Received: 10 May 2018 | Revised: 30 October 2018 | Accepted: 4 November 2018
DOI: 10.1002/spe.2672

SHORT COMMUNICATION

WILEY

From source code to test cases: A comprehensive benchmark for resource leak detection in Android apps

Oliviero Riganelli | Daniela Micucci | Leonardo Mariani

Dipartimento Informatica Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Milan, Italy

Correspondence
Oliviero Riganelli, Dipartimento Informatica Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Viale Sarca 336, 20126 Milan, Italy.
Email: oliviero.riganelli@unimib.it

Funding information
Horizon 2020 LEARN Project funded under the European Research Council Consolidator Grant 2014 program, Grant/Award Number: 646867

Summary

Android apps share resources, such as sensors, cameras, and Global Positioning System, that are subject to specific usage policies whose correct implementation is left to programmers. Failing to satisfy these policies may cause resource leaks, that is, apps may acquire but never release resources. This might have different kinds of consequences, such as apps that are unable to use resources or resources that are unnecessarily active wasting battery. Researchers have proposed several techniques to detect and fix resource leaks. However, the unavailability of public benchmarks of faulty apps makes comparison between techniques difficult, if not impossible, and forces researchers to build their own data set to verify the effectiveness of their techniques (thus, making their work burdensome). The aim of our work is to define a public benchmark of Android apps affected by resource leaks. The resulting benchmark, called AppLeak, is publicly available on GitLab and includes faulty apps, versions with bug fixes (when available), test cases to automatically reproduce the leaks, and additional information that may help researchers in their tasks. Overall, the benchmark includes a body of 40 faults that can be exploited to evaluate and compare both static and dynamic analysis techniques for resource leak detection.

KEYWORDS

Android app, benchmark, bug detection, resource leak

1 | INTRODUCTION

Mobile devices are extremely popular and pervasively present in our society. Among the many devices available, the ones that run Android are the largest majority, currently corresponding to 85% of the devices sold worldwide.[1] Indeed, the Android ecosystem is extremely rich, with a huge offer in terms of apps that can be installed and used. For instance, by March 2017, the number of apps available for download in Google Play was equal to 3.7 million.[2]

Many of the available apps interact with the resources available in the device, such as cameras, Global Positioning System, and sensors. In the Android environment, it is a responsibility of the developers to implement these interactions properly. Unfortunately, many apps fail to correctly handle the life cycle of resources, causing misbehaviors and instability in the execution environment.[3-8] In particular, apps are frequently affected by *resource leaks*, that is, apps acquire but never release resources.[6,9] Resource leaks can be responsible for the degradation of the performance of the device, for instance, due to excessive memory or battery utilization, or even for app and system crashes, due to apps that fail to acquire the resources that are permanently acquired by other apps.

In the last years, several static and/or dynamic analysis techniques have been proposed to detect resource leaks.[6,7,10-13] For example, Relda2 is a static analysis technique to detect resource leaks in Android apps,[6] whereas RelFix extends Relda2 with the capability to fix resource leaks[12]; Proactive libraries can detect and heal resource leaks[7]; and EnergyPatch uses a combination of static and dynamic analysis techniques to detect, validate, and repair energy faults caused by misuses of energy-intensive resources.[13]

Most published articles on the detection of resource leaks in mobile apps demonstrate the effectiveness of the techniques through an empirical evaluation. These techniques defined their own set of faulty apps to study the effectiveness of the proposed approaches. This resulted in a waste of effort due to the time spent looking for faulty apps in marketplaces and open-source repositories. Even worse, the different sets of faults that have been selected make it difficult, if not even impossible, to compare these techniques. The construction of a benchmark is thus essential as it provides a common and reliable basis on which to assess and compare the various techniques. For example, different techniques can be compared based on the number of resource leaks they can find, the rate of false alarms, and the performance overhead. Furthermore, the creation of a standard benchmark can lead to progress in this research field, improving science and community cohesion as demonstrated by past experience in different research areas, such as Text Retrieval Conference (TREC) for the information retrieval community[14] and Transaction Processing Performance Council (TPC) for the database community.[15]

DroidLeaks was a first attempt to create a benchmark of Android apps affected by resource leaks.[16] However, DroidLeaks is limited in usability, that is, it contains only the source code of *potentially* faulty apps and does *not include* artifacts that can confirm the presence of faults and facilitate their reproduction. For instance, the executable code of the faulty apps has not been collected, and the test cases that reveal the faults have not been implemented and made available for download. Since resource leaks do not always cause easily recognizable failures (eg, a resource leak may cause a waste of memory or battery, which causes visible misbehaviors only after the app has been used for a long time), making automatic test cases that reveal faults available is extremely important to simplify the use of the benchmark.

This paper describes the methodology we used to collect, analyze, and reproduce several resource leaks affecting Android apps and presents the resulting benchmark called AppLeak, which can be effectively exploited by researchers to evaluate and compare static and dynamic analysis techniques. We created the benchmark from the faulty apps that have been already used to study approaches for resource leak detection and repair, obtaining a total of 26 releases of faulty apps and a total of 40 resource leaks. For each resource leak, we made available the source code and the executable code of both the faulty app and the fixed app when available, the test case that exercises the resource leak, and information about specific configurations that might be needed to reproduce the problem. Compared to other benchmarks that are available, AppLeak originally includes all the artifacts necessary to reproduce resource leaks with minimal effort. In fact, apps are shipped as ready-to-use executables (apk files) associated with automatic system test cases that reproduce problems and the corresponding source code to investigate the cause of the problem. Our benchmark is available for download from https://goo.gl/forms/JZWWaeOK5TMbkacA2.

The interested researchers and practitioners can use AppLeak to download the apps affected by resource leaks from our repository, deploy these apps on an Android emulator, and run the test cases we made available to concretely observe the sequences of actions that reproduce the resource leaks. In addition, they can use our benchmark to assess techniques for resource leak detection and repair. For instance, they can run static analysis techniques on the source code of the apps that we collected to check how many resource leaks could be revealed; they can use dynamic analysis techniques jointly with our test cases to assess the ability of these techniques to detect resource leaks when they happen; they can use test case generation techniques to assess their ability to reveal resource leaks and compare the generated tests with the tests we made available; and, finally, they can use automatic repair techniques to assess their ability to fix resource leaks.

This paper is organized as follows. Section 2 describes the methodology that we followed to build the benchmark. Section 3 describes our benchmark in detail. Section 4 explains how to use the benchmark. Section 5 discusses related work. Finally, Section 6 provides final remarks.

2 | METHODOLOGY

The methodology that we used to build our benchmark of Android apps affected by resource leaks consists of four main steps.

1. *Selection of the eligible sets of apps*. In this step, we identify the repositories and data sets of apps affected by resource leaks that we consider to create our benchmark.
2. *Identification of the apps that satisfy our reproducibility requirements*. In this step, we filter out the apps that do not satisfy our reproducibility requirements from the overall set of apps selected in the previous step.
3. *Compilation and execution of the apps*. In this step, we work on the selected apps to make sure they can be compiled and executed, which are necessary conditions to reproduce failures.
4. *Reproduction of the resource leak*. In this last step, we implement an automatic test case that reproduces each resource leak by interacting with the graphical user interface of the application.

We describe each step in detail below.

Selection of the eligible sets of apps

For our benchmark, we selected Android apps affected by one or more resource leaks, starting from sets of apps that have been already studied in scientific papers. We focused on recent work in the area to select apps developed for the most recent versions of the Android application programming interfaces (APIs). In fact, the Android APIs evolve quickly, at the rate of 115 API updates per month based on the study by McDonnell et al,[17] and running old applications would imply using outdated versions of the Android APIs.

Our search identified three main sets of resource leaks and corresponding applications that can be used to produce a benchmark of *reproduced* and *reproducible* faults: the cases in DroidLeaks by Liu et al,[16] Wu et al,[6] and Banerjee et al.[13] Some of the resource leaks in these sets have been further used in recent studies by Liu et al[12] and Riganelli et al.[7]

DroidLeaks consists of 176 releases of Android apps affected by a total of 292 resource leaks. This data set includes links to the source code of both the faulty and fixed versions. The apk files of the apps and any other artifact that can facilitate the reproduction of the problem are not available. However, the data set specifies the resource affected by the resource leak, the name of the methods that have been modified to implement the fix, and the name of the source file that implements the method to be fixed. In some cases, the link to the bug report is also present.

The data set used by Wu et al[6] consists of 43 Android apps and 67 resource leaks: 35 apps are from the Google Play* and the Chinese Wandoujia† marketplaces, and eight apps are from F-droid.‡ No information is available about the resource leaks present in these apps, and no information is available about the apps beside their names. To have enough information to work with these resource leaks, we contacted the authors who provided us the output generated by their analysis tool for all the open-source cases. The output includes information about the resource that is leaked and the place that generates the leak.

The data set used by Banerjee et al[13] consists of a suite of 12 apps and 13 resource leaks that have an impact on energy consumption and that have been downloaded from various online repositories, such as F-droid, GitHub,§ Google Code,¶ and Google Play. The resource leaks are described succinctly, with one sentence, and only the name and the version of the app affected by the leak are provided.

Since the data sets used by Banerjee et al[13] and by Wu et al[6] share the same release of an app, at the end of this phase, the total numbers of app releases and resource leaks considered to build our benchmark are 231 and 372, respectively.

Identification of the apps that satisfy our reproducibility requirements

This step is quite simple because we only have two requirements that must be satisfied: (1) the apps must be available with the *source code*, and (2) the leaks must occur on Android-specific resources, that is, we only consider resources whose class is in the Android package. This makes our benchmark relevant also to static analysis techniques and not only to testing and dynamic analysis techniques, and specific to Android.

This step reduces the sets of app releases and resource leaks relevant for the benchmark to 133 and 228, respectively.

Compilation and execution of the apps

In order to reproduce the resource leaks, it is mandatory that the apps can be *executed* and that the executed app is obtained from the available source code. We thus worked on the compilation and execution of each app performing the following steps.

1. We download the source code of the app.

*https://play.google.com/store/apps
†http://www.wandoujia.com/apps/
‡https://f-droid.org/
§https://github.com/github
¶https://code.google.com/

- For the apps in DroidLeaks,[16] we had direct access to the source code hosted in GitHub.
- For the apps used by Wu et al[6] and Banjeree et al,[13] we searched in GitHub, exploiting the available information: the app name for the apps used by Wu et al[6] and both the app name and the version for the apps used by Banerjee et al.[13]

2. We imported the downloaded projects into an Android integrated development environment (IDE) to make debugging easier. We used two different Android IDEs according to the type of project: Android Studio,[#] when a gradle script[‖] was present, and Eclipse Android Development Tools,[**] otherwise.
3. We compiled the code of the apps using the selected IDE. When performing this task, we always first checked the presence of compilation instructions that can help us.
4. We worked on the fix of the compilation errors, if any.
   When the app did not compile with the first attempt, we tried to fix the compilation problems. In most of the cases, the fix required one of the following actions: (1) specify in the project properties the same Android version indicated in the Android manifest file or in the grade file, (2) import external libraries (eg, Jars or external projects), (3) fix XML errors (eg, change dp to dpi), and (4) fix simple Java errors (eg, errors in character codification).

After step (1), we managed to successfully download the source code of 129 app releases. Step (2) did not suppress any app because all the 129 apps have been successfully imported into their respective IDEs. After step (3), 23 app releases passed the compilation stage without requiring any fix, whereas 106 failed to compile. In step (4), we fixed 69 of these apps, whereas the remaining 37 apps have been dropped. The fixes that we implemented are distributed as follows (note that we implemented multiple fixes for some apps): seven app releases had an incompatibility between the Android version declared in the configuration file and the one in the project properties, 46 required external libraries, 31 presented errors in the XML files, and 21 presented Java errors.

This activity resulted in 92 app releases that have been compiled. Some of these apps could not be used, although they have been compiled because they either crash at startup or use external services that are no longer available (eg, a mailbox-finding service that is no longer available at www.findcdn.org). This resulted in 62 apps and 122 resource leaks that can be executed.

Reproduction of the resource leak

This is the last step of our process, that is, the *reproduction* of the resource leak and the implementation of an *automatic test case* that reveals the fault. To achieve both these objectives, we performed the following steps.

1. We first *identified the resource that is leaked* by the app. In the case of the resource leaks in DroidLeaks[16] and that in the work of Wu et al,[6] this information is explicit. In the case of the resource leaks reported by Banerjee et al,[13] we manually analyzed the applications based on the description in the paper trying to identify the resource affected by the leak. We had to discard one case where we could not identify the leaked resource.
2. We then identified the *statement in the code where the leaked resource is acquired*. In most of the cases, the identification task could be trivially completed manually.
3. We thus identified the *faulty method*. The faulty method is a method of an Activity that misses to release a resource that the activity has previously acquired. We need to identify this method because the reproduction of the resource leak requires producing an execution that first acquires the resource, that is, executes the statement identified in the previous step, and then misses to release it executing the faulty method, that is, the method identified in this step.
   The identification of the faulty method for the resource leaks in DroidLeaks[16] and the cases considered by Wu et al[6] was trivial since this information was explicitly documented. In the cases reported by Banerjee et al,[13] we manually analyzed the code, looking for a violation of resource utilization policies.
4. Once we had the information about the locations that should be traversed by the execution that exposes the resource leak, we worked on the *identification of the sequence of end-user operations* (ie, interactions with the user interface) that produce that execution. Completing this activity might be challenging because there are many layers that are involved in an execution. To identify the right sequence of actions, in addition to our intuition and experience, we exploited the GitHub repository of the apps, particularly the information in the issue tracker system; Simple tools for code analysis, such as "Call Hierarchy" (Eclipse) and "Find Usage" (Android Studio); and Debugging tools.
   In several cases, it has been impossible to reproduce the resource leak. However, a vast majority of the resource leaks we started from are *potential* resource leaks, that is, they are leaks reported by static analysis tools that have never been

[#] https://developer.android.com/studio/index.html
[‖] https://gradle.org
[**] https://developer.android.com/studio/tools/sdk/eclipse-adt.htm

confirmed. Thus, many of them are likely to be infeasible to reproduce, as confirmed by our experience. We finally managed to fully reproduce 40 resource leaks.

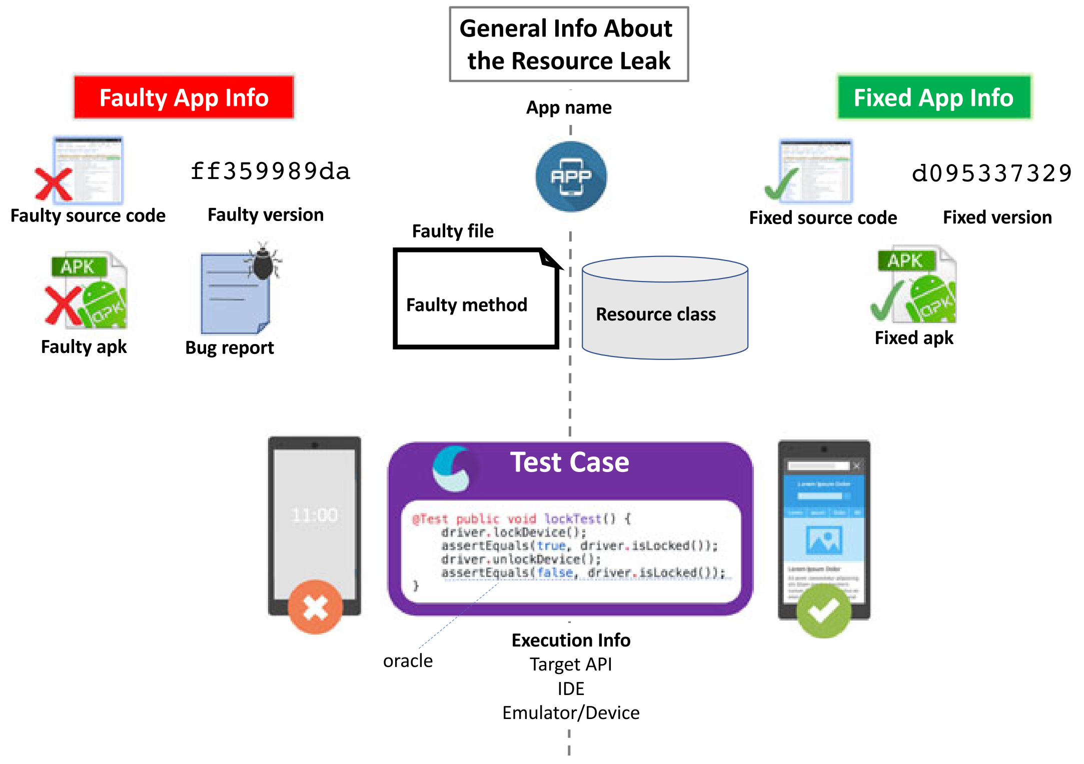
5. We implemented an *automatic test case* for each resource leak we manually reproduced. We used Appium[††] to implement the test case and the Genymotion emulator.[‡‡] When the leaked resource was impossible to emulate with Genymotion (eg, the Global Positioning System), we used a Samsung device running API 21.
6. In the majority of the cases, the reproduction of the leak implied no immediately visible effect on the app and never implied the crash of the app. When possible, we implemented an *oracle* that checks the behavior of the app and makes the test fails when the leak is reproduced (eg, we run `adb` to read the number of wakelocks held by the target app and implement a checkpoint that verifies the presence of the leak). Overall, seven test cases include an oracle, whereas 33 test cases have no oracle.

3 | RESOURCE LEAK BENCHMARK

The benchmark consists of 40 reproducible resource leaks and is hosted on GitLab at the following address: https://goo.gl/forms/JZWWaeOK5TMbkacA2. The root of the project includes a folder for each Android app affected by resource leaks. The name of the folder is the same as the name of the app. The root of the project also hosts the Android_Apps_Leak.csv file that reports information about every resource leak that has been reproduced. In particular, as graphically illustrated in Figure 1, each resource leak is associated with the following information and artifacts.

- General information about the resource leak
  - *App*: the name of the app affected by the resource leak
  - *Resource class*: the name of the Java class that identifies the resource that is leaked
  - *Faulty file*: the name of the file with the class that misses to invoke the release method
  - *Faulty method*: the name of the method in the *Faulty file*, which misses to release the leaked resource
- Faulty app information
  - *Faulty version*: the identifier of the faulty version of the app in the version control system of the app
  - *Faulty source code*: the name of the zip file containing the source code of the faulty version that is available in our benchmark under the folder of the app
  - *Faulty apk*: the name of the apk file corresponding to the *Faulty source code* that is available in our benchmark under the folder of the app
  - *Bug report*: the identifier of the bug report that describes the resource leak
- Fixed app information
  - *Fixed version*: the identifier of the app version with the fix as it appears in the version control system of the app
  - *Fixed source code*: the name of the zip file containing the source code of the fixed version that is available in our benchmark under the folder of the app, which includes the manual fixes that we implemented to compile and execute the app
  - *Fixed apk*: the name of the apk file corresponding to the *Fixed source code* that is available in our benchmark under the folder of the app
- Execution information
  - *Test case*: the name of the zip file that contains an Appium automatic test case that reveals the fault, which is available in our benchmark under the folder of the app
  - *Target (compiled) API*: the version of the Android API declared as target in the manifest file and the version of the Android API that we used to reproduce the failure, respectively
  - *IDE*: the Android IDE that we used to compile the app
  - *Emulator/Device*: it indicates whether the emulator is sufficient to reproduce the failure or a device is required
  - *Oracle*: it indicates if the test case includes an oracle

[††]http://appium.io
[‡‡]https://www.genymotion.com



```
@Test public void lockTest() {
    driver.lockDevice();
    assertEquals(true, driver.isLocked());
    driver.unlockDevice();
    assertEquals(false, driver.isLocked());
}
```

FIGURE 1 Artifacts available in the AppLeak benchmark. API, application programming interface; IDE, integrated development environment [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Two sample resource leaks of the benchmark

	App	**Resource class**		**Faulty file**	**Faulty method**
General info	AnkiDroid	android.database.Cursor		Sched.java	eta()
	Aripuca	android.location.LocationManager		MainActivity.java	onPause()
	Faulty version	**Faulty source code**		**Faulty apk**	**Bug report**
Faulty app info	ff359989da	ff359989da.zip		ff359989da.apk	Pull 275
	0fafe089c	0fafe089c.zip		0fafe089c.apk	N/A
	Fixed version	**Fixed source code**		**Fixed apk**	
Fixed app info	d095337329	d095337329.zip		d095337329.apk	
	N/A	N/A		N/A	
	Test case	**Target (Compiled) API**	**IDE**	**Emulator/Device**	**Oracle**
Execution info	d095337329_eta.zip	18 (18)	Eclipse/ADT	Emulator	no
	test_0fafe089c.zip	17 (18)	Eclipse/ADT	Device	no

Abbreviations: ADT, Android Development Tools; API, application programming interface; IDE, integrated development environment.

Table 1 shows two sample entries of the benchmark. The AnkiDroid app is a flashcard app affected by a cursor leak.[§§] The app erroneously overwrites the reference to a cursor object without closing the cursor first. The cursor is thus indefinitely left open, causing a loss of resources. The fooCam app is an app to record tracks and save waypoints. The app does not use the location service correctly.[¶¶] When the app is paused, the location updates are not disabled; as a consequence, the system location service constantly sends updates to the app even if these updates are no longer necessary.

[§§]https://play.google.com/store/apps/details?id=com.ichi2.anki&hl=en
[¶¶]https://play.google.com/store/apps/details?id=net.phunehehe.foocam2

4 | BENCHMARK USAGE

The benchmark can be used to study the effectiveness of techniques for identifying, analyzing, and fixing resource leaks, based on confirmed and reproduced resource leaks. If the benchmark is used to assess static analysis techniques, the focus would be on the source code of the collected apps, and the test cases could be exploited to confirm that the right resource leaks have been detected. If the benchmark is used to assess testing techniques, the available test cases work as a reference for the study. Otherwise, if the benchmark is used to assess dynamic analysis techniques, the focus would be on the detection of the leaks while reproducing them by running the available test cases. Finally, when evaluating automatic repair techniques,[18] the available test cases could be used to assess the effectiveness of the generated fixes.

In general, the benchmark could be exploited by all the researchers and practitioners who study how to efficiently and effectively address resource leaks in Android applications.

In all the cases, the first step is reproducing the available resource leaks locally. To do this, the recommended procedure is as follows.

1. Clone the benchmark repository in the target computer.
2. If not already present, install Appium (we used version 1.3.1).
3. Configure and run the Appium server. The test cases assume that the app under test runs on the same machine of the Appium client and that the Appium server is available on the port 4723. If it is not the case, the test case must be changed accordingly.
4. Set *ANDROID_HOME* and *adb* in the environment variables.
5. Start the emulator/device according to the attribute *MobileCapabilityType.VERSION* specified in the test case code.
6. Install the apk of the app, launch Appium, and finally launch the test case to reproduce the resource leak.

We illustrated this procedure in a tutorial available on our repository (see file README.md).

5 | RELATED WORK

In this paper, we presented how we built and made available to the community a benchmark with 40 real, reproduced, and reproducible resource leaks affecting Android apps. The benchmark is designed to support research in resource leak detection and fixing. There are two distinct research areas that are related to our contribution: work on the detection of API misuses, which could be exploited to reveal resource leaks, and work on benchmarking, which may have done a work similar to ours for different classes of faults.

Detection of API misuses. Android devices have embedded resources (eg, cameras and sensors) that must be explicitly handled, for instance, acquired and then released. A missing release of such resources can cause system crashes, poor responsiveness, battery drain, and a negative user experience. For this reason, there is a growing body of work on analyzing and/or fixing resource leaks of mobile apps based on static analysis,[6,12] dynamic analysis,[7,8] or a combination of both.[13]

Guo et al[9] developed a static analysis tool, called Relda, to detect resource leaks in Android apps. The approach builds a functional call graph and discovers potential resource leaks by searching depth-first on the graph for paths where the resources are acquired but not released. Wu et al[6] extended this work to an interprocedural analysis to obtain more precise resource leak reports. Riganelli et al[7,8] developed a dynamic technique, called proactive library, that augments classic libraries with the capability of proactively detecting and healing resource leaks at runtime. Proactive libraries collect data from a monitored app, check if the resources are used correctly by the app, and heal executions as soon as an incorrect usage is detected. Banerjee et al[13] developed EnergyPatch, a framework that combines static and dynamic analysis techniques to detect, validate, and repair resource leaks that cause the battery to drain in Android apps. During testing, EnergyPatch extracts a model of the app, automatically exploring the app's graphical user interface. Then, EnergyPatch analyzes this model using a lightweight static analysis technique to detect program paths that could potentially lead to a resource leak. These potentially incorrect program paths are then explored using symbolic execution to confirm the presence of the resource leak.

This body of work on the detection, localization, and fix of resource leaks in Android apps uses empirical evaluation to demonstrate the effectiveness of the technique. However, there is no public data set of real, reproduced, and reproducible resource leaks in Android apps that can facilitate the comparison of these techniques. This motivates our work to produce a benchmark that can be the starting point of providing a common ground for the comparison of techniques that can deal with resource leaks.

Benchmark of faults. A benchmark is an artifact that can serve as a basis for the cost-effective evaluation and comparison of solutions that address the same problem. A good benchmark accepted by the whole community can make assessments more rigorous and convincing, and new ideas can be compared objectively.[19-21] Several benchmarks of software faults have been created in the past for these purposes, such as the ManyBugs and IntroClass benchmarks for evaluating the automated repair of C programs[22] and Defects4J for enabling controlled testing studies for Java programs.[23]

In recent years, there has been little attempts to build benchmarks of real faults from open-source Android apps.[16,24] MUBench[25] is a data set of 89 API misuses that were collected by reviewing over 1200 reports from existing data sets of faults and conducting a developer survey. An API misuse is a usage of library methods that violates the API's contract. Differently from our work, MUBench does not focus on Android projects and API misuses that result in resource leaks. DroidLeaks[16] is the first collection of possible resource leaks in large-scale Android apps. Unfortunately, the resource leaks available in DroidLeaks are only potential leaks that have not been confirmed. Moreover, there is no artifact available for their reproduction. In AppLeak, we used DroidLeaks as one of the sources for the creation of the benchmark. However, all the leaks finally occurring in our benchmark have been confirmed and have been made available in a way that they are easy to reproduce, for instance, they all include an automatic test that reproduces the leak.

6 | CONCLUSION

Android apps are frequently affected by resource leaks, that is, apps acquire resources without timely and properly releasing them.[6,9] In the last few years, several static and dynamic analysis techniques have been proposed to detect resource leaks.[6,7,10-13] Unfortunately, techniques are often experimented on different sets of apps and faults, making comparison extremely hard.

In this paper, we have presented AppLeak, our benchmark of Android resource leaks. Interestingly, our benchmark includes not only the faults but also the apk and automatic test cases to reproduce these faults, delivering a total of 40 reproducible resource leaks that can be used to evaluate both static and dynamic techniques. In particular, AppLeak includes a ready-to-use copy of all the useful artifacts, facilitating the work to any intended user of the repository, who can simply download the apk files and run the test cases to reproduce the resource leaks and exploit the source code of the project to investigate the cause of the problem. The fact that we compiled all the projects before including them in the repository is a further guarantee that any combination of static and dynamic analysis techniques can be successfully experienced with our benchmark. We thus expect AppLeak to support and facilitate advancements in the detection of resource leaks in Android apps.

ACKNOWLEDGEMENT

This work has been partially supported by the Horizon 2020 LEARN Project, which has been funded under the European Research Council (ERC) Consolidator Grant 2014 program (ERC grant agreement 646867).

ORCID

Oliviero Riganelli https://orcid.org/0000-0003-2120-2894
Daniela Micucci https://orcid.org/0000-0003-1261-2234
Leonardo Mariani https://orcid.org/0000-0001-9527-7042

REFERENCES

1. IDC. Smartphone OS market share. 2017. https://www.idc.com/promo/smartphone-market-share/os. Accessed January 8, 2018.
2. AppBrain. Number of Android applications on Google Play. 2018. https://www.appbrain.com/stats/number-of-android-apps. Accessed April 21, 2018.
3. Azim MT, Neamtiu I, Marvel LM. Towards self-healing smartphone software via automated patching. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE); 2014; Vasteras, Sweden.
4. Banerjee A, Chong LK, Chattopadhyay S, Roychoudhury A. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE); 2014; Hong Kong, China.

5. Riganelli O, Micucci D, Mariani L. Healing data loss problems in Android apps. In: Proceedings of the International Workshop on Software Faults (IWSF), co-located with the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2016; Ottawa, Canada.
6. Wu T, Liu J, Xu Z, et al. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Trans Softw Eng*. 2016;42(11):1054-1076.
7. Riganelli O, Micucci D, Mariani L. Policy enforcement with proactive libraries. In: Proceedings of the 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS); 2017; Buenos Aires, Argentina.
8. Riganelli O, Micucci D, Mariani L, Falcone Y. Verifying policy enforcers. In: Proceedings of the 17th International Conference on Runtime Verification (RV); 2017; Seattle, WA.
9. Guo C, Zhang J, Yan J, Zhang Z, Zhang Y. Characterizing and detecting resource leaks in Android applications. In: Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2013; Silicon Valley, CA.
10. Yan D, Yang S, Rountev A. Systematic testing for resource leaks in Android applications. In: Proceedings of the 2013 24th International Symposium on Software Reliability Engineering (ISSRE); 2013; Pasadena, CA.
11. Liu Y, Xu C, Cheung SC, Lü J. GreenDroid: automated diagnosis of energy inefficiency for smartphone applications. *IEEE Trans Softw Eng*. 2014;40(9):911-940.
12. Liu J, Wu T, Yan J, Zhang J. Fixing resource leaks in Android apps with light-weight static analysis and low-overhead instrumentation. In: Proceedings of the 2016 27th International Symposium on Software Reliability Engineering (ISSRE); 2016; Ottawa, Canada.
13. Banerjee A, Chong LK, Ballabriga C, Roychoudhury A. EnergyPatch: repairing resource leaks to improve energy-efficiency of Android apps. *IEEE Trans Software Eng*. 2018;44(5):470-490.
14. Voorhees EM, Harman DK. *TREC: Experiment and Evaluation in Information Retrieval*. Cambridge, MA: The MIT Press; 2005.
15. Gray J. *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA: Morgan Kaufmann Publishers Inc; 1992.
16. Liu Y, Wei L, Xu C, Cheung S-C. DroidLeaks: benchmarking resource leak bugs for Android applications. arXiv:1611.08079. 2016. ePrint.
17. McDonnell T, Ray B, Kim M. An empirical study of API stability and adoption in the Android ecosystem. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM); 2013; Eindhoven, The Netherlands.
18. Gazzola L, Micucci D, Mariani L. Automatic Software Repair: a Survey. *IEEE Trans Softw Eng*. 2017. https://doi.org/10.1109/TSE.2017.2755013
19. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empir Softw Eng*. 2005;10(4):405-435.
20. Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE); 2007; Atlanta, GA.
21. Sim SE, Easterbrook S, Holt RC. Using benchmarking to advance research: a challenge to software engineering. In: Proceedings of the 25th International Conference on Software Engineering (ICSE); 2003; Portland, OR.
22. Le Goues C, Holtschulte N, Smith EK, et al. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans Softw Eng*. 2015;41(12):1236-1256.
23. Just R, Jalali D, Ernst MD. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA); 2014; San Jose, CA.
24. Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M. MUBench: a benchmark for API-misuse detectors. In: Proceedings of the 13th Working Conference on Mining Software Repositories (MSR); 2016; Austin, TX.
25. Amann S. MUBench. 2018. https://github.com/stg-tud/MUBench. Accessed January 8, 2018.

How to cite this article: Riganelli O, Micucci D, Mariani L. From source code to test cases: A comprehensive benchmark for resource leak detection in Android apps. *Softw: Pract Exper*. 2019;49:540–548. https://doi.org/10.1002/spe.2672