2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

A Benchmark of Data Loss Bugs for Android Apps

Oliviero Riganelli, Marco Mobilio, Daniela Micucci, Leonardo Mariani
Department of Informatics, Systems and Communication
University of Milano - Bicocca, Milan, Italy
{oliviero.riganelli, marco.mobilio, daniela.micucci, leonardo.mariani}@unimib.it

***Abstract*—Android apps must be able to deal with both *stop event*s, which require immediately stopping the execution of the app without losing state information, and *start event*s, which require resuming the execution of the app at the same point it was stopped. Support to these kinds of events must be explicitly implemented by developers who unfortunately often fail to implement the proper logic for saving and restoring the state of an app. As a consequence apps can lose data when moved to background and then back to foreground (e.g., to answer a call) or when the screen is simply rotated. These faults can be the cause of annoying usability issues and unexpected crashes.**

This paper presents a public benchmark of 110 data loss faults in Android apps that we systematically collected to facilitate research and experimentation with these problems. The benchmark is available on GitLab and includes the faulty apps, the fixed apps (when available), the test cases to automatically reproduce the problems, and additional information that may help researchers in their tasks.

***Index Terms*—Data loss, Android, benchmark, bug detection**

I. Introduction

Interactions with Android apps may produce many *stop event*s, which require suspending the execution of the apps, and *start event*s, which require resuming the execution of the apps. Many frequent situations may trigger these kinds of events: switching between applications, rotating the screen, switching between windows in the same app, and so on.

Handling stop and start events requires implementing the logic necessary to save the state of the app, when the stop event is received, and to restore the state of the app, when the start event is received [1]. This logic must be implemented within the callbacks methods that are invoked by the Android framework when stop and start events occur. For instance, when the user rotates the screen, the Android framework first invokes the `onSaveInstanceState()` callback method to save the state of the current activity, then destroys the activity, adjusts the orientation of the screen, reloads the activity, and finally recovers the saved state by invoking the `onRestoreInstanceState()` callback method. There are indeed several possible causes for data loss problems. For example, the callback methods involved in this process might be missing or might be implemented incorrectly; or framework upgrades may change the generation of the callbacks breaking the logic of the app [2]. Saving and restoring state information might be tricky because each app requires a specific implementation that depends on the nature of the data that must be saved and resumed. In fact, developers can easily miss to properly save or resume some fields for some components and introduce data loss faults in their apps [3]–[5].

A data loss fault causes some of the program variables to lose their values, which are replaced by default values once the app is resumed (e.g., numeric data types are assigned with 0 and objects are assigned with `null`). This might have a range of consequences depending on the variables that are assigned with incorrect values. In the best case, users may just loose the values they entered into the app, in other cases, variables with wrong values might be the cause of incorrect computations and crashes.

In addition to generic approaches for the detection, localization, and repair of Android faults [6]–[11], it is important to design techniques that can help developers with data loss problems [2]–[4], [12]. To be able to experiment with a wide class of faults related to data loss and compare techniques, it is important to exploit publicly available data sets that include extensive sets of apps and faults, covering a variety of cases.

This paper presents the result of our effort in the creation of a public repository of 110 real data loss faults affecting 48 Android apps. The repository is designed to facilitate the reproduction of the data loss faults: it includes the faulty apps, the fixed apps (when available), and test cases that fail on the faulty apps but pass on the fixed apps that any third party can use to automatically reproduce the data loss problems. The repository includes additional information that may help researchers with their tasks, such as the version of both the emulator and the Android API that we used to reproduce the faults. The interested researchers and professionals can access the repository at the following url: https://gitlab.com/learnERC/DataLossRepository and can use our benchmark to evaluate their techniques against data loss defects. We expect our benchmark to support and facilitate the definition of methods and techniques to detect and fix data loss faults in Android apps.

The rest of the paper presents the methodology that we adopted to create the benchmark (Section II), describes the artefacts that are part of the benchmark (Section III), illustrates how the benchmark can be used (Section IV), describes challenges, limitations, and improvements (Section V), and provides final remarks (Section VI).

II. Methodology

The methodology that we used to build our benchmark of apps affected by data loss faults consists of four main steps:

1) *Selection of the eligible apps*. In this step, we identify the repositories and data sets of apps affected by data loss faults that we consider to create our benchmark.

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00087

2) *Identification of the apps that satisfy the 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 data loss faults*. In this last step, for each fault we implement an automatic test case that interacts with the graphical user interface of the application to reproduce it.

We describe each step in detail below.

A. Selection of the eligible apps

To create the benchmark, we considered the Android apps with one or more data loss faults available on F-Droid (https://f-droid.org/en/) in early June 2018. F-Droid is a well-known software repository that at the time of our search contained 1,420 free and open-source apps for the Android platform. In order to select significant and mature app projects, we applied the following selection criteria:

- *Traceable*: the app should have a public version control and issue tracking system.
- *Popular*: The app should have more than 10,000 downloads on Google Play (https://play.google.com/).
- *Maintained*: The app should contain more than 100 code revisions.
- *Non-Trivial*: The app should contain at least 1,000 lines of Java code.

The application of these criteria generated a list of 428 apps from F-Droid whose software repositories are mainly hosted on GitHub (https://github.com/). We thus analyzed the project history of these apps to discover the potential data loss faults by performing keyword search in commits and bug reports. In order to reduce the risk of losing real data loss problems, we leverage on general keywords that are *data*, *loss*, *landscape*, *save*, *rotate*, *screen*, *portrait*, and *restore*, with the aim of maximizing the coverage of data loss issues. We also used *onSaveInstanceState* and *onRestoreInstanceState* as keywords since they are the names of the callback methods implemented to save and restore the app data. For each general keyword, we used the appropriate word forms, such as conjugations and declensions. For example, the keyword *rotate* was also searched for *rotation*, *rotated*, and *rotating*. The search resulted in more than 2,500 cases of potential data loss faults.

B. Identification of the apps that satisfy the reproducibility requirements

In this step, we manually analyze the commits and the bug reports selected in the previous step to make sure that the occurrence of the keywords is not incidental and that the data loss faults are actually present. Since the search operation was conservatively inclusive, for instance selecting every bug report that simply mentions the word *rotate*, we manage to quickly discard most of the entries by inspecting the bug reports, and only a few cases required extensive checks. Our manual analysis finally confirmed the presence of 168 bugs affecting 82 different apps as actual cases of data loss faults.

Out of 428 apps that satisfy our requirements about maturity and significance, 82 apps (19.2%) were affected by at least a data loss fault, which shows the pervasiveness of data loss issues in mobile apps. Data loss faults may be not trivial to localize and repair as witnessed by discussions and commits: faults require on average an online discussion of 3 comments to be clarified (15 comments in the worst case and 1 comment in the best case); fixes can spread over multiple methods and files, in fact they required the modification of at least two files in 68% of the cases, and they required the modification of 44 lines of code on average (1 line of code in the best case, and 1,856 lines of code in the worst case). The studied faults required from 1 day to almost a year to be fixed.

To facilitate the compilation and reproduction tasks, we considered only faulty apps developed in Android Studio (https://developer.android.com/studio/). This produces a negligible reduction of our data set, in fact of these 168 cases, only 16 apps affected by 22 data loss faults were not developed in Android Studio. We thus ended up with 146 data loss faults affecting 78 releases of 66 apps.

C. Compilation and execution of the apps

In order to reproduce data loss faults, it is mandatory that each app release can be *executed* and that the executed app is consistent with the available source code. We thus worked on the compilation and execution of each app release performing the following steps.

1) We downloaded the source code of the 78 app releases affected by the data loss faults from the public version control system of the app linked by F-Droid. When available, we also downloaded the app releases containing the fixes.
2) We imported all the downloaded projects into the Android Studio IDE.
3) We compiled the code of the apps.
   - We first checked the presence of compilation instructions that can help us with this task.
   - We then compiled each app using the Android Studio IDE.
4) When experiencing compilation errors, we tried to fix the app. In total we experienced compilation problems for 39 app releases. We managed to successfully fix the compilation problems for 16 of them by applying one or more of these actions.
   - Change the version of the Gradle (https://gradle.org) plugin in the dependencies (in most cases with version 2.3.3).
   - Change the Gradle version to the Gradle Wrapper if the Android Gradle plugin and Gradle versions are not compatible (in most cases with version 3.3).
   - Update the SDK Build Tools revision in case the version is not compatible.

- Import the Google Maven repository (https://mvnrepository.com/artifact/com.google).
- Download the configuration file google-services.json from Firebase (https://firebase.google.com/) and add to the app directory.
- Update some configurations if obsolete (e.g., 'compile' replaced with 'implementation').
- Configure a build.gradle file in case it is not already configured.

The main problem with unfixed app releases is outdated dependencies that were difficult or impossible to satisfy. This activity finally resulted in 55 app releases of 49 apps that have been successfully compiled.

We then executed all the 55 app releases that we compiled. In particular, we ran 53 app releases on an emulated Google Nexus 5 with Android 6.0, and we ran 2 app releases on an emulated Google Nexus 5 with Android 5.1. This resulted in a total of 116 data loss faults affecting 55 releases of 49 apps that can be potentially reproduced by using the apps.

D. Reproduction of the data loss faults

This is the last and most difficult step of our process, that is, the *reproduction* of the data loss faults and the implementation of the *automatic test case*s that reveal the faults. To achieve both these objectives, we performed the following steps.

1) We extracted *information useful to reproduce the data loss fault*. In particular, we analyze the app to identify the variables whose values are lost and the conditions that cause the data loss. When possible, we started our investigation from the bug report. If the bug report was not available, we started from the comments in the commits and the code that fixes the discovered bugs to infer what values are lost and under what conditions. Overall, we successfully analyzed 116 data loss faults.
2) We worked on the *identification of the sequence of end-user operations* (i.e., interactions with the user interface) that made the app fail because of the data loss. In 6 cases, it was impossible to understand if the data loss indicated in the commit operation could be feasibly reproduced. We however managed to reproduce 110 out of the 116 data loss faults that have been selected.
3) We implemented an *automatic test case* for each data loss we manually reproduced. We used Appium (http://appium.io) and the Genymotion emulator (https://www.genymotion.com) to implement the test case.
4) In all the cases the reproduction of the data loss implied immediately visibile effects on the faulty app. We managed to implement an automatic *oracle* in the vast majority of the cases (98 out of 110). The oracle checks the behavior of the app and makes the test fail when the data loss is reproduced (e.g., we verify that after a rotation the text written in a form is not lost). The few cases without an automatic oracle depend on Appium not being able to read the properties of the widget presenting the data loss problem.

III. Data Loss Benchmark

The benchmark consists of 110 reproducible data loss faults present in 54 releases of 48 different apps and is hosted on GitLab at the following address: https://gitlab.com/learnERC/DataLossRepository.

The root of the project includes a folder for each Android app affected by at least a data loss. The name of the folder is the same as the name of the app. The folder of an app further includes folders with the faulty releases and the fixed releases. The root of the project also hosts the Data_Loss_Apps.html file that reports information about every data loss fault that has been reproduced. In particular, each data loss fault is associated with the following information (see Table I for two sample entries):

- General Info
  - *App Name*: the name of the app affected by the data loss.
  - *Category*: the category to which the app belongs to.
  - *Issue Report*: the identifier of the bug report that describes the data loss.
- Faulty app info
  - *Faulty Version*: the release of the app affected by the data loss fault in the version control system of the app.
  - *Faulty Apk*: the name of the apk file corresponding to the *Faulty Source Code* that is available in our benchmark in the folder 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 in the folder of the app. It includes the manual fixes that we implemented to compile and execute the app.
  - *Faulty Activity*: the name of the Activity class that is affected by the data loss issue.
- 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 Apk*: the name of the apk file corresponding to the *Fixed Source Code* that is available in our benchmark in the folder 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 in the folder of the app. It includes the manual fixes that we implemented to compile and execute the app.
- Execution info
  - *Test Case*: the name of the zip file that contains an Appium automatic test case that reveals the bug. It is available in our benchmark in the folder of the app.
  - *Oracle*: it indicates if the test case includes an oracle.
  - *Tested API*: it indicates the API version of the emulator we used to reproduce the failure.
  - *Target API (Comp - Min)*: the highest API level against which the application was designed, the version of the

Android API with which the app is compiled, and the minimum API level with which the app is compatible, respectively.

TABLE I
TWO SAMPLES OF DATA LOSSES IN THE BENCHMARK.

General info

App Name	Category	Issue Report
Amaze File Manager	Tools	Issue 1034
OpenTasks	Productivity	Issue 658

Faulty app info

Faulty Version	Faulty Apk	Faulty Source Code	Faulty Activity
v3.1.0-beta.1	v3.1.0-beta.1.apk	v3.1.0-beta.1.zip	MainActivity
v1.1.13	v1.1.13.apk	v1.1.13.zip	EditTaskActivity

Fixed app info

Fixed Version	Fixed Apk	Fixed Source Code
884c16c	884c16c.apk	884c16c.zip
N/A	N/A	N/A

Execution info

Test Case	Oracle	Tested API	Target API (Comp-Min)
TestCase 1034	yes	22	25 (25 - 14)
TestCase 658	yes	23	25 (25 - 15)

IV. BENCHMARK USAGE

The benchmark can be used to study the effectiveness of techniques for identifying, analyzing, and fixing faults in Android apps. The first step normally is reproducing the available data loss faults 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) and its dependencies (the JDK and Android SDK).
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 data loss.

The dataset contains all the precompiled apks of the available apps, which were built starting from the releases specified in the Data_loss_Apps.html file of the dataset.

The benchmark can be potentially used to answer a number of research questions. Indeed it allows to study the *effectiveness* and *efficiency* of a range of analysis and testing techniques for Android apps. In particular, the benchmark can be used to assess *static analysis* techniques, since it makes the source code of the faulty and fixed apps available; to assess *dynamic analysis* techniques, since it makes the automatic test cases for failure reproduction available; and *testing* techniques, since it makes the faulty apps available. Moreover, it can be exploited to study the *evolution* of these bugs, since the faulty and fixed versions of the apps are explicitly mapped to the corresponding revisions in GitHub.

V. CHALLENGES, LIMITATIONS AND IMPROVEMENTS

Although there exist benchmarks of vulnerability faults [13] and resource leaks [14] in Android apps, our benchmark *originally* includes data loss faults equipped with automatic test cases and both source and compiled apps with faults and fixes.

To create this benchmark we faced several *challenges*. For instance, we performed a significant magnitude of manual work by manually analyzing thousand of reports and commits, we fixed several tricky compilation problems and packaged the apps in artefacts ready to be used, and we invested significant effort in carefully reproducing all the faults, de facto confirming them, and implementing automatic test cases that can be inexpensively executed by any third party.

A *limitation* of our benchmark is that projects have been compiled with the Android Studio IDE, and users who want to use a different IDE may have to fix some compilation problems. This is anyway a marginal problem since it is not an obstacle to the usage of the artefacts (e.g., the apk and the test cases) that are part of the benchmark.

The benchmark is already quite broad including 110 data loss faults. It could be however further improved including other faults and apps extracted from other official repositories. Another possible improvement is the creation of artefacts that can be executed with virtually no configuration effort, such as making containers with the Android emulators already set up available. However, combining multiple virtualization levels can be problematic, and our benchmark requires only a few operations to be used.

VI. CONCLUSION

Android apps are frequently affected by data loss faults (19.2% of the analyzed projects in our investigation). It is therefore important to have techniques that can detect, localize, and repair these problems. To be able to experiment on a wide class of data loss faults and assess testing and analysis techniques, we produced a publicly available data set that includes an extensive set of apps and faults. Interestingly our benchmark includes not only the faults, but also the apk and the automatic test cases to reproduce these bugs, delivering a total of 110 data loss faults. We expect our benchmark to facilitate progresses in detecting and repairing data loss faults in Android apps.

Acknowledgements: This work has been partially supported by the EU H2020 Learn project, funded under the ERC Consolidator Grant 2014 program (Grant Agreement n. 646867) and the GAUSS national research project, funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX).

REFERENCES

[1] "Understand the Activity Lifecycle," https://developer.android.com/guide/components/activities/activity-lifecycle, [Online; accessed 01-Feb-2019].

[2] O. Riganelli, D. Micucci, and L. Mariani, "Healing data loss problems in android apps," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW'16)*, 2016.

[3] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Proceedings of the European Conference on Computer Systems (EuroSys'14)*, 2014.

[4] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)*, 2015.

[5] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my android application? from GUI failures to code faults," *Softw Test Verif Reliab.*, vol. 28, no. 1, 2018.

[6] P. Machado, J. Campos, and R. Abreu, "MZoltar: Automatic debugging of android application," in *Proceedings of the International Workshop on Software Development Lifecycle for Mobile (DeMobile'13)*, 2013.

[7] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, 2018.

[8] O. Riganelli, D. Micucci, and L. Mariani, "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.

[9] O. Riganelli, D. Micucci, L. Mariani, and Y. Falcone, "Verifying policy enforcers," in *Proceedings of the 17th International Conference on Runtime Verification (RV'17)*, 2017.

[10] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13)*, 2013.

[11] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 1, pp. 34–67, 2019.

[12] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'14)*, 2014.

[13] J. Mitra and V.-P. Ranganath, "Ghera: A repository of android app vulnerability benchmarks," in *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'17)*, 2017.

[14] O. Riganelli, D. Micucci, and L. Mariani, "From source code to test cases: A comprehensive benchmark for resource leak detection in android apps," *Software: Practice and Experience*, vol. 49, no. 3, pp. 540–548, 2019.