Call Graph Soundness in Android Static Analysis

Jordan Samhi
CISPA Helmholtz Center for Information Security
Saarbrucken, Germany
jordan.samhi@cispa.de

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

Michael D. Ernst
University of Washington
Seattle, USA
mernst@cs.washington.edu

Jacques Klein
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Abstract

Static analysis is sound in theory, but an implementation may unsoundly fail to analyze all of a program's code. Any such omission is a serious threat to the validity of the tool's output. Our work is the first to measure the prevalence of these omissions. Previously, researchers and analysts did not know what is missed by static analysis, what sort of code is missed, or the reasons behind these omissions. To address this gap, we ran 13 static analysis tools and a dynamic analysis on 1000 Android apps. Any method in the dynamic analysis but not in a static analysis is an unsoundness.

Our findings include the following. ① Apps built around external frameworks challenge static analyzers. On average, the 13 static analysis tools failed to capture 61% of the dynamically-executed methods. ② A high level of precision in call graph construction is a synonym for a high level of unsoundness. ③ No existing approach significantly improves static analysis soundness. This includes those specifically tailored for a given mechanism, such as DroidRA to address reflection. It also includes systematic approaches, such as EdgeMiner, capturing all callbacks in the Android framework systematically. ④Modeling entry point methods challenges call graph construction which jeopardizes soundness.

CCS Concepts

• **Software and its engineering → Automated static analysis**; **Dynamic analysis**.

Keywords

Android, Static Analysis, Dynamic Analysis, Call Graph Soundness

ACM Reference Format:
Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. 2024. Call Graph Soundness in Android Static Analysis. 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/10.1145/3650212.3680333

1 Introduction

A static analysis can be sound and conservative by using a model that over-approximates the code's behavior. In contrast, dynamic analysis is precise, capturing actual run-time behavior, but it is unsound as it ignores other executions and uncovered code. Hence, clients can choose soundness (static analysis) or precision (dynamic analysis) [18]. However, this is to some extent a false choice: soundness is often unattainable in practice due to unsound static models. This paper investigates this issue in the context of Android.

One reason for unsoundness when analyzing Android apps is that apps are event- and callback-driven, resulting in a complex execution flow that static analysis struggles to capture [29, 31]. For instance, static analysis fails to fully and automatically account for the *implicit* invocation of methods by frameworks, such as the Android framework, Flutter [23], Xamarin [59], etc. The calls to some application methods are in a framework, which (for scaling reasons) is often not statically analyzed along with the app. Missing these implicit calls leads to an incomplete understanding of the app's behavior (e.g., missing nodes and edges in the call graph) [7, 35]. This blind spot can be exploited by attackers to circumvent state-of-the-art static analysis tools, such as FlowDroid [6], and hide malicious code [25]. Hence, developing the most effective static data leak detector or static malware detector is little help if the analysis is run over an unsound static model of the app.

Numerous approaches have attempted to refine call graphs by accounting for specific mechanisms [6, 8, 19, 24, 34, 35, 44, 46], such as reflection. In addition, there have been attempts to systematically analyze the Android framework to collect callbacks [11, 12]. None of these approaches is comprehensive. Although some studies show that several mechanisms, such as implicit calls, cause unsound static analysis [7, 12], the extent of under-approximation of static models is not known. As a result, there is a need to systematically explore the amount of methods missed during static analysis and study their underlying causes.

Our work consists of three parts. First, we identified static analysis tools via a systematic literature search (section 3.2). Second, we obtained static and dynamic call graphs for 1000 recent apps. To do this, we slightly modified each static analysis tool to extract the call graph it builds. Then, we ran each static analysis tool on 1000 recent apps (section 3.3). We also ran a dynamic analysis to determine which methods the apps call at run time (section 3.4). Third, we examined the differences between the statically- and dynamically-generated call graphs — in particular, the methods invoked at run time that do not appear in static models (section 4).

Our study seeks to provide directions to the research community on how can static analysis of Android apps be improved, particularly with recent apps utilizing external frameworks.

Our results indicate that call graph construction is slow, even when using high-performance computing hardware. On average, only 58% of apps can be statically modeled (only building the call graph, not applying the analysis that uses the call graph) within 1 hour by the static analyzers (the average size of apps in our dataset is 24MB and the call graphs computed in our study have an average of 16 633 nodes and 229 901 edges). Tools putatively using the same call graph construction algorithms have *very* different call graphs and run times. *Every* static tool suffers significant unsoundness by missing a substantial number of methods in their static model that are called at run time. Furthermore, more precise call graph construction implementations suffer more unsoundness.

Although computing precise call graphs is important, it is problematic when it compromises soundness, especially with recent malware using, e.g., implicit call mechanisms. The Scylla Android malware illustrates how sophisticated threats operate [25]. It uses the Android framework's JobScheduler to activate payloads only under specific conditions. It evades detection by dynamic analyzers when conditions aren't met. It evades detection by static analyzers that overlook JobScheduler's implicit mechanisms. Consequently, such malware can infiltrate platforms like Google Play. We have investigated the root causes for this unsoundness. In general, static analyzers suffer from lack of understanding of the Android framework and external frameworks, leading to missing many implicit calls that, in turn, trigger many methods left unanalyzed statically.

To better illustrate this problem, we ran the tools in our study against the Scylla malware. Only 4 of the 13 tools could statically reach the trigger point of the malicious code using the call graph. This suggests that most current tools are insufficiently sound to handle real-world threats. This poses a significant risk for end-users, who may unknowingly use malware that enters app markets such as Google Play because the malware is not detected by these tools. Nearly every week, new malware is detected in official app markets.

The main contributions of this paper are as follows:

- We conducted a large-scale empirical study by comparing call graphs yielded by 13 static analyzers over 1000 Android apps against call graphs yielded by dynamic analysis.
- While the problem of unsoundness is known in the literature, we are the first to quantify the extent of the problem in Android app static analysis. In particular, we show that, in the best case, at least 40% of the methods called at runtime are overlooked by static analyzers.
- We show that static analyzers are unsound because they are not properly modeling entry point methods, i.e., root nodes in the dynamically generated call graph.

Findings' Implications: Previous papers proposing new static analysis techniques (e.g., addressing inter-component communication, reflection, or dynamic loading) often have a formulaic mention of soundness in the threats to validity section, but do not acknowledge the *massive* scale of the problem or the fact that improving soundness is more relevant in practice than improving precision. Our research findings suggest that: ① researchers should pause their work on new analyses and on call graph precision until they have solved problems of call graph soundness; and ② library methods are the most often missed by static analysis, so handling libraries is probably the most important avenue for future work.

As a result, our study sets the stage for more sound static analysis of Android apps, which can lead to safer applications and, ultimately, protect end-users more effectively.

Artifacts. We make all our artifacts available: https://github.com/JordanSamhi/Call-Graph-Soundness-in-Android-Static-Analysis

2 Motivation

A static analysis uses call graphs to represent the calling relationships between methods. Most static analyses aim to be sound. When the analysis is unable to resolve a program behavior such as aliasing, method dispatch, reflection, etc., a sound analysis must over-approximate the program's possible run-time behaviors. Other challenges for call graph construction in Android apps include inter-component communication (ICC), callbacks, calls into and out of native code, asynchronous tasks, GUI-related events, etc., for which the target of method calls cannot be inferred statically. Unsoundness occurs when the estimate of the targets of a method call omits a target that is called on some execution of the call site [43].

Listing 1 shows how unsoundness occurs with an implicit call. An implicit call is a method call to a method executed at run time without a call to the method in the app's code. Implicit calls, as established by our research (cf. Section 4.3), pose a significant obstacle to call graph construction in static analysis. Omitting them makes a static analyzer unaware of certain interactions and dependencies between different parts of the program.

Listing 1 contains: ① a class called `MyTask` (line 1) extending the Android framework `AsyncTask` class; and ② an Activity called `MainActivity` (line 11), the first Activity launched when the app is run, its `onCreate()` method is executed (lines 13–16). On line 14, an object `myTask` of type `MyTask` is created. Then, on line 15, the `execute()` method is called on the `myTask` object. This call will *not trigger* any method called *execute()* in the app's code. Instead, the Android framework implicitly triggers the execution of the `doInBackground` method in the app code (lines 3-5). Additionally, the `onPostExecute` method will execute after `doInBackground` completes, even though the app source code does not call this method.

Implicit calls present a significant challenge for static analysis. They require additional information to model and analyze an app's behavior. However, it raises the question of how analysts can be expected to possess this knowledge. The Android framework, third-party frameworks, and libraries lack comprehensive documentation regarding the existence and usage of implicit mechanisms. This work measures the amount of such methods, among others, missed by existing state-of-the-art static analyzers.

3 Empirical Study Setup

Our experimental methodology compares static models with methods invoked at run time. Dynamic analysis reports actual calls,

Listing 1 Example: how implicit calls can be triggered in apps.

```java
① public class MyTask extends AsyncTask<Void, Void, String> {
      @Override
      protected String doInBackground(Void... params) {
          return "Background task completed";
      }

      @Override
      protected void onPostExecute(String result) {
          textView.setText(result); // Called after doInBackground
      }
  }

② public class MainActivity extends AppCompatActivity {
      @Override
      protected void onCreate(Bundle savedInstanceState) {
          MyTask myTask = new MyTask();
          myTask.execute(); // implicitly calls doInBackground
      }
  }
```

allowing us to identify unsoundness in static models: a method that has been called at run time *must* appear in the static call graph. Current state-of-the-art static analyzers overlook many methods, showing that statically representing Android apps is not trivial and is still an open problem. Our goal is to discern why such discrepancies occur and identify the reasons contributing to this problem. To this end, our study aims to answer the following research questions:

RQ1: How do static analyzers' call graphs compare to each other?

RQ2: How (un)sound are static analyzers? That is, what proportion of methods executed at run time is missed by static analyzers?

RQ3: What are the root causes of unsoundness in static analyzers?

RQ4: To what extent would approaches collecting callbacks from the Android framework systematically improve the models of static analyzers?

The remainder of this section describes the setup of our study. Our approach consists of three phases: ① building call graphs of Android apps using several state-of-the-art static analyzers (section 3.3); ② executing the same apps to collect the methods called at run time (section 3.4); and ③ comparing and studying both sets of methods (section 4).

3.1 Dataset

We randomly collected a representative sample of 1000 recent real-world apps from the 24 000 000+ APKs in the AndroZoo repository [3] (with a confidence level of 99% and a confidence interval of ± 5%), all of which were collected in 2023 by AndroZoo from app markets. The average size of apps in our dataset is 24MB, the median is 15MB, and the standard deviation is 24MB. These 1000 apps contain an average of 132 687 methods, an average of 108 973 non-library methods, and the medians are, respectively, 110 378 and 96 571. Our dataset is available in our project's artifacts, along with our experimental framework.

3.2 Tools

Our selection criteria for tools included: ① tools relying on call graphs; and ② open-source (which allows us to slightly modify them for extracting their call graphs).

To gather tools, we relied on a frequently used strategy in systematic literature reviews (SLRs): identifying relevant keywords to help us identify a large number of potential papers from well-known databases. We searched for "android" AND "static" AND ("call-graph" OR "call graph" OR "call-graph" OR "model" OR "implicit" OR "callback" OR "component"). We searched in three databases: IEEE Xplore, ACM DL, and Science Direct. This search yielded 423 distinct papers, which describe 61 static analysis tools using a call graph. Among these tools, 21/61 (34%) match our criteria (i.e., use a call graph, and are open-source).

Tool	Runnable	Tool	Runnable	Tool	Runnable
ACID [36]	✓	DroiDel [9]		IccTA [34]	✓
AppoScopy [20]		DroidRA [35]	✓	Jicer [40]	✓
ArpDroid [16]	✓	DroidSafe [24]	✓	MaMaDroid [39]	✓
BackDroid [57]	✓	ELEGANT [33]		NatiDroid [32]	✓
BackStage [28]		FlowDroid [6]	✓	NaDroid [21]	
DidFAIL [27]		Gator [61]	✓	RAICC [44]	✓
Difuzer [47]	✓	HybriDroid [30]		SootFX [26]	✓

Table 1: Open-source static analysis tools that use a call graph. "Runnable" means we could adapt, build, and run the tool.

Table 1 shows the list of tools that match our criteria. Among these tools, we were unable to build 6 of them (i.e., Hybridroid, ELEGANT, Backstage, DroidDEL, DIDFail, and AppoScopy) and unable to run one of them (i.e., NaDroid). We have contacted the authors of these 7 tools. We have received three answers, among which two to help us build the tools (DidFail and ELEGANT), but eventually we were not able to build them, and in the third answer regarding Backstage, one of the authors said that they will try to contact the developer, we have not received any news so far.

We ran 14 static analysis tools on all the apps, but we dropped DroidSafe from our experiments as it could only analyze 6 out of the 1000 apps. Therefore, this paper's results contain 13 tools.

Note that while each considered tool computes a call graph, each tool is built differently and may have been designed for different goals. We briefly describe the main goal of each tool in Table 2. The supplementary material gives the configuration of each tool as run in our experiments.

(1)	FlowDroid	detects data leaks in Android apps
(2)	IccTA	detects potential data leaks in apps with an ICC sensitivity
(3)	RAICC	extends FlowDroid with additional ICC methods
(4)	DroidRA	extends FlowDroid to resolve reflective calls and improve call graphs
(5)	NatiDroid	performs cross-language static analyses of both bytecode and native code
(6)	MaMaDroid	detects malware based on app behavior
(7)	BackDroid	on-the-fly bytecode search to improve inter-procedural analysis
(8)	SootFX	extracts features for machine learning
(9)	ACID	detects API compatibility issues
(10)	Gator	performs callback-sensitive static analysis
(11)	Jicer	is a bytecode slicer
(12)	ArpDroid	detects and repairs incompatible uses of the runtime permission
(13)	Difuzer	detects hidden sensitive information in apps (via data flow analysis)

Table 2: Description of the tools.

3.3 Running Static Analysis

For each static analysis tool that we have considered, we modified them slightly to extract relevant data, hence our modifications have no impact on the analyses. Note that several tools, such as DroidRA, modify the original call graph used. Hence, before extracting the call graphs, we let the tools exercise the call graphs so that we extract the call graphs they use for their analyses.

We then ran the modified tool (with different call graph construction algorithms when possible; see fig. 2 and table 4 for all 25

Listing 2 Instrumentation for logging methods called at run time.

```
1  public void m() {
2  + Log.d("MY_LOGGER", "<Class: void m()>");
3    // method body
4  + Log.d("MY_LOGGER", "<Class: void m()>--><Cat: void <init>()>");
5    Cat c = new Cat();
6  + Log.d("MY_LOGGER", "<Class: void m()>--><Cat: void walk()>");
7    c.walk();
8  }
```

configurations) with a 1-hour timeout and with the default configuration described by the developers. Prior to using the 1-hour timeout, we performed the same experiment with a 10-minute timeout. The findings remained unchanged. Moreover, with a 1-hour timeout, the number of successfully analyzed apps did not increase. Thus, increasing the timeout is unlikely to ① change the findings of this paper; and ② significantly increase the number of successfully analyzed apps. The interested reader can check our artifacts for all the results. Note that, when the timeout is reached, there is no call-graph, analyzers do not yield a partial call graph.

The output of this process is, for each app and each call graph construction algorithm, six sets of methods and two sets of edges:

(1) *SM*: the set of all methods in the app (to obtain this information, we have iterated over all classes and counted all methods present in these classes)[1]
(2) SM_{cg}: the set of all methods in the call graph
(3) $SM_{\neg cg} = SM - SM_{cg}$: the set of all methods that do not appear in the call graph
(4) $SM^{\neg l}$: set of all non-library methods
(5) $SM^{\neg l}_{cg}$: set of all non-library methods in the call graph
(6) $SM^{\neg l}_{\neg cg} = SM^{\neg l} - SM^{\neg l}_{cg}$: the set of all methods that are neither in the call graph nor classified as libraries
(7) *SE*: the set of all edges in the call graph
(8) $SE^{\neg l}$ the set of all edges in the call graph whose targets are non-library methods.

To determine whether a method is a library or not, we relied on the list of Android libraries given in [45].

3.4 Running Dynamic Analysis

We built a dynamic call graph analysis. Its main component is an instrumentation tool called AndroLog [48] that inserts a log statement at: ① the beginning of each method in the app; and ② each method call *in the app*. The dynamic call graph analysis uses the log to construct three sets per app: *DM* a set of methods called; $DM^{\neg l}$ a set of non-library methods called; and *DE* a set of dynamically collected edges. Note than only the code inside apps is instrumented, the Android framework is not.

As an example, consider the code in Listing 2. On line 2, a simple log statement is added with the name of the method (in Jimple format) that is being executed. Lines 4 and 6 demonstrate how our instrumentation tool would insert log statements to record that the current method is invoking the method called within it. Our implementation is more sophisticated; for example, it correctly handles the case when the call is in a subexpression that might or might not get executed. Our implementation does not record the actual targets of reflective calls. At a call that is executed via dynamic dispatch, the target is recorded as it appears in the bytecode, rather than all the method implementations among which dispatch might choose. Missing calls in the dynamic call graph mean that this paper may under-report the unsoundness of static analysis tools.

After instrumentation, we signed each app and installed it on a headless Android emulator based on Google's android-33 system image (x86_64).

Subsequently, we exercised each app for 5 min with Monkey [22], generating random inputs. We used Monkey because of empirical evidence [53, 54] that, despite the existence of more complex approaches to augment code coverage, Monkey still achieves the best coverage performance in practice. Also, a recent study has shown that after 5 min, the proportion of code covered using Monkey reaches a "plateau" [54].

The dynamic analysis observed 310 595 043 method calls (including implicit invocations in the apps) to 1 082 265 unique methods across the 1000 apps of our dataset[2]. The average code coverage (at the method level) for our dynamic analysis over all 1000 apps is 8% and the median is 4%. We acknowledge that the code coverage is low. However, we remind the reader that our goal is not to reach high code coverage, and that a low code coverage will actually reinforce our findings. Indeed, if the number of executed methods is low, and if static analyzers miss a high proportion of these methods, then the problem of unsoundness that this paper reports is a lower bound of the actual problem.

Figure 1 shows the distributions of the number of method calls and unique methods called during the dynamic analysis with and without library methods. Table 3 shows the mean and median numbers of method calls and unique methods collected from the dynamic analysis. Results indicate that, on average, there are more than twice as many library calls in Android apps during execution as non-library calls.

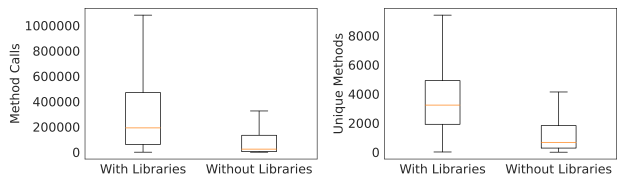

Figure 1: The number of methods called at run time.

	Mean	Median
Method Calls (*DM*)	310 595	191 862
Unique Methods	3 885	3 236
Non Library Method Calls ($DM^{\neg l}$)	136 374	24 310
Non Library Unique Methods	1 355	678

Table 3: Mean and Median of the number of method calls and the number of unique methods with and without libraries

4 Empirical Findings

4.1 RQ1: Comparison of Static Analyzers Model

Figure 2 reports the number of apps successfully analyzed by each tool. For instance, FlowDroid extracted 624 call graphs using the

[1]We compute these sets using the Jimple intermediate representation.

[2]For instance, a method m(), i.e., a given method implementation, can be called *n* times, but is only counted once in the set of unique methods.

CHA algorithm and 429 call graphs using the SPARK algorithm. For the remaining apps, i.e., 1000 − 624 = 376 for FlowDroid-CHA, FlowDroid crashed 7 times and reached the timeout 369 times. For SPARK, FlowDroid reached the timeout for 566 apps, and crashed for 5 apps (the detailed results are available in our artifacts). Overall, most of the considered static analysis tools can only successfully analyze about half of the 1000 apps. This is a threat to the validity of previous work that used these tools, which may not have been run on a representative sample of apps. The most robust tools are NatiDroid, SootFX, ACID, and Gator.

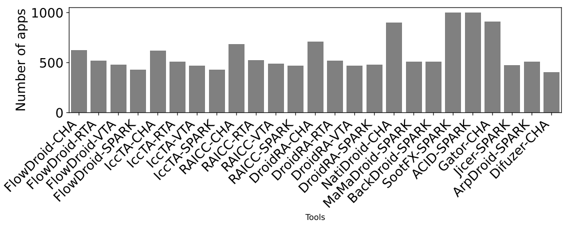

Figure 2: Number of apps successfully analyzed per tool

Only 126 apps were successfully analyzed by every tool. The remainder of this paper focuses on those apps, in order to permit a fair comparison among the tools. The supplementary material provides the results for all apps. These 126 apps are probably among the simplest of the 1000, since they did not trigger bugs or limitations in the static analysis tools. Therefore, the actual unsoundness of static analysis tools in practice is probably worse than reported by this paper.

Table 4 provides data about the apps together with their libraries (the "With libraries" columns), and also data about only the app code (the "Without libraries" columns). Both sets columns are data from the same analysis run, but the "Without libraries" columns only count the part of the computed call graph that comes from developer code. Each $|SM|$ is the number of methods that the analysis tool discovered in the app. The "%M in CG" columns report how many of those methods appear in the call graph: $|SM_{cg}|/|SM|$ (see section 3.3 for definitions). provide statistics about the methods identified by the tools within the successfully analyzed apps.

Exhaustively comparing the tools is outside the scope of this paper, because our objective is to quantify unsoundness by using dynamic analysis as a ground truth. Nonetheless, we note a few observations from the data.

Different tools find different methods $|SM|$ in an app. One set of tools (ACID, ArpDroid, BackDroid, Difuzer, MaMaDroid, NatiDroid, and SootFX) find about61 000 methods per app; another set of tools (DroidRA, FlowDroid, IccTA, Jicer, and RAICC) finds about 71 000 per app, and Gator finds 111 000. We have investigated and could find the following explanation for these differences. Firstly, we confirm that none of the following tools: ACID, ArpDroid, BackDroid, Difuzer, MaMaDroid, NatiDroid, and SootFX consider all dex files in apps for the static analysis, they only consider the main "classes.dex" file and therefore miss many methods if additional ".dex" files are present in apps, which explain the low number of methods discovered statically compared to the rest. Secondly, we confirm that the following tools: DroidRA, FlowDroid, IccTA, Jicer, and RAICC consider all ".dex" files in apps which explains why they are able to find about 10 000 additional methods compared to the first set of tools. For the last tool, i.e., Gator, we confirm that it also consider all ".dex" files in apps but could not find any additional hint about why it finds 40 000 additional methods in apps.

Table 4: Methods gathered statically in apps, for 126 apps. (CG = call graph, M. = Methods, Avg. = Average). The denominator for the % columns is the number of methods in apps.

		With libraries			Without libraries		
		Avg. $|SM|$	% M. in CG	Avg. $|SE|$	Avg. $|SM^{\neg l}|$	% M. in CG	Avg. $|SE^{\neg l}|$
FlowDroid	CHA	71 051	38%	399 975	6651	66%	48 218
	RTA	71 046	24%	227 493	6651	52%	33 802
	VTA	71 045	18%	109 519	6651	42%	16 788
	SPARK	71 031	5%	15 250	6649	12%	2391
IccTA	CHA	71 051	38%	399 981	6651	66%	48 220
	RTA	71 046	24%	227 541	6651	52%	33 746
	VTA	71 045	18%	109 023	6651	41%	16 703
	SPARK	71 031	5%	15 249	6649	12%	2391
RAICC	CHA	71 051	38%	397 791	6651	66%	47 894
	RTA	71 046	24%	224 574	6651	52%	33 271
	VTA	71 045	19%	111 151	6651	41%	16 605
	SPARK	71 031	6%	16 264	6650	12%	2434
DroidRA	CHA	71 053	38%	397 872	6652	66%	47 903
	RTA	71 048	24%	224 992	6652	52%	33 452
	VTA	71 047	19%	111 188	6652	42%	16 749
	SPARK	71 033	6%	16 437	6650	12%	2491
NatiDroid	CHA	61 758	81%	469 025	4837	88%	40 398
MaMaDroid	SPARK	60 500	5%	12 592	4791	14%	2007
BackDroid	SPARK	60 500	5%	12 592	4791	14%	2007
SootFX	SPARK	61 707	0%	101	4798	1%	9
ACID	SPARK	61 707	8%	54 169	4798	48%	4124
Gator	CHA	110 824	73%	1 920 412	31 342	90%	655 813
Jicer	SPARK	71 144	6%	15 763	6651	11%	2302
ArpDroid	SPARK	60 500	5%	12 593	4791	14%	2007
Difuzer	CHA	60 567	34%	245 987	4809	65%	31 060

IccTA, RAICC, and DroidRA add few edges to the call graph. These tools are built upon FlowDroid and are designed to add edges to its call graph; in other words, they are intended to correct unsoundness in FlowDroid. Corresponding rows in table 4 are little different — always well under 10% and usually closer to 0% different. As shown later in this paper, they do not address the most important sources of unsoundness in FlowDroid. In some cases, RAICC-CHA and DroidRA-CHA have *fewer* edges than FlowDroid-CHA, even though those algorithms are designed to *add* edges. We do not have an explanation for this behavior, though nondeterminism may play a part [50].

More precise algorithms succeed in pruning the call graph. SPARK is more precise than VTA, VTA than RTA, and RTA than CHA. Table 4 shows that the more precise algorithms lead to graphs containing fewer methods and fewer edges. This aligns with the fact that enhancing precision results in reducing over-approximation.

Static analysis considers large portions of apps to be dead code. In the "With libraries" columns, most code is dead code (i.e., methods not in call graph), likely because no app exercises all parts of a library that it depends on. More surprising are the "Without libraries" columns, where FlowDroid considers apps to contain 1/3 to 7/8 dead code (depending on the call graph construction algorithm). This likely reflects unsoundness (many methods are present but are not modeled statically in the call graph), since it seems unlikely that developers would ship 8 times as much code as an app needs, most of it useless. NatiDroid and Gator more plausibly claim that 88–90% of the app code may be executed.

Implementations of the same call graph construction algorithm differ. Different implementations of an algorithm may yield slightly different results due to different modeling and implementation choices. The implementations of CHA in FlowDroid and Difuzer retain very similar percentages of methods. However, the CHA implementations in NatiDroid and Gator retain dramatically more methods. In the "Without libraries" columns, all SPARK implementations retain 11–14% of methods — except SootFX which retains 1% and ACID which retains 48%. We investigated SootFX and found that its problem is its set of entry points: SootFX uses only `Threads` as an entry point, which misses most of the application. The set of entry points is at least as important as the call graph construction algorithm.

RQ1 answer: Our comparison of the static analyzers shows that: ① only a small proportion, i.e., 58% on average, of apps can be analyzed by static analysis tools in a 1-hour timeframe; ② static analysis approaches supposed to improve call graphs' soundness show little variation in the number of methods and edges in their call graphs; and ③ although static analysis tools share similar call graph construction algorithms, they exhibit different (sometimes significantly divergent) call graphs.

4.2 RQ2: Unsoundness: Methods Missed by Static Analyzers

Figure 3 shows the precision, recall, and f_1 score of the methods in static call graphs compared to *methods in dynamic call graphs* (since the dynamic analysis does not yield complete call graphs given that coverage is limited, we report precision with respect to the call graphs collected dynamically) for each app and each configuration, i.e., tools and call graph construction algorithms. For a fair head-to-head comparison, fig. 3 reports results for the 126 apps that were analyzed by all tools. The supplementary material provides the results for all apps successfully analyzed by each tool.

The goal of a more precise call graph construction algorithm, such as SPARK, is to soundly improve the precision of the call graph. That is, a more precise algorithm should remove infeasible nodes and edges from the call graph *without* removing feasible ones. Figure 3 shows that these algorithms fail at their goal. While they do improve precision, they do so at the cost of increased unsoundness. More specifically, in fig. 3, SPARK tends to have the greatest unsoundness, CHA tends to have the least, and RTA and VTA are in between. But even the CHA-based tools report unacceptable unsoundness with 21%-67% recall. We were not able to determine whether the more precise algorithms are fundamentally flawed (as fielded in the given tools), or the implementations of the algorithms are defective. Although the more precise algorithms are more complex to implement, we did not expect that complexity to lead to systematically more bugs.

One outlier is the CHA-based Gator tool, which has nearly the highest level of unsoundness (lowest recall). After investigation, we found that Gator relies on a unique two-step technique to build its call graph: first, it considers all methods within the app as a potential entry point which is a considerable over-approximation; and second, it refines the set of potential entry points with many rules based on permissions, the manifest, etc. Future tools should not adopt its novel call graph construction technique.

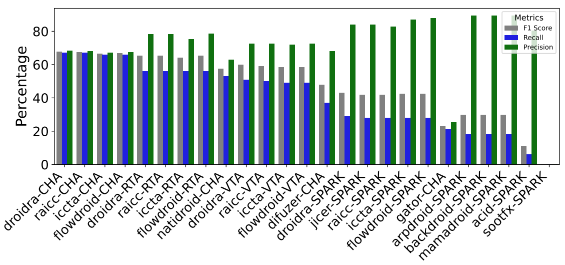

Figure 3: Comparison of recall, precision, and f_1 score of all configurations of tools and call graph construction algorithms.

Table 5: Methods called at run time but not modeled statically.

Tool	Algorithm	Missing methods ($DM \setminus SM_{cg}$)				
		Mean	Median	Min	Max	Total
FlowDroid	CHA	1294	685	14	14 397	163 099
	RTA	1499	936	29	14 399	188 845
	VTA	1621	1110	41	14 399	204 220
	SPARK	1965	1216	42	14 430	247 577
IccTA	CHA	1294	685	14	14 397	163 094
	RTA	1498	938	29	14 399	188 799
	VTA	1621	1110	41	14 399	204 268
	SPARK	1965	1216	42	14 430	247 577
RAICC	CHA	1295	684	14	14 397	163 225
	RTA	1522	983	41	14 399	191 785
	VTA	1614	1064	41	14 399	203 374
	SPARK	1935	1188	42	14 430	243 871
DroidRA	CHA	1295	683	14	14 397	163 195
	RTA	1521	976	41	14 399	191 606
	VTA	1612	1064	41	14 399	203 174
	SPARK	1933	1188	42	14 430	243 570
NatiDroid	CHA	1299	704	0	14 382	163 639
MaMaDroid	SPARK	2085	1344	42	14 430	262 671
BackDroid	SPARK	2085	1344	42	14 430	262 671
SootFX	SPARK	2641	1828	84	15 322	332 824
ACID	SPARK	2286	1704	84	15 322	288 068
Gator	CHA	2213	1462	6	14 815	278 840
Jicer	SPARK	1962	1194	42	14 430	247 237
ArpDroid	SPARK	2085	1344	42	14 430	262 671
Difuzer	CHA	1542	926	14	14 397	194 256

Our finding shows that (in current implementations) the more precise the algorithm, the more unsound: it misses many methods, i.e., apps' code, during analysis. Most research in call graph construction algorithms has the goal of improving precision. Our research throws doubt on the desirability of precision-focused call graph construction algorithms since they are time-consuming to build and often result in a significant proportion of code being overlooked. They are strictly worse for analyses related to security, among other applications.

We again remind the reader that the methods collected dynamically were gathered in a mere 5 minutes of random execution. Hence, our findings likely underestimate the volume of methods static analyzers miss.

The "Total" column in table 5 is proportional to missed methods, and inversely proportional to soundness, in fig. 3. Table 5 breaks the data down more finely.

We can make several observations: ① although CHA is the highest degree of over-approximation in existing call graph construction algorithms, it still falls short in capturing a large quantity of methods; ② SPARK, the most precise call-graph construction algorithm for Android apps, falls behind, missing almost twice as many methods on average as CHA does, i.e., SPARK is less sound than CHA (and RTA and VTA); and ③ with SootFX and ACID at least one app shows up to 15 322 methods missing, a statistic that could prove critical if the app is, in fact, a malware.

We observe, in the "Min" column that for some apps, the minimum number of methods missed is low, e.g., 14 for FlowDroid-CHA. This could indicate that, for some apps, the static models appear to be sound, i.e., it does not miss many methods called dynamically. To investigate we have plotted the proportion of methods missed per app successfully analyzed for FlowDroid-CHA in Figure 4. We can see that indeed, for some apps, the proportion of methods missed is low (i.e., for 6 apps, less than 10% of methods were missed). However, the proportion of methods missed grows quickly and is critical for most apps.

Furthermore, Figure 5 shows the proportion of code covered during the dynamic analysis for each of the 126 apps (in the same order as in Figure 4). We see no correlation between unsoundness and code coverage, i.e., this is not because code coverage is better, that unsoundness is worse.

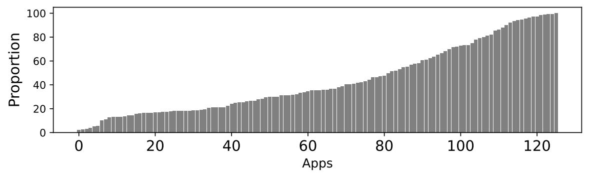

Figure 4: Proportion of dynamically-executed methods missed by FlowDroid-CHA.

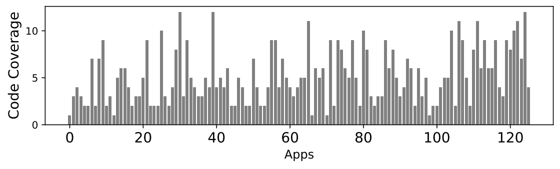

Figure 5: Code Coverage of the dynamic analysis for the 126 apps successfully analyzed by all tools. The code coverage is at the method level and is expressed in %.

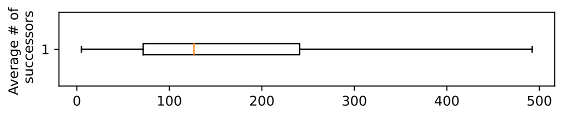

Figure 6: Distributions of the average number of transitive successors of methods dynamically called.

Moreover, let us now focus on the dynamically-extracted call graph. For each app, we have computed the following: for each node in the call graph, the number of transitive successors of the node. Figure 6 shows the distribution of the average number of transitive successors for each node for each app. We can see that if a method m is missed by a static analyzer, this is in fact hundreds of methods that will not be modeled, and not analyzed if m is the only entrypoint to these methods.

Qualitative Analysis of the Methods Missed On our reduced dataset, there were 25 successful tool invocations on each of 126 apps = 3150 total invocations. Table 6 shows the top 10 most frequently missed methods. The top 5 methods belong to the com.ryanheise package, which is a Flutter plugin [49]. Flutter is a framework for building apps. The following 5 most frequently missed methods have their class name obfuscated, preventing us from identifying their origin. In addition, 8 out of these 10 methods are class constructors. (In Java, <init> refers to a constructor method, and <clinit> a static class initializer.) The remaining 2 methods, onCreate()–a lifecycle method of the AudioService class that extends the MediaBrowserServiceCompat class–and size(), were called during run time but not modeled statically. This is unusual, given that static analyzers like FlowDroid are expected to handle lifecycle methods and other standard methods such as constructors.

Table 6: Top 10 most missed methods by static analyzers.

Occurrences	Method
2594	com.ryanheise.audioservice.AudioService.<clinit>()
2594	com.ryanheise.audioservice.AudioService.<init>()
2562	com.ryanheise.audioservice.AudioService$d.<init>(AudioService)
2555	com.ryanheise.audioservice.AudioService$a.<init>(AudioService,int)
2551	com.ryanheise.audioservice.AudioService.onCreate()
2422	l.b.<init>()
2358	k.a.<init>()
2315	k.b.<init>()
2290	l.b.size()
2290	r.g.<init>()

Table 7: Top 10 most missed methods by static analyzers without obfuscated class names.

Occurrences	Method
2594	com.ryanheise.audioservice.AudioService.<clinit>()
2594	com.ryanheise.audioservice.AudioService.<init>()
2562	com.ryanheise.audioservice.AudioService$d.<init>(AudioService)
2555	com.ryanheise.audioservice.AudioService$a.<init>(AudioService,int)
2551	com.ryanheise.audioservice.AudioService.onCreate()
2203	com.unity3d.player.h.onActivityStopped(Activity)
2191	com.unity3d.player.h.onActivityPaused(Activity)
2125	vn.hunghd.flutterdownloader.FlutterDownloaderInitializer.onCreate()
2125	vn.hunghd.flutterdownloader.FlutterDownloaderInitializer.a(Context)
2098	vn.hunghd.flutterdownloader.FlutterDownloaderInitializer$a.<init>(g)

Among the next 5 methods with non-obfuscated class names (Table 7), two methods come from the unity3d framework [52], and the other 3 belong to a Flutter plugin [17]. For all of the above 15 methods, some algorithms included them in the call graph.

Tables 6 and 7 underscore that frameworks present a significant obstacle to the precise static modeling of Android apps[5, 9], with quantitative data beyond previous investigations.

We have contacted the authors of all the tools considered in our study. We have received 5 replies from the authors of RAICC, BackDroid, Gator, ArpDroid, and Difuzer. None of them is surprised about the unsoundness of their model and say that the goal of their tool is not compute a sound and complete call graph.

RQ2 answer: Our findings reveal an inverse relationship between the precision and soundness of call graph algorithms. Less precise algorithms seem more sound but still have significant issues. CHA-based tools miss at least 40% of methods, while SPARK-based tools can miss up to 100% for SootFX.

4.3 RQ3: Root Causes of Unsoundness

Section 4.2 showed that static modeling misses many methods — that is, it is significantly unsound. This section asks, why are they overlooked?

Dynamic Call Graph and Entry Point Methods. Our dynamic analysis (section 3.4) captures every call from within the app, and every execution of every method in the app. We remind we instrumented all apps to log every method call during execution, as detailed in Section 3.4. Using the set of dynamically collected edges *DE*, this procedure generates a call graph for each app, call graph that we will call *Dynamic Call Graph* (or DCG in short) to differentiate this call graph from the one obtained via static analysis. If a method is ever executed without being called from within the app, we call it an *entry point. We hypothesize that these entry point methods are a major cause of unsoundness.*

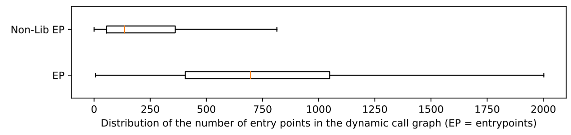

Figure 7: Distribution of the number of entry point methods called during the dynamic analysis.

Figure 7 shows the distribution of the number of entry point methods in the DCG for the 1000 apps of our dataset (with and without considering libraries: note that libraries are embedded in Android apps when apps are packaged; thus libraries can be entry points). When libraries are considered, the median is 697, representing 16.37% of the total number of methods in the DCG, and the mean is 833 per app. For non-library methods, the median is 136, representing 4% of the total number of methods and 20.2% of non-library methods. The mean is 284 per app. Both distributions are significantly different, as confirmed by a Mann-Whitney-Wilcoxon (MWW) test [37] (significance level set at 0.05). Our results indicate that, on average, 1/6 of the methods in the DCG are entry points, and a 1/5 of non-library methods are entry points. Also, this result indicates that most entry point methods are methods from libraries.

These entry point methods are key in our study because they are hard for static analyzers to discover. Indeed, there is no call to these methods in the app code. So, without proper modeling of these entry points methods, a static analyzer will simply miss them. Moreover, as shown in Section 4.2, if one method is missed, there are hundreds of additional transitive methods potentially missed.

Let us now further investigate these entry point methods. Table 8 shows the top ten entry point methods (with libraries) in the DCGs of apps in our dataset. Six methods are from the `androidx` package and four are from the `com.google` package, which provides additional libraries developed by Google.

Entry Point Methods Missed by Static Analyzers: Previously, we have seen that many different methods from many classes are entry points in the DCG, but we have not yet checked if static analyzers miss these methods. In fact, all the top 10 methods of Table 8 and Table 9 are missed by static analyzers. We have investigated further, and we observed that all static analyzers overlooked the top 5352 entry point methods among the 776 075 entry points identified. Among these identified entry points, 34.5% (267 843) were missed by static analyzers. Regarding non-library entry point methods, representing 272 313 of the total, the top 5863 methods were not modeled. In total, 95 309 methods were missed, representing 35% of the non-library entry points. Our results highlight a substantial gap in the coverage provided by static analysis tools concerning identifying entry points of the DCGs.

Table 8: Top 10 entry point methods in the DCG of the 1000 apps (ranked per number of occurrences)

Occurrences	Method
961	androidx.core.app.CoreComponentFactory.<init>()
955	androidx.core.app.CoreComponentFactory.instantiateApplication(ClassLoader,String)
944	androidx.core.app.CoreComponentFactory.instantiateProvider(ClassLoader,String)
794	androidx.startup.InitializationProvider.onCreate()
790	com.google.firebase.provider.FirebaseInitProvider.onCreate()
786	androidx.startup.InitializationProvider.<init>()
781	com.google.firebase.provider.FirebaseInitProvider.attachInfo(Context,ProviderInfo)
780	com.google.firebase.provider.FirebaseInitProvider.<init>()
763	androidx.core.app.CoreComponentFactory.instantiateActivity(ClassLoader,String,Intent)
713	com.google.android.gms.dynamite.DynamiteModule.<clinit>()

Table 9: Top 10 entry point non-library methods in the 1000 dynamic call graphs

Occurrences	Method
191	com.unity3d.player.UnityPlayer$5.run()
185	com.ryanheise.audioservice.AudioService.onCreate()
185	com.ryanheise.audioservice.AudioService.<init>()
184	com.ryanheise.audioservice.AudioService.<clinit>()
184	com.unity3d.player.UnityPlayer$d.<clinit>()
183	com.unity3d.player.UnityPlayer$1.onClick(DialogInterface,int)
181	k.a.<clinit>()
174	b1.a.<clinit>()
170	com.unity3d.player.UnityPlayer.<clinit>()
168	b1.b.<clinit>()

Note that these missed entry point methods represent 20.3% of the total methods missed by the static analyzers. The rest of the methods missed (i.e., 79.7%) are transitive methods of the dynamic entry points. This result validates our hypothesis that entry points in the DCG are one of the main causes of unsoundness in Android app static analyzers.

Let us now examine the top 40 method names that were overlooked by static analyzers among the entry points identified through dynamic analysis. Note that, in this part, we only look at *method names*, i.e., no matter their class (as studied in Tables 8 and 9). Figure 8 illustrates the number of occurrences of method names found in the set of dynamic entry points that were not captured by static analyzers. First, we notice that static constructors (i.e., $< clinit >$) are by far the most missed methods with 56 708 occurrences. By comparison, constructors (i.e., $< init >$) have 4209 occurrences. This result indicates that constructors are particularly prone to being overlooked by static analysis tools. We observe that a high proportion of obfuscated methods are also hard to model by static analyzers (i.e., methods with names such as “a”, or “b”). Interestingly, several methods that are indicative of well-known implicit mechanisms, such as run(), call(), onCreate(), execute(), onClick(), etc., continue to be neglected by static analyzers. This observation shows significant opportunities for improving the soundness of existing tools and techniques.

Further, Figure 9 shows the same information as in Figure 8 but without obfuscated method names. We still observe well-known implicit mechanisms with methods starting with “on”, such as onConfigure, onResame, etc. But, we also see that several other methods are overlooked, such as accept, read, close, build, values, etc. Our

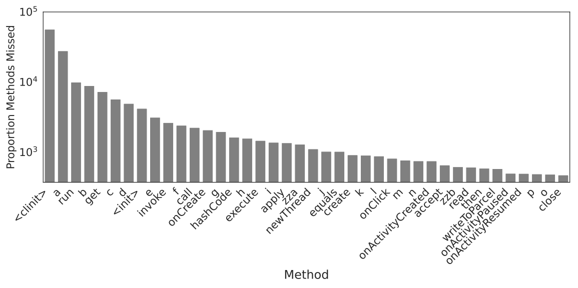

Figure 8: Top 40 most unique missed entry point method' names by static analyzers (logarithmic scale)

results give insights into which direction future research should dig into to improve the soundness of static analysis. We provide, in our replication package, a list of 7137 unique method names and a list of 220 127 methods to provide insights for future research to improve the soundness of static analysis of Android apps.

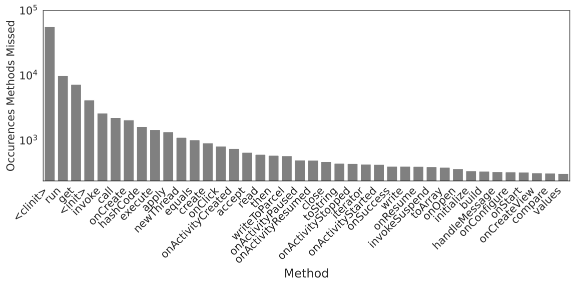

Figure 9: Top 40 most unique non-obfuscated missed entry point method' names by static analyzers

RQ3 answer: Determining the root causes of missed methods is complex, necessitating a thorough investigation of all missed methods in apps. Despite this complexity, our study offers several insights: ① static analyzers lack understanding of the Android and external frameworks; and ② modeling entry point methods challenge call graph construction. Our study highlights many opportunities for future research and paves the way for improving the soundness of static analysis tools.

4.4 Case Study

This section presents examples of methods missed by static analysis tools. These examples were selected randomly among the methods that were missed by static analyzers.

Case 1: Method com.lyokone.location.FlutterLocationService.a-(Activity)[3] is invoked by h.j.a.b(), which is called by h.j.a.a(), and ultimately by h.j.a.onDetachedFromActivity(), a Flutter method called implicitly defined in class io.flutter.embedding.engine.plugins.activity.ActivityAware. To conclude, method com.lyokone.location.FlutterLocationService.a(Activity) is missed because of an **initial implicit call** that is not modeled since the h.j.a.onDetachedFrom-Activity() methods is an entry point in the DCG.

[3]in app: 78064E0B68067BC764102B47391F0D912F8C250E17A80FDC3828EBBEA53F497F

Case 2: Method com.gulfbrokers.android.app.database.MyDatabase_Impl.createInvalidationTracker()[4] which is invoked in the static initializer of the abstract RoomDatabase class. MyDatabase_-Impl (a subclass of RoomDatabase) is instantiated in the onOpen() method, which is declared in an anonymous object created in the app that extends the androix.room.RoomOpenHelper.Delegate class (which necessitates onOpen() to be overridden). Method onOpen() is called implicitly by androidx (part of the Android framework) and is, therefore, an entry point in the DCG.

Case 3: The method c.a.a.a.i.x.j.h0.onOpen(android.database-.sqlite.SQLiteDatabase)[5], inherited from the android.database.sqlite-.SQLiteOpenHelper class, which implements five implicit mechanisms, was not modeled successfully. This method is called implictly by the android framework. Again, it is an entry point in the DCG.

Similarly, methods from the Cordova and Flutter frameworks were also missed. Specifically, methods com.getcapacitor.cordova.-MockCordovaWebViewImpl$CapacitorEvalBridgeMode.onNative-ToJsMessageAvailable(org.apache.cordova.NativeToJsMessageQueue)[6] and com.mr.flutter.plugin.filepicker.FilePickerPlugin.onAttachedToEngine(io.flutter.embedding.engine.plugins.FlutterPlugin$F-lutterPluginBinding)[7]. These methods, called implicitly by the respective frameworks, are **never called within the apps**, which prevented the tools from modeling them.

These examples indicate limitations to handle implicit mechanisms across various frameworks from existing static analysis tools. This suggests a research gap, indicating that further investigation into these frameworks is needed. By studying these frameworks and identifying their implicit mechanisms, there is a potential to improve static modeling significantly. Hence, our research suggests that the Android framework still contains numerous unexplored implicit mechanisms, highlighting the necessity for continued exploration and analysis.

4.5 RQ4: Can Systematically-Collected Callbacks Improve Soundness?

To the best of our knowledge, two papers, in which the authors devise techniques to collect implicit mechanisms from the Android Framework, have been presented. The first one presents EdgeMiner [12], a technique to collect callbacks (i.e., a type of implicitly invoked method) systematically from the Android framework. Their technique involves the static analysis of the Android framework in order to generate summaries of API methods describing the implicit control flow transitions of callbacks. The second one presents Columbus [11] an automated technique that statically analyzes the Android framework and apps under test. Their technique involves identifying apps' methods that override framework methods to build a mapping between registration and callback methods.

To determine whether systematic approaches aiming to capture implicit mechanisms in the Android framework cover the methods missed by the static analysis tools studied, we have contacted the authors of EdgeMiner and Columbus. Unfortunately, we could not gather data from Columbus due to a lack of response and an empty

[4]in app: E44DDFB0FDE572171BA60595B7AD6BC95AA7ACFA8AA932473C4AE6CBC0A3589C
[5]in app: 14DDDDBCB6395363B490A32C33A8924E16F94295F77E2DC27A1453D754465ABC
[6]in app: 6E2AB5488A78E61BF63EE4CFD942E85D92C2CB10F99F86E2338448E8346555D1
[7]in app: 78064E0B68067BC764102B47391F0D912F8C250E17A80FDC3828EBBEA53F497F

repository [42]. EdgeMiner's authors shared their data, allowing us to conduct a comparative analysis.

By extracting potential callback methods from the EdgeMiner dataset and comparing these with the methods called dynamically in our study, we could evaluate the potential improvement EdgeMiner would bring to static analyzers. We have computed the difference between the set of unique called methods (DM) and the set of EdgeMiner's unique potential callbacks (EM), expressed as $DM \setminus EM$.

With $|DM|$ being 1 082 265 and $|EM|$ being 19 510, we found that $|DM \setminus EM|$ equaled 1 081 886. This suggests that EdgeMiner would only help static analyzers to model 1 082 265 – 1 081 886 = 379 additional methods out of the 1 082 265 called dynamically. Our research suggests that the findings reported in the EdgeMiner's paper may not hold the same level of efficiency for improving the soundness of static analyzers. This limited improvement that EdgeMiner would bring to static analyzers might be attributed to several factors: ① The potential presence of false-positive results in the EdgeMiner dataset (i.e., methods collected not being callbacks); and ② The fact that EdgeMiner focuses on the Android Framework, thereby potentially missing callbacks from external frameworks like React Native, or Flutter, as well as those triggered by languages such as C++ or JavaScript, which can also trigger bytecode methods.

RQ4 answer: Existing approaches that systematically collect implicit mechanisms from the Android framework dot not significantly augment the static models of current static analyzers.

5 Threats to validity

As discussed in section 3.4, the runs in our experiments achieve low coverage. This means that our results underestimate the true extent of the missed methods.

Our evaluation is over only 13 static analysis tools and 25 configurations. This was all the open-source tools we could find and run after extensive efforts. Other tools may behave differently, but 13 tools already offer substantial breadth to our investigation.

Our study's scope was limited to Dalvik bytecode, which is the compiled form of Java/Kotlin Android code and is the dominant form of code that runs in Android apps. If an app used other languages, e.g., C or JavaScript, they would be treated by our experiment like library code. To the best of our knowledge, among the tools considered in our study, only NatiDroid models C code to get a more accurate specification of Android API protection. When a static tool does not model C and JavaScript code, its unsoundness is greater than reported in our experiments.

Static analysis might be hindered by packed apps. Packing is a technique to obscure code or data. The process involves compressing or encrypting code, which is uncompressed or decrypted at run time. This makes static analysis more difficult since the code is hidden and only revealed during execution. To check for packing, we used the ApkId tool [4]. It was observed that only two apps out of the 1000 apps of our dataset were using packed code. In both cases, the packer used was DexProtector [15], it is used to protect apps from tampering, reverse-engineering, and cracking. As a result, since only 2 apps use packing, we can eliminate packing as a potential barrier to creating sound call graphs in our dataset.

Similarly, static analysis can be hindered by dynamic loading. For instance, classes might be downloaded from an external server and loaded at run time. In this case, existing static analyzers cannot account for code loaded dynamically. Packing and dynamic loading can also affect our dynamic analysis since we cannot instrument the packed or loaded code dynamically. This is mitigated by the fact that, if we had instrumented more methods, we would have shown that the problem of unsoundness is worse than it appears.

The performance of a static analysis is affected by its configuration. We used each tool's default configuration, but it is possible that some other configuration would have yielded better call graphs.

Finally, the 1-hour timeout to compute a call graph may not be adequate for some apps or tools, as increasing the timeout could allow to fully analyze more apps.

6 Related Work

The literature contains many approaches to handle implicit mechanisms for the Android platform. Section 6.1 describes studies addressing specific implicit mechanisms in apps and proposes a static model for them. Section 6.2 presents studies to systematically analyze the Android framework for implicit mechanisms. Section 6.3 presents studies measuring call graph soundness.

6.1 Handling particular implicit mechanisms

Callbacks. Callback mechanisms register methods executed by the Android framework in response to events like clicks. These methods are called implicitly, without explicit calls in the app code, posing challenges for static analyzers. Techniques have been developed to account for these callbacks.

FlowDroid [6] was a pioneer in statically modeling callbacks in Android apps. It would construct a call graph per component, identifying calls to system methods with callback interfaces (defined in layout XML files) and incrementally extending the call graph with newly added method calls. Furthermore, FlowDroid also includes handpicked callback methods in configuration files. Yang et al. [62] conducted a study on lifecycle and user-driven callbacks in Android apps. Their approach utilizes a GUI model to capture the app's graphical user interface and generates a callback control flow graph. By analyzing the generated GUI model, the authors extract possible sequences of user GUI events that correspond to valid paths in the model. Likewise, Wu et al. [58] introduced a callback-aware technique focusing on two callback methods: system-triggered and user-triggered. The former includes lifecycle and callback methods of resource classes, such as onDestroy(), while the latter represents callbacks triggered by user interactions with the GUI.

Inter-Component Communication. Android apps consist of various components that communicate with each other through inter-component communication (ICC) methods provided by the Android framework, such as startActivity() and sendBroadcast(). These ICC methods trigger the execution of lifecycle methods implemented by each component, including onCreate() and onReceive(). The communication between components through ICC methods involves implicit calls to these lifecycle methods from the Android framework. Numerous research efforts have been dedicated to resolving the target components of ICC communication.

IccTA [34] relies on composite constant propagation [38] and instrumentation to infers the potential targets of `Intents`, while Amandroid [56] generates data flow and data dependence graphs to

infer possible target components. DroidSafe [24] employs string and class analysis to infer target components and modifies ICC method calls to explicit lifecycle method calls. RAICC [44] addresses atypical ICC methods, such as SmsManager.sendTextMessage(), by resolving potential targets with constant propagation and instrumenting the app to include ICC method calls (e.g., startActivity()). ICCBot [60] is a recently released tool that performs context-sensitive and inter-procedural analysis to infer component transitions connected via fragments, modeling data carried by ICC objects like Intents. Additionally, Chen et al. [13, 14] also recently developed an approach to construct an Activity Transition Graph to create storyboards for apps using ICC-related information to improve activity coverage.
Reflection. The reflection mechanism allows for run-time introspection, enabling the execution of methods without explicitly calling them in the source code, i.e., using reflective calls.

More than ten years ago, TamiFlex [10] was proposed to boost static analyzers with information dynamically gathered about reflective calls. TamiFlex introduced an instrumentation engine to insert regular method calls into apps to boost static analyzers. DroidRA [35] is an instrumentation-based analysis technique that enhances apps by resolving reflective calls using the COAL solver [38] to infer reflection targets. By instrumenting the app and adding explicit calls for each resolved reflective call, DroidRA improves analysis accuracy. In addition, Barros et al. [7] propose a two-fold solution for resolving reflection call targets. Their approach includes a reflection-type system for inferring class and method names and a reflection solver for estimating invocable method signatures.

➠ Contrary to these approaches, we do not aim at identifying and focusing on single implicit mechanisms in Android apps. Our empirical study has been devised to capture any method call – implicit or not– happening at execution time but not modeled statically to understand and thus improve static analysis models.

6.2 Systematic studies

EdgeMiner. EdgeMiner [12] systematically analyzes multiple Android framework versions to identify callbacks and their registration methods using inter-procedural backward data flow analysis.
Columbus. The latest work that systematically analyzes the Android framework to search for callbacks is Columbus [11]. The authors statically analyze the Android framework to identify callbacks by considering protected or public methods with at least one caller in the framework. Then they construct a call graph of the framework and over-approximates possible targets when type inference fails. Additionally, the authors consolidate callbacks inherited from superclasses by traversing the class hierarchy to produce a mapping from the registration method to the callback method.

➠ Contrary to these works, our study aims at revealing the extent to which static analyzers' model are unsound with a high level a precision. Also, our work do not focus on the Android framework and *callbacks*, rather it encompasses any method that could be missed statically, and from any source, e.g., an external framework.

6.3 Call graph soudness

Ali et al. [1] investigate the efficacy of JVM-bytecode-based static analysis across various JVM-hosted languages, including Scheme, Scala, OCaml, Groovy, Clojure, Python, and Ruby. The authors found that the analyses produce sound call graphs for Scheme, Scala, and OCaml, similar to Java, but fails to do so for Groovy, Clojure, Python, and Ruby due to their extensive use of reflection and invokedynamic instructions. Reif et al. [41] have presented Judge, a toolchain designed to identify sources of unsoundness in call graphs. The authors leverage Judge and a test suite to compare different call graph implementations (Soot, WALA, DOOP, and OPAL), evaluate language features and APIs' prevalence that impact soundness in modern Java bytecode. Sui et al. [51] study the prevalence of dynamic language features in modern programming languages. It catalogs dynamic features for Java and presents a micro-benchmark that helps investigate the soundness of static analysis framework (i.e., Soot, Wala, and Doop). Aljawder [2] compares static call graphs built with FlowDroid with dynamic call graphs resulting from execution with Monkey. They found that 62/92 apps were missing edges in the static call graph, but did not investigate root causes. Wang et al. [55] address the unsoundness in static analysis of Android GUIs, highlighting mismatches between existing tools like FlowDroid, IccTA, GATOR, and runtime behavior. Their study shows that these tools often miss runtime sequences of callbacks and parameters, due to the complexity of the Android framework. The paper suggests improving static analysis with runtime behavior insights to reduce unsoundness.

➠ Contrary to these studies, we do not concentrate on specific language features. We examine real-world call graphs produced by static analysis tools and dynamic analysis. We then compare these call graphs, highlighting the discrepancies, i.e., methods called at runtime that *must* appear in the static call graph but do not. Our study reveals that many methods are missed in real-world apps due to factors like implicit calls, not just language features.

7 Conclusion

We conducted an empirical study to measure how much static analyzers under-approximate their models and to identify discrepancies between static models and dynamic data. Our investigation revealed key insights: ① both Android and external frameworks present significant challenges for current static analyzers in building sound models; ② highly precise call graph algorithms result in significant unsoundness; ③ no existing systematic or specific approach drastically ameliorates soundness; and ④ modeling entry point methods poses a major challenge to call graph construction and soundness. Our findings indicate that ① even the most effective and precise static analysis is useless with unsound static models; and ② the issue is likely worse than described due to low code coverage in our dynamic analysis. Therefore, innovative techniques are needed to improve the soundness of Android static analysis.

8 Data Availability

To promote transparency and facilitate reproducibility, our artifacts are publicly available: https://github.com/JordanSamhi/Call-Graph-Soundness-in-Android-Static-Analysis

Acknowledgment

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant references 16344458 (REPROCESS) and 18154263 (UNLOCK).

References

[1] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2021. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2644–2666. https://doi.org/10.1109/TSE.2019.2956925

[2] Dana Aljawder. 2016. *Identifying unsoundness of call graphs in Android static analysis tools.* Master's thesis. Boston University. https://open.bu.edu/handle/2144/17085

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[4] ApkID. 2023. https://github.com/rednaga/APKiD. Accessed December 2023.

[5] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 725–735. https://doi.org/10.1145/2884781.2884816

[6] 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 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[7] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 669–679. https://doi.org/10.1109/ASE.2015.69

[8] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, USA, 669–679.

[9] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. 2015. Droidel: A General Approach to Android Framework Modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Portland, OR, USA) *(SOAP 2015)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/2771284.2771288

[10] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[11] Priyanka Bose, Dipanjan Das, Saastha Vasan, Sebastiano Mariani, Ilya Grishchenko, Andrea Continella, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2023. COLUMBUS: Android App Testing Through Systematic Callback Exploration. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 45th International Conference on Software Engineering, ICSE 2023.

[12] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.

[13] Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu. 2022. Automatically Distilling Storyboard with Rich Features for Android Apps. *IEEE Transactions on Software Engineering* (2022).

[14] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 596–607. https://doi.org/10.1109/ICSE.2019.00070

[15] DexProtector. 2023. https://dexprotector.com. Accessed December 2023.

[16] Malinda Dilhara, Haipeng Cai, and John Jenkins. 2018. Automated Detection and Repair of Incompatible Uses of Runtime Permissions in Android Apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 67–71. https://doi.org/10.1145/3197231.3197255

[17] Flutter Downloader. 2023. https://github.com/fluttercommunity/flutter_downloader. Accessed December 2023.

[18] Michael D. Ernst. [n. d.]. Static and dynamic analysis: Synergy and duality. 24–27.

[19] Yonghui Liu et al. 2023. ReuNify: A Step Towards Whole Program Analysis for React Native Android Apps. In *Proceedings of the 38th ACM/IEEE International Conference on Automated Software Engineering* (Luxembourg, France) *(ASE 2023)*. Association for Computing Machinery.

[20] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

[21] Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. NAdroid: Statically Detecting Ordering Violations in Android Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 62–74. https://doi.org/10.1145/3168829

[22] Google. 2023. Android Monkey, https://developer.android.com/studio/test/monkey. Accessed July 2023.

[23] Google. 2023. Flutter. https://flutter.dev Accessed December 2023.

[24] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[25] Satori Threat Intelligence and Research Team. 2022. Poseidon's Offspring: Charybdis and Scylla, https://www.humansecurity.com/learn/blog/poseidons-offspring-charybdis-and-scylla. Accessed July 2023.

[26] Kadiray Karakaya and Eric Bodden. 2021. SootFX: A Static Code Feature Extraction Tool for Java and Android. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 181–186. https://doi.org/10.1109/SCAM52516.2021.00030

[27] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. 1–6.

[28] Konstantin Kuznetsov, Vitalii Avdiienko, Alessandra Gorla, and Andreas Zeller. 2018. Analyzing the User Interface of Android Apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3197231.3197232

[29] Duling Lai and Julia Rubin. 2020. Goal-Driven Exploration for Android Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 115–127. https://doi.org/10.1109/ASE.2019.00021

[30] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: Static Analysis Framework for Android Hybrid Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 250–261. https://doi.org/10.1145/2970276.2970368

[31] Youn Kyu Lee, Jae young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A SEALANT for Inter-App Security Holes in Android. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 312–323. https://doi.org/10.1109/ICSE.2017.36

[32] Chaoran Li, Xiao Chen, Ruoxi Sun, Minhui Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2022. Cross-Language Android Permission Specification. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 772–783. https://doi.org/10.1145/3540250.3549142

[33] Cong Li, Chang Xu, Lili Wei, Jue Wang, Jun Ma, and Jian Lu. 2018. ELEGANT: Towards Effective Location of Fragmentation-Induced Compatibility Issues for Android Apps. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 278–287. https://doi.org/10.1109/APSEC.2018.00042

[34] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 280–291.

[35] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 318–329. https://doi.org/10.1145/2931037.2931044

[36] Tarek Mahmud, Meiru Che, and Guowei Yang. 2022. ACID: An API Compatibility Issue Detector for Android Apps. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3510454.3516854

[37] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[38] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 77–88.

[39] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2019. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). *ACM Trans. Priv. Secur.* 22, 2, Article 14 (apr 2019), 34 pages. https://doi.org/10.1145/3313391

[40] Felix Pauck and Heike Wehrheim. 2021. Jicer: Simplifying Cooperative Android App Analysis Tasks. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 187–197. https://doi.org/10.1109/SCAM52516.2021.00031

[41] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

[42] Columbus repository. 2023. https://github.com/ucsb-seclab/columbus. Accessed December 2023.

[43] Jordan Samhi. 2023. *Analyzing the Unanalyzable: an Application to Android Apps*. Ph. D. Dissertation. University of Luxembourg, Luxembourg City, Luxembourg. http://hdl.handle.net/10993/54372

[44] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1398–1409. https://doi.org/10.1109/ICSE43902.2021.00126

[45] J. Samhi, T. F. Bissyande, and J. Klein. 2024. AndroLibZoo: A Reliable Dataset of Libraries Based on Software Dependency Analysis. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*.

[46] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1232–1244. https://doi.org/10.1145/3510003.3512766

[47] J. Samhi, L. Li, T. F. Bissyande, and J. Klein. 2022. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 723–735. https://doi.org/10.1145/3510003.3510135

[48] Jordan Samhi and Andreas Zeller. 2024. AndroLog: Android Instrumentation and Code Coverage Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 597â€“601. https://doi.org/10.1145/3663529.3663806

[49] Audio Service. 2023. https://github.com/ryanheise/audio_service. Accessed December 2023.

[50] Dakota Soles. 2023. An Empirical Study of Nondeterministic Behavior and Its Causes in Static Analysis Tools. ECOOP and ISSTA 2023 Student Research Competition.

[51] 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*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.

[52] Unity. 2023. Unity. https://unity.com Accessed December 2023.

[53] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/3460319.3464828

[54] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 738–748. https://doi.org/10.1145/3238147.3240465

[55] Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the Unsoundness of Static Analysis for Android GUIs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Santa Barbara, CA, USA) *(SOAP 2016)*. Association for Computing Machinery, New York, NY, USA, 18–23. https://doi.org/10.1145/2931021.2931026

[56] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[57] Daoyuan Wu, Debin Gao, Robert H. Deng, and Chang Rocky K. C. 2021. When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 543–554. https://doi.org/10.1109/DSN48987.2021.00063

[58] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1054–1076. https://doi.org/10.1109/TSE.2016.2547385

[59] XAMARIN. 2023. https://dotnet.microsoft.com/apps/xamarin. Accessed December 2023.

[60] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. 2022. ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 105–109. https://doi.org/10.1109/ICSE-Companion55297.2022.9793791

[61] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 89–99. https://doi.org/10.1109/ICSE.2015.31

[62] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 89–99. https://doi.org/10.1109/ICSE.2015.31

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