and commits by root cause, we find that the majority of nondeterminisms are caused by concurrency issues, incorrect analysis logic, or assumed orderings of unordered data structures, which have shared patterns. We also perform a quantitative analysis, where we use two strategies and diverse input programs and configurations to detect yet-unknown nondeterministic behaviors. We discover such behavior in 8 out of the 12 tools, including 3 which had no results from the qualitative analysis. We find that nondeterminism often appears in multiple configurations on a variety of input programs. We communicated all identified nondeterminism to the developers, and received confirmation of five tools. Finally, we detail a case study of fixing FlowDroid's nondeterministic behavior.**

***Index Terms*—nondeterminism, staic analysis, software testing.**

I. Introduction

Static analyses are powerful, complex algorithms with many important applications including bug finding and software optimization. It is critical that these analyses are implemented correctly and produce reliable results, in order for them to live up to their potential utility as important development aides. In recent years, there has been a push in the research literature to better ensure the quality and reliability of static analysis tools. Various approaches have been proposed to help tool developers and users test and debug static analysis [1], [2], [3], [4]. However, one significant issue that these works do not touch on is that of *nondeterminism*. Nondeterministic behavior has been observed in many different types of software, often in the context of flaky tests [5], [6], [7]. Past research has found common causes of nondeterminism include concurrency issues, caching, and test order dependency [7], [5].

Static analysis tools exhibit many of these risk factors. While the analysis algorithms themselves are deterministic, there are implementation details, such as thread synchronization, usage of ordered/non-ordered data structures, and memorization that have previously been associated with nondeterministic behavior. Despite these risk factors, there has not been a thorough exploration of the extent to which static analyzers exhibit nondeterministic behavior. We hypothesize that there is yet-undiscovered nondeterminism in static analysis tools that may seriously impede their reliability.

To test this hypothesis, we performed the first systematic empirical study to detect, analyze, and understand nondeterministic behavior in static analysis tools. We selected 12 popular static analysis tools as subjects of this study: WALA [8], Soot [9], DOOP [10], and OPAL [11] for Java; TAJS [12] and WALA for JavaScript; PyCG [13] and Code2Flow [14] for Python; FlowDroid [15], Amandroid [16], and DroidSafe [17] for Android; and Infer [18] for C. These tools represent some of the most popular static analysis tools for these 5 types of projects. Note that we consider WALA-Java and WALA-JS as two distinct tools because there exist different analysis logic and options in WALA's codebase for analyzing Java and JavaScript programs. We designed two separate evaluations to understand the occurrence of nondeterminism in these tools: one qualitative study, and one quantitative evaluation.

Our qualitative study aims to understand the extent to which tool users and maintainers are aware of nondeterminism. We aim to extract common characteristics of this behavior, as well as common fixes. We analyzed the repositories of each tool in order to find issues and commits that identify nondeterministic behavior. We extracted 265 issues and 241 commits in total, by searching for keywords relevant to nondeterminism. Then, we manually analyzed these issues and commits to identify distinct results that report or fix nondeterminism. We categorized these results based on the root cause of nondeterminism into 6 categories: *concurrency*, *ordering of data structure*, *analysis logic*, *system-dependent or random value*, *early termination*, and *dependency*. We also summarized 9 distinct patterns of

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00125

nondeterminism and their solutions that may help future static analysis developers to avoid common pitfalls that lead to nondeterminism.

Our quantitative study aims to discover previously unknown nondeterminism in the 12 tools we evaluated. We ran each tool multiple times across diverse input programs and configurations to detect nondeterministic behavior. For each target platform (Java, JavaScript, Python, Android, and C), we selected 2 input program datasets, including microbenchmarks and real-world programs. We sampled configurations of these tools and ran them on the input programs in order to discover nondeterminism that may only occur under certain combination of options. We additionally applied a second strategy, adapting an existing tool [19] that modifies standard libraries in order to detect more nondeterminism, to four compatible tools. Overall, we detected nondeterminism in 8 out of 12 tools, specifically 75 total analyzer configurations on 150 input programs. These results indicate widespread, previously unreported nondeterminism in the current versions of popular static analysis tools. We analyzed the characteristics of the nondeterministic behavior and communicated all identified nondeterminism to the developers of each tool. The developers of FlowDroid, Soot, DOOP, OPAL, and Infer confirmed the reported nondeterminism and provided useful insights and/or bug fixes. We additional conducted and report a case study of FlowDroid's bug fixes to demonstrate the prevalence and complexity of nondeterminism in this tool.

This work makes the following contributions:

- The first systematic empirical study of the nondeterministic behavior in static analysis tools, revealing widespread, previously unknown nondeterministic issues in these tools.
- A qualitative analysis of the tool repositories that shows common patterns of fixes for nondeterministic issues and categorizes their root causes.
- A quantitative analysis identifying extant nondeterminism in 8 out of 12 studied static analysis tools.
- An in-depth analysis of the detected nondeterministic behavior, useful for future development and research of static analysis tools to increase their reliability.

II. Study Goal and Scope

To demonstrate the importance and the complex nature of studying nondeterminism in static analysis tools, we use a FlowDroid bug detected by our study (and reported/fixed, described in Section IV-D2) as a motivating example. FlowDroid performs taint analysis on Android applications to detect security leaks. It produces a set of flows, each made up of a source method (that produces sensitive data) and a sink (that writes/sends data outside of the application). It is expected that when FlowDroid successfully terminates and does not timeout, the set of flows it produces for a given configuration and input program will always be the same.

Figure 1 shows part of a program on which FlowDroid exhibits nondeterministic behavior. This program, `AnonymousClass1`, is from DroidBench [20], [21], a popular microbenchmark used to evaluate Android taint analysis tools. In this program, the `locationListener` (line 5)

```
1  public class AnnonymousClass1 extends Activity { ...
2    private static double latitude;
3    private static double longitude;
4    private LocationManager locationManager;
5    LocationListener locationListener = new
         LocationListener() { ...
6      @Override
7      public void onLocationChanged(Location location) {
8        longitude = location.getLongitude();
9        latitude = location.getLatitude();
10     } ...
11   @Override
12   protected void onResume() { ...
13     Log.i("LOG","Latitude: "+latitude+"Longitude: "+
           longitude);
14   }
15 }
```

Fig. 1: `AnonymousClass1` from DroidBench.

listens for location updates. When the location changes, the method `onLocationChanged` is called and the variables `longitude` and `latitude` are updated on lines 8 and 9. These lines contain tainted data as they are initialized by `getLongitude` and `getLatitude`, which are sources defined in FlowDroid. Finally, the method `onResume` is called and sends the sensitive location data to a log (line 13), which is a sink. This program contains two leaks (i.e., both sources on lines 8 and 9 to the sink on line 13).

When we ran FlowDroid on this program 5 times using the same configuration (that runs a flow-insensitive analysis), it produced three different results, sometimes reporting either leak and sometimes reporting both. Such nondeterministic behavior can lead to users missing real leaks in their applications.

Nondeterministic issues are often complex to detect and to debug; one reason is because of the probabilistic nature of nondeterminism. This is further exacerbated by the complexity of static analysis tools. FlowDroid has 18 configuration options that can each individually affect the expected result of the analysis. In addition, this particular bug was only observed running `AnonymousClass1` and not any of the other 192 DroidBench programs. Furthermore, debugging nondeterminism in static analysis can be a daunting task. We reported this detected nondeterminism bug to the developers of FlowDroid [22]. Ultimately, this bug was fixed in a commit by changing hundreds of lines of code (detailed in Section IV-D2).

A. Research Questions

The goal of this study is to fill the knowledge gap that exists in understanding the existence and significance of nondeterministic behavior in static analysis tools, which is critical for tool development and maintenance. We answer two important research questions in pursuit of this goal:

- **RQ1:** Are there existing nondeterminism issues in static analysis repositories, and how are they usually addressed?
- **RQ2:** Are there unknown nondeterministic behaviors in static analysis tools, what are the characteristics, and how do tool developers respond to the nondeterminism?

RQ1 aims to understand how nondeterminism plays into the development history of static analysis tools. It is answered through a qualitative study that analyzes tool repositories

TABLE I: Information of subject tools.

Tool	# of Options	LoC	Target	Analysis Result
Soot	29	517,679	Java	Call Graph
WALA-Java	8	217,584	Java	Call Graph
DOOP	16	32,597	Java	Call Graph
OPAL	1	402,452	Java	Call Graph
FlowDroid	18	106,668	Android	Alarm
Amandroid	5	15,077	Android	Alarm
DroidSafe	18	55,014	Android	Alarm
TAJS	25	88,308	JavaScript	Call Graph
WALA-JS	5	217,584	JavaScript	Call Graph
PyCG	1	6,119	Python	Call Graph
Code2Flow	2	14,573	Python	Call Graph
Infer	6	546,222	C	Alarm

looking for issues and commits that address nondeterministic behavior. We extract such commits and issues and categorize them to understand how tool developers and users find and fix nondeterminism. RQ2 looks for yet-undiscovered nondeterminism in the current versions of popular static analysis tools. We answer it through a quantitative empirical evaluation, which runs tools multiple times across various input programs and configurations to detect nondeterminism.

B. Subject Tools

We aim to study a wide variety of static analysis tools, covering multiple target languages and analysis types. This helps lend generality to our results. Our methodology is to search for tools that are popular, frequently used in existing evaluations, and open source, so that we could investigate the code of each tool, as well as inspect its issues and commit history. We ultimately select frameworks that perform call graph construction for Java, JavaScript and Python, because these analyses provide fundamental information for developing many inter-procedural dataflow analyses. We target taint analysis tools for Android, because most Android analysis research has focused on the security issues in Android apps, and the most popular analyses are those that perform taint analysis to find software flaws. For C, we use one of the most widely adopted bug detection tools. In the end, we target 12 analysis tools.

Table I shows the details of the selected tools. Column 2 shows the number of configuration options in each tool; tool options usually allow the user to tune the analysis algorithms (e.g., context sensitivity and reflection handling). These configurations present tradeoffs between performance, soundness, and precision. The large numbers of options existing in most tools demonstrate the complexity of the selected tools. Note that in Table I we list WALA-Java and WALA-JS as two tools while they share the same repository (thus, the same count of lines of code in column 3). This is because there exists different analysis logic and options in WALA's codebase for analyzing Java and JavaScript programs; we consider the difference large enough to call them separate tools. Among these tools, Infer is implemented in OCaml, OPAL and Amandroid are implemented in Scala, while the rest are implemented in Java.

III. RQ1: Existing Reports and Fixes of Nondeterministic Behavior

The goal of our qualitative study is to understand the extent to which tool users and developers are aware of nondeterminism in static analysis tools. Furthermore, we aim to learn how tool developers address nondeterminism when it arises. Towards these goals, we performed a manual study of the tool repositories we selected. All the repositories were hosted on GitHub, except for DOOP which was on BitBucket [10]. We first extracted all the issues and commits from each repository, resulting in 6135 issues and 55920 commits in total. Repositories had between 27 (TAJS) and 15575 (OPAL) commits, and between 11 (DroidSafe) and 1963 (Soot) issues.

Next, we aimed to find those commits and issues that affect nondeterministic behavior. We performed a keyword search through the commit messages and the issue texts, using six keywords: *deterministic*, *determinism*, *flaky*, *flakiness*, *concurrent*, and *concurrency*. We originally considered the following additional keywords, but removed them because they resulted in a large number of false positives: *consistent*, *different*, *parallel*, and *thread*. This keyword search returned 265 issues and 241 commits. The number of results containing *concurrent* or *concurrency* was significantly higher than others. After a manual investigation, we noticed that many of these issues and commits only contain these keywords as part of a code package name (e.g., `java.util.concurrent`). We automatically excluded any issue or commit that included *concurrency* or *concurrent* only within a code snippet; this allowed us to remove 143 issues and 8 commits.

This methodology resulted in 122 issues and 233 commits, representing 2% and 0.4% of all the retrieved issues and commits, respectively. For each result, one author inspected all of its text (commit messages and/or issue comments), including the code diff for commits. In this process, the author also deduplicated issues and commits that discuss the same problem. If an issue/commit explicitly references another issue/commit that reports the same problem, we consider them duplicates. From this, the author classified each result as a true positive (i.e., is relevant to a nondeterminism bug), a false positive (i.e., is not relevant) or an inconclusive result (i.e., not enough information is provided to conclude if the issue is relevant). When the author was uncertain if a result is a true positive, the decision was made through the discussion with two other authors. This manual investigation resulted in 125 true positives.

For each true positive, one author wrote a summary of the issue/commit that answers the following questions: (1) What is the root cause of the nondeterminism? (2) Is it fixed? And if yes, how? (3) Are there any other interesting characteristics of the bug (e.g., was it high priority)? Two other authors then reviewed each summary, along with the associated issue/commit. Any disagreements in the summary were resolved via discussion among the three authors. After reviewing each issue/commit, the three authors categorized the nondeterminism bugs based on the nature of its root cause, into following categories:

- Analysis logic (AL): Problematic analysis algorithms.

TABLE II: Categorization of qualitative study results.

Repository	AL	CO	DE	ET	ODS	SRV	Total
Soot	3	19			6	1	29
WALA				1	4	2	7
DOOP	2						2
OPAL	6	2					8
FlowDroid	1	1			1		3
DroidSafe		1					1
Infer	16	15	1	1	1	1	35
Total	28	38	1	2	12	4	85

- Concurrency (CO): Incorrect synchronization between concurrent processes.
- Dependency (DE): Issue in an external dependency.
- Early termination (ET): Terminating an analysis early.
- Ordering of data structure (ODS): Traversal of a data structure with unpredictable iteration order.
- System-dependent or random value (SRV): Using unstable values for variable names.

A. Results

Table II shows the results categorized by their root causes, organized by repository. We were unable to categorize the root cause of 38 results (15 in Soot, 11 in Infer, 7 in OPAL, 3 in FlowDroid, 1 in WALA, and 1 in DroidSafe) because the developers could not locate the cause or insufficient information was provided in the issue/commit. In addition, 2 results were excluded because the nondeterminism was only observed in the build process and not in the actual analysis implementation. Therefore, the total number of results displayed in Table II is 85, with concurrency (38), analysis logic (28), and ordering of data structure (12) being the most frequently observed causes. Seven repositories have nondeterminism-related issues whose root causes have been identified and fixed. Infer (35) and Soot (29) having the most results.

We now describe these results in detail. Due to the page limit, we focus on those that share common patterns, and on patterns that were exhibited across multiple repositories. For each pattern, we also included the corresponding fix pattern and specific cases that tie to particular stages unique to static analysis. The full list of patterns and results are available in our artifact [23].

a) Concurrency: 38 results from Soot (19), Infer (15), OPAL (2), FlowDroid (1), and DroidSafe (1) are associated with concurrency. We have identified 29 results with three distinct common patterns in this category. Patterns CO-1 and CO-2 are described in Table III. Soot-9 (see Figure 2a) is an example of Pattern CO-1; it changes code that iterates over a collection to use a snapshot iterator (i.e., an iterator over a copy of the list) instead. Soot-15 (see Figure 2b) is an example of Pattern CO-2; it is fixed by replacing `HashMaps` with `ConcurrentHashMaps` and `ArrayLists` with `CopyOnWriteArrayLists`. An additional pattern (CO-3 [24]) only occurs in Infer, and deals with how Infer relies on an intermediate file to store procedure properties for analyses, which can sometimes become inaccessible due to multi-threaded execution, leading analyzers to resort to alternative sources and thereby introducing nondeterminism into the analysis results. There are 8 results with a root cause of concurrency that do not fit into any pattern.

TABLE III: Distinct common patterns of nondeterminism.

Pattern ID: **CO-1**
Description: If one thread modifies a mutable collection while another thread is iterating over it, nondeterministic behavior can occur.
Solution: Avoid iterating directly over collections; use a snapshot iterator instead.
Occurrence in each tool: Soot (10), DroidSafe (1).
Related results: Soot-9, 10, 11, 14, 20, 21, 24, 26, 27, 28, DroidSafe-1.

Pattern ID: **CO-2**
Description: Using data structures that are not thread-safe can result in exceptions due to concurrent access.
Solution: Avoid using thread-unsafe data structures; instead, use their thread-safe counterparts or other data structures that guarantee thread safety.
Occurrence in each tool: Soot (5), FlowDroid (1).
Related results: Soot-1, 15, 22, 23, 29, FlowDroid-3.

Pattern ID: **AL-1**
Description: Intermediate and final results can depend on the order of class and file processing. Nondeterministic behavior can occur if this order changes.
Solution: Induce a consistent order on classes and files (e.g., always iterate through user-provided classes first when searching for a main method) (Soot-19).
Occurrence in each tool: Infer (2), Soot (1).
Related results: Infer-13, 29, Soot-19.

Pattern ID: **ODS-1**
Description: Nondeterminstic behavior occurs when an assumption is made about the iteration order of a data structure (e.g., that it will be the same as the insertion order) that does not hold.
Solution: Avoid using unordered data structures; instead, use their ordered counterparts or other data structures that guarantee iteration order.
Occurrence in each tool: Soot (6), WALA (4), FlowDroid (1), Infer (1).
Related results: Soot-2, 7, 12, 16, 17, 18, FlowDroid-2, WALA-3, 4, 5, 7, Infer-31.

Pattern ID: **SRV-1**
Description: Nondeterministic behavior occurs due to the usage of system-dependent or random values when naming or identifying intermediate files or variables (e.g., `System.identityHashCode`). This does not necessarily make the analysis result incorrect, but it makes comparing the results of multiple runs difficult.
Solution: Avoid using system-dependent or random values in this use case, instead defining deterministic naming or identification schemes.
Occurrence in each tool: WALA (2), Soot (1), Infer (1).
Related results: WALA-2, 6, Soot-6, Infer-11.

Pattern ID: **ET-1**
Description: Nondeterministic behavior arises from using timeouts for internal analysis, leading to flaky and incomplete results.
Solution: Avoid using timeouts for analysis, instead utilizing iteration limits to stop an analysis early.
Occurrence in each tool: WALA (1), Infer (1).
Related results: WALA-1, Infer-3.

b) Analysis Logic: 28 results from Infer (16), OPAL (6), Soot (3), DOOP (2), and FlowDroid (1) are associated with analysis logic. We have identified 13 results with three distinct common patterns in this category. Pattern AL-1 is described in Table III. Soot-19 is an example of Pattern AL-1; it is resolved by iterating through user-provided classes first, rather than searching through all the classes on the classpath for a `main` method to use as an entry point to start an analysis. One pattern (AL-2 [24]) occurs only in Infer, and deals with the analysis' handling of mutually recursive functions. Another pattern (AL-3 [24]) occurs only in OPAL, and deals with collaborative analysis dependencies. There are 15 results in this category that do not have shared root causes.

c) Ordering of Data Structure: 12 results from Soot (6), WALA (4), FlowDroid (1) and Infer (1) are associated with this category, sharing one distinct common pattern (ODS-1). Pattern ODS-1 is described in Table III. FlowDroid-2 (see Figure 2c) is an example of Pattern ODS-1; it is fixed by replacing the old data structure, `Set`, which does not guarantee iteration order, with an `ArrayList`. This fix ensures the correct ordering of classes during class loading.

d) System-Dependent or Random Values: We identified 4 results from WALA (2), Soot (1), and Infer (1) in this category, sharing one distinct common pattern (SRV-1). Pattern SRV-1 is described in Table III. Soot-6 (see Figure 2d) is an example of Pattern SRV-1; it addresses a problem of using `System.identityHashCode` in the variable naming algorithm by replacing it with a method that generates deterministic variable names.

e) Early Termination: 2 results from WALA (1) and Infer (1) are in this category, sharing one common pattern (ET-1). Pattern ET-1 is described in Table III. WALA-1 is an example of Pattern ET-1; the pull request introduces a limit to the number of fixed point iterations the algorithm is allowed to perform. It also discusses the alternative of using a timeout, noting that "this approach is less flaky than just setting a timeout."

f) Dependency: We identified 1 result from Infer (Infer-18) in this category. It describes the nondeterminism introduced by problematic dependencies. The issue is found in a test case after upgrading the OCaml compiler to version 4.04.0. The test case is then disabled to avoid flakiness.

Summary: We identify 85 results that indicate nondeterministic behavior across 7 repositories of the subject tools. We categorize our results into 6 root causes, with concurrency, analysis logic, and ordering of data structures being the most frequent. We also identify 9 distinct common patterns of nondeterminism and their respective solutions across these six categories, which can serve as anti-patterns to help static analysis developers avoid common pitfalls that lead to nondeterminism.

```
// the following is a snapshot iterator; this is
    necessary because it can happen that phantom
    methods are added during resolution.
- Iterator<SootMethod> methodIt =
    cl.getMethods().iterator();
+ Iterator<SootMethod> methodIt = new
    ArrayList<SootMethod>(cl.getMethods().iterator());
```

(a) Soot-9

```
+ import java.util.concurrent.ConcurrentHashMap;
+ import java.util.concurrent.CopyOnWriteArrayList;
- public static Map method2Locals2REALTypes =
    new HashMap();
    ...
+ public static Map method2Locals2REALTypes = new
    ConcurrentHashMap();
    ...
- static List<Transformer> jbcotransforms = new
    ArrayList<Transformer>();
+ static List<Transformer> jbcotransforms = new
    CopyOnWriteArrayList<>();
```

(b) Soot-15

```
- Set<SootClass> res = new HashSet<>();
+ // This has to be a list such that an iterator
+ // walks the hierarchy upwards. Otherwise,
+ // iterating on it might return an interface
+ // with more callees than the nearest
+ // superclass leading to more definitions.
+ ArrayList<SootClass> res = new ArrayList<>();
```

(c) FlowDroid-2

```
Jimple jimple = Jimple.v();
- Local vnew = jimple.newLocal("tmp", useType);
- vnew.setName("tmp$" + System.identityHashCode(vnew));
+ Local vnew = localGenerator.generateLocal(useType);
```

(d) Soot-6

Fig. 2: Illustrative examples of common patterns.

IV. RQ2: Detection and Analysis of Nondeterministic Behavior

Now we answer RQ2, which aims to detect and analyze previously unknown nondeterministic behavior in static analysis tools. We answer this with three sub-RQs.

- **RQ2.1:** Is nondeterministic behavior widely observed in analysis tools across different programming languages?
- **RQ2.2:** What are the result consistency and the distinct result counts for the reported nondeterministic behavior in analysis tools?
- **RQ2.3:** How do analysis tool developers respond to the reported nondeterministic behavior?

A. Experimental Setup

a) Target Programs: Finding representative and diverse input programs is important to understand the behavior of static analysis tools. Existing evaluations of static analysis tools have been mainly conducted on two kinds of program datasets: microbenchmarks, which typically consist of hand-crafted (or automatically generated) programs that cover a set

TABLE IV: Selected input programs. M denotes microbenchmarks; R denotes real-world benchmarks.

Dataset	# of Progs	Average LoC	Language	M/R
CATS [2]	112	19	Java	M
DaCapo-2006 [25]	11	85,000	Java	R
DroidBench [26]	193	61	Android	M
Fossdroid [27], [28]	38	38,880	Android	R
Sunspider [29]	27	170	JavaScript	M
jQuery [30]	1	6,759	JavaScript	R
PyCG-Micro [13]	127	7	Python	M
PyCG-Macro [13]	5	2,006	Python	R
ITC [31]	110	374	C	M
Toybox [32]	1	83,535	C	R
SQLite [33]	1	935,041	C	R
OpenSSL [34]	1	693,427	C	R

of pre-defined program or language features; and real-world programs, which have larger codebases for evaluating tool performance. Since these two types of dataset evaluate the tool in different ways, we choose to run each studied tool on both kinds. For each language, we select at least two datasets that have been used in past evaluations, shown in Table IV.

For Java analysis tools, we use CATS [2], a manually created test suite designed to test and compare the soundness of different call graph algorithms [2], [35], [36], and DaCapo-2006, a widely used benchmark of real-world programs for the purpose of performance testing [25]. For Android tools, we use DroidBench as the microbenchmark. It has been widely used to evaluate Android taint analysis tools [37], [38], [26]. As for the real-world programs, we use the FossDroid dataset by Mordahl et al. which consists of 38 open-source real apps collected for the purpose of evaluating Android taint analysis tools' configuration spaces [28]. For Python tools, both datasets are collected from the evaluation of PyCG [39]. PyCG-Micro and PyCG-Macro were created to evaluate call-graph generation algorithms and claim to cover a large fraction of Python's language features. For JavaScript tools, we use the Sunspider performance test suite as the microbenchmark, while we use jQuery, a widely used program for evaluating JavaScript static analyses [40], [30], as the real-world benchmark. Finally, we choose Toyota ITC [31], generated to evaluate different static analyzers [41], as the microbenchmark for the C tool. In addition, we use three well-known C programs, Toybox-0.8.11 [32], SQLite-3.42.0 [33] and OpenSSL-3.3.0 [34], as the real-world programs. These programs have been used in past evaluations of C static bug detectors [42], [43].

b) Timeouts: Because static analysis tools can be both memory and time-intensive, it is important to select an appropriate timeout for each tool and benchmark combination. We decide the timeouts for each tool experimentally. We ran each tool on the corresponding datasets and collected the times to produce each result. For each tool and benchmark, we choose a timeout that allows a majority of runs to finish. For the Java and Android tools on their respective microbenchmarks, WALA-Java, OPAL, Amandroid, and FlowDroid have a timeout of 5 minutes, while Soot, DOOP, and DroidSafe have timeouts of 15, 30, and 60 minutes, respectively. For the real-world datasets, OPAL has a 30 minute timeout, WALA-Java, Amandroid, and FlowDroid have a 1 hour timeout while Soot, DOOP, and DroidSafe have a 2 hour timeout. For the JavaScript and Python tools, on both the microbenchmarks and real-world datasets, the timeout is 5 minutes. The only exception is WALA-JS on jQuery, which has a timeout of 2 hours. For Infer, the timeout is 30 minutes for both types of dataset. Note that all the reported nondeterminism in Section IV-B was detected only among the runs that were completed successfully. We did not use any partial result produced by a timed-out execution to decide nondeterminism.

c) Tool configuration sampling: Different configurations expose different analysis algorithms, each of which may or may not contain nondeterminism. Because it is infeasible to evaluate all combinations of configuration options, we sample the configuration space of each tool to detect nondeterministic behavior in these samples. Note that *configuration option* refers to the tunable algorithmic options provided by the target tools. For example, FlowDroid has 18 configuration options. Some are binary (e.g., `-aliasflowins`), and some are categorical (e.g., `-cgalgo`), resulting in 2,488,320 possible combinations. Specifically, we generate samples from a 2-way covering arrays [44]; this means that every pair of configuration option settings will appear together at least once in the configuration sample. We use ACTS [45], a combinatorial interaction tool, to generate the configuration samples. However, not every configuration can be run successfully, due to unknown constraints in the configuration space. To winnow down the configuration samples only to those which can reliably produce results, we find configurations that terminate successfully on at least 70% of the input programs. We delta debug these configurations to identify failure-inducing options, and then prune these. In the end, the number of sampled configurations ranges from 3 (for PyCG) to 122 (for WALA-Java). All sampled configurations for each tool are available in our artifact [46].

d) Detecting nondeterministic behavior: Following the above setup, we created a framework that detects nondeterministic behavior in static analysis tools, using two strategies.

Strategy I simply runs each combination of tool configuration and input program multiple times. To select the number of times to run each combination, we performed a preliminary experiment using 50 repetitions. Our results show that while more repetitions may allow detecting more nondeterministic behavior, most of such behavior can be detected within a smaller number of repetitions. Considering that analysis run times are generally slow, especially on large programs, for tools that run quickly (TAJS, PyCG, Code2Flow, and WALA-JS), we use 10 repetitions for each combination; for the other tools, we use 5 repetitions. Soot, WALA-Java, DOOP, OPAL, WALA-JS, TAJS, PyCG, and Code2Flow produce static call graphs as outputs. We treat these call graphs as a set of edges, each edge representing a caller-callee relationship. FlowDroid, DroidSafe, Amandroid, and Infer outputs are in the form of sets of alarms (information leaks by the Android tools and bug reports of different types by Infer). We say that nondeterminism behavior is detected if the sets of results (call graph edges or

alarms) are different between any repetitions running the same configuration of a tool on the same program.

Strategy II adapts an existing flaky test detection tool, NonDex [19], to detect additional nondeterminism due to *ordering of data structure*, a common cause of known nondeterministic behavior reported in Section III. Shi et al. [47] studied how nondeterministic methods within the Java Standard Library contribute to the occurrence of flaky tests, and introduced NonDex to identify such flaky tests. For example, assuming a specific iteration order of a `HashSet` can lead to issues when the iteration order changes across JDK versions. NonDex can randomly explore different iteration orders to proactively detect tests failures that iterate over `HashSet` [19]. We integrated NonDex into our framework to explore different iteration orders of collections and applied it to all compatible analysis tools (Soot, FlowDroid, Amandroid and TAJS). We are unable to run NonDex on WALA-Java, OPAL and WALA-JS due to JDK version incompatibility and on DOOP and DroidSafe due to the substantial time and memory required. To detect additional nondeterminism, we ran these four tools with NonDex under the same settings as employed in Strategy I. We say that nondeterministic behavior is detected if the set of results produced by any Strategy II repetition is different from those produced by Strategy I, running the same configuration of a tool on the same program.

e) Hardware environment: All experiments were conducted on two servers to accelerate the process and run more experiments simultaneously. The experiments of DOOP, Soot, WALA-Java, OPAL, DroidSafe and Infer were conducted on a server with 384GB of RAM and 2 Intel Xeon Gold 5218 16-core CPUs@2.30GHz running Ubuntu 20.04. The experiments of FlowDroid, Amandroid, PyCG, Code2Flow, TAJS and WALA-JS were conducted on a server with 144GB of RAM and 2 Intel Xeon Silver 4116 12-core CPUs@2.10GHz running Ubuntu 18.04. To ensure a consistent environment, we execute the studied tools in Docker containers [48].

B. RQ2.1

Overall, *nondeterministic behavior is detected in 8 out of the 12 tools we studied, suggesting that there are widely-spread, previously unknown nondeterminism bugs in these important tools.* Among the 8 tools, 3 tools (Amandroid, Code2Flow and PyCG) did not contain issues related to nondeterminism in their repositories (Section III). Interestingly, despite the fact that there have been past nondeterminism bugs reported in WALA and DroidSafe, we did not detect new nondeterminism in either WALA variation or DroidSafe.

Tables V and VI show details of the detected nondeterminism. Note that not all tools exhibit nondeterminism. Columns **Micro** and **Real-world** in Table V indicate the number of nondeterministic configurations detected in the respective microbenchmarks and real-world benchmarks for each tool. Similarly, these columns in Table VI represent the number of programs where nondeterministic behavior was detected while running each tool. A plus sign indicates additional results detected by Strategy II; for example, Strategy I detected 18 nondeterministic Soot configurations on the microbenchmark, and Strategy II found 3 more (Table V). Column **Pct.** in Table V shows the percentage of nondeterministic configurations out of all sampled configurations of each tool. In total, we detected nondeterminism in 75 configurations. Column **Pct.** in Table VI shows the percentage of nondeterministic programs out of all programs from both benchmarks. Overall, we detected nondeterminism in 150 programs. Note that this number is not the sum of Column **Total** in Table VI due to overlaps in programs detected by multiple tools.

TABLE V: Number of nondeterministic configurations in each tool. The number of ND Configs represents the overlap of nondeterministic configurations detected in both microbenchmarks and real-world benchmarks with two strategies.

Tool	Micro	Real-world	ND/Sample Configs	Pct.
Soot	18 + 3	18 + 1	21 / 30	70%
DOOP	2	7	7 / 9	77.78%
OPAL	0	4	4 / 4	100%
FlowDroid	5 + 5	13 + 4	18 / 19	94.74%
Amandroid	0 + 0	4 + 2	6 / 25	24%
PyCG	2	0	2 / 3	66.67%
Code2Flow	4	0	4 / 4	100%
Infer	0	13	13 / 20	65%

TABLE VI: Number of programs exhibiting nondeterminism detected by each tool.

Tool	Micro	Real-world	Total	Pct.
Soot	112 + 0	7 + 0	119	96.75%
DOOP	105	7	112	91.06%
OPAL	0	1	1	0.81%
FlowDroid	4 + 2	13 + 2	21	9.09%
Amandroid	0 + 0	2 + 3	5	2.16%
PyCG	3	0	3	2.27%
Code2Flow	2	0	2	1.52%
Infer	0	2	2	1.77%

First, as shown in the last column of Table V, all the sampled configurations for OPAL and Code2Flow exhibited nondeterminism, and the majority of Soot, DOOP, FlowDroid, PyCG and Infer sampled configurations have nondeterminism. This result indicates that these tools may contain nondeterminism bugs in common logic and/or there are bugs widely spread in their implementations of configuration options. Indeed, we manually confirmed that the default configurations of Soot, OPAL, FlowDroid, PyCG, Code2Flow and Infer are nondeterministic. As for Amandroid, 6 out of 25 (24%) sampled configurations exhibit nondeterminism. All nondeterministic configurations of Amandroid set the `approach` option to `COMPONENT_BASED`. This approach focuses on analyzing Android applications by treating their various components as discrete units that can communicate through different channels, such as Intents and static fields [37]. This indicates that there may be nondeterminism local to the implementation of the component-based approach.

Additionally, we identified nondeterminism using both microbenchmarks and real-world programs for Soot, DOOP, and FlowDroid (Table VI). For OPAL, Amandroid and Infer, only real-world programs reveal the presence of nondeterminism.

Nondeterminism in both Python tools can only be detected running the PyCG-Micro microbenchmark; few programs in this benchmark, which contains 127 programs, enable the detection of nondeterminism. As for Soot and DOOP, most programs in the CATS microbenchmark allow detection of nondeterministic behavior. The above result suggests that microbenchmarks and real-world benchmarks complement each other in detecting nondeterminism.

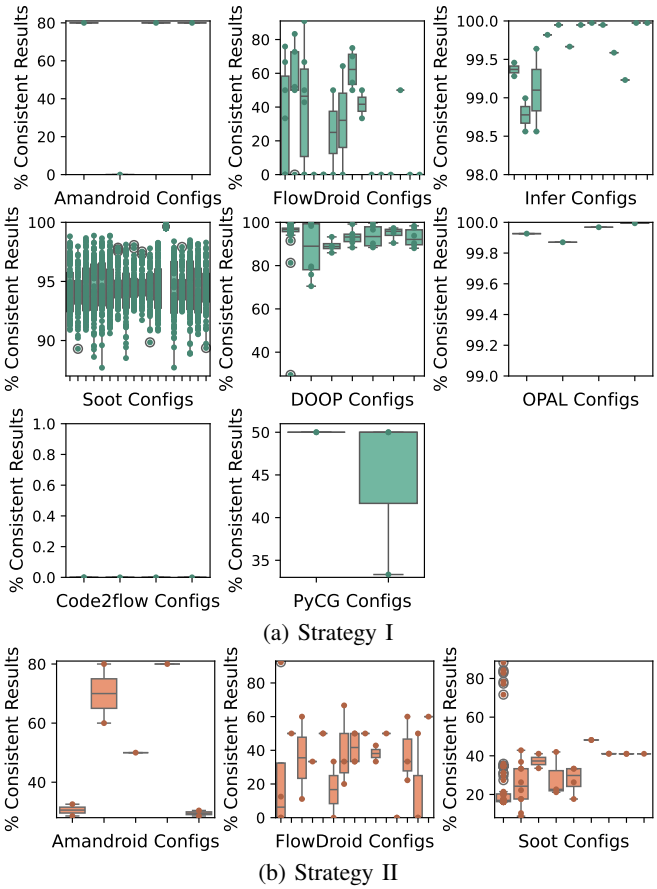
Finally, Strategy II uncovers additional nondeterministic configurations in Soot (3), FlowDroid (5), and Amandroid (2), demonstrating its effectiveness. Recall that Strategy II shuffles the order of elements in certain data structures; thus, the detected nondeterminism likely falls into the *ordering of data structure* category identified in Section III.

Summary: Nondeterministic behavior is detected in 8 out of the 12 tools we studied. Many configurations in these tools exhibit nondeterministic behavior; microbenchmarks and real-world benchmarks complement each other in the detection of nondeterminism, and Strategy II is useful in detecting more nondeterminism. In addition, only 5 tools among the 8 have discussed nondeterministic behavior in their repositories. This result indicates there exists widespread nondeterminism in static analysis tools that has been overlooked.

C. RQ2.2

We first aim to understand how severely the tools' results suffer from nondeterminism. Figure 3 shows box plots that depict how different each tools' results are as a result of nondeterministic behavior under both strategies. The x-axes represent configurations of each tool. Each box represents all of the input programs which demonstrated nondeterministic behavior under a certain configuration; boxes that contain a single point mean that for that configuration, only one input program enabled nondeterministic behavior. The y-axes indicate the percentage of consistent results (i.e., call graph edges or alarms) across the repetitions. A value near 100% indicates that the results were largely the same between repetitions, while a value near 0% means that the results were almost completely different between repetitions.

Analyzing Strategy I results in Figure 3a, Soot, DOOP, OPAL and Infer have relatively higher percentages of consistent results, averaged 95%, 95%, 99% and 99%, respectively. However, for Soot and DOOP, this does not mean few edges in these call graph results are inconsistent. In fact, the call graphs produced by Soot and DOOP often consist of thousands of edges, meaning that there are still hundreds of call graph edges not appearing in every repetition. The two taint analysis tools, Amandroid and FlowDroid, contain 60% and 33% consistent leaks, on average. This means that many of these tools' leaks are produced in a unpredictable manner, which can severely impact a user's confidence in using these tools to identify security vulnerabilities. For FlowDroid, the percentages of consistent leaks vary both by configuration and by program.



(a) Strategy I

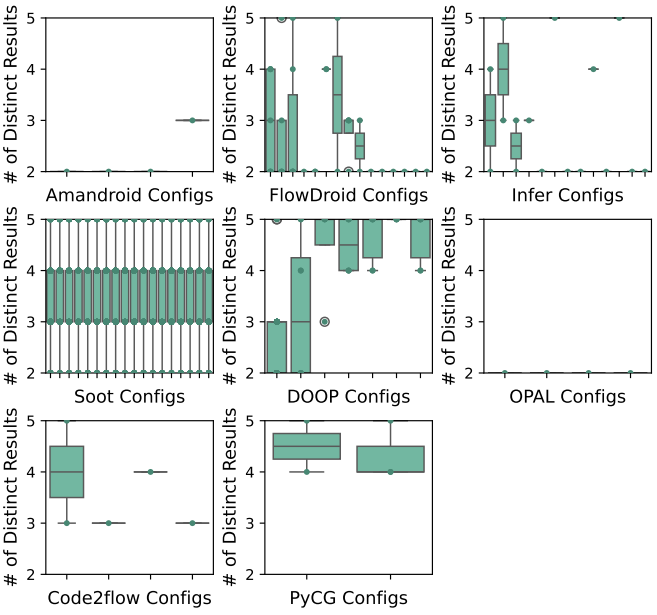
(b) Strategy II

Fig. 3: Percentage of consistent results across configs for each tool under both strategies.

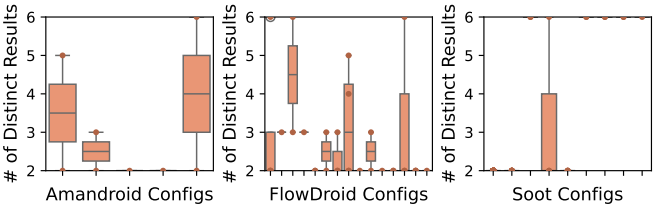
While many results (16) had 0% consistent results (i.e., no leak is consistently found in all repetitions), there are 4 results that have greater than 75% consistent results. Code2Flow has 0% consistent results in all of its detected nondeterminism, which suggests Code2Flow's nondeterminism affects the whole result. Finally, on average, over half (53%) of PyCG's call graph edges are nondeterministically produced.

Strategy II results in Figure 3b are notably lower than those in Strategy I for each tool on which it is applied. On average, the percentages of consistent results are 49%, 34%, 25% for Amandroid, FlowDroid and Soot, respectively. In particular, the results consistency rate experiences a substantial decrease (70%) for Soot, indicating that the ordering of data structures significantly contributes to its nondeterminism.

Figure 4 shows a set of box plots of the number of distinct repetitions, organized by nondeterministic configuration of each tool detected by both strategies. We say a repetition is a distinct repetition if there exists any result (call graph edge or alarm) in its result set that differs from another repetition. Fewer distinct repetitions means that the detected nondeterminism is more *stable* in that the tool does not produce many variations of different results. Analyzing Strategy I results in Figure 4a, on average, the number of distinct repetitions ranges from 2.0 (for OPAL) to 4.4 (for PyCG). Among all studied analyzers, OPAL is the most stable tool, with all of its results having only 2



(a) Strategy I



(b) Strategy II

Fig. 4: Number of distinct repetitions across configs for each tool under both strategies.

distinct repetitions. However, the majority of configurations in Soot and DOOP have 3 or 4 distinct repetitions, indicating the nondeterminism bugs cause more unstable results and are potentially more complex than bugs that only cause 2 distinct repetitions.

As for the Strategy II results in Figure 4b, the detected nondeterminism appears to be more unstable for Android tools compared to Soot. Specifically, only 4 out of 40 results contain two distinct repetitions for Amandroid and FlowDroid, compared to 135 out of 145 results for Soot. Interestingly, none of the results for Soot in Strategy II includes any repetition that matches those from Strategy I; this is consistent with our previous observation that the ordering of data structures is crucial to the nondeterminism in Soot.

Summary: The percentages of consistent results vary by tool, but even for the tools with higher percentages (Soot and DOOP), the impact is significant, affecting hundreds of call graph edges. In addition, tools varied significantly in how many distinct result sets they produced. Adding NonDex to the tools (Strategy II) allows us to detect additional nondeterminism due to *ordering of data structures*. It has a larger impact on Soot, suggesting this is an important source of nondeterminism in this tool.

TABLE VII: Summary of reported nondeterminism.

ID	Tool	Status
1	FlowDroid	Fix provided, nondeterminism persists.
2	FlowDroid	Fix provided, nondeterminism persists.
3	FlowDroid	Fix provided, nondeterminism persists.
4	Soot	Confirmed, no action planned.
5	Soot	No response from developer.
6	DOOP	Confirmed and resolved.
7	OPAL	Confirmed, no action planned.
8	Infer	Workaround provided, fixes planned.
9	Amandroid	No response from developer.
10	PyCG	No response from developer.
11	Code2Flow	No response from developer.

D. RQ2.3

We communicated all identified nondeterminism to the developers of each tool by raising issues on their respective repositories. In total, we created 3 issues for FlowDroid, 2 for Soot, and 1 each for DOOP, OPAL, Infer, Amandroid, PyCG and Code2Flow. Note that we did not report a separate issue for each configuration of the tools that we detected nondeterminism because there may be shared root causes of the nondeterminism. Instead, we conducted manual analysis to determine the common symptoms, and then reported them in groups. We received responses from the developers of FlowDroid, Soot, DOOP, OPAL and Infer. Table VII summarizes the status of each reported issue, all of which are included in the artifact [23]. Next, we provide a detailed discussion of the developers' responses for each tool.

1) Status of reported nondeterminism:

a) FlowDroid: In response to our reported issues, FlowDroid developers provided three fixes. To assess the impact of these fixes, we conducted a case study running FlowDroid on DroidBench with each provided fix (see Section IV-D2).

b) Soot: The first reported issue has the symptom that the same identifiers in the IR have different names in different repetitions, often appended with a different number for the same identifier. One developer responded that identifiers only need to be consistent within the same repetition. While it is true this nondeterminism does not affect the correctness of analysis result in each repetition, it affects the ability to compare results across multiple runs, as needed when debugging the tool. Indeed, our qualitative analysis reports past bug fixes in Soot and WALA addressing similar issues (Soot-6, WALA-6). Another issue we reported is that certain groups of call graph nodes are always missed together in some repetitions nondeterministically. As of the writing of this paper, the investigation remains ongoing.

c) DOOP: The developer of DOOP responded to our reported issue with two possible sources of the nondeterminism in DOOP. First, DOOP relies on Soot as its facts generator; before running the point-to analysis, it invokes Soot to generate either Jimple or Shimple IR to use as the input of the point-to analysis. The developer mentioned that Soot's output is influenced by the order of finding classes, which inherently introduces nondeterminism into DOOP's input facts. They suggest running the first analysis with the `--facts-only`

`--id` option to generate the facts and using the `--input-id` option to load these facts for subsequent analyses. This ensures that all repetitions use the same set of input facts. Second, certain algorithms in DOOP's reflection analysis involve randomization, selecting a representative from an equivalence class of classes and processing the classes based on these representatives. To verify the proposed root causes, we conducted additional experiments on the DOOP configurations and programs that exhibited nondeterminism, following the developer's suggestion to use the same set of input facts for subsequent analyses. We observed no nondeterminism in this experiment, suggesting that (some) detected nondeterminism in DOOP may stem from its fact generator, Soot.

d) OPAL: The OPAL developer investigated the reported issue and identified the root cause of nondeterminism, which stemmed from conflicting dependencies in the target program. This nondeterminism was observed in eclipse.jar from DaCapo-2006 benchmark. The issue arose because two dependencies of eclipse.jar, `ee.foundation.jar` and `ee.minimum.jar`, contained different versions of the `java.lang.Thread` class. OPAL appears to nondeterministically choose which version of the class file to retain, resulting in diverging results in the call graph. While the developer of OPAL considers this as an improper practice of building a project, we believe that static analyzers should implement more deterministic mechanisms to handle conflicting dependencies. Our qualitative analysis reports similar discussions in Soot and Infer, which address nondeterminisms caused by the order in which classes and files are processed (Infer-13, Infer-29, Soot-19).

e) Infer: The Infer developer provided two possible sources of nondeterminism: (1) Infer's `--biabduction` checker and (2) how it processes recursive functions. We conducted additional experiments to run Infer on OpenSSL with the `--biabduction` checker deactivated, and observed no nondeterminism. However, disabling this checker significantly reduces the number of reported warnings, drastically lowering recall. According to the Infer developer, the biabduction analysis in Infer has a hard wall-clock time limit, causing the analysis of a procedure to stop once this limit is reached. Because the performance of each analysis can vary due to external factors like CPU load, using a hard time limit can lead to inconsistent and incomplete results. Note that the Infer developers have been aware of the nondeterministic behavior for a long time (Section III) and are still actively fixing the related issues. Several fixes to nondeterminism have landed on the development branch since the latest release with additional fixes currently in the planning phase, according to Infer developers.

2) Case study on FlowDroid: We opened 3 issues on the FlowDroid GitHub repository to report the observed nondeterministic behavior. In response, developers provided three fixes, and we ran FlowDroid on DroidBench to validate each fix. Figure 5 illustrates the fluctuation of nondeterministic configurations and programs across four versions of FlowDroid. The versions and fixes (anonymized) are represented along a timeline. The circles above the timeline indicate the number of nondeterministic configurations detected, while the circles below indicate the number of affected programs. Red arrows represent configurations or programs that disappear after a version or fix, and green arrows represent configurations or programs that newly appear or reappear.

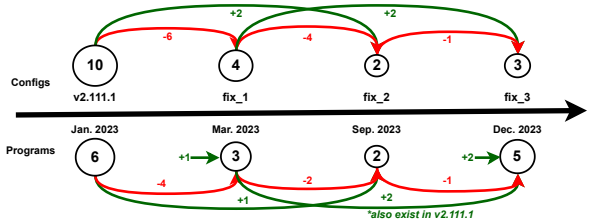


Fig. 5: FlowDroid nondeterminism across four versions.

`fix_1` corresponds to the first reported issue. The developer believed the nondeterminism bug happened during the taint propagation stage, where the redundant taint abstractions were pruned from the abstraction chain to enhance performance. However, the incorrect comparison of taint abstractions led to the erroneous discarding of nodes. This bug was fixed by including the current statement and call site information in the comparison to determine whether a node should be added as a neighbor in the taint abstraction chain.

However, our results on `fix_1` still reveal nondeterminism across 4 configurations and 3 programs. The FlowDroid developer then provided another fix, `fix_2`, which corrects an error in the logic for verifying taint abstractions. This error led to the removal of non-isomorphic abstraction graphs and caused nondeterminism based on the iteration order of the result set. Luo et al. [4] also observed this nondeterminism. FlowDroid developer confirmed that `fix_2` addresses the issue.

After running the experiments with the second fix, we found that nondeterminism still appears in 2 configurations and 2 programs, which are a subset of all previously detected configurations and programs. The FlowDroid developer then provided `fix_3`. `fix_3` addresses nondeterminism caused by multiple neighbors with equal statements, especially when joining after returning from a function. Previously, it was assumed that structurally equal neighbors (facts in the taint graph that reached a fixed point in the IFDS algorithm) were unnecessary. However, this assumption is incorrect in some cases. The fix ensures that all neighbors of a taint abstraction are always kept, eliminating this source of nondeterminism.

Upon running the experiments with the latest fix, we still encountered nondeterminism across 3 configurations in 5 programs. As illustrated in Figure 5, the nondeterministic behavior persists across all three fixes, with each fix potentially causing regressions of previously resolved nondeterministic behaviors. As of the latest fix `fix_3`, the detected nondeterministic configurations are a subset of all previously detected ones. `fix_3` includes a regression that caused 2 configurations that were nondeterministic in v2.111.1 and after `fix_1` but not after `fix_2` to become nondeterministic again.

These results indicate that none of the fixes eliminated all sources of nondeterminism during taint propagation, highlighting the significant challenge of resolving this issue in FlowDroid. Additionally, according to the developers, some

level of nondeterminism is intentional, as certain settings are implemented as cutoffs to prevent runtime explosions.

Summary: FlowDroid, Soot, DOOP, OPAL and Infer developers respond to our bug reports with useful information and/or bug fixes. We perform experiments to confirm some unaddressed, known issues in these tools; moreover, our case study on FlowDroid showcases the challenges in addressing all nondeterminism, particularly because some instances only manifest under specific configurations.

V. Threats to Validity

There are several potential threats to validity. First, both the qualitative and quantitative analyses focus on the 12 subject tools that perform call graph analysis, taint analysis or bug detection targeting 5 types of projects. Our observations may not be generalized to tools that perform different types of analysis or target different programming languages. Nevertheless, our subject tools are among those that are the most widely used, performing important analyses and targeting important languages. Second, our protocol of the qualitative analysis (e.g., searching keywords and categorization of results) may result in incorrectly removing relevant issues or mis-categorization. We mitigate this threat by having three authors with the domain expertise in static analysis agreeing on the categorization. Third, our quantitative analysis does not consider several factors that may allow detection of nondeterminism in tools, including different execution environment and more repetitions. Nevertheless, our results demonstrate its effectiveness in detecting many previously unknown nondeterministic behavior.

VI. Related Work

a) Flaky tests: Flaky tests refer to the tests that exhibit nondeterministic outcomes. Luo et al. performed an extensive empirical study on flaky tests by studying 201 commits that likely fix flaky tests in 51 open-source projects [5]. They classified the most common root cause of flaky tests, identified approaches that could manifest flaky behaviors, and described common strategies that developers use to fix flaky tests. Lam et al. [7] partially classified the nondeterministic test failures into *order-dependent tests* and *non-order-dependent* tests, studying the impact of test reordering on the number of detected flaky tests. In addition, many approaches (e.g., [6], [49], [50], [51]) have been presented to detect flaky tests. Our work is the first to focus specifically on static analysis, revealing how widely the nondeterministic behavior exists across different tools and the impact of this observed nondeterminism.

b) Testing and debugging of static analysis: Various works have focused on testing and debugging static analysis. Mordahl et al. presented ECSTATIC to automatically test and debug configurable static analysis [1], [52]. ECSTATIC leverages the relationships between configuration options [28] to find and delta debug issues in static analysis. While ECSTATIC aims to find issues with an analyzer's correctness, our approach aims to detect nondeterminism. Do et al. introduced VISUFLOW, which provides tools like an IR mapping and a Control Flow Graph (CFG) viewer to help a user debug an incorrect static analysis [53]. Reif et al. introduced Judge, a toolchain for testing static call graph algorithms for unsoundness [2]. Cadar and Donaldson argued for performing program analyses on the analyzers themselves to improve their reliability and outlined the ideas for cross-checking analysis results and using program transformations as a basis for checking analyzers [54]. These approaches all focus on finding sources of incorrectness or imprecision in static analysis, while we are the first to specifically identify nondeterministic behavior.

c) Nondeterminism in program analyzers: There exist other works that discuss nondeterminism in program analyzers. Sui et al. [55] studied the soundness of static analysis tools in the presence of dynamic features, and found that the presence of some call graph edges depends on JVM versions due to the limited support for dynamic features. The nondeterminism we investigate refers to scenarios where certain edges are inconsistently missing, even under identical experimental conditions. Schemmel et al. [56] identified six design principles that memory allocators for dynamic symbolic execution should follow, two of which deal with determinism. They implemented KDALLOC, a deterministic memory allocator, and integrated it into KLEE [57]. Our work explores nondeterminism in a wide variety of static analyzers.

VII. Conclusions and Future Work

We present the first extensive empirical evaluation of the occurrence and cause of nondeterministic behavior in static analysis tools. We studied 12 popular tools, using both qualitative and quantitative analysis. In our qualitative analysis, we classified 85 results by their root causes, and found that most are caused by concurrency bugs, complicated analysis logic or unordered data structures. Our quantitative study detected nondeterministic behavior in 8 out of 12 tools, including 3 which had no issues of nondeterminism reported from the qualitative analysis. We found that nondeterminism occurs commonly, across a variety of configurations and input programs. These results suggest that nondeterminism is endemic in static analysis tools, which is potentially a severe impediment to their adoption and usage.

Our work can help raise awareness of the nondeterministic behaviors among the static analysis community. The dataset, including the identified patterns, and the detection toolchain will provide a foundation for avoiding nondeterminism in important static analyzers and for future research work. Moving forward, we plan to extend our quantitative evaluation to evaluate how system factors (e.g., operating system and system load) affect the occurrence of nondeterminism. We also plan to develop tools to help static analysis developers avoid common pitfalls that can lead to nondeterminism in static analysis.

Acknowledgment

We thank Darko Marinov for guidance in using NonDex [19]. This work was partly supported by NSF grants CCF-2008905 and CCF-2047682.

REFERENCES

[1] A. Mordahl, Z. Zhang, D. Soles, and S. Wei, "ECSTATIC: An extensible framework for testing and debugging configurable static analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 550–562.

[2] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. 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*, 2019, pp. 251–261.

[3] M. Reif, M. Eichberg, B. Hermann, and M. Mezini, "Hermes: assessment and creation of effective test corpora," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2017, pp. 43–48.

[4] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "TaintBench: Automatic real-world malware benchmarking of android taint analyses," *Empirical Software Engineering*, vol. 27, pp. 1–41, 2022.

[5] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.

[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 433–444.

[7] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322.

[8] "Wala," https://github.com/wala/WALA, 2024.

[9] "Soot," https://github.com/soot-oss/soot, 2024.

[10] "Doop," https://bitbucket.org/yanniss/doop/src/master/, 2024.

[11] "Opal," https://github.com/opalj/opal, 2024.

[12] "Tajs," https://github.com/cs-au-dk/TAJS, 2024.

[13] "Pycg," https://github.com/vitsalis/PyCG, 2024.

[14] "Code2flow," https://github.com/scottrogowski/code2flow, 2024.

[15] "Flowdroid," https://github.com/secure-software-engineering/FlowDroid, 2024.

[16] "Amandroid," https://github.com/arguslab/Argus-SAF, 2024.

[17] "Droidsafe," https://github.com/MIT-PAC/droidsafe-src, 2024.

[18] "Infer," https://github.com/facebook/infer, 2024.

[19] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 993–997.

[20] "Droidbench," https://github.com/secure-software-engineering/DroidBench, 2024.

[21] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 331–341.

[22] "Nondeterministic results from dataflowsolver flow insensitive," https://github.com/secure-software-engineering/FlowDroid/issues/583, 2023.

[23] "Artifact for an extensive empirical study of nondeterministic behavior in static analysis tools," https://github.com/UTD-FAST-Lab/NDSAStudy, 2024.

[24] "Nd-patterns-list," https://github.com/UTD-FAST-Lab/NDSAStudy/blob/main/data/rq1/RQ1_ND_PATTERN_LIST.pdf, 2024.

[25] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[27] "Fossdroid," https://fossdroid.com, 2024.

[28] A. Mordahl and S. Wei, "The impact of tool configuration spaces on the evaluation of configurable taint analysis for android," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 466–477.

[29] "Sunspider," https://chromium.googlesource.com/v8/deps/third_party/benchmarks/+/refs/heads/master, 2015.

[30] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 17–31.

[31] "Static analysis benchmarks from Toyota itc," https://github.com/regehr/itc-benchmarks, 2024.

[32] "Toybox," https://github.com/toy-box/toybox, 2024.

[33] "Sqlite," https://github.com/sqlite/sqlite, 2024.

[34] "Openssl," https://github.com/openssl/openssl, 2024.

[35] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[36] D. Lehmann, M. Thalakottur, F. Tip, and M. Pradel, "That's a tough call: Studying the challenges of call graph construction for WebAssembly," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 892–903.

[37] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[38] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in DroidSafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[39] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in Python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1646–1657.

[40] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy, "Static JavaScript call graphs: A comparative study," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 177–186.

[41] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2015, pp. 12–15.

[42] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, "An empirical study of real-world variability bugs detected by variability-oblivious tools," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 50–61.

[43] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–33, 2018.

[44] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[45] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 370–375.

[46] "Sampled configurations for each tool," https://github.com/UTD-FAST-Lab/NDSAStudy/tree/main/code/NDDetector/src/resources/configurations, 2024.

[47] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 80–90.

[48] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[49] A. Groce and J. Holmes, "Practical automatic lightweight nondeterminism and flaky test detection and debugging for Python," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 188–195.

[50] C. Ziftci and D. Cavalcanti, "De-flake your tests: Automatically locating root causes of flaky tests in code at google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 736–745.

[51] R. Mudduluru, J. Waataja, S. Millstein, and M. Ernst, "Verifying determinism in sequential programs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 37–49.

[52] A. Mordahl, D. Soles, M. Miao, Z. Zhang, and S. Wei, "ECSTATIC: Automatic configuration-aware testing and debugging of static analysis tools," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1479–1482.

[53] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 697–709, 2018.

[54] C. Cadar and A. F. Donaldson, "Analysing the program analyser," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 765–768.

[55] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. 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*. Springer, 2018, pp. 69–88.

[56] D. Schemmel, J. Büning, F. Busse, M. Nowack, and C. Cadar, "A deterministic memory allocator for dynamic symbolic execution," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[57] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.