Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016

DOI: xxx/xxxx

ARTICLE TYPE

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

Summary

Android apps share resources, such as sensors, cameras, and GPS, 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 both makes comparison between techniques difficult, if not impossible, and forces researchers to build their own dataset 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:
Benchmark, Android App, Resource Leak, Bug Detection

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, GPS, 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 lifecycle of resources, causing misbehaviours and instability in the execution environment[3,4,5,6,7,8]. In particular, apps are frequently affected by *resource leaks*, that is, apps acquire but never release resources[6,9]. Resource leaks can

[†]This work has been partially supported by the H2020 Learn project, which has been funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n. 646867).

be responsible of the degradation of the performance of the device, for instance due to excessive memory or battery utilization, or even of 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 [6,7,12,10,11,13]. For example, Relda2 is a static analysis technique to detect resource leaks in Android apps [6], while 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 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 TREC for the information retrieval (IR) community [14] and 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 have 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 (e.g., a resource leak may cause a waste of memory or battery which causes visible misbehaviours only after the app has been used for 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 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 artefacts 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.

The paper is organized as follows. Section 2 describes the methodology that we followed to build the benchmark. Section 3 describes our benchmark in details. 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 datasets 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 reproducibiity 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 GUI of the application.

We describe each step in details 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 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[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 dataset includes links to the source code of both the faulty and fixed version. The apk files of the apps and any other artefact that can facilitate the reproduction of the problem are not available. However, the dataset 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 dataset used by Wu et al.[6] consists of 43 Android apps and 67 resource leaks: 35 apps are from the Google Play[1] and the Chinese Wandoujia[2] marketplaces, and 8 apps are from F-droid[3]. 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 dataset 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[4], Google Code[5], 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 datasets 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 number 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: (*i*) the apps must be available with the *source code* and (*ii*) 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 set 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.
   - For the apps in DroidLeaks[16], we had direct access to the source code hosted in GitHub.

[1] https://play.google.com/store/apps
[2] http://www.wandoujia.com/apps/
[3] https://f-droid.org/
[4] https://github.com/github
[5] https://code.google.com/

- 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 version for the apps used by Banerjee et al.[13].

2. We imported the downloaded projects into an Android IDE to make debugging easier. We used two different Android IDEs according to the type of project: Android Studio[6], when a gradle script[7] was present, and Eclipse ADT[8], 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 (e.g., Jars or external projects), (3) fix XML errors (e.g., change dp with dpi), (4) fix simple Java errors (e.g., 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 IDE. After step (3), 23 app releases passed the compilation stage without requiring any fix, while 106 failed to compile. In step (4) we fixed 69 of these apps, while 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): 7 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 not available anymore (e.g., a mailbox finding service that is not anymore 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 reource leaks in DroidLeak[16] and in 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 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* (i.e., 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, in particular the information in the issue tracker system; Simple tools for code analysis, such as "Call Hierarchy" (Eclipse) and "Find usage" (Android Studio); Debugging tools.

[6] https://developer.android.com/studio/index.html
[7] https://gradle.org
[8] "https://developer.android.com/studio/tools/sdk/eclipse-adt.htm

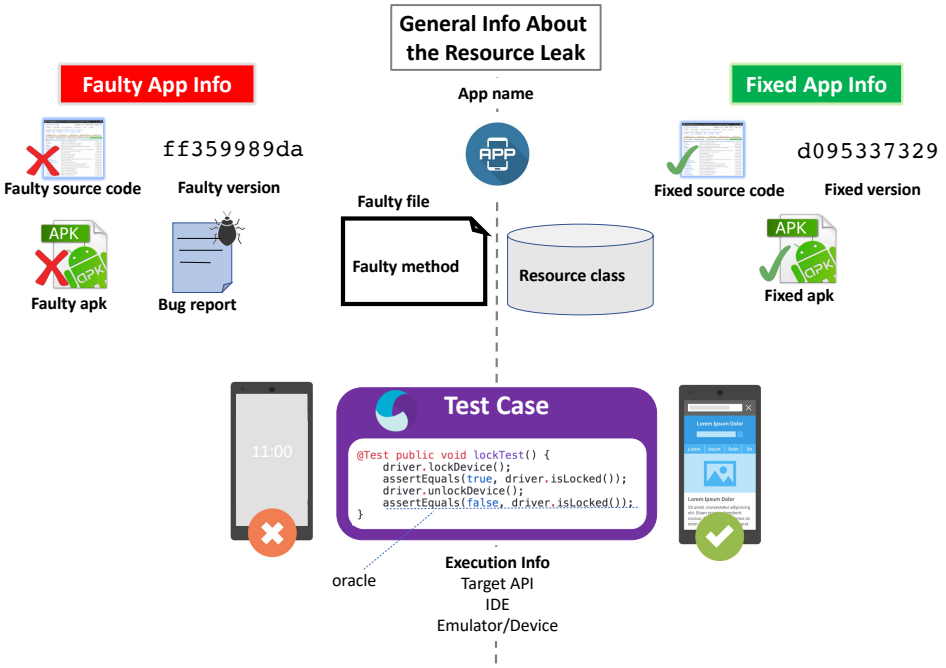


FIGURE 1 Artefacts available in the AppLeak benchmark

In several cases, it has been impossible to reproduce the resource leak. However, the 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 been never 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[9] to implement the test case and the Genymotion emulator[10]. When the leaked resource was impossible to emulate with Genymotion (e.g., the GPS), we used a Samsung device running API 21.

6. In the majority of the cases the reproduction of the leak implied no immediately visibile 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 (e.g., we run `adb` to read the number of wakelocks hold by the target app and implement a checkpoint that verifies the presence of the leak). Overall, 7 test cases include an oracle, while 33 test cases have no oracle.

3 | RESOURCE LEAKS 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 than 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 artefacts:

- General Info 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,

[9] http://appium.io
[10] https://www.genymotion.com

  - *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 info
  - *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 folder of the app,
  - *Bug report*: the identifier of the bug report that describes the resource leak.

- Fixed app info
  - *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, it 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 info
  - *Test case*: the name of the zip file that contains an Appium automatic test case that reveals the fault, it 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.

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

Table 1 shows two sample entries of the benchmark. The AnkiDroid app is a flashcard app affected by a cursor leak[11]. 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[12]. 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.

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 , 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 researches 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 the following one:

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 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 (e.g., 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] extends this work to an inter-procedural analysis to obtaining more precise resource leak reports. Riganelli et al.[7,8] developed a dynamic technique, called proactive library, which augments classic libraries with the capability of proactively detecting and healing resource leaks at runtime. Proactive libraries collect data from a monitored

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

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 technique to detect, validate and repair resource leaks which cause the battery to drain in Android apps. During testing, EnergyPatch extracts a model of the app, automatically exploring the app GUI. 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 dataset 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 for provide a common ground for the comparison of techniques that can deal with resource leaks.

Benchmark of faults A benchmark is an artefact that can serve as a basis for the cost-effective evaluation and comparison of solutions that address a same problem. A good benchmark accepted by the whole community can make assessments more rigorous and convincing, and new ideas can be compared objectively[18,19,20]. Several benchmarks of software faults have been created in the past for these purposes, such as the ManyBugs and IntroClass benchmarks for evaluating automated repair of C programs[21] and Defects4J for enabling controlled testing studies for Java programs[22].

In recent years, there has been little attempts to build benchmarks of real faults from open-source Android apps[16,23]. MUBench[24] is a dataset of 89 API misuses that were collected by reviewing over 1200 reports from existing datasets 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 artefact 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[9,6]. In the last few years, several static and dynamic analysis techniques have been proposed to detect resource leaks[10,11,6,12,7,13]. Unfortunately, techniques are often experimented on different sets of apps and faults, making comparison extremely hard.

In this paper we presented AppLeak, our benchmark of Android resource leaks. Interestingly our benchmark includes not only the faults, but also the apk and the 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 artefacts, 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 technique 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.

References

1. IDC . *IDC. Smartphone OS Market Share.* https://www.idc.com/promo/smartphone-market-share/os. Accessed January 8, 2018; 2017.

2. AppBrain . *Number of Android applications.* https://www.appbrain.com/stats/number-of-android-apps. Accessed April 21, 2018; 2018.

3. Azim M. T., Neamtiu I., Marvel L. M.. Towards Self-healing Smartphone Software via Automated Patching. In: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14); 2014.

4. Banerjee A., Chong L. K., 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'14); 2014.

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 International Symposium on Software Reliability Engineering (ISSRE); 2016.

6. Wu T., Liu J., Xu Z., et al. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering.* 2016;42(11):1054–1076.

7. Riganelli O., Micucci D., Mariani L.. Policy Enforcement with Proactive Libraries. In: Proceedings of the 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17); 2017.

8. Riganelli O., Micucci D., Mariani L., Falcone Y.. Verifying Policy Enforcers. In: Proceedings of the 17th International Conference on Runtime Verification (RV'17); 2017.

9. Guo C., Zhang J., Yan J., Zhang Z., Zhang Y.. Characterizing and Detecting Resource Leaks in Android Applications. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13); 2013.

10. 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 27th International Symposium on Software Reliability Engineering (ISSRE'16); 2016.

11. Yan D., Yang S., Rountev A.. Systematic testing for resource leaks in Android applications. In: Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE'13):411-420; 2013.

12. Liu Y., Xu C., Cheung S. C., Lü J.. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering.* 2014;40(9):911-940.

13. Banerjee A., Chong L. K., Ballabriga C., Roychoudhury A.. EnergyPatch: Repairing Resource Leaks to Improve Energy-efficiency of Android Apps. *IEEE Transactions on Software Engineering.* 2018;44(5):470–490.

14. Voorhees E. M., Harman D. K.. *TREC: Experiment and Evaluation in Information Retrieval.* Cambridge, MA, USA: The MIT Press; 2005.

15. Gray J.. *Benchmark Handbook: For Database and Transaction Processing Systems.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1992.

16. Liu Y., Wei L., Xu C., Cheung S.-C.. DroidLeaks: Benchmarking Resource Leak Bugs for Android Applications. In: eprint arXiv:1611.08079; 2016.

17. McDonnell T., Ray B., Kim M.. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'13); 2013.

18. Gazzola L., Micucci D., Mariani L.. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering.* to appear;.

19. Do H., Elbaum S., Rothermel G.. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering.* 2005;10(4):405–435.

20. Dallmeier V., Zimmermann T.. Extraction of Bug Localization Benchmarks from History. In: Proceedings of theInternational Conference on Automated Software Engineering (ASE'07):433–436; 2007.

21. Sim S. E., Easterbrook S., Holt R. C.. Using benchmarking to advance research: a challenge to software engineering. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03):74-83; 2003.

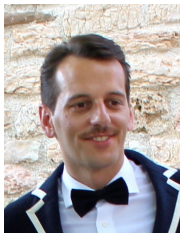
22. Goues C. Le, Holtschulte N., Smith E. K., et al. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering.* 2015;41(12):1236-1256.

23. Just R., Jalali D., Ernst M. D.. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14):437–440; 2014.

24. Amann S., Nadi S., Nguyen H. A., Nguyen T. N., Mezini M.. MUBench: A Benchmark for API-Misuse Detectors. In: Proceedings of the 13th Working Conference on Mining Software Repositories (MSR'16):464-467; 2016.

25. Amann, S. . *MUBench.* https://github.com/stg-tud/MUBench. Accessed January 8, 2018; 2018.

AUTHOR BIOGRAPHY



Oliviero Riganelli. Oliviero Riganelli is an Assistant Professor at the University of Milano Bicocca. He holds a Ph.D, in Computer Science and Complex System from the University of Camerino in 2009. He is a Software Engineer Scientist with a passion for Software Quality and Test Automation. His research interests focus on effective methods for designing and developing high-quality systems, including software testing, static and dynamic software analysis, and self-healing and self-adaptive systems. He is and has been involved in several international research and development projects in close collaboration with leading European information and communication companies. He is currently active in ERC LEARN project, H2020 NGPaaS project and the national PRIN GAUSS project. He is also regularly involved in the program committees of workshops and conferences in his areas of interest.

Daniela Micucci. Daniela Micucci obtained her Ph.D. in Mathematics, Statistics, Computational Sciences and Computer Science from the University of Milano in 2004. She is currently an assistant professor at the University of Milano Bicocca. Her research interests include software engineering, in particular software architectures, real-time systems, and self-healing and self-repairing systems. She is currently active in several European and National projects. She is also regularly involved in the program committees of workshops and conferences in her areas of interest.

Leonardo Mariani. Leonardo Mariani is Full Professor at the University of Milano Bicocca. He holds a Ph.D. in Computer Science received from the same university in 2005. Leonardo Mariani authored more than 50 papers, including papers appeared at top software engineering conference, such as ICSE and ESEC/FSE, and journals, such as IEEE TSE and TOSEM. His research interests include software engineering, in particular software testing, static and dynamic analysis, automated debugging, and self-healing and self-repairing systems. He has been awarded with the ERC Consolidator Grant 2014 and the ERC PoC 2018, and he is currently active in several European and National projects, including the H2020 NGPaaS project and the national PRIN GAUSS project. He is regularly involved in the organizing and program committees of major software engineering conferences.