# AndroLog: Android Instrumentation and Code Coverage Analysis

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

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

## ABSTRACT

Dynamic analysis has emerged as a pivotal technique for testing Android apps, enabling the detection of bugs, malicious code, and vulnerabilities. A key metric in evaluating the efficacy of tools employed by both research and practitioner communities for this purpose is *code coverage*. Obtaining code coverage typically requires planting probes within apps to gather coverage data during runtime. Due to the general unavailability of source code to analysts, there is a necessity for instrumenting apps to insert these probes in black-box environments. However, the tools available for such instrumentation are limited in their reliability and require intrusive changes interfering with apps' functionalities.

This paper introduces *AndroLog*, a novel tool developed on top of the *Soot* framework, designed to provide *fine-grained coverage information* at multiple levels, including class, methods, statements, and Android components. In contrast to existing tools, AndroLog leaves the responsibility to test apps to analysts, and its motto is *simplicity*. As demonstrated in this paper, AndroLog can instrument up to 98% of recent Android apps compared to existing tools with 79% and 48% respectively for COSMO and ACVTool. AndroLog also stands out for its potential for future enhancements to increase granularity on demand. We make AndroLog available to the community and provide a video demonstration of AndroLog (see section 8).

## 1 INTRODUCTION

In recent years, analysts have identified numerous threats to Android security, even in official app markets like Google Play, making it easy for malware to reach millions of users. As a result, ensuring the security and privacy of Android devices has become increasingly important. Effective protection against these threats is paramount, as they can have severe consequences for individuals and organizations. Researchers have developed various approaches to combat security and privacy threats in Android apps in response to these threats. These approaches include static analyses [7, 12, 20, 21], dynamic analyses [16, 25], and machine learning techniques [5, 15, 19].

Dynamic analysis is a prevalent technique in Android app analysis [8]. Its widespread use is largely attributed to its effectiveness to uncover app security issues [1, 3, 4, 9, 14, 26]. A key challenge in dynamic analysis is determining the parts of the code being executed, a concept referred to as *code coverage* [22, 27]. With access to source code, analysts can easily modify the code to add probes that track execution behaviors. But in reality, most apps are distributed in a "black-box" manner, i.e., the source code is inaccessible to analysts. This lack of access significantly complicates modifying the code to insert probes. Hence, the ability to insert these probes in a black-box manner is paramount for analyzing apps on a large scale.

The most recent tools *available* (most researchers developed ad-hoc techniques not publicly available) to instrument apps in a black-box manner are BBoxTester [29] (based on Emma [6]), COSMO [18] (based on JaCoCo [11]), and ACVTool [17]. These tools, however, come with a range of limitations and challenges. Firstly, they are significantly outdated, with no maintenance for 8, 4, and 4 years, respectively for BBoxTester, ACVTool, and COSMO. This lack of updates renders them almost inapplicable for recent apps due to the rapid evolution of the Android platform. Secondly, they require complex modifications to the apps (due to the use of existing frameworks to instrument Java programs, such as JaCoCo [11] and Emma [6]). This involves adding new intrusive classes, Android components, resources, and structures in memory, and modifying the AndroidManifest.xml file with additional permissions, which can lead to unexpected or even altered behavior. Thirdly, user experience with existing tools is cumbersome due to their complex setup and usage requirements (e.g., writing on the device's SD card).

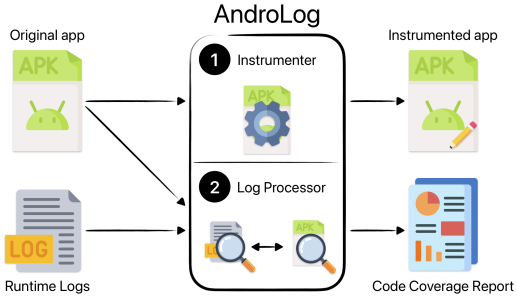


**Figure 1: AndroLog's Architecture. The AndroLog *Instrumenter* ❶ takes an app and instruments it. the AndroLog *Log Processor* ❷ takes the app and the logs produced by the instrumented app to generate a code coverage report.**

We introduce AndroLog, a novel open-source framework for instrumenting Android apps in a black-box manner that overcomes all of the abovementioned deficiencies with a more efficient and simple solution. Indeed, AndroLog is conceptually designed with *simplicity* and *flexibility* in mind. Most importantly, as instrumentation must not interfere with existing code, AndroLog takes a streamlined approach by only adding log statements to monitor app behavior, with no new permissions, resources, Android components, or manifest modification. This minimizes the risk of interfering with apps' original functionalities. Additionally, AndroLog: ① is *test-independent*, allowing analysts to choose whether to use AndroLog for code coverage computation. This approach simplifies the entire process and gives analysts more control over their testing and analysis procedures; ② simplifies the instrumentation and code coverage computation process to just two main steps: (1) providing an APK that is then instrumented into an APK′, and (2) executing APK′ and feeding the resulting logs into AndroLog to obtain a coverage report (note that the resulting logs is a valuable resource that can independently be used for any other downstream task); ③ relies

on the Soot framework [23], ensuring its long-term viability and compatibility with newer apps (Soot is well tested and constantly maintained); and ④ allows for high customization for new levels of granularity as demanded by the evolving needs of dynamic analysis.

In summary, we make the following contributions:

- We introduce *AndroLog*, a black-box instrumentation tool to compute code coverage of Android apps.
- AndroLog works at various *levels of granularity* (e.g., classes, methods, statements, and Android components) and can evolve on demand.
- We show AndroLog's effectiveness in instrumenting the most recent apps compared to existing tools.

AndroLog is conceived as a modern, efficient, easy-to-use, and adaptable tool, addressing the limitations of existing solutions and paving the way for more effective dynamic analysis and code coverage computation in Android apps.

## 2 ARCHITECTURE

AndroLog's architecture is simple and straightforward, it is depicted in Figure 1 and described hereafter.

AndroLog is divided into two main components: ❶ the first component is the instrumenter. Its primary function is to process an APK, the standard file format for Android apps. The instrumenter role involves adding log statements at various levels within the APK (as described in Section 3). The output of this phase is an instrumented APK', ready for testing; and ❷ the second component is the log processor. This module is designed to handle two inputs: the original APK file and the runtime logs collected by the analyst during the testing phase. The responsibility of the log processor is to analyze these inputs and generate a code coverage report.

**AndroLog's Workflow.** The operation of AndroLog is encapsulated in a straightforward, three-step workflow, as depicted in Fig. 2:

(1) **Intrumentation phase.** The workflow begins with the analysts using AndroLog to instrument the original app in order to add probes at various levels of granularity.
(2) **Testing phase.** The next step involves the execution/testing of this modified app on a device or an emulator to collect runtime logs. This is the analyst's responsibility. By design, AndroLog has no role in this phase.
(3) **Code Coverage phase.** The final step is the generation of the code coverage report. Here, the analyst provides AndroLog with two inputs: the original APK and the log report. AndroLog then computes the code coverage report and displays it to the analyst.

**Functionalities** AndroLog allows the analyst to ① sets the output folder of the instrumented app; ② set the log identifier to parse the logs easily; ③ choose what to measure in terms of code coverage (i.e., classes, methods, statements, Android components, etc.); and ④ choose whether to include libraries for computing code coverage.

The interested reader can check our Github page for more information and glance at our online demonstration (see section 8).

## 3 DESIGN

In this section, we give the implementation details of AndroLog. As described in the previous section, AndroLog is made of two main components: ① the *Instrumenter*; and ② the *Log Processor*. The overview of AndroLog's design is depicted in Figure 3; we describe both components hereafter.

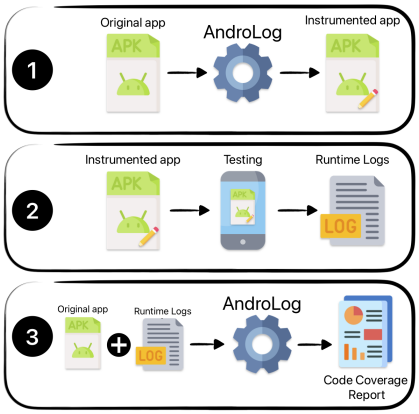


**Figure 2: AndroLog's Workflow. ❶ The analyst uses AndroLog to instrument an app. ❷ The analyst tests the instrumented app and collects runtime logs. ❸ The analyst uses AndroLog to generate the code coverage report.**

```
1  public class LogCheckerClass {
2      private static final Set LOGS = new HashSet();
3
4      public static void log(String log, String tag) {
5          if (LOGS.contains(log)) {
6              return;
7          }
8          Log.d(tag, log);
9          LOGS.add(log);
10     }
11 }
```

**Listing 1: LogCheckerClass injected into apps.**

### 3.1 Instrumenter Component

AndroLog first transforms the Dalvik bytecode into the Jimple IR [24] using the Soot framework. Then, before the logs are inserted, and to ensure that the execution logs of an app are not overloaded and the app is not slowed down, AndroLog injects a special class `LogCheckerClass` that keeps track of elements already logged and only prints a log if it was not previously logged. Listing 1 depicts the class injected. Any instrumentation, described hereafter, then simply involves adding a call to the method `LogCheckerClass.log()` (Line 4 in Listing 1) as depicted in Listing 2.

```
1  public void run() {
2  +   LogCheckerClass.log("METHOD=<e: void run()>", "TAG");
3      [...]
4  }
```

**Listing 2: Example of the instrumentation of a method.**

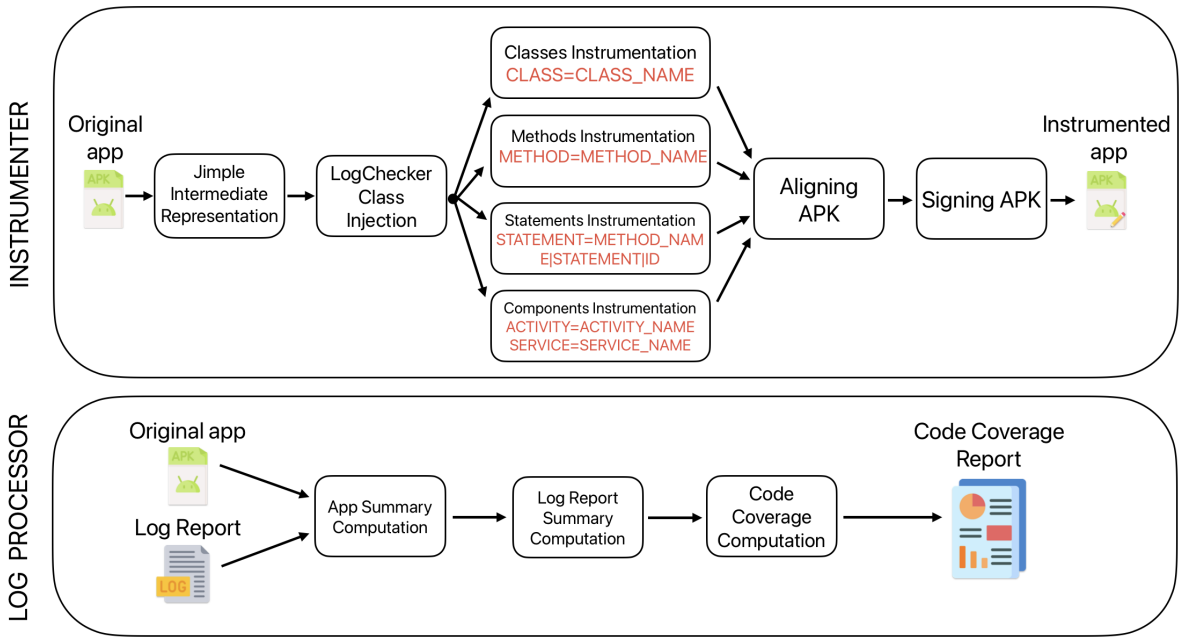


**Figure 3: AndroLog's Design. *Instrumenter:* AndroLog transforms the Dalvik bytecode of an app into Jimple and instruments it. Then, AndroLog aligns and signs the app which is ready for testing. *Log Processor:* AndroLog takes the original app (i.e., the non-instrumented version) and the execution logs to produce the code coverage report.**

The instrumentation process is then divided into four distinct phases, each targeting different app components (activated by the analyst): ① In the first phase, AndroLog iterates over all the classes within the app. For each class, it instruments both their constructors (identified as `void <init>()` in Jimple) and static constructors (`void <clinit>()` in Jimple). The rationale behind this is that any class usage, whether through instantiation or static access, involves these constructors. The instrumentation process involves adding a log statement as the first executed statement in these constructors. For example, a log statement would be: `LogCheckerClass.log("CLASS=com.example.MyClass", "ANDROLOG")`. This ensures that any interaction with the class is logged; ② The second phase involves iterating over all methods in the app. For each method, AndroLog inserts a log statement at the very beginning of the code (see Listing 2). A method log statement would be: `LogCheckerClass.log("METHOD=com.example.Class.foo()", "ANDROLOG")`. This step ensures that the invocation of every method within the app is captured in the logs; ③ The third phase is the most granular, involving the iteration of all statements in all methods of all classes. For each statement, a log statement is added immediately after the statement to be logged. This approach guarantees that if a statement is executed, it will be reflected in the log report. An example log entry would be: `LogCheckerClass.log("STATEMENT=com.example.MyClass.foo()|$r3 = (android.telephony.TelephonyManager) $r2|4", "ANDROLOG")`. The number 4 is the line number that uniquely identifies the statement in the methods (this is needed since there might be similar statements in a given method); and ④ The final phase focuses on Android components such as Activities, Services, BroadcastReceivers, and ContentProviders. AndroLog checks each class to determine if it is an Android component and, if so, adds a log statement in its lifecycle methods. For instance, in an Activity, the log would be added, not only, to the onCreate method, with a log statement like: `LogCheckerClass.log("ACTIVITY=com.example.MyActivity", "ANDROLOG")` or `LogCheckerClass.log("SERVICE=com.example.MyService", "ANDROLOG")`. After these phases are completed, the app is repackaged, aligned, and signed, making it ready for testing, manually or automatically.

## 3.2 Log Processor Component

AndroLog's approach to computing code coverage after an app has been tested is divided into three main steps. The process begins with AndroLog generating a summary of the app. This summary includes quantifying the number of classes, methods, statements, and Android components in the app. This initial summary serves as a baseline for understanding the full scope of the app and computes the code coverage report. The next step involves AndroLog processing the log report previously generated during the app's execution. This log report, provided by the analyst, contains detailed information about the app elements that were executed. By parsing this report, AndroLog identifies which classes, methods, statements, and components were actively used during the app's runtime. In the final step, AndroLog computes the code coverage by comparing the initial app summary with the execution summary derived from the log report. This comparison allows AndroLog to determine the extent to which various app elements were executed. By juxtaposing the execution scope (as outlined in the initial summary) with the actual execution data (from the log report), AndroLog can quantify

the coverage. The outcome of this comparison is a code coverage report presented to the analyst.

## 4 FUTURE RESEARCH OPPORTUNITIES

The introduction of AndroLog creates opportunities for a variety of analytical and exploratory studies. This section outlines two research opportunities that this tool can help address.

**Comparative Effectiveness of Dynamic Analyses:** *How do existing dynamic analysis techniques, such as fuzzing, compare in terms of code coverage for Android applications?* This question uses AndroLog to evaluate and contrast different dynamic analysis methodologies. By assessing the code coverage achieved through these techniques, researchers can better understand their effectiveness in identifying, e.g., vulnerabilities or bugs in Android apps. This comparative analysis would be invaluable in refining dynamic analysis strategies.

**Impact of Code Coverage on Defects Detection:** *What is the impact of code coverage on the detection of runtime behaviors and vulnerabilities?* This question explores the relationship between the extent of code coverage in dynamic analysis and the effectiveness of identifying, e.g., bugs in apps. Using AndroLog to measure this coverage, researchers can gain insights into how thorough code exploration influences the detection of potential security and performance issues.

## 5 PRELIMINARY EMPIRICAL EVALUATION

In this section, we present preliminary empirical results that assess AndroLog into instrumenting apps. To that end, we have randomly selected 2000 recent apps (i.e., apps from 2023) from the AndroZoo dataset [2]. The average size of our dataset is 64 MB, and the median is 56 MB. We performed our experiment on a Linux server (Debian 5.10.0-7-amd64) with an AMD EPYC 7552 48-Core Processor CPU with 96 cores and 630GB of RAM. Subsequently, for each app, we use AndroLog to perform the instrumentation (with all levels of granularity activated), during which we also record the time taken for this instrumentation to complete. AndroLog was able to instrument 1957 apps successfully (i.e., 98%). The average instrumentation time is 34 seconds, and the median is 1 seconds. The failure to successfully instrument 43 apps is attributed to the oversized DEX files generated by Soot, which require splitting for 17 apps. This issue is not a problem in general, except for pre-Lollipop Android apps (API 22). For the remaining 26 apps, the error is due to a bug in Soot related to backward analysis on empty sets.

To compare AndroLog against existing tools, we have considered COSMO, ACVTool, and BBoxTester, and performed the instrumentation step. COSMO could instrument 1588 (79%) apps, and ACVTool could instrument 966 (48%) apps. We encountered difficulties with BBoxTester, while the installation was successful, we faced challenges in the instrumentation process, as it was impossible to instrument any app due to many different crashes and errors. We attribute these problems to obsolescence. All artifacts are available in the project's repository.

## 6 RELATED WORK

This section discusses existing research prototypes to instrument Android apps in a black-box manner.

InsDal [13], CovDroid [28], and the approach by Huang et al. [10] have been introduced as tools for computing code coverage in Android apps, but their unavailability to the public as open-source limits their practical application. InsDal, built on top of ApkTool, operates at the smali level, offering only instrumentation at the class and method levels. Similarly, CovDroid executes its instrumentation at the smali level, inserting probes at the method level. The method devised by Huang et al. also functions at the smali level but necessitates additional modifications to the Android manifest, including integrating new permissions. Their tool demonstrates a limited success rate in instrumenting apps, achieving only 36%.

BBoxTester, introduced in [29], is a black-box code coverage tool for Android apps. It converts Dalvik bytecode into Java bytecode using dex2jar, and then leverages Emma [6] for the instrumentation process. However, BBoxTester's approach necessitates modifying the Android manifest and adding app resources, making it less flexible and potentially intrusive. Similarly, COSMO [18] is designed to transform Dalvik bytecode into Java source code, subsequently using JaCoCo [11] for instrumentation. An aspect of COSMO is its method of recording code coverage summaries directly on the device's SDCard. ACVTool [17] is another tool to instrument Android apps. Operating at the smali level, ACVTool's process requires several steps. This includes modifying the Android manifest to incorporate additional components and new permissions. Furthermore, like COSMO, ACVTool records reports directly onto the device's SDCard. These procedures introduce potential intrusiveness.

AndroLog, in contrast to the tools discussed, is an openly available Android app instrumentation framework that offers non-intrusive, granular-level analysis without requiring additional permissions or Android manifest modifications. AndroLog achieves high instrumentation success with simple two-command-line usage. Built on the well-maintained Soot framework, AndroLog ensures compatibility with newer apps and adaptability for future needs. This approach not only simplifies the instrumentation process but also minimizes interference with the original app functionality.

## 7 CONCLUSION

In this paper, we presented a novel framework, called AndroLog, for automatically inserting logging probes into Android apps to compute code coverage in a black-box manner. AndroLog, built on top of the Soot framework, is designed to be *simple* and to easily evolve on demand. AndroLog can insert probes at different levels of granularity such as classes, methods, statements, and Android components with a 98% success rate. Most importantly, AndroLog adopts a streamlined approach to ensure non-interference with existing app code; it only adds log statements for behavior monitoring, avoiding new permissions, resources, Android components, or manifest modifications, thereby minimizing the risk of affecting the app's original functionality.

## 8 DATA AVAILABILITY

To promote transparency and facilitate reproducibility, we make AndroLog and our artifacts available to the community at:

https://github.com/JordanSamhi/AndroLog

We also provide a video demonstration of AndroLog:

https://www.youtube.com/watch?v=NEcYg98k_o4

# REFERENCES


[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong. 2015. GroddDroid: a gorilla for triggering malicious behaviors. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. 119–127. https://doi.org/10.1109/MALWARE.2015.7413692

[2] 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

[3] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: a platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 916–919. https://doi.org/10.1145/3238147.3240479

[4] Yuning Cui, Yi Sun, and Zhaowen Lin. 2023. DroidHook: a novel API-hook based Android malware dynamic analysis sandbox. *Automated Software Engineering* 30, 1 (24 Feb 2023), 10. https://doi.org/10.1007/s10515-023-00378-w

[5] Nadia Daoudi, Jordan Samhi, Abdoul Kader Kabore, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2021. DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection Based on Image Representation of Bytecode. In *Deployable Machine Learning for Security Defense*, Gang Wang, Arridhana Ciptadi, and Ali Ahmadzadeh (Eds.). Springer International Publishing, Cham, 81–106.

[6] Emma. 2024. Emma: a free Java code coverage tool.http://emma.sourceforge.net. Accessed January 2024.

[7] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. 2016. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2016.7792435

[8] Google. 2023. Google Play Protectionhttps://developers.google.com/android/play-protect/cloud-based-protections. Accessed November 2023.

[9] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. Association for Computing Machinery, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[10] Chun-Ying Huang, Ching-Hsiang Chiu, Chih-Hung Lin, and Han-Wei Tzeng. 2015. Code Coverage Measurement for Android Dynamic Analysis Tools. In *2015 IEEE International Conference on Mobile Services*. 209–216. https://doi.org/10.1109/MobServ.2015.38

[11] JaCoCO. 2024. JaCoCo repositoryhttps://www.eclemma.org/jacoco/. Accessed January 2024.

[12] Hyunjae Kang, Jae wook Jang, Aziz Mohaisen, and Huy Kang Kim. 2015. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 479174. https://doi.org/10.1155/2015/479174 arXiv:https://doi.org/10.1155/2015/479174

[13] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2017. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 502–506. https://doi.org/10.1109/SANER.2017.7884662

[14] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[15] N. Peiravian and X. Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 300–305. https://doi.org/10.1109/ICTAI.2013.53

[16] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security* (Amsterdam, The Netherlands) *(EuroSec '14)*. Association for Computing Machinery, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2592791.2592796

[17] Aleksandr Pilgun, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw. 2020. Fine-Grained Code Coverage Measurement in Automated Black-Box Android Testing. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 23 (jul 2020), 35 pages. https://doi.org/10.1145/3395042

[18] Andrea Romdhana, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella. 2021. COSMO: Code Coverage Made Easier for Android. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 417–423. https://doi.org/10.1109/ICST49551.2021.00053

[19] J. Sahs and L. Khan. 2012. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*. 141–147. https://doi.org/10.1109/EISIC.2012.34

[20] 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

[21] Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé F. Bissyandé, and Jacques Klein. 2022. Demystifying Hidden Sensitive Operations in Android apps. In *ACM Transactions on Software Engineering and Methodology*.

[22] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 86–96.

[23] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[24] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. *no* (1998).

[25] Victor Van Der Veen, Herbert Bos, and Christian Rossow. 2013. Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam* (2013).

[26] Tanapuch Wanwarang, Nataniel P. Borges, Leon Bettscheider, and Andreas Zeller. 2020. Testing Apps With Real-World Inputs. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test* (Seoul, Republic of Korea) *(AST '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3387903.3389310

[27] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[28] Chao-Chun Yeh and Shih-Kun Huang. 2015. CovDroid: A Black-Box Testing Coverage System for Android. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. 447–452. https://doi.org/10.1109/COMPSAC.2015.125

[29] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. Towards Black Box Testing of Android Apps. In *2015 10th International Conference on Availability, Reliability and Security*. 501–510. https://doi.org/10.1109/ARES.2015.70